An Automata-Theoretic Approach to Infinite-State Systems*

Orna Kupferman[1], Nir Piterman[2], and Moshe Y. Vardi[3]

[1] Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
Email: orna@cs.huji.ac.il, URL: http://www.cs.huji.ac.il/~orna
[2] Imperial College London, Department of Computing, London SW7 2AZ, UK
Email: nir.piterman@doc.ic.ac.uk, URL: http://www.doc.ic.ac.uk/~npiterma
[3] Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/~vardi

Amir has had a profound influence on the three of us, as a teacher, an advisor, a mentor, and a collaborator. His fundamental ideas on the temporal logics of programs have, to a large extent, set the course for our professional careers. His sudden passing away has deprived us of many more years of wonderful interaction, intellectual engagement, and friendship. We miss him profoundly. His wisdom and pleasantness will stay with us forever.

Abstract. In this paper we develop an automata-theoretic framework for reasoning about infinite-state sequential systems. Our framework is based on the observation that states of such systems, which carry a finite but unbounded amount of information, can be viewed as nodes in an infinite tree, and transitions between states can be simulated by finite-state automata. Checking that a system satisfies a temporal property can then be done by an alternating two-way tree automaton that navigates through the tree. We show how this framework can be used to solve the model-checking problem for μ-calculus and LTL specifications with respect to pushdown and prefix-recognizable systems. In order to handle model checking of linear-time specifications, we introduce and study *path automata on trees*. The input to a path automaton is a tree, but the automaton cannot split to copies and it can read only a single path of the tree.
As has been the case with finite-state systems, the automata-theoretic framework is quite versatile. We demonstrate it by solving the realizability and synthesis problems for μ-calculus specifications with respect to prefix-recognizable environments, and extending our framework to handle systems with *regular labeling regular fairness constraints* and μ-calculus with *backward modalities*.

1 Introduction

One of the most significant developments in the area of formal design verification is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [CES86,LP85,QS82,VW86a]. In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal

* The paper is based on the papers [KV00a,KPV02].

logic formula that specifies this behavior (for a survey, see [CGP99]). Symbolic methods that enable model checking of very large state spaces, and the great ease of use of fully algorithmic methods, led to industrial acceptance of temporal model checking [BLM01,CFF$^+$01].

An important research topic over the past decade has been the application of model checking to infinite-state systems. Notable success in this area has been the application of model checking to real-time and hybrid systems (cf. [HHWT95,LPY97]). Another active thrust of research is the application of model checking to *infinite-state sequential systems*. These are systems in which a state carries a finite, but unbounded, amount of information, e.g., a pushdown store. The origin of this thrust is the important result by Müller and Schupp that the monadic second-order theory (MSO) of *context-free graphs* is decidable [MS85]. As the complexity involved in that decidability result is nonelementary, researchers sought decidability results of elementary complexity. This started with Burkart and Steffen, who developed an exponential-time algorithm for model-checking formulas in the *alternation-free* μ-calculus with respect to context-free graphs [BS92]. Researchers then went on to extend this result to the μ-calculus, on one hand, and to more general graphs on the other hand, such as *pushdown graphs* [BS95,Wal96], *regular graphs* [BQ96], and *prefix-recognizable graphs* [Cau96].

On the theoretical side, the limits of MSO decidability have been pushed forward. Walukiewicz and Caucal show that MSO decidability is maintained under certain operations on graphs [Wal02,Cau03]. Further studies of these graphs show that they are the configuration graphs of *high-order pushdown automata* [CW03], and provide an elementary time solution for model checking μ-calculus over these graphs [Cac03]. Recently, the decidability of MSO and μ-calculus with respect to graphs produced by higher-order recursion was established [KNUW05,Ong06].

From a practical point of view, model checking of pushdown graphs (or pushdown systems) provides a framework for software model checking where the store of the pushdown system corresponds to the function call stack. This led to the implementation of pushdown model-checkers such as Mops [CW02], Moped [ES01,Sch02], and Bebop [BR00] (to name a few). Of the mentioned three, the industrial application, Bebop, enables only model checking of safety properties. Successful applications of these model-checkers to the verification of software are reported, for example, in [BR01,CW02,EKS06]. Researchers then considered more expressive logics that are tailored for pushdown graphs [AEM04][4] and showed how to handle restricted cases of communicating pushdown systems [KIG05,BTP06,KG06,KGS06,KG07]. Recently, model checking and analysis of pushdown systems has been shown to have uses also in security and authentication [SSE06,JSWR06]. Extensions like module checking, probabilistic model checking, and exact computational complexity of model checking with respect to branching time logics were studied as well [BMP05,EE05,Boz06].

In this paper, we develop an automata-theoretic framework for reasoning about infinite-state sequential systems. The automata-theoretic approach uses the theory of automata as a unifying paradigm for system specification, verification, and synthesis

[4] See also extensive research on visibly pushdown automata and visibly pushdown languages and games that resulted from the research of this logic [AM04,LMS04,BLS06,AM06,ACM06].

[WVS83,EJ91,Kur94,VW94,KVW00]. Automata enable the separation of the logical and the algorithmic aspects of reasoning about systems, yielding clean and asymptotically optimal algorithms. Automata are the key to techniques such as on-the-fly verification [GPVW95], and they are useful also for modular verification [KV98], partial-order verification [GW94,WW96], verification of real-time systems and hybrid systems [HKV96,DW99], and verification of open systems [AHK97,KV99]. Many decision and synthesis problems have automata-based solutions and no other solution for them is known [EJ88,PR89,KV00b]. Automata-based methods have been implemented in industrial automated-verification tools (c.f., COSPAN [HHK96] and SPIN [Hol97,VB00]).

The automata-theoretic approach, however, has long been thought to be inapplicable for effective reasoning about infinite-state systems. The reason, essentially, lies in the fact that the automata-theoretic techniques involve constructions in which the state space of the system directly influences the state space of the automaton (e.g., when we take the product of a specification automaton with the graph that models the system). On the other hand, the automata we know to handle have finitely many states. The key insight, which enables us to overcome this difficulty, and which is implicit in all previous decidability results in the area of infinite-state sequential systems, is that in spite of the somewhat misleading terminology (e.g., "context-free graphs" and "pushdown graphs"), the classes of infinite-state graphs for which decidability is known can be described by finite-state automata. This is explained by the fact the the states of the graphs that model these systems can be viewed as nodes in an infinite tree and transitions between states can be expressed by finite-state automata. As a result, automata-theoretic techniques can be used to reason about such systems. In particular, we show that various problems related to the analysis of such systems can be reduced to the membership and emptiness problems for *alternating two-way tree automata*, which was shown to be decidable in exponential time [Var98].

We first show how the automata-theoretic framework can be used to solve the μ-calculus model-checking problem with respect to pushdown and prefix-recognizable systems. As explained, the solution is based on the observation that states of such systems correspond to a location in an infinite tree. Transitions of the system, can be simulated by a finite state automaton that reads the infinite tree. Thus, the model-checking problem of μ-calculus over pushdown and prefix-recognizable graphs is reduced to the membership problem of 2-way alternating parity tree automata, namely, the question whether an automaton accepts the tree obtained by unwinding a given finite labeled graph. The complexity of our algorithm matches the complexity of previous algorithms.

The μ-calculus is sufficiently strong to express all properties expressible in the linear temporal logic LTL (and in fact, all properties expressible by an ω-regular language) [Dam94]. Thus, by translating LTL formulas into μ-calculus formulas we can use our solution for μ-calculus model checking in order to solve LTL model checking. This solution, however, is not optimal. This has to do both with the fact that the translation of LTL to μ-calculus is exponential, as well as the fact that our solution for μ-calculus model checking is based on tree automata. A tree automaton splits into several copies when it runs on a tree. While splitting is essential for reasoning about branching properties, it has a computational price. For linear properties, it is sufficient to follow a single computation of the system, and tree automata seem too strong for this task. For example, while the application of the framework developed above to pushdown systems and LTL properties results in an algorithm that is doubly-exponential in the formula and exponential in the system, the problem is known to be EXPTIME-complete in the formula and polynomial in the system [BEM97].

In order to handle model checking of linear-time properties, we introduce *path automata on trees*. The input to a path automaton is a tree, but the automaton cannot split to copies and it can read only a single path of the tree. In particular, *two-way* nondeterministic path automata enable exactly the type of navigation that is required in order to check linear properties of infinite-state sequential systems. We study the expressive power and the complexity of the decision problems for (two way) path automata. The fact that path automata follow a single path in the tree makes them very similar to two-way nondeterministic automata on infinite words. This enables us to reduce the membership problem (whether an automaton accepts the tree obtained by unwinding a given finite labeled graph) of two-way nondeterministic path automata to the emptiness problem of one-way alternating Büchi automata on infinite words, which was studied in [VW86b]. This leads to a quadratic upper bound for the membership problem for two-way nondeterministic path automata.

Using path automata we are able to solve the problem of LTL model checking with respect to pushdown and prefix-recognizable systems by a reduction to the membership problem of two-way nondeterministic path automata. Usually, automata-theoretic solutions to model checking use the emptiness problem, namely whether an automaton accepts some tree. We note that for (linear-time) model checking of sequential infinite-state system both simplifications, to the membership problem vs. the emptiness problem, and to path automata vs. tree automata are crucial: as we prove the emptiness problem for two-way nondeterministic Büchi path automata is EXPTIME-complete, and the membership problem for two-way alternating Büchi tree automata is also EXPTIME-complete[5]. Our automata-theoretic technique matches the known upper bound for model-checking LTL properties on pushdown systems [BEM97,EHRS00]. In addition, the automata-theoretic approach provides the first solution for the case where the system is prefix-recognizable. Specifically, we show that we can solve the model-checking problem of an LTL formula φ with respect to a prefix-recognizable system R of size n in time and space $2^{O(n+|\varphi|)}$. We also prove a matching EXPTIME lower bound.

Usually, the labeling of the state depends on the internal state of the system and the top of the store. Our framework also handles *regular labeling*, where the label depends on whether the word on the store is a member in some regular language. The complexity is exponential in the nondeterministic automata that describe the labeling, matching the known bound for pushdown systems and linear-time specifications [EKS01]. The

[5] In contract, the membership problem for one-way alternating Büchi tree automata can be reduced to the emptiness problem of the 1-letter alternating word automaton obtained by taking the product of the labeled graph that models the tree with the one-way alternating tree automaton [KVW00]. This technique cannot be applied to two-way automata, since they can distinguish between a graph and its unwinding. For a related discussion regarding past-time connectives in branching temporal logics, see [KP95].

automata-theoretic techniques for handling regular labeling and for handling the regular transitions of a prefix-recognizable system are very similar. This leads us to the understanding that regular labeling and prefix-recognizability have exactly the same power. Formally, we prove that model checking (for either μ-calculus or LTL) with respect to a prefix-recognizable system can be reduced to model checking with respect to a pushdown system with regular labeling, and vice versa. For linear-time properties, it is known that model checking of a pushdown system with regular labeling is EXPTIME-complete [EKS01]. Hence, our reductions suggest an alternative proof of the exponential upper and lower bounds for the problem of LTL model checking of prefix-recognizable systems.

While most of the complexity results established for model checking of infinite-state sequential systems using our framework are not new, it appears to be, like the automata-theoretic framework for finite-state systems, very versatile, and it has further potential applications. We proceed by showing how to solve the *realizability* and *synthesis* problem of μ-calculus formulas with respect to infinite-state sequential environments. Similar methods are used to solve realizability of LTL [ATM03]. We discuss how to extend the algorithms to handle graphs with *regular fairness constraints*, and to μ-calculus with *backward modalities*. In both these problems all we demonstrate is a (fairly simple) extension of the basic algorithm; the (exponentially) hard work is then done by the membership-checking algorithm. The automata-theoretic framework for reasoning about infinite-state sequential systems was also extended to global model checking [PV04] and to classes of systems that are more expressive than prefix-recognizable [Cac03,PV03]. It can be easily extended to handle also CARET specifications [AEM04].

Since the publication of the preliminary versions of this work [KV00a,KPV02], this method has been used extensively. Cachat uses the connection between pushdown-systems and 2-way tree automata to show that μ-calculus model checking over high-order pushdown automata is decidable [Cac03]. Gimbert uses these techniques to consider games over pushdown arenas where the winning conditions are combination of parity and unboundedness [Gim03][6]. Serre shows how these techniques can achieve better upper bounds in the restricted case of counter machines [Ser06].

2 Preliminaries

Given a finite set Σ, a *word* over Σ is a finite or infinite sequence of symbols from Σ. We denote by Σ^* the set of finite sequences over Σ and by Σ^ω the set of infinite sequences over Σ. Given a word $w = \sigma_0\sigma_1\sigma_2 \cdots \in \Sigma^* \cup \Sigma^\omega$, we denote by $w_{\geq i}$ the suffix of w starting at σ_i, i.e., $w_{\geq i} = \sigma_i\sigma_{i+1}\cdots$. The *length* of w is denoted by $|w|$ and is defined to be ω for infinite words.

[6] See also [BSW03] for a different solution when the parity conditions are restricted to index three.

2.1 Labeled Transition Graphs and Rewrite Systems

A *labeled transition graph* is $G = \langle \Sigma, S, L, \rho, s_0 \rangle$, where Σ is a finite set of labels, S is a (possibly infinite) set of states, $L : S \rightarrow \Sigma$ is a labeling function, $\rho \subseteq S \times S$ is a transition relation, and $s_0 \in S_0$ is an initial state. When $\rho(s, s')$, we say that s' is a *successor* of s, and s is a *predecessor* of s'. For a state $s \in S$, we denote by $G^s = \langle \Sigma, S, L, \rho, s \rangle$, the graph G with s as its initial state. An s-*computation* is an infinite sequence of states $s_0, s_1, \ldots \in S^\omega$ such that $s_0 = s$ and for all $i \geq 0$, we have $\rho(s_i, s_{i+1})$. An s-computation $s_0, s_1, \ldots$ induces the s-*trace* $L(s_0) \cdot L(s_1) \cdots$. Let $\mathcal{T}_s$ be the set of all s-traces.

A *rewrite system* is $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$, where Σ is a finite set of labels, V is a finite alphabet, Q is a finite set of states, $L : Q \times V^* \rightarrow \Sigma$ is a labeling function, T is a finite set of rewrite rules, to be defined below, q_0 is an initial state, and $x_0 \in V^*$ is an initial word. The set of *configurations* of the system is $Q \times V^*$. Intuitively, the system has finitely many control states and an unbounded store. Thus, in a configuration $(q, x) \in Q \times V^*$ we refer to q as the *control state* and to x as the *store*. A configuration $(q, x) \in Q \times V^*$ indicates that the system is in control state q with store x. We consider here two types of rewrite systems. In a *pushdown* system, each rewrite rule is $\langle q, A, x, q' \rangle \in Q \times V \times V^* \times Q$. Thus, $T \subseteq Q \times V \times V^* \times Q$. In a *prefix-recognizable* system, each rewrite rule is $\langle q, \alpha, \beta, \gamma, q' \rangle \in Q \times \textit{reg}(V) \times \textit{reg}(V) \times \textit{reg}(V) \times Q$, where $\textit{reg}(V)$ is the set of regular expressions over V. Thus, $T \subseteq Q \times \textit{reg}(V) \times \textit{reg}(V) \times \textit{reg}(V) \times Q$. For a word $w \in V^*$ and a regular expression $r \in \textit{reg}(V)$ we write $w \in r$ to denote that w is in the language of the regular expression r. We note that the standard definition of prefix-recognizable systems does not include control states. Indeed, a prefix-recognizable system without states can simulate a prefix-recognizable system with states by having the state as the first letter of the unbounded store. We use prefix-recognizable systems with control states for the sake of uniform notation.

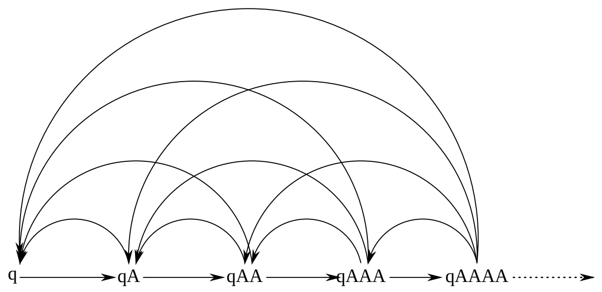
We consider two types of labeling functions, *simple* and *regular*. The labeling function associates with a configuration $(q, x) \in Q \times V^*$ a symbol from Σ. A simple labeling function depends only on the first letter of x. Thus, we may write $L : Q \times (V \cup \{\epsilon\}) \rightarrow \Sigma$. Note that the label is defined also for the case that x is the empty word ϵ. A regular labeling function considers the entire word x but can only refer to its membership in some regular set. Formally, for every state q there is a partition of V^* to $|\Sigma|$ regular languages $R_1, \ldots R_{|\Sigma|}$, and $L(q, x)$ depends on the regular set that x belongs to. For a letter $\sigma \in \Sigma$ and a state $q \in Q$ we set $R_{\sigma,q} = \{x \mid L(q, x) = \sigma\}$ to be the regular language of store contents that produce the label σ (with state q). We are especially interested in the cases where the alphabet Σ is the powerset 2^{AP} of the set of atomic propositions. In this case, we associate with every state q and proposition p a regular language $R_{p,q}$ that contains all the words w for which the proposition p is true in configuration (q, x). Thus $p \in L(q, x)$ iff $x \in R_{p,q}$. Unless mentioned explicitly, the system has a simple labeling.

The rewrite system R induces the labeled transition graph whose states are the configurations of R and whose transitions correspond to rewrite rules. Formally, $G_R = \langle \Sigma, Q \times V^*, L, \rho_R, (q_0, x_0) \rangle$, where $Q \times V^*$ is the set of configurations of R and $\langle (q, z), (q', z') \rangle \in \rho_R$ if there is a rewrite rule $t \in T$ leading from configuration (q, z) to configuration (q', z'). Formally, if R is a pushdown system, then $\rho_R((q, A\cdot y), (q', x\cdot y))$ if $\langle q, A, x, q'\rangle \in T$; and if R is a prefix-recognizable system, then $\rho_R((q, x \cdot y), (q', x' \cdot y))$ if there are regular expressions α, β, and γ such that $x \in \alpha$, $y \in \beta$, $x' \in \gamma$, and $\langle q, \alpha, \beta, \gamma, q'\rangle \in T$. Note that in order to apply a rewrite rule in state $(q, z) \in Q \times V^*$ of a pushdown graph, we only need to match the state q and the first letter of z with the second element of a rule. On the other hand, in an application of a rewrite rule in a prefix-recognizable graph, we have to match the state q and we should find a partition of z to a prefix that belongs to the second element of the rule and a suffix that belongs to the third element. A labeled transition graph that is induced by a pushdown system is called a *pushdown graph*. A labeled transition system that is induced by a prefix-recognizable system is called a *prefix-recognizable graph*.

Example 1. The pushdown system $P = \langle 2^{\{p_1,p_2\}}, \{A, B\}, \{q_0\}, L, T, q_0, A\rangle$, where L is defined by $R_{q_0,p_1} = \{A, B\}^* \cdot B \cdot B \cdot \{A, B\}^*$ and $R_{q_0,p_2} = A \cdot \{A, B\}^*$ and $T = \{\langle q_0, A, AB, q_0\rangle, \langle q_0, A, \varepsilon, q_0\rangle, \langle q_0, B, \varepsilon, q_0\rangle\}$, induces the labeled transition graph below.

$$\begin{tikzcd}
p_2 & p_2 & p_1, p_2 & p_1, p_2 & \\
(q_0,\mathrm{A}) \arrow[r] \arrow[d] & (q_0,\mathrm{AB}) \arrow[r] \arrow[d] & (q_0,\mathrm{ABB}) \arrow[r] \arrow[d] & (q_0,\mathrm{ABBB}) \arrow[d] & \cdots \\
(q_0, \varepsilon) & (q_0,\mathrm{B}) \arrow[l] & (q_0,\mathrm{BB}) \arrow[l] & (q_0,\mathrm{BBB}) \arrow[l] & \cdots \\
 & & p_1 & p_1 &
\end{tikzcd}$$

Example 2. The prefix-recognizable system $\langle 2^{\emptyset}, \{A\}, \{q\}, L, T, q_0, A\rangle$, where $T = \{\langle q, A^*, A^*, \varepsilon, q\rangle, \langle q, \varepsilon, A^*, A, q\rangle\}$ induces the labeled transition graph below.



Consider a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0\rangle$. For a rewrite rule $t_i = \langle s, \alpha_i, \beta_i, \gamma_i, s'\rangle \in T$, let $\mathcal{U}_\lambda = \langle V, Q_\lambda, \eta_\lambda, q^0_\lambda, F_\lambda\rangle$, for $\lambda \in \{\alpha_i, \beta_i, \gamma_i\}$, be the nondeterministic automaton for the language of the regular expression λ. We assume that all initial states have no incoming edges and that all accepting states have no outgoing edges. We collect all the states of all the automata for α, β, and γ regular expressions. Formally, $Q_\alpha = \bigcup_{t_i \in T} Q_{\alpha_i}$, $Q_\beta = \bigcup_{t_i \in T} Q_{\beta_i}$, and $Q_\gamma = \bigcup_{t_i \in T} Q_{\gamma_i}$. We assume that we have an automaton whose language is $\{x_0\}$. We denote the final state of this automaton by x_0 and add all its states to Q_γ. Finally, for a regular labeling function L, a state $q \in Q$, and a letter $\sigma \in \Sigma$, let $\mathcal{U}_{\sigma,q} = \langle V, Q_{\sigma,q}, q^0_{\sigma,q}, \rho_{\sigma,q}, F_{\sigma,q}\rangle$ be the nondeterministic automaton for the language $R_{\sigma,q}$. In a similar way given a state $q \in Q$

and a proposition $p \in AP$, let $\mathcal{U}_{p,q} = \langle V, Q_{p,q}, q^0_{p,q}, \rho_{p,q}, F_{p,q}\rangle$ be the nondeterministic automaton for the language $R_{p,q}$.

We define the *size* $\|T\|$ of T as the space required in order to encode the rewrite rules in T. Thus, in a pushdown system, $\|T\| = \sum_{\langle q,A,x,q'\rangle \in T} |x|$, and in a prefix-recognizable system, $\|T\| = \sum_{\langle q,\alpha,\beta,\gamma,q'\rangle \in T} |\mathcal{U}_\alpha| + |\mathcal{U}_\beta| + |\mathcal{U}_\gamma|$. In the case of a regular labeling function, we also measure the labeling function $\|L\| = \sum_{q\in Q}\sum_{\sigma\in\Sigma} |\mathcal{U}_{\sigma,q}|$ or $\|L\| = \sum_{q\in Q}\sum_{p\in AP} |\mathcal{U}_{p,q}|$.

2.2 Temporal Logics

We give a short introduction to the tempora logics LTL [Pnu77] and μ-calculus [Koz83].

The logic LTL augments propositional logic with temporal quantifiers. Given a finite set AP of propositions, an LTL formula is one of the following.

– $\mathbf{true}$, $\mathbf{false}$, p for all $p \in AP$;
– $\neg\varphi_1,\ \varphi_1 \vee \varphi_2,\ \varphi_1 \wedge \varphi_2,\ \bigcirc\varphi_1$ and $\varphi_1 U \varphi_2$, for LTL formulas φ_1 and φ_2;

The semantics of LTL formulas is defined with respect to an infinite sequence $\pi \in (2^{AP})^\omega$ and a location $i \in \mathbb{N}$. We use $(\pi, i) \models \psi$ to indicate that the word π in the designated location i satisfies the formula ψ.

– For a proposition $p \in AP$, we have $(\pi, i) \models p$ iff $p \in \pi_i$;
– $(\pi, i) \models \neg f_1$ iff not $(\pi, i) \models f_1$;
– $(\pi, i) \models f_1 \vee f_2$ iff $(\pi, i) \models f_1$ or $(\pi, i) \models f_2$;
– $(\pi, i) \models f_1 \wedge f_2$ iff $(\pi, i) \models f_1$ and $(\pi, i) \models f_2$;
– $(\pi, i) \models \bigcirc f_1$ iff $(\pi, i+1) \models f_1$;
– $(\pi, i) \models f_1 \mathcal{U} f_2$ iff there exists $k \geq i$ such that $(\pi, k) \models f_2$ and for all $i \leq j < k$, we have $(\pi, j) \models f_1$;

If $(\pi, 0) \models \psi$, then we say that π *satisfies* ψ. We denote by $L(\psi)$ the set of sequences π that satisfy ψ.

The *μ-calculus* is a modal logic augmented with least and greatest fixpoint operators. Given a finite set AP of atomic propositions and a finite set Var of variables, a μ-calculus formula (in a positive normal form) over AP and Var is one of the following:

– $\mathbf{true}$, $\mathbf{false}$, p and $\neg p$ for all $p \in AP$, or y for all $y \in Var$;
– $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, for μ-calculus formulas φ_1 and φ_2;
– $\Box\varphi$ or $\Diamond\varphi$ for a μ-calculus formula φ.
– $\mu y.\varphi$ or $\nu y.\varphi$, for $y \in Var$ and a μ-calculus formula φ.

A *sentence* is a formula that contains no free variables from Var (that is, every variable is in the scope of some fixed-point operator that binds it). We define the semantics of μ-calculus with respect to a labeled transition graph $G = \langle 2^{AP}, S, L, \rho, s_0\rangle$ and a valuation $\mathcal{V} : Var \to 2^S$. Each formula ψ and valuation $\mathcal{V}$ then define a set $[[\psi]]^G_\mathcal{V}$ of states of G that satisfy the formula. For a valuation $\mathcal{V}$, a variable $y \in Var$, and a set $S' \subseteq S$, we denote by $\mathcal{V}[y \leftarrow S']$ the valuation obtained from $\mathcal{V}$ by assigning S' to y. The mapping $[[\psi]]^G_\mathcal{V}$ is defined inductively as follows:

- $[[\mathbf{true}]]^G_\mathcal{V} = S$ and $[[\mathbf{false}]]^G_\mathcal{V} = \emptyset$;
- For $y \in Var$, we have $[[y]]^G_\mathcal{V} = \mathcal{V}(y)$;
- For $p \in AP$, we have $[[p]]^G_\mathcal{V} = \{s \mid p \in L(s)\}$ and $[[\neg p]]^G_\mathcal{V} = \{s \mid p \notin L(s)\}$;
- $[[\psi_1 \wedge \psi_2]]^G_\mathcal{V} = [[\psi_1]]^G_\mathcal{V} \cap [[\psi_2]]^G_\mathcal{V}$;
- $[[\psi_1 \vee \psi_2]]^G_\mathcal{V} = [[\psi_1]]^G_\mathcal{V} \cup [[\psi_2]]^G_\mathcal{V}$;
- $[[\Box\psi]]^G_\mathcal{V} = \{s \in S :$ for all s' such that $\rho(s, s'),$ we have $s' \in [[\psi]]^G_\mathcal{V}\}$;
- $[[\Diamond\psi]]^G_\mathcal{V} = \{s \in S :$ there is s' such that $\rho(s, s')$ and $s' \in [[\psi]]^G_\mathcal{V}\}$;
- $[[\mu y.\psi]]^G_\mathcal{V} = \bigcap\{S' \subseteq S : [[\psi]]^G_{\mathcal{V}[y \leftarrow S']} \subseteq S'\}$;
- $[[\nu y.\psi]]^G_\mathcal{V} = \bigcup\{S' \subseteq S : S' \subseteq [[\psi]]^G_{\mathcal{V}[y \leftarrow S']}\}$.

The *alternation depth* of a formula is the number of alternations in the nesting of least and greatest fixpoints. For a full exposition of μ-calculus we refer the reader to [Eme97].

Note that $[[\psi]]^G_\mathcal{V}$ depends only on the valuations of the free variables in ψ. In particular, no valuation is required for a sentence and we write $[[\psi]]^G$. For a state $s \in S$ and a sentence ψ, we say that ψ holds at s in G, denoted $G, s \models \psi$ iff $s \in [[\psi]]^G$. Also, $G \models \psi$ iff $G, s_0 \models \psi$. We say that a rewrite system R satisfies a μ-calculus formula ψ if $G_R \models \psi$.

While LTL is a linear temporal logic and we have defined its semantics with respect to infinite sequences, we often refer also to satisfaction of LTL formulas in labeled transition graphs. Intuitively, all the sequences induced by computations of the graph should satisfy the formula. Formally, given a graph G and a state s of G, we say that s satisfies an LTL formula φ, denoted $(G, s) \models \varphi$, iff $\mathcal{T}_s \subseteq \mathcal{L}(\varphi)$. A graph G satisfies an LTL formula φ, denoted $G \models \varphi$, iff its initial state satisfies it; that is $(G, s_0) \models \varphi$.

The *model-checking problem* for a labeled transition graph G and an LTL or μ-calculus formula φ is to determine whether G satisfies φ. Note that the transition relation of R need not be total. There may be finite paths but satisfaction is determined only with respect to infinite paths. In particular, if the graph has only finite paths, its set of traces is empty and the graph satisfies every LTL formula [7]. We say that a rewrite system R satisfies an LTL formula φ if $G_R \models \varphi$. [8]

Theorem 1. *The model-checking problem for a pushdown system R and a formula φ is solvable*

- *in time $\|T\|^3 \cdot 2^{O(|\varphi|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|)}$ in the case φ is an LTL formula and L is a simple labeling function* [EHRS00].
- *in time $\|T\|^3 \cdot 2^{O(\|L\|+|\varphi|)}$ and space $\|T\|^2 \cdot 2^{O(\|L\|+|\varphi|)}$ in the case φ is an LTL formulas and L is a regular labeling function. The problem is EXPTIME-hard in $\|L\|$ even for a fixed formula* [EKS01].
- *in time $2^{O(\|T\| \cdot |\psi| \cdot k)}$ in the case φ is a μ-calculus formula with alternation depth k* [Wal96,Bur97].

[7] It is also possible to consider finite paths. In this case, the nondeterministic Büchi automaton in Theorem 5 has to be modified so that it can recognize also finite words (cf. [GO03]). Our results are easily extended to consider also finite paths.

[8] Some work on verification of infinite-state system (e.g., [EHRS00]), consider properties given by nondeterministic Büchi word automata, rather than LTL formulas. Since we anyway translate LTL formulas to automata, we can easily handle also properties given by automata.

2.3 Alternating Two-Way Automata

Given a finite set Υ of directions, an *Υ-tree* is a set $T \subseteq \Upsilon^*$ such that if $\upsilon \cdot x \in T$, where $\upsilon \in \Upsilon$ and $x \in \Upsilon^*$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $\upsilon \in \Upsilon$ and $x \in T$, the node x is the *parent* of $\upsilon \cdot x$. Each node $x \neq \varepsilon$ of T has a *direction* in Υ. The direction of the root is the symbol $\perp$ (we assume that $\perp \notin \Upsilon$). The direction of a node $\upsilon \cdot x$ is υ. We denote by $dir(x)$ the direction of node x. An Υ-tree T is a *full infinite tree* if $T = \Upsilon^*$. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$ there exists a unique $\upsilon \in \Upsilon$ such that $\upsilon \cdot x \in \pi$. Note that our definitions here reverse the standard definitions (e.g., when $\Upsilon = \{0, 1\}$, the successors of the node 0 are 00 and 10 (rather than 00 and 01)[9].

Given two finite sets Υ and Σ, a *Σ-labeled Υ-tree* is a pair $\langle T, \tau \rangle$ where T is an Υ-tree and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ. When Υ and Σ are not important or clear from the context, we call $\langle T, \tau \rangle$ a labeled tree. We say that an $((\Upsilon \cup \{\perp\}) \times \Sigma)$-labeled Υ-tree $\langle T, \tau \rangle$ is *Υ-exhaustive* if for every node $x \in T$, we have $\tau(x) \in \{dir(x)\} \times \Sigma$.

A labeled tree is *regular* if it is the unwinding of some finite labeled graph. More formally, a *transducer* $\mathcal{D}$ is a tuple $\langle \Upsilon, \Sigma, Q, \eta, q_0, L \rangle$, where Υ is a finite set of directions, Σ is a finite alphabet, Q is a finite set of states, $\eta : Q \times \Upsilon \to Q$ is a deterministic transition function, $q_0 \in Q$ is a start state, and $L : Q \to \Sigma$ is a labeling function. We define $\eta : \Upsilon^* \to Q$ in the standard way: $\eta(\varepsilon) = q_0$ and $\eta(ax) = \eta(\eta(x), a)$. Intuitively, a transducer is a labeled finite graph with a designated start node, where the edges are labeled by Υ and the nodes are labeled by Σ. A Σ-labeled Υ-tree $\langle \Upsilon^*, \tau \rangle$ is regular if there exists a transducer $\mathcal{D} = \langle \Upsilon, \Sigma, Q, \eta, q_0, L \rangle$, such that for every $x \in \Upsilon^*$, we have $\tau(x) = L(\eta(x))$. The size of $\langle \Upsilon^*, \tau \rangle$, denoted $\|\tau\|$, is $|Q|$, the number of states of $\mathcal{D}$.

Alternating automata on infinite trees generalize nondeterministic tree automata and were first introduced in [MS87]. Here we describe alternating *two-way* tree automata. For a finite set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas $\mathbf{true}$ and $\mathbf{false}$, and, as usual, $\wedge$ has precedence over $\vee$. For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y *satisfies* θ iff assigning $\mathbf{true}$ to elements in Y and assigning $\mathbf{false}$ to elements in $X \setminus Y$ makes θ true. For a set Υ of directions, the *extension* of Υ is the set $ext(\Upsilon) = \Upsilon \cup \{\varepsilon, \uparrow\}$ (we assume that $\Upsilon \cap \{\varepsilon, \uparrow\} = \emptyset$). An *alternating two-way automaton* over Σ-labeled Υ-trees is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(ext(\Upsilon) \times Q)$ is the transition function, $q_0 \in Q$ is an initial state, and F specifies the acceptance condition.

A run of an alternating automaton $\mathcal{A}$ over a labeled tree $\langle \Upsilon^*, \tau \rangle$ is a labeled tree $\langle T_r, r \rangle$ in which every node is labeled by an element of $\Upsilon^* \times Q$. A node in T_r, labeled by (x, q), describes a copy of the automaton that is in the state q and reads the node x of Υ^*. Note that many nodes of T_r can correspond to the same node of Υ^*; there is no one-to-one correspondence between the nodes of the run and the nodes of the tree. The labels of a node and its successors have to satisfy the transition function. Formally, a

[9] As will get clearer in the sequel, the reason for that is that rewrite rules refer to the prefix of words.

run $\langle T_r, r\rangle$ is a Σ_r-labeled Γ-tree, for some set Γ of directions, where $\Sigma_r = \Upsilon^* \times Q$ and $\langle T_r, r\rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (\varepsilon, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (x, q)$ and $\delta(q, \tau(x)) = \theta$. Then there is a (possibly empty) set $S \subseteq ext(\Upsilon) \times Q$, such that S satisfies θ, and for all $\langle c, q'\rangle \in S$, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and the following hold:
   – If $c \in \Upsilon$, then $r(\gamma \cdot y) = (c \cdot x, q')$.
   – If $c = \varepsilon$, then $r(\gamma \cdot y) = (x, q')$.
   – If $c = \uparrow$, then $x = \upsilon \cdot z$, for some $\upsilon \in \Upsilon$ and $z \in \Upsilon^*$, and $r(\gamma \cdot y) = (z, q')$.

Thus, ε-transitions leave the automaton on the same node of the input tree, and $\uparrow$-transitions take it up to the parent node. Note that the automaton cannot go up the root of the input tree, as whenever $c = \uparrow$, we require that $x \neq \varepsilon$.

A run $\langle T_r, r\rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. We consider here *parity* acceptance conditions [EJ91]. A parity condition over a state set Q is a finite sequence $F = \{F_1, F_2, \ldots, F_m\}$ of subsets of Q, where $F_1 \subseteq F_2 \subseteq \ldots \subseteq F_m = Q$. The number m of sets is called the *index* of $\mathcal{A}$. Given a run $\langle T_r, r\rangle$ and an infinite path $\pi \subseteq T_r$, let $inf(\pi) \subseteq Q$ be such that $q \in inf(\pi)$ if and only if there are infinitely many $y \in \pi$ for which $r(y) \in \Upsilon^* \times \{q\}$. That is, $inf(\pi)$ contains exactly all the states that appear infinitely often in π. A path π satisfies the condition F if there is an even i for which $inf(\pi) \cap F_i \neq \emptyset$ and $inf(\pi) \cap F_{i-1} = \emptyset$. An automaton accepts a labeled tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts. The automaton $\mathcal{A}$ is *nonempty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$. *Büchi* acceptance condition [Büc62] is a private case of parity of index 3. Büchi condition $F \subseteq Q$ is equivalent to parity condition $\langle \emptyset, F, Q\rangle$. A path π satisfies Büchi condition F iff $inf(\pi) \cap F \neq \emptyset$. *Co-Büchi* acceptance condition is the dual of Büchi. Co-Büchi condition $F \subseteq Q$ is equivalent to parity condition $\langle F, Q\rangle$. A path π satisfies co-Büchi condition F iff $inf(\pi) \cap F = \emptyset$.

The size of an automaton is determined by the number of its states and the size of its transition function. The size of the transition function is $\eta = \Sigma_{q \in Q} \Sigma_{\sigma \in \Sigma} |\eta(q, a)|$ where, for a formula in $B^+(ext(\Upsilon) \times Q)$ we define $|(\Delta, q)| = |\mathbf{true}| = |\mathbf{false}| = 1$ and $|\theta_1 \vee \theta_2| = |\theta_1 \wedge \theta_2| = |\theta_1| + |\theta_2| + 1$.

We say that $\mathcal{A}$ is advancing if δ is restricted to formulas in $B^+((\Upsilon \cup \{\varepsilon\}) \times Q)$, it is one-way if δ is restricted to formulas in $B^+(\Upsilon \times Q)$. We say that $\mathcal{A}$ is nondeterministic if its transitions are of the form $\bigvee_{i \in I} \bigwedge_{\upsilon \in \Upsilon} (\upsilon, q^i_\upsilon))$, in such cases we write $\delta : Q \times \Sigma \to 2^{Q^{|\Upsilon|}}$. In particular, a nondeterministic automaton is 1-way. It is easy to see that a run tree of a nondeterministic tree automaton visits every node in the input tree exactly once. Hence, a run of a nondeterministic tree automaton on tree $\langle T, \tau\rangle$ is $\langle T, r\rangle$ where $r : T \to Q$. Note, that τ and r use the same domain T. In the case that $|\Upsilon| = 1$, $\mathcal{A}$ is a word automaton. In the run of a word automaton, the location of the automaton on the word is identified by the length of its location. Hence, instead of marking the location by υ^i, we mark it by i. Formally, a run of a word automaton is $\langle T, r\rangle$ where $r : T \to \mathbb{N} \times Q$ and a node $x \in T$ such that $r(x) = (i, q)$ signifies that the automaton in state q is reading the ith letter of the word. In the case of word automata, there is only one direction $\upsilon \in \Upsilon$. Hence, we replace the atoms $(d, q) \in ext(\Upsilon) \times Q$ in the transition of $\mathcal{A}$ by atoms from $\{-1,0,1\} \times Q$ where -1 means read the previous letter, 0 means read again the same letter, and 1 means read the next letter. Accordingly, the pair $(i,q),(j,q')$ satisfies the transition of $\mathcal{A}$ if there exists $(d,q') \in \delta(q,w_i)$ such that $j = i + d$. In the case that the automaton is 1-way the length of x uniquely identifies the location in the word. That is, we use $r : T \to Q$ and $r(x) = q$ signifies that state q is reading letter $|x|$. In the case that a word automaton is nondeterministic, its run is an infinite sequence of locations and states. Namely, $r = (0,q_0),(i_1,q_1),\ldots$. In addition, if the automaton is 1-way the location in the sequence identifies the letter read by the automaton and we write $r = q_0, q_1, \ldots$.

Theorem 2. *Given an alternating two-way parity tree automaton $\mathcal{A}$ with n states and index k, we can construct an equivalent nondeterministic one-way parity tree automaton whose number of states is exponential in nk and whose index is linear in nk* [Var98], *and we can check the nonemptiness of $\mathcal{A}$ in time exponential in nk* [EJS93].

We use acronyms in $\{2,\varepsilon,1\} \times \{A,N,D\} \times \{P,B,C,F\} \times \{T,W\}$ to denote the different types of automata. The first symbol stands for the type of movement: 2 for 2-way automata, ε for advancing, and 1 for 1-way (we often omit the 1). The second symbol stands for the branching mode: A for alternating, N for nondeterministic, and D for deterministic. The third symbol stands for the type of acceptance: P for parity, B for Büchi, C for co-Büchi, and F for finite (i.e., automata that read finite words or trees), and the last symbol stands for the object the automaton is reading: T for trees and W for words. For example, a 2APT is a 2-way alternating parity tree automaton and an NBW is a 1-way nondeterministic Büchi word automaton.

The *membership problem* of an automaton $\mathcal{A}$ and a regular tree $\langle \Upsilon^*, \tau \rangle$ is to determine whether $\mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$; or equivalently whether $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(A)$. It is not hard to see that the membership problem for a 2APT can be solved by a reduction to the emptiness problem. Formally we have the following.

Theorem 3. *Given an alternating two-way parity tree automaton $\mathcal{A}$ with n states and index k, and a regular tree $\langle \Upsilon^*, \tau \rangle$ we can check whether $\mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$ in time $(\|\tau\| nk)^{O((nk)^2)}$.*

Proof: Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ be a 2APT and $\langle \Upsilon^*, \tau \rangle$ be a regular tree. Let the transducer inducing the labeling of τ be $\mathcal{D}_\tau = \langle \Upsilon, \Sigma, D, \eta, d_0, L \rangle$. According to Theorem 2, we construct a 1NPT $N = \langle \Sigma, S, \rho, s_0, \alpha \rangle$ that accepts the language of A.

Consider the 1NPT $N' = \langle \{a\}, D \times S, \rho', (d_0, s_0), \alpha' \rangle$ where $\rho'(d,s)$ is obtained from $\rho(s, L(d))$ by replacing every atom (υ, s') by $(\upsilon, (\eta(d,\upsilon), s'))$ and α' is obtained from α by replacing every set F by the set $D \times F$. It follows that $\langle \Upsilon^*, \tau \rangle$ is accepted by $\mathcal{A}$ iff N' is not empty. The number of states of N' is $\|\tau\|(nk)^{O(nk)}$ and its index is $O(nk)$. $\square$

2.4 Alternating Automata on Labeled Transition Graphs

Consider a labeled transition graph $G = \langle \Sigma, S, L, \rho, s_0 \rangle$. Let $\Delta = \{\varepsilon, \Box, \Diamond\}$. An alternating automaton on labeled transition graphs (*graph automaton*, for short) [JW95][10] is

[10] The graph automata in [JW95] are different than these defined here, but this is only a technical difference.

a tuple $\mathcal{S} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ, Q, q_0, and F are as in alternating two-way automata, and $\delta : Q \times \Sigma \to \mathcal{B}^+(\Delta \times Q)$ is the transition function. Intuitively, when $\mathcal{S}$ is in state q and it reads a state s of G, fulfilling an atom $\langle \Diamond, t \rangle$ (or $\Diamond t$, for short) requires $\mathcal{S}$ to send a copy in state t to some successor of s. Similarly, fulfilling an atom $\Box t$ requires $\mathcal{S}$ to send copies in state t to all the successors of s. Thus, like symmetric automata [DW99,Wil99], graph automata cannot distinguish between the various successors of a state and treat them in an existential or universal way.

Like runs of alternating two-way automata, a run of a graph automaton $\mathcal{S}$ over a labeled transition graph $G = \langle S, L, \rho, s_0 \rangle$ is a labeled tree in which every node is labeled by an element of $S \times Q$. A node labeled by (s, q), describes a copy of the automaton that is in the state q of $\mathcal{S}$ and reads the state s of G. Formally, a run is a Σ_r-labeled Γ-tree $\langle T_r, r \rangle$, where Γ is an arbitrary set of directions, $\Sigma_r = S \times Q$, and $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (s_0, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (s, q)$ and $\delta(q, L(s)) = \theta$. Then there is a (possibly empty) set $S \subseteq \Delta \times Q$, such that S satisfies θ, and for all $\langle c, q' \rangle \in S$, the following hold:
   – If $c = \varepsilon$, then there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s, q')$.
   – If $c = \Box$, then for every successor s' of s, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.
   – If $c = \Diamond$, then there is a successor s' of s and $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.

A run $\langle T_r, r \rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. The graph G is accepted by $\mathcal{S}$ if there is an accepting run on it. We denote by $\mathcal{L}(\mathcal{S})$ the set of all graphs that $\mathcal{S}$ accepts. We denote by $\mathcal{S}^q = \langle \Sigma, Q, \delta, q, F \rangle$ the automaton $\mathcal{S}$ with q as its initial state.

We say that a labeled transition graph G satisfies a graph automaton $\mathcal{S}$, denoted $G \models \mathcal{S}$, if $\mathcal{S}$ accepts G. It is shown in [JW95] that graph automata are as expressive as μ-calculus. In particular, we have the following.

Theorem 4. [JW95] *Given a μ-calculus formula ψ, of length n and alternation depth k, we can construct a graph parity automaton $\mathcal{S}_\psi$ such that $L(\mathcal{S}_\psi)$ is exactly the set of graphs satisfying ψ. The automaton $\mathcal{S}_\psi$ has n states and index k.*

A graph automaton whose transitions are restricted to disjunctions over $\{\Diamond\} \times Q$ is in fact a nondeterministic automaton. We freely confuse between such graph automata with the Büchi acceptance condition and NBW. It is well known that every LTL formula can be translated into an NBW that accepts all traces that satisfy the formula. Formally, we have the following.

Theorem 5. [VW94] *For every LTL formula φ, we can construct an NBW N_φ with $2^{O(|\varphi|)}$ states such that $L(N_\varphi) = L(\varphi)$.*

3 Model-Checking Branching-Time Properties

In this section we present an automata-theoretic approach solution to model-checking branching-time properties of pushdown and prefix-recognizable graphs. We start by demonstrating our technique on model checking of pushdown systems. Then we show how to extend it to prefix-recognizable systems. Consider a rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and let $G_R = \langle \Sigma, Q \times V^*, L, \rho_R, (q_0, x_0) \rangle$ be its induced graph. recall that a configuration of G_R is a pair $(q, x) \in Q \times V^*$. Thus, the store x corresponds to a node in the full infinite V-tree. An automaton that reads the tree V^* can memorize in its state space the state component of the configuration and refer to the location of its reading head in V^* as the store. We would like the automaton to “know” the location of its reading head in V^*. A straightforward way to do so is to label a node $x \in V^*$ by x. This, however, involves an infinite alphabet, and results in trees that are not regular.

We show that labeling every node in V^* by its direction is sufficiently informative to provide the 2-way automaton with the information it needs in order to simulate transitions of the rewrite system. Thus, if R is a pushdown system and we are at node $A \cdot y$ of the V-tree (with state q memorized), an application of the transition $\langle q, A, x, q' \rangle$ takes us to node $x \cdot y$ (with state q' memorized). If R is a prefix-recognizable system and we are at node y of the V-tree (with state q memorized), an application of the transition $\langle q, \alpha, \beta, \gamma, q' \rangle$ takes us to node xz (with state q' memorized) where $x \in \gamma$, $z \in \beta$, and $y = z'z$ for some $z' \in \alpha$. Technically, this means that we first move up the tree, and then move down. Such a navigation through the V-tree can be easily performed by two-way automata.

3.1 Pushdown Graphs

We present our solution for pushdown graphs in details. Let $\langle V^*, \tau_V \rangle$ be the V-labeled V-tree such that for every $x \in V^*$ we have $\tau_V(x) = dir(x)$ ($\langle V^*, \tau_V \rangle$ is the exhaustive V-labeled V-tree). Note that $\langle V^*, \tau_V \rangle$ is a regular tree of size $|V| + 1$. We construct a 2APT $\mathcal{A}$ that reads $\langle V^*, \tau_V \rangle$. The state space of $\mathcal{A}$ contains a component that memorizes the current state of the rewrite system. The location of the reading head in $\langle V^*, \tau_V \rangle$ represents the store of the current configuration. Thus, in order to know which rewrite rules can be applied, $\mathcal{A}$ consults its current state and the label of the node it reads (note that $dir(x)$ is the first letter of x). Formally, we have the following.

Theorem 6. *Given a pushdown system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $(V \cup \{\perp\})$-labeled V-trees such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff $G_{\mathcal{R}}$ satisfies $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\|)$ states, and has the same index as $\mathcal{S}$.*

Proof: We define $\mathcal{A} = \langle V \cup \{\perp\}, P, \eta, p_0, \alpha \rangle$ as follows.

– $P = (W \times Q \times heads(T))$, where $heads(T) \subseteq V^*$ is the set of all prefixes of words $x \in V^*$ for which there are states $q, q' \in Q$ and $A \in V$ such that $\langle q, A, x, q' \rangle \in T$. Intuitively, when $\mathcal{A}$ visits a node $x \in V^*$ in state $\langle w, q, y \rangle$, it checks that G_R with initial state $(q, y \cdot x)$ is accepted by $\mathcal{S}^s$. In particular, when $y = \varepsilon$, then G_R with initial state (q, x) (the node currently being visited) needs to be accepted by $\mathcal{S}^w$.

States of the form $\langle w, q, \varepsilon\rangle$ are called *action states*. From these states $\mathcal{A}$ consults δ and T in order to impose new requirements on the exhaustive V-tree. States of the form $\langle w, q, y\rangle$, for $y \in V^+$, are called *navigation states*. From these states $\mathcal{A}$ only navigates downwards y to reach new action states.

– In order to define $\eta : P \times \Sigma \to \mathcal{B}^+(ext(V) \times P)$, we first define the function $apply_T : \Delta \times W \times Q \times V \to \mathcal{B}^+(ext(V) \times P)$. Intuitively, $apply_T$ transforms atoms participating in δ to a formula that describes the requirements on G_R when the rewrite rules in T are applied to words of the form $A \cdot V^*$. For $c \in \Delta$, $w \in W$, $q \in Q$, and $A \in V$ we define

$$apply_T(c, w, q, A) = \begin{cases} \langle \varepsilon, (w, q, \varepsilon)\rangle & \text{if } c = \varepsilon \\ \bigwedge_{\langle q,A,y,q'\rangle \in T} \langle \uparrow, (w, q', y)\rangle & \text{if } c = \Box \\ \bigvee_{\langle q,A,y,q'\rangle \in T} \langle \uparrow, (w, q', y)\rangle & \text{if } c = \Diamond \end{cases}$$

Note that T may contain no tuples in $\{q\} \times \{A\} \times V^* \times Q$ (that is, the transition relation of G_R may not be total). In particular, this happens when $A = \bot$ (that is, for every state $q \in Q$ the configuration (q, ε) of G_R has no successors). Then, we take empty conjunctions as $\mathbf{true}$, and take empty disjunctions as $\mathbf{false}$.

In order to understand the function $apply_T$, consider the case $c = \Box$. When $\mathcal{S}$ reads the configuration $(q, A \cdot x)$ of the input graph, fulfilling the atom $\Box w$ requires $\mathcal{S}$ to send copies in state w to all the successors of $(q, A \cdot x)$. The automaton $\mathcal{A}$ then sends to the node x copies that check whether all the configuration $(q', y \cdot x)$, with $\rho_R((q, A \cdot x), (q', y \cdot x))$, are accepted by $\mathcal{S}$ with initial state w.

Now, for a formula $\theta \in \mathcal{B}^+(\Delta \times S)$, the formula $apply_T(\theta, q, A) \in \mathcal{B}^+(ext(V) \times P)$ is obtained from θ by replacing an atom $\langle c, w\rangle$ by the atom $apply_R(c, w, q, A)$. We can now define η for all $A \in V \cup \{\bot\}$ as follows.

- $\eta(\langle w, q, \varepsilon\rangle, A) = apply_T(\delta(w, L(q, A)), q, A)$.
- $\eta(\langle w, q, y \cdot B\rangle, A) = (B, \langle w, q, y\rangle)$.

Thus, in action states, $\mathcal{A}$ reads the direction of the current node and applies the rewrite rules of R in order to impose new requirements according to δ. In navigation states, $\mathcal{A}$ needs to go downwards $y \cdot B$, so it continues in direction B.

– $p_0 = \langle w_0, q_0, x_0\rangle$. Thus, in its initial state $\mathcal{A}$ checks that G_R with initial configuration (q_0, x_0) is accepted by $\mathcal{S}$ with initial state w_0.

– α is obtained from F by replacing each set F_i by the set $S \times F_i \times heads(T)$.

We show that $\mathcal{A}$ accepts $\langle V^*, \tau_V\rangle$ iff $R \models \mathcal{S}$. Assume that $\mathcal{A}$ accepts $\langle V^*, \tau_V\rangle$. Then, there exists an accepting run $\langle T, r\rangle$ of $\mathcal{A}$ on $\langle V^*, \tau_V\rangle$. Extract from this run the subtree of nodes labeled by action states. That is, consider the following tree $\langle T', r'\rangle$ defined by induction. We know that $r(\varepsilon) = (\varepsilon, (w_0, q_0, x_0))$. It follows that there exists a unique minimal (according to the inverse lexicographic order on the nodes of T) node $y \in T$ labeled by an action state. In our case, $r(y) = (x_0, (w_0, q_0, \varepsilon))$. We add ε to T' and label it $r'(\varepsilon) = ((q_0, x_0), w_0)$. Consider a node z' in T' labeled $r'(z') = ((q, x), w)$. By the construction of $\langle T', r'\rangle$ there exists $z \in T$ such that $r(z) = (x, (w, q, \epsilon))$. Let $\{z_1 \cdot z, \ldots, z_k \cdot z\}$ be the set of minimal nodes in T such that $z_i \cdot z$ is labeled by an action state, $r(z_i \cdot z) = (x_i, (w_i, q_i, \varepsilon))$. We add k successors $a_1 z', \ldots a_k z'$ to z' in T' and set $r'(a_i z') = ((q_i, x_i), w_i)$. By the definition of η, the tree $\langle T', r'\rangle$ is a valid run tree of $\mathcal{S}$ on G_R. Consider an infinite path π' in $\langle T', r'\rangle$. The labels of nodes in π' identify a unique path π in $\langle T, r\rangle$. It follows that the minimal rank appearing infinitely often along π' is the minimal rank appearing infinitely often along π. Hence, $\langle T', r'\rangle$ is accepting and $\mathcal{S}$ accepts G_R.

Assume now that $G_R \models \mathcal{S}$. Then, there exists an accepting run tree $\langle T', r'\rangle$ of $\mathcal{S}$ on G_R. The tree $\langle T', r'\rangle$ serves as the action state skeleton to an accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V\rangle$. A node $z \in T'$ labeled by $((q, x), w)$ corresponds to a copy of $\mathcal{A}$ in state (w, q, ε) reading node x of $\langle V^*, \tau_V\rangle$. It is not hard to extend this skeleton into a valid and accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V\rangle$. $\square$

Pushdown systems can be viewed as a special case of prefix-recognizable systems. In the next subsection we describe how to extend the construction described above to prefix-recognizable graphs, and we also analyze the complexity of the model-checking algorithm that follows for the two types of systems.

3.2 Prefix-Recognizable Graphs

We now extend the construction described in Subsection 3.1 to prefix-recognizable systems. The idea is similar: two-way automata can navigate through the full V-tree and simulate transitions in a rewrite graph by a chain of transitions in the tree. While in pushdown systems the application of rewrite rules involved one move up the tree and then a chain of moves down, here things are a bit more involved. In order to apply a rewrite rule $\langle q, \alpha, \beta, \gamma, q'\rangle$, the automaton has to move upwards along a word in α, check that the remaining word leading to the root is in β, and move downwards along a word in γ. As we explain below, $\mathcal{A}$ does so by simulating automata for the regular expressions participating in T.

Theorem 7. *Given a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, T, L, q_0, x_0\rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F\rangle$, we can construct a 2APT $\mathcal{A}$ over $(V \cup \{\perp\})$-labeled V-trees such that $\mathcal{A}$ accepts $\langle V^*, \tau_V\rangle$ iff G_R satisfies $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\|)$ states, and its index is the index of $\mathcal{S}$ plus 1.*

Proof: As in the case of pushdown systems, $\mathcal{A}$ uses the labels on $\langle V^*, \tau_V\rangle$ in order to learn the location in V^* that each node corresponds to. As there, $\mathcal{A}$ applies to the transition function δ of $\mathcal{S}$ the rewrite rules of R. Here, however, the application of the rewrite rules on atoms of the form $\Diamond w$ and $\Box w$ is more involved, and we describe it below. Assume that $\mathcal{A}$ wants to check whether $\mathcal{S}^w$ accepts $G_R^{(q,x)}$, and it wants to proceed with an atom $\Diamond w'$ in $\delta(w, L(q, x))$. The automaton $\mathcal{A}$ needs to check whether $\mathcal{S}^{w'}$ accepts $G_R^{(q',y)}$ for some configuration (q', y) reachable from (q, x) by applying a rewrite rule. That is, a configuration (q', y) for which there is $\langle q, \alpha, \beta, \gamma, q'\rangle \in T$ and partitions $x' \cdot z$ and $y' \cdot z$, of x and y, respectively, such that x' is accepted by $\mathcal{U}_\alpha$, z is accepted by $\mathcal{U}_\beta$, and y' accepted by $\mathcal{U}_\gamma$. The way $\mathcal{A}$ detects such a state y is the following. From the node x, the automaton $\mathcal{A}$ simulates the automaton $\mathcal{U}_\alpha$ upwards (that is, $\mathcal{A}$ guesses a run of $\mathcal{U}_\alpha$ on the word it reads as it proceeds on direction $\uparrow$ from x towards the root of the V-tree). Suppose that on its way up to the root, $\mathcal{A}$ encounters a state in F_α as it reads the node $z \in V^*$. This means that the word read so far is in α, and can serve as the prefix x' above. If this is indeed the case, then it is left to check that the word z is accepted by $\mathcal{U}_\beta$, and that there is a state that is obtained from z by prefixing it with a word $y' \in \gamma$ that is accepted by $\mathcal{S}^{w'}$. To check the first condition, $\mathcal{A}$ sends a copy in direction $\uparrow$ that simulates a run of $\mathcal{U}_\beta$, hoping to reach a state in F_β as it reaches the root (that is, $\mathcal{A}$ guesses a run of $\mathcal{U}_\beta$ on the word it reads as it proceeds from z up to the root of $\langle V^*, \tau_V \rangle$). To check the second condition, $\mathcal{A}$ simulates the automaton $\mathcal{U}_\gamma$ downwards starting from F_γ. A node $y' \cdot z \in V^*$ that $\mathcal{A}$ reads as it encounters q^0_γ can serve as the state y we are after. The case for an atom $\Box w'$ is similar, only that here $\mathcal{A}$ needs to check whether $\mathcal{S}^{w'}$ accepts $G_R^{(q',y)}$ for all configurations (q', y) reachable from (q, x) by applying a rewrite rule, and thus the choices made by $\mathcal{A}$ for guessing the partition $x' \cdot z$ of x and the prefix y' of y are now treated dually. More formally, we have the following.

We define $\mathcal{A} = \langle V \cup \{\bot\}, P, \eta, p_0, \alpha \rangle$ as follows.

– $P = P_1 \cup P_2$ where $P_1 = \{\exists, \forall\} \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma)$ and $P_2 = \{\exists, \forall\} \times T \times Q_\beta$. States in P_1 serve to simulate automata for α and γ regular expressions and states in P_2 serve to simulate automata for β regular expressions. A state marked by $\exists$ participates in the simulation of a $\Diamond s$ atom of $\mathcal{S}$, and a state marked by $\forall$ participates in the simulation of a $\Box s$ atom of $\mathcal{S}$. A state in P_1 marked by the transition $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ and a state $s \in Q_{\alpha_i}$ participates in the simulation of a run of $\mathcal{U}_{\alpha_i}$. When $s \in F_{\alpha_i}$ (recall that states in F_{α_i} have no outgoing transitions) $\mathcal{A}$ spawns a copy (in a state in P_2) that checks that the suffix is in β_i and continues to simulate $\mathcal{U}_{\gamma_i}$. A state in P_1 marked by the transition $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ and a state $s \in Q_{\gamma_i}$ participates in the simulation of a run of U_{γ_i}. When $s = q_o^{\gamma_i}$ (recall that the initial state $q^0_{\gamma_i}$ has no incoming transitions) the state is an action state, and $\mathcal{A}$ consults δ and T in order to impose new restrictions on $\langle V^*, \tau_V \rangle$. [11]
– In order to define $\eta : P \times \Sigma \to \mathcal{B}^+(ext(V) \times P)$, we first define the function $apply_T : \Delta \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma) \to \mathcal{B}^+(ext(V) \times P)$. Intuitively, $apply_T$ transforms atoms participating in δ to a formula that describes the requirements on G_R when the rewrite rules in T are applied to words from V^*. For $c \in \Delta$, $w \in W$, $q \in Q$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T$, and $s = q^0_{\gamma_i}$ we define

$$apply_T(c, w, q, t_i, s) = \begin{bmatrix} \langle \varepsilon, (\exists, w, q, t_i, s) \rangle & \text{if } c = \varepsilon \\ \bigwedge_{t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle \in T} (\varepsilon, (\forall, w, q', t_{i'}, q^0_{\alpha_{i'}})) & \text{if } c = \Box \\ \bigvee_{t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle \in T} (\varepsilon, (\exists, w, q', t_{i'}, q^0_{\gamma_{i'}})) & \text{if } c = \Diamond \end{bmatrix}$$

In order to understand the function $apply_T$, consider the case $c = \Box$. When $\mathcal{S}$ reads the configuration (q, x) of the input graph, fulfilling the atom $\Box w$ requires $\mathcal{S}$ to send copies in state w to all the successors of (q, x). The automaton $\mathcal{A}$ then sends copies that check whether all the configurations (q', y') with $\rho_R((q, x), (q', y'))$ are accepted by $\mathcal{S}$ with initial state w.
For a formula $\theta \in \mathcal{B}^+(\Delta \times W)$, the formula $apply_T(\theta, q, t_i, s) \in \mathcal{B}^+(ext(V) \times P)$ is obtained from θ by replacing an atom $\langle c, w \rangle$ by the atom $apply_T(c, w, q, t_i, s)$. We

[11] Note that a straightforward representation of P results in $O(|W| \cdot |Q| \cdot |T| \cdot \|T\|)$ states. Since, however, the states of the automata for the regular expressions are disjoint, we can assume that the rewrite rule in T that each automaton corresponds to is uniquely defined from it.

can now define η for all $w \in W$, $q \in Q$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \in T$, $s \in Q_{\alpha_i} \cup Q_{\gamma_i}$, and $A \in V \cup \{\perp\}$ as follows.

- $\eta((\exists, w, q, t_i, s), A) =$

$$\begin{cases} apply_T(\delta(w, L(q, A)), q, t_i, s) & \text{if } s = q^0_{\gamma_i} \\ \bigvee_{B\in V} \bigvee_{s\in\eta_{\gamma_i}(s',B)} (B, (\exists, w, q, t_i, s')) & \text{if } s \in Q_{\gamma_i} \setminus \{q^0_{\gamma_i}\} \\ \bigvee_{s'\in\eta_{\alpha_i}(s,A)} (\uparrow, (\exists, w, q, t_i, s')) & \text{if } s \in Q_{\alpha_i} \setminus F_{\alpha_i} \\ (\varepsilon, (\exists, t_i, q^0_{\beta_i})) \wedge \left(\bigvee_{s'\in F_{\gamma_i}} (\varepsilon, (\exists, w, q, t_i, s'))\right) & \text{if } s \in F_{\alpha_i} \end{cases}$$

- $\eta((\forall, w, q, t_i, s), A) =$

$$\begin{cases} apply_T(\delta(w, L(q, A)), q, t_i, s) & \text{if } s = q^0_{\gamma_i} \\ \text{if } \bigwedge_{B\in V} \bigwedge_{s\in\eta_{\gamma_i}(s',B)} (B, (\forall, w, q, t_i, s')) & \text{if } s \in Q_{\gamma_i} \setminus \{q^0_{\gamma_i}\} \\ \text{if } \bigwedge_{s'\in\eta_{\alpha_i}(s,A)} (\uparrow, (\forall, w, q, t_i, s')) & \text{if } s \in Q_{\alpha_i} \setminus F_{\alpha_i} \\ (\varepsilon, (\forall, t_i, q^0_{\beta_i})) \vee \left(\bigwedge_{s'\in F_{\gamma_i}} (\varepsilon, (\forall, w, q, t_i, s'))\right) & \text{if } t \in F_{\alpha_i} \end{cases}$$

Thus, when $s \in Q_\alpha$ the 2APT $\mathcal{A}$ either chooses a successor s' of s and goes up the tree or in case s is an accepting state of $\mathcal{U}_{\alpha_i}$, it spawns a copy that checks that the suffix is in β_i and moves to a final state of $\mathcal{U}_{\gamma_i}$.
When $s \in Q_\gamma$ the 2APT $\mathcal{A}$ either chooses a direction B and chooses a predecessor s' of s or in case that $s = q^0_{\gamma_i}$ is the initial state of $\mathcal{U}_{\gamma_i}$, the automaton $\mathcal{A}$ uses the transition δ to impose new restrictions on $\langle V^*, \tau_V\rangle$.
We define η for all $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle$, $s \in Q_{\beta_i}$, and $A \in V \cup \{\perp\}$ as follows.

$$\eta((\exists, t_i, s), A) = \begin{cases} \bigvee_{s'\in\eta_{\beta_i}(s,A)} (\uparrow, (\exists, t_i, s')) & \text{if } A \neq \perp \\ \mathbf{true} & \text{if } s \in F_{\beta_i} \text{ and } A = \perp \\ \mathbf{false} & \text{if } s \notin F_{\beta_i} \text{ and } A = \perp \end{cases}$$
$$\eta((\forall, t_i, s), A) = \begin{cases} \bigwedge_{s'\in\eta_{\beta_i}(s,A)} (\uparrow, (\forall, t_i, s')) & \text{if } A \neq \perp \\ \mathbf{false} & \text{if } s \in F_{\beta_i} \text{ and } A = \perp \\ \mathbf{true} & \text{if } s \notin F_{\beta_i} \text{ and } A = \perp \end{cases}$$

If $s \in Q_\beta$, then in existential mode, the automaton $\mathcal{A}$ makes sure that the suffix is in β and in universal mode it makes sure that the suffix is not in β.

– $p_0 = \langle \exists, w_0, q_0, t, x_0\rangle$. Thus, in its initial state $\mathcal{A}$ starts a simulation (backward) of the automaton that accepts the unique word x_0. It follows that $\mathcal{A}$ checks that G_R with initial configuration (q_0, x_0) is accepted by $\mathcal{S}$ with initial state w_0.

– Let $F_\gamma = \bigcup_{t_i\in T} F_{\gamma_i}$. The acceptance condition α is obtained from F by replacing each set F_i by the set $\{\exists, \forall\} \times F_i \times Q \times T \times F_\gamma$. We add to α a maximal odd set and include all the states in $\{\exists\} \times W \times Q \times T \times (Q_\gamma \setminus F_\gamma)$ in this set. We add to α a maximal even set and include all the states in $\{\forall\} \times W \times Q \times T \times (Q_\gamma \setminus F_\gamma)$ in this set[12]. The states in $\{\exists, \forall\} \times W \times Q \times T \times Q_\alpha$ and P_2 are added to the maximal set (notice that states marked by a state in Q_α appear in finite sequences and states in P_2 appear only in suffixes of finite paths in the run tree).

[12] Note that if the maximal set in F is even then we only add to α a maximal odd set. Dually, if the maximal set in F is odd then we add to α a maximal even set.

Thus, in a path that visits infinitely many action states, the action states define it as accepting or not accepting. A path that visits finitely many action states is either finite or ends in an infinite sequence of Q_γ labeled states. If these states are existential, then the path is rejecting. If these states are universal, then the path is accepting.

We show that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff $R \models \mathcal{S}$. Assume that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$. Then, there exists an accepting run $\langle T, r \rangle$ of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. Extract from this run the subtree of nodes labeled by action states. Denote this tree by $\langle T', r' \rangle$. By the definition of δ, the tree $\langle T', r' \rangle$ is a valid run tree of $\mathcal{S}$ on G_R. Consider an infinite path π' in $\langle T', r' \rangle$. The labels of nodes in π' identify a unique path π in $\langle T, r \rangle$. As π' is infinite, it follows that π visits infinitely many action states. As all navigation states are added to the maximal ranks the minimal rank visited along π must be equal to the minimal rank visited along π'. Hence, $\langle T', r' \rangle$ is accepting and $\mathcal{S}$ accepts G_R.

Assume now that $G_R \models \mathcal{S}$. Then, there exists an accepting run tree $\langle T', r' \rangle$ of $\mathcal{S}$ on G_R. The tree $\langle T', r' \rangle$ serves as the action state skeleton to an accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. A node $z \in T'$ labeled by $((q, x), w)$ corresponds to a copy of $\mathcal{A}$ in state (d, w, q, t, s) reading node x of $\langle V^*, \tau_V \rangle$ for some $d \in \{\exists, \forall\}$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T$ and $s = q^0_{\gamma_i}$. In order to extend this skeleton into a valid and accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$ we have to complete the runs of the automata for the different regular expressions appearing in T. □

The constructions described in Theorems 6 and 7 reduce the model-checking problem to the membership problem of $\langle V^*, \tau_V \rangle$ in the language of a 2APT. By Theorem 3, we then have the following.

Theorem 8. *The model-checking problem for a pushdown or a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$ and k is the index of $\mathcal{S}$.*

Together with Theorem 4, we can conclude with an EXPTIME bound also for the model-checking problem of μ-calculus formulas matching the lower bound in [Wal96]. Note that the fact the same complexity bound holds for both pushdown and prefix-recognizable rewrite systems stems from the different definition of $\|T\|$ in the two cases.

4 Path Automata on Trees

We would like to enhance the approach developed in Section 3 to linear time properties. The solution to μ-calculus model checking is exponential in both the system and the specification and it is EXPTIME-complete [Wal96]. On the other hand, model-checking linear-time specifications is polynomial in the system [BEM97]. As we discuss below, both the emptiness and membership problems for 2APT are EXPTIME-complete. While 2APT can reason about many computation paths simultaneously, in linear-time model-checking we need to reason about a single path that does not satisfy a specification. It follows, that the extra power of 2APT comes at a price we cannot pay. In this section we introduce *path automata* and study them. In Section 5 we show that path automata give us the necessary tool in order to reason about linear specifications.

Path automata resemble *tree walking automata*. These are automata that read finite trees and expect the nodes of the tree to be labeled by the direction and by the set of successors of the node. Tree walking automata are used in XML queries. We refer the reader to [EHvB99,Nev02].

4.1 Definition

Path automata on trees are a hybrid of nondeterministic word automata and nondeterministic tree automata: they run on trees but have linear runs. Here we describe *two-way* nondeterministic Büchi path automata.

A *two-way nondeterministic Büchi path automaton* (2NBP, for short) on Σ-labeled Υ-trees is in fact a 2ABT whose transitions are restricted to disjunctions. Formally, $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$, where Σ, P, p_0, and F are as in an NBW, and $\delta : P \times \Sigma \to 2^{(ext(\Upsilon) \times P)}$ is the transition function. A path automaton that is in state p and reads the node $x \in T$ chooses a pair $(d, p') \in \delta(p, \tau(x))$, and then follows direction d and moves to state p'. It follows that a *run* of a 2NBP $\mathcal{P}$ on a labeled tree $\langle \Upsilon^*, \tau \rangle$ is a sequence of pairs $r = (x_0, p_0), (x_1, p_1), \ldots$ where for all $i \geq 0$, $x_i \in \Upsilon^*$ is a node of the tree and $p_i \in P$ is a state. The pair (x, p) describes a copy of the automaton that reads the node x of Υ^* and is in the state p. Note that many pairs in r may correspond to the same node of Υ^*; Thus, $\mathcal{S}$ may visit a node several times. The run has to satisfy the transition function. Formally, $(x_0, p_0) = (\varepsilon, q_0)$ and for all $i \geq 0$ there is $d \in ext(\Upsilon)$ such that $(d, p_{i+1}) \in \delta(p_i, \tau(x_i))$ and

– If $\Delta \in \Upsilon$, then $x_{i+1} = \Delta \cdot x_i$.
– If $\Delta = \varepsilon$, then $x_{i+1} = x_i$.
– If $\Delta = \uparrow$, then $x_i = \upsilon \cdot z$, for some $\upsilon \in \Upsilon$ and $z \in \Upsilon^*$, and $x_{i+1} = z$.

Thus, ε-transitions leave the automaton on the same node of the input tree, and $\uparrow$-transitions take it up to the parent node. Note that the automaton cannot go up the root of the input tree, as whenever $d = \uparrow$, we require that $x_i \neq \varepsilon$. A run r is *accepting* if it visits $\Upsilon^* \times F$ infinitely often. An automaton accepts a labeled tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{P})$ the set of all Σ-labeled trees that $\mathcal{P}$ accepts. The automaton $\mathcal{P}$ is *nonempty* iff $\mathcal{L}(\mathcal{P}) \neq \emptyset$. We measure the size of a 2NBP by two parameters, the number of states and the size, $|\delta| = \Sigma_{p \in P} \Sigma_{a \in \Sigma} |\delta(s, a)|$, of the transition function.

Readers familiar with tree automata know that the run of a tree automaton starts in a single copy of the automaton reading the root of the tree, and then the copy splits to the successors of the root and so on, thus the run simultaneously follows many paths in the input tree. In contrast, a path automaton has a single copy at all times. It starts from the root and it always chooses a single direction to go to. In two-way path automata, the direction may be "up", so the automaton can read many paths of the tree, but it cannot read them simultaneously.

The fact that a 2NBP has a single copy influences its expressive power and the complexity of its nonemptiness and membership problems. We now turn to study these issues.

4.2 Expressiveness

One-way nondeterministic path automata can read a single path of the tree, so it is easy to see that they accept exactly all languages $\mathcal{T}$ of trees such that there is an ω-regular language L of words and $\mathcal{T}$ contains exactly all trees that have a path labeled by a word in L. For two-way path automata, the expressive power is less clear, as by going up and down the tree, the automaton can traverse several paths. Still, a path automaton cannot traverse all the nodes of the tree. To see that, we prove that a 2NBP cannot recognize even very simple properties that refer to all the branches of the tree (*universal* properties for short).

Theorem 9. *There are no 2NBP $\mathcal{P}_1$ and $\mathcal{P}_2$ over the alphabet $\{0, 1\}$ such that $L(\mathcal{P}_1) = L_1$ and $L(\mathcal{P}_2) = L_2$ where $|\Upsilon| > 1$ and*

– $L_1 = \{\langle \Upsilon^*, \tau\rangle : \tau(x) = 0$ *for all* $x \in T\}$.
– $L_2 = \{\langle \Upsilon^*, \tau\rangle :$ *for every path* $\pi \subseteq T$, *there is* $x \in \pi$ *with* $\tau(x) = 0\}$.

Proof: Suppose that there exists a 2NBP $\mathcal{P}_1$ that accepts L_1. Let $T = \langle \Upsilon^*, \tau\rangle \in L_1$ be some tree accepted by $\mathcal{P}_1$. There exists an accepting run $r = (x_0, p_0), (x_1, p_1), \ldots$ of $\mathcal{P}_1$ on T. It is either the case that r visits some node in Υ^* infinitely often or not.

– Suppose that there exists a node $x \in \Upsilon^*$ visited infinitely often by r. There must exist $i < j$ such that $x_i = x_j = x$, $p_i = p_j$, and there exists $i \leq k < j$ such that $p_k \in F$. Consider the run $r' = (x_0, p_0), \ldots, (x_{i-1}, p_{i-1}) \left((x_i, p_i), \ldots, (x_{j-1}, p_{j-1})\right)^\omega$. Clearly, it is a valid and accepting run of $\mathcal{P}_1$ on T. However, r' visits only a finite number of nodes in T. Let $W = \{x_i | x_i \text{ visited by } r'\}$. It is quite clear that the same run r' is an accepting run of $\mathcal{P}_1$ on the tree $\langle \Upsilon, \tau'\rangle$ such that $\tau'(x) = \tau(x)$ for $x \in W$ and $\tau'(x) = 1$ for $x \notin W$. Clearly, $\langle \Upsilon^*, \tau'\rangle \notin L_1$.
– Suppose that every node $x \in \Upsilon^*$ is visited only a finite number of times. Let (x_i, p_i) be the last visit of r to the root. It must be the case that $x_{i+1} = \upsilon$ for some $\upsilon \in \Upsilon$. Let $\upsilon' \neq \upsilon$ be a different element in Υ. Let $W = \{x_{i'} \in \Upsilon^* \cdot \upsilon' \mid x_{i'} \text{ visited by } r\}$ be the set of nodes in the subtree of υ' visited by r. Clearly, W is finite and we proceed as above.

The proof for the case of $\mathcal{P}_2$ and L_2 is similar. $\square$

There are, however, universal properties that a 2NBP can recognize. Consider a language $L \subseteq \Sigma^\omega$ of infinite words over the alphabet Σ. A finite word $x \in \Sigma^*$ is a *bad prefix* for L iff for all $y \in \Sigma^\omega$, we have $x \cdot y \notin L$. Thus, a bad prefix is a finite word that cannot be extended to an infinite word in L. A language L is a *safety* language iff every $w \notin L$ has a finite bad prefix. A language $L \subseteq \Sigma^\omega$ is *clopen* if both L and its complement are safety languages, or, equivalently, L corresponds to a set that is both closed and open in Cantor space. It is known that a clopen language is bounded: there is an integer k such that after reading a prefix of length k of a word $w \in \Sigma^\omega$, one can determine whether w is in L [KV01]. A 2NBP can then traverse all the paths of the input tree up to level k (given L, its bound k can be calculated), hence the following theorem.

Theorem 10. *Let $L \subseteq \Sigma^\omega$ be a clopen language. There is a 2NBP $\mathcal{P}$ such that $L(\mathcal{P}) = \{\langle \Upsilon^*, \tau\rangle : $ for all paths $\pi \subseteq \Upsilon^*$, we have $\tau(\pi) \in L\}$.*

Proof: Let k be the bound of L and $\Upsilon = \{\upsilon_1, \ldots, \upsilon_m\}$. Consider, $w = w_0, \ldots, w_r \in \Upsilon^*$. Let i be the maximal index such that $w_i \neq \upsilon_m$. Let $succ(w)$ be as follows

$$succ(w) = w_0, \ldots w_{i-1}, w'_i, w_{i+1}, \ldots, w_r,$$

where if $w_i = \upsilon_j$ then $w'_i = \upsilon_{j+1}$. That is, if we take $w = (\upsilon_1)^k$ then by using the $succ$ function we pass on all elements in Υ^k according to the lexicographic order (induced by $\upsilon_1 < \upsilon_2 < \ldots < \upsilon_m$). Let $\mathcal{N} = \langle \Sigma, N, \delta, n_0, F\rangle$ be an NBW accepting L. According to [KV01], $\mathcal{N}$ is cycle-free and has a unique accepting sink state. Formally, $\mathcal{N}$ has an accepting state n_{acc} such that for every $\sigma \in \Sigma$ we have $\delta(n_{acc}, \sigma) = \{n_{acc}\}$ and for every run $r = n_0, n_1, \ldots$ and every $i < j$ either $n_i \neq n_j$ or $n_i = n_{acc}$.

We construct a 2NBP that scans all the paths in Υ^k according to the order induced by using $succ$. The 2NBP scans a path and simulates $\mathcal{N}$ on this path. Once our 2NBP ensures that this path is accepted by $\mathcal{N}$ it proceeds to the next path. Consider the following 2NBP $\mathcal{P} = \langle \Sigma, Q, \eta, q_0, \{q_{acc}\}\rangle$ where

- $Q = (\{u, d\} \times \Upsilon^k \times [k] \times N) \cup \{q_{acc}\}$. A state consists of 4 components. The symbols u and d are acronyms for up and $down$. A state marked by d means that the 2NBP is going down the tree while scanning a path. A state marked by u means that the 2NBP is going up towards the root where it starts scanning the next path. The word $w \in \Upsilon^k$ is the current explored path. The number $i \in [k]$ denotes the location in the path w. The state $n \in N$ denotes the current state of the automaton $\mathcal{N}$.
- For every state $q \in Q$ and letter $\sigma \in \Sigma$, the transition function $\eta : Q \times \Sigma \to 2^{ext(\Upsilon) \times Q}$ is defined as follows:

$$\eta((d, w, i, n), \sigma) = \left[\begin{array}{ll} \{(w_{i+1}, (d, w, i{+}1, n')) \mid n' \in \delta(n, \sigma)\} & \text{if } i \neq k \\ \emptyset & \text{if } i = k \text{ and } n \neq n_{acc} \\ \{(\varepsilon, (u, succ(w), i, n))\} & \text{if } i = k, n = n_{acc}, \text{ and } w \neq (\upsilon_m)^k \\ \{(\varepsilon, q_{acc})\} & \text{if } i = k, n = n_{acc}, \text{ and } w = (\upsilon_m)^k \end{array}\right.$$

$$\eta((u, w, i, n), \sigma) = \left[\begin{array}{ll} \{(\uparrow, (u, w, i-1, n))\} & \text{if } i \neq 0 \\ \{(\varepsilon, (d, w, 0, n_0))\} & \text{if } i = 0 \end{array}\right.$$

$$\eta(q_{acc}, \sigma) = \{(\varepsilon, q_{acc})\}$$

  Intuitively, in d-states the automaton goes in the direction dictated by w and simulates $\mathcal{N}$ on the labeling of the path w. Once the path w is explored, if the $\mathcal{N}$ component is not in n_{acc} this means the run of $\mathcal{N}$ on w failed and the run is terminated. If the $\mathcal{N}$ component reaches n_{acc} this means that the run of $\mathcal{N}$ on w succeeded and the 2NBP proceeds to a u-state with $succ(w)$. If $succ(w)$ does not exist (i.e., $w = (\upsilon_m)^k$) the 2NBP accepts. In u-states the 2NBP goes up towards the root; when it reaches the root it initiates a run of $\mathcal{N}$ on the word w.
- $q_0 = (d, (\upsilon_1)^k, 0, n_0)$. Thus, in the initial state, $\mathcal{P}$ starts to simulate $\mathcal{N}$ on the first path $(\upsilon_1)^k$.

Let $\mathcal{L} = \{\langle \Upsilon^*, \tau \rangle :$ for all paths $\pi \subseteq \Upsilon^*$, we have $\tau(\pi) \in L\}$. Consider a tree t in $\mathcal{L}$. We show that t is accepted by $\mathcal{P}$. Consider a path w in t. Let $r_w = n_0 \cdots n_k$ be the accepting run of $\mathcal{N}$ on the word labeling the path w in t. We use the sequence $(d, w, n_0, 0) \cdots (d, w, n_k, k)$ as part of the run of $\mathcal{P}$ on t. We add the parts $(u, w, k - 1, n) \cdots (u, w, 0, n)$ that connect these different sequences and finally add an infinite sequence of q_{acc}.

In the other direction consider a tree t accepted by $\mathcal{P}$. For a path w in t we extract from the accepting run of $\mathcal{P}$ the part that relates to w. By the definition of the transition it follows that if we project this segment on the states of $\mathcal{N}$ we get an accepting run of $\mathcal{N}$ on the word labeling w in t. As w is arbitrary it follows that every path in tree is labeled by a word in L and that $t \in \mathcal{L}$. $\square$

Recently, it was shown that deterministic walking tree automata are less expressive than nondeterministic walking tree automata [BC04] and that nondeterministic walking tree automata do not accept all regular tree languages [BC05]. That is, there exist languages recognized by nondeterministic walking tree automata that cannot be recognized by deterministic walking tree automata and there exist languages accept by deterministic tree automata that cannot be recognized by nondeterministic walking tree automata. Using standard techniques to generalize results about automata over finite objects to automata over infinite objects we can show that 2DBP are less expressive than 2NBP. Similarly, the algorithms described in the next subsection can be modified to handle the respective problems for walking tree automata.

4.3 Decision Problems

Given a 2NBP $\mathcal{S}$, the *emptiness problem* is to determine whether $\mathcal{S}$ accepts some tree, or equivalently whether $\mathcal{L}(\mathcal{S}) = \emptyset$. The *membership problem* of $\mathcal{S}$ and a regular tree $\langle \Upsilon^*, \tau \rangle$ is to determine whether $\mathcal{S}$ accepts $\langle \Upsilon^*, \tau \rangle$, or equivalently $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(\mathcal{S})$. The fact that 2NBP cannot spawn new copies makes them very similar to word automata. Thus, the membership problem for 2NBP can be reduced to the emptiness problem of εABW over a 1-letter alphabet (cf. [KVW00]). The reduction yields a polynomial time algorithm for solving the membership problem. In contrast, the emptiness problem of 2NBP is EXPTIME-complete.

We show a reduction from the membership problem of 2NBP to the emptiness problem of εABW with a 1-letter alphabet. The reduction is a generalization of a construction that translates 2NBW to εABW [PV03]. The emptiness of εABW with a 1-letter alphabet is solvable in quadratic time and linear space [KVW00]. We show that in our case the membership problem of a 2NBP is solved in cubic time and quadratic space in the size of the original 2NBP. Formally, we have the following.

Theorem 11. *Consider a 2NBP $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$. The membership problem of the regular tree $\langle \Upsilon^*, \tau \rangle$ in the language of $\mathcal{S}$ is solvable in time $O(|P|^2 \cdot |\delta| \cdot \|\tau\|)$ and space $O(|P|^2 \cdot \|\tau\|)$.*

Proof: We construct an εABW on 1-letter alphabet $\mathcal{A} = \langle \{a\}, Q, \eta, q_0, \alpha \rangle$ such that $L(\mathcal{A}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in L(\mathcal{P})$. The εABW $\mathcal{A}$ has $O(|P|^2 \cdot \|\tau\|)$ states and the size of its transition function is $O(|P|^2 \cdot |\delta| \cdot \|\tau\|)$. As $\langle \Upsilon^*, \tau \rangle$ is a regular tree, there exists a transducer that produces it. In order to construct $\mathcal{A}$ we combine this transducer with a construction that converts 2-way automata to 1-way automata. In [PV03] we show that given a 2NBW we can construct an ϵABW that accepts the same language. The conversion of 2-way movement to 1-way movement relies on the following basic paradigm. We check whether the 2-way automaton accepts the word aw from state s by checking that it can get from state s to state t reading aw and that it accepts aw from state t. In order to check that the 2-way automaton can get from state s to state t reading a suffix aw, the 1-way automaton either guesses that the 2-way automaton gets from s to some state p and from p to t, or that there is a transition from s reading a and going forward to state s', a transition from some state t' reading the first letter of w going backward to t, and that the 2-way automaton can get from s' to t' reading w. We use a similar idea here. Consider a regular tree that is the unwinding of a transducer from state d. The 2NBP accepts this tree from state s if there exists a state t such that the 2NBP reaches from s to t reading the tree and accepts the tree starting from t. In order to get from s to t reading the tree the 2NBP either reaches the root again in state p (i.e., reach from s to p and from p to t) or there is a transition from s reading the label of d and going in direction γ to state s', a transition from some state t' reading the label of the γ successor of d going backward to t, and that the 2-way automaton can get from s' to t' reading the regular tree that is the unwinding of the transducer from state d'.

Let $\mathcal{D}_\tau = \langle \Upsilon, \Sigma, D_\tau, \rho_\tau, d_0^\tau, L_\tau \rangle$ be the transducer that generates the labels of τ. For a word $w \in \Upsilon^*$ we denote by $\rho_\tau(w)$ the unique state that $\mathcal{D}_\tau$ gets to after reading w. We construct the εABW $\mathcal{A} = \langle \{a\}, Q, \eta, q_0, \alpha \rangle$ as follows.

– $Q = (P \cup (P \times P)) \times D_\tau \times \{\bot, \top\}$. States in $P \times D_\tau \times \{\bot, \top\}$, which hold a single state from P, are called *singleton states*. Similarly, we call states in $P \times P \times D_\tau \times \{\bot, \top\}$ *pair states*.
– $q_0 = (p_0, d_0^\tau, \bot)$.
– $\alpha = (F \times D_\tau \times \{\bot\}) \cup (P \times D_\tau \times \{\top\})$.

In order to define the transition function we have the following definitions. Two functions $f_\alpha : P \times P \to \{\bot, \top\}$ where $\alpha \in \{\bot, \top\}$, and for every state $p \in P$ and alphabet letter $\sigma \in \Sigma$ the set C_p^σ is the set of states from which p is reachable by a sequence of ϵ-transitions reading letter σ and one final $\uparrow$-transition reading σ. Formally,

$$f_\bot(p, p') = \bot.$$

$$f_\top(p, p') = \left[\begin{array}{ll} \bot & \text{if } p \in F \text{ or } p' \in F \\ \top & \text{otherwise.} \end{array}\right.$$

$$C_p^\sigma = \left\{ p' \;\middle|\; \begin{array}{l} \exists t_0, t_1, \ldots, t_n \in P^+ \text{ such that } t_0 = p',\ t_n = p, \\ (\epsilon, t_i) \in \delta(t_{i-1}, \sigma) \text{ for all } 0 < i < n, \text{ and } (\uparrow, p_n) \in \delta(p_{n-1}, \sigma) \end{array} \right\}.$$

Now η is defined for every state in Q as follows (recall that $\mathcal{A}$ is a word automaton, hence we use directions 0 and 1 in the definition of η, as $\Sigma = \{a\}$, we omit the letter a from the definition of η).

$$
\begin{aligned}
\eta(p,d,\alpha) = \quad & \bigvee_{p'\in P}\bigvee_{\beta\in\{\bot,\top\}} (0,(p,p',d,\beta)) \wedge (0,(p',d,\beta)) && \vee\\
& \bigvee_{\upsilon\in\Upsilon}\bigvee_{(\upsilon,p')\in\delta(p,L_\tau(d))} (1,(p',\rho_\tau(d,\upsilon),\bot)) && \vee\\
& \bigvee_{\langle\epsilon,p'\rangle\in\delta(p,L_\tau(d))} (0,(p',d,\bot))
\end{aligned}
$$

$$
\begin{aligned}
\eta(p_1,p_2,d,\alpha) = & \bigvee_{\langle\epsilon,p'\rangle\in\delta(p_1,L_\tau(d))} (0,(p',p_2,d,f_\alpha(p',p_2))) && \vee\\
& \bigvee_{p'\in P}\bigvee_{\beta_1+\beta_2=\alpha} (0,(p_1,p',d,f_{\beta_1}(p_1,p'))) \wedge (0,(p',p_2,d,f_{\beta_2}(p',p_2))) && \vee\\
& \bigvee_{\upsilon\in\Upsilon}\bigvee_{\langle\upsilon,p'\rangle\in\delta(p_1,L_\tau(d))}\bigvee_{p''\in C_{p_2}^{L_\tau(d)}} (1,(p',p'',\rho_\tau(d,\upsilon),f_\alpha(p',p'')))
\end{aligned}
$$

Finally, we replace every state of the form $\{(p,p,d,\alpha) \mid \text{either } p \in P \text{ and } \alpha = \bot \text{ or } p \in F \text{ and } \alpha = \top\}$ by $\mathbf{true}$.

Claim. $L(\mathcal{A}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau\rangle \in L(\mathcal{P})$.

The proof is very similar to the proof in [PV03] and is described in detail in Appendix A.

The emptiness of an εABW can be determined in linear space [EL86]. For an εABW $\mathcal{A}$ with 1-letter alphabet, we have the following.

Theorem 12. [VW86b] *Given an εABW over 1-letter alphabet $\mathcal{A} = \langle\{a\}, Q, \eta, q_0, \alpha\rangle$ we can check whether $L(\mathcal{A})$ is empty in time $O(|Q|\cdot|\eta|)$ and space $O(|Q|)$.*

Vardi and Wolper give an algorithm that solves the emptiness problem of an ABW over 1-letter alphabet [VW86b]. We note that emptiness of εABW over 1-letter alphabet can be reduced to that of an ABW over 1-letter alphabet by replacing every ϵ-transition by a transition that advances to the next letter. As the input word is infinite, there is no difference between advancing and not advancing.

The automaton $\mathcal{A}$ constructed above has a special structure. The transition of $\mathcal{A}$ from states of the form $P \times P \times D_\tau \times \{\bot,\top\}$ includes only states of the same form. In addition, all these states are not accepting. This suggests that if in the emptiness algorithm we handle these states first, the quadratic part of the algorithm can be applied only to the states of the form $P \times D_\tau \times \{\bot,\top\}$. Using these ideas, we show in [PV03] that the emptiness of $\mathcal{A}$ can be decided in time $O(|\eta|)$ and space $O(|Q|)$. $\square$

Theorem 13. *The emptiness problem for 2NBP is EXPTIME-complete.*

Proof: The upper bound follows immediately from the exponential time algorithm for the emptiness for 2APT [Var98].

For the lower bound we use the EXPTIME-hard problem of whether a linear space alternating Turing machine accepts the empty tape [CKS81]. We reduce this problem to the emptiness problem of a 2NBP with a polynomial number of states. We start with definitions of alternating linear space Turing machines. An alternating Turing machine is $M = \langle \Gamma, S_u, S_e, \mapsto, s_0, F_{acc}, F_{rej} \rangle$, where the four sets of states S_u, S_e, F_{acc}, and F_{rej} are disjoint, and contain the universal, the existential, the accepting, and the rejecting states, respectively. We denote their union (the set of all states) by S. Our model of alternation prescribes that $\mapsto \subseteq S \times \Gamma \times S \times \Gamma \times \{L, R\}$ has a binary branching degree. When a universal or an existential state of M branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(s, a) \mapsto^l (s_l, b_l, \Delta_l)$ and $(s, a) \mapsto^r (s_r, b_r, \Delta_r)$ to indicate that when M is in state $s \in S_u \cup S_e$ reading input symbol a, it branches to the left with (s_l, b_l, Δ_l) and to the right with (s_r, b_r, Δ_r). (Note that the directions left and right here have nothing to do with the movement direction of the head; these are determined by Δ_l and Δ_r.)

Recall that we consider here alternating linear-space Turing machines. Let $f : \mathbb{N} \to \mathbb{N}$ be the linear function such that M uses $f(n)$ cells in its working tape in order to process an input of length n. We encode a configuration of M by a string in $\{\sharp\} \cdot \Gamma^i \cdots (S \times \Gamma) \cdot \Gamma^{f(n)-i-1}$. That is, a configuration starts with the symbol $\sharp$, all its other letters are in Γ, except for one letter in $S \times \Gamma$. The meaning of such a configuration is that the j^{th} cell in the configuration, for $1 \leq j \leq f(n)$, is labeled γ_j, the reading head points at cell $i{+}1$, and M is in state s. For example, the initial configuration of M is $\sharp \cdot (s_0, b) b \cdots b$ (with $f(n){-}1$ occurrences of b's) where b stands for an empty cell. A configuration c' is a successor of configuration c if c' is a left or right successor of c. We can encode now a computation of M by a tree whose branches describe sequences of configurations of M. The computation is legal if a configuration and its successors satisfy the transition relation.

Note that though M has an existential (thus nondeterministic) mode, there is a single computation tree that describes all the possible choices of M. Each run of M corresponds to a pruning of the computation tree in which all the universal configurations have both successors and all the existential configurations have at least one successor. The run is accepting if all the branches in the pruned tree reach an accepting configuration.

We encode the full run tree of M into the labeling of the full infinite binary tree. We construct a 2NBP that reads an input tree and checks that it is indeed a correct encoding of the run tree of M. In case the input tree is a correct encoding, the 2NBP further checks that there exists a subtree that represents an accepting computation of M.

We now explain how the labeling of the full binary tree is used to encode the run tree of M. Let $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ be a configuration and $\sharp \cdot \sigma^l_1 \cdots \sigma^l_{f_n}$ be its left successor. We set σ_0 and σ^l_0 to $\sharp$. Formally, let $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma)$ and let $next_l : V^3 \to V$ where $next_l(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$ denotes our expectation for σ^l_i. We define $next_l(\sigma, \sharp, \sigma') = \sharp$ and

$$next_l(\sigma, \sigma', \sigma'') = \begin{cases} \sigma' & \text{if } \{\sigma, \sigma', \sigma''\} \subseteq \{\sharp\} \cup \Gamma \\ \sigma' & \text{if } \sigma'' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', R) \\ (s', \sigma') & \text{if } \sigma'' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', L) \\ \sigma' & \text{if } \sigma = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', L) \\ (s', \sigma') & \text{if } \sigma = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', R) \\ \gamma' & \text{if } \sigma' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', \alpha) \end{cases}$$

The expectation $next_r : V^3 \to V$ for the letters in the right successor is defined analogously.

The run tree of M is encoded in the full binary tree as follows. Every configuration is encoded by a string of length $f(n)+1$ in $\{\sharp\} \times \Gamma^* \times (S \times \Gamma) \times \Gamma^*$. The encoding of a configuration $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ starts in a node x that is labeled by $\sharp$. The 0 successor of x, namely $0 \cdot x$, is labeled by σ_1 and so on until $0^{f(n)} \cdot x$ that is labeled by $\sigma_{f(n)}$. The configuration $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ has its right successor $\sharp \cdot \sigma_1^r \cdots \sigma_{f(n)}^r$ and its left successor $\sharp \cdot \sigma_1^l \cdots \sigma_{f(n)}^l$. The encoding of $\sharp \cdot \sigma_1^r \cdots \sigma_{f(n)}^r$ starts in $1 \cdot 0^{f(n)} \cdot x$ (that is labeled by $\sharp$) and the encoding of $\sharp \cdot \sigma_1^l \cdots \sigma_{f(n)}^l$ starts in $0 \cdot 0^{f(n)} \cdot x$ (that is labeled by $\sharp$). We also demand that every node be labeled by its direction. This way we can infer from the label of the node labeled by $\sharp$ whether its the first letter in the left successor or the first letter in the right successor. For example, the root of the tree is labeled by $\langle \perp, \sharp \rangle$, the node 0 is labeled by $\langle 0, (s_0, b) \rangle$ and for every $1 < i \leq f(n)$ the node 0^i is labeled by $\langle 0, b \rangle$ (here b stands for the blank symbol). We do not care about the labels of other nodes. Thus, the labeling of 'most' nodes in the tree does not interest us.

The 2NBP reads an infinite binary tree. All trees whose labeling does not conform to the above are rejected. A tree whose labeling is a correct encoding of the run tree of M is accepted only if there exists an accepting pruning tree. Thus, the language of the 2NBP is not empty iff the Turing machine M accepts the empty tape.

In order to check that the input tree is a correct encoding of the run tree of M, the 2NBP has to check that every configuration is followed by its successor configurations. When checking location i in configuration a, the NBW memorizes the three letters around location i ($i-1$, i, $i+1$), it goes $f(n)$ steps forward to the next configuration and checks that it finds there the correct $next_l$ or $next_r$ successor. Then the 2NBP returns to location $i+1$ in configuration a and updates its three letters memory to check consistency of this next location.

We now explain the construction in more detail. The 2NBP has two main modes of operation. In *forward* mode, the 2NBP checks that the next (right or left) configuration is indeed the correct successor. Then it moves to check the next configuration. If it reaches an accepting configuration, this means that the currently scanned pruning tree may still be accepting. Then it moves to *backward* mode and remembers that it should check other universal branches. If it reaches a rejecting configuration, this means that the currently scanned pruning tree is rejecting. The 2NBP has to move to the next pruning tree. It moves to *backward* mode and remembers that it has to check other existential branches. In *backward universal* mode, the 2NBP goes backward until it gets to a universal configuration and the only configuration to be visited below it is the left successor. Then the 2NBP goes back to forward mode but remembers that the next configuration to visit is the right successor. If the root is reached in backward universal mode then there are no more branches to check, the pruning tree is accepting and the 2NBP accepts. In *backward existential* mode, the 2NBP goes backward until it gets to an existential configuration and the only configuration to be visited below it is the left successor. Then the 2NBP goes to forward mode but remembers that the next configuration to visit is the right successor. If the root is reached in backward existential mode then there are no more pruning trees to check and the 2NBP rejects.

The full formal construction is given in Appendix B. $\square$

We note that the membership problem for 2-way alternating Büchi automata on trees is EXPTIME-complete. Indeed, CTL model checking of pushdown systems, proven to be EXPTIME-hard in [Wal00], can be reduced to the membership problem of a regular tree in the language of a 2ABT. Given a pushdown system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a CTL formula φ, we can construct a graph automaton $\mathcal{S}$ accepting the set of graphs that satisfy φ [KVW00]. This graph automaton is linear in φ and it uses the Büchi acceptance condition. Using the construction in Section 3, CTL model checking then reduces to the membership problem of $\langle V^*, \tau_V \rangle$ in the language of a 2ABT. EXPTIME-hardness follows. Thus, path automata capture the computational difference between linear and branching specifications.

5 Model-Checking Linear-Time Properties

In this section we solve the LTL model-checking problem by a reduction to the membership problem of 2NBP. We start by demonstrating our technique on LTL model checking of pushdown systems. Then we show how to extend it to prefix-recognizable systems. For an LTL formula φ, we construct a 2NBP that navigates through the full infinite V-tree and simulates a computation of the rewrite system that does not satisfy φ. Thus, our 2NBP accepts the V-tree iff the rewrite system does not satisfy the specification. Then, we use the results in Section 4: we check whether the given V-tree is in the language of the 2NBP and conclude whether the system satisfies the property. For pushdown systems we show that the tree $\langle V^*, \tau_V \rangle$ gives sufficient information in order to let the 2NBP simulate transitions. For prefix-recognizable systems the label is more complex and reflects the membership of a node x in the regular expressions that are used in the transition rules and the regular labeling.

5.1 Pushdown Graphs

Recall that in order to apply a rewrite rule of a pushdown system from configuration (q, x), it is sufficient to know q and the first letter of x. We construct a 2NBP $\mathcal{P}$ that reads $\langle V^*, \tau_V \rangle$. The state space of $\mathcal{P}$ contains a component that memorizes the current state of the rewrite system. The location of the reading head in $\langle V^*, \tau_V \rangle$ represents the store of the current configuration. Thus, in order to know which rewrite rules can be applied, $\mathcal{P}$ consults its current state and the label of the node it reads (note that $dir(x)$ is the first letter of x). Formally, we have the following.

Theorem 14. *Given a pushdown system $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0 \rangle$ and an LTL formula φ, there is a 2NBP $\mathcal{P}$ on V-trees such that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff $G_R \not\models \varphi$. The automaton $\mathcal{P}$ has $|Q| \cdot \|T\| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.*

Proof: According to Theorem 5, there is an NBW $\mathcal{S}_{\neg\varphi} = \langle 2^{AP}, W, \eta_{\neg\varphi}, w_0, F \rangle$ such that $L(\mathcal{S}_{\neg\varphi}) = (2^{AP})^\omega \setminus L(\varphi)$. The 2NBP $\mathcal{P}$ tries to find a trace in G_R that satisfies $\neg\varphi$. The 2NBP $\mathcal{P}$ runs $\mathcal{S}_{\neg\varphi}$ on a guessed (q_0, x_0)-computation in R. Thus, $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff there exists an (q_0, x_0)-trace in G_R accepted by $\mathcal{S}_{\neg\varphi}$. Such a (q_0, x_0)-trace does not satisfy φ, and it exists iff $R \not\models \varphi$. We define $\mathcal{P} = \langle \{V \cup \{\perp\}, P, \delta, p_0, \alpha \rangle$, where

– $P = W \times Q \times heads(T)$, where $heads(T) \subseteq V^*$ is the set of all prefixes of words $x \in V^*$ for which there are states $q, q' \in Q$ and $A \in V$ such that $\langle q, A, x, q' \rangle \in T$. Intuitively, when $\mathcal{P}$ visits a node $x \in V^*$ in state $\langle w, q, y \rangle$, it checks that R with initial configuration $(q, y \cdot x)$ is accepted by $\mathcal{S}^w_{\neg\varphi}$. In particular, when $y = \varepsilon$, then R with initial configuration (q, x) needs to be accepted by $\mathcal{S}^w_{\neg\varphi}$. States of the form $\langle w, q, \varepsilon \rangle$ are called *action states*. From these states $\mathcal{S}$ consults $\eta_{\neg\varphi}$ and T in order to impose new requirements on $\langle V^*, \tau_V \rangle$. States of the form $\langle w, q, y \rangle$, for $y \in V^+$, are called *navigation states*. From these states $\mathcal{P}$ only navigates downwards y to reach new action states.
– The transition function δ is defined for every state in $\langle w, q, x \rangle \in S \times Q \times heads(T)$ and letter in $A \in V$ as follows.
  • $\delta(\langle w, q, \epsilon \rangle, A) =$

$$\{(\uparrow, \langle w', q', y \rangle) \ : \ w' \in \eta_{\neg\varphi}(w, L(q, A)) \text{ and } \langle q, A, y, q' \rangle \in T\}.$$

  • $\delta(\langle w, q, y \cdot B \rangle, A) = \{(B, \langle w, q, y \rangle)\}$.
  Thus, in action states, $\mathcal{P}$ reads the direction of the current node and applies the rewrite rules of R in order to impose new requirements according to $\eta_{\neg\varphi}$. In navigation states, $\mathcal{P}$ needs to go downwards $y \cdot B$, so it continues in direction B.
– $p_0 = \langle w_0, q_0, x_0 \rangle$. Thus, in its initial state $\mathcal{P}$ checks that R with initial configuration (q_0, x_0) contains a trace that is accepted by $\mathcal{S}$ with initial state w_0.
– $\alpha = \{\langle w, q, \epsilon \rangle \ : \ w \in F \text{ and } q \in Q\}$. Note that only action states can be accepting states of $\mathcal{P}$.

We show that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff $R \not\models \varphi$. Assume first that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$. Then, there exists an accepting run $(p_0, x_0), (p_1, x_1), \ldots$ of $\mathcal{P}$ on $\langle V^*, \tau_V \rangle$. Extract from this run the subsequence of action states $(p_{i_1}, x_{i_1}), (p_{i_2}, x_{i_2}), \ldots$. As the run is accepting and only action states are accepting states, we know that this subsequence is infinite. Let $p_{i_j} = \langle w_{i_j}, q_{i_j}, \varepsilon \rangle$. By the definition of δ, the sequence $(q_{i_1}, x_{i_1}), (q_{i_2}, x_{i_2}), \ldots$ corresponds to an infinite path in the graph G_R. Also, by the definition of α, the run $w_{i_1}, w_{i_2}, \ldots$ is an accepting run of $\mathcal{S}_{\neg\varphi}$ on the trace of this path. Hence, G_R contains a trace that is accepted by $\mathcal{S}_{\neg\varphi}$, thus $R \not\models \varphi$.

Assume now that $R \not\models \varphi$. Then, there exists a path $(q_0, x_0), (q_1, x_1), \ldots$ in G_R whose trace does not satisfy φ. There exists an accepting run $w_0, w_1, \ldots$ of $\mathcal{S}_{\neg\varphi}$ on this trace. The combination of the two sequences serves as the subsequence of action states in an accepting run of $\mathcal{P}$. It is not hard to extend this subsequence to an accepting run of $\mathcal{P}$ on $\langle V^*, \tau_V \rangle$. □

5.2 Prefix-Recognizable Graphs

We now turn to consider prefix-recognizable systems. Again a configuration of a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ consists of a state in Q and a word in V^*. So, the store content is still a node in the tree V^*. However, in order to apply a rewrite rule it is not enough to know the direction of the node. Recall that in order to represent the configuration $(q, x) \in Q \times V^*$, our 2NBP memorizes the state q as part of its state space and it reads the node $x \in V^*$. In order to apply the rewrite rule

$t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$, the 2NBP has to go up the tree along a word $y \in \alpha_i$. Then, if $x = y \cdot z$, it has to check that $z \in \beta_i$, and finally guess a word $y' \in \gamma_i$ and go downwards y' to $y' \cdot z$. Finding a prefix y of x such that $y \in \alpha_i$, and a new word $y' \in \gamma_i$ is not hard: the 2NBP can emulate the run of the automaton $\mathcal{U}_{\alpha_i}$ while going up the tree and the run of the automaton $\mathcal{U}_{\gamma_i}$ backwards while going down the guessed y'. How can the 2NBP know that $z \in \beta_i$? In Subsection 3.2 we allowed the 2APT to branch to two states. The first, checking that $z \in \beta_i$ and the second, guessing y'. With 2NBP this is impossible and we provide a different solution. Instead of labeling each node $x \in V^*$ only by its direction, we can label it also by the regular expressions β for which $x \in \beta$. Thus, when the 2NBP runs $\mathcal{U}_{\alpha_i}$ up the tree, it can tell, in every node it visits, whether z is a member of β_i or not. If $z \in \beta_i$, the 2NBP may guess that time has come to guess a word in γ_i and run $\mathcal{U}_{\gamma_i}$ down the guessed word.

Thus, in the case of prefix-recognizable systems, the nodes of the tree whose membership is checked are labeled by both their directions and information about the regular expressions β. Let $\{\beta_1, \ldots, \beta_n\}$ be the set of regular expressions β_i such that there is a rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle \in T$. Let $\mathcal{D}_{\beta_i} = \langle V, D_{\beta_i}, \eta_{\beta_i}, q^0_{\beta_i}, F_{\beta_i} \rangle$ be the deterministic automaton for the reverse of the language of β_i. For a word $x \in V^*$, we denote by $\eta_{\beta_i}(x)$ the unique state that $\mathcal{D}_{\beta_i}$ reaches after reading the word x^R. Let $\Sigma = V \times \Pi_{1 \leq i \leq n} D_{\beta_i}$. For a letter $\sigma \in \Sigma$, let $\sigma[i]$, for $i \in \{0, \ldots n\}$, denote the i-th element in σ (that is, $\sigma[0] \in V$ and $\sigma[i] \in D_{\beta_i}$ for $i > 0$). Let $\langle V^*, \tau_\beta \rangle$ denote the Σ-labeled V-tree such that $\tau_\beta(\epsilon) = \langle \bot, q^0_{\beta_1}, \ldots, q^0_{\beta_n} \rangle$, and for every node $A \cdot x \in V^+$, we have $\tau_\beta(A \cdot x) = \langle A, \eta_{\beta_1}(A \cdot x), \ldots, \eta_{\beta_n}(A \cdot x) \rangle$. Thus, every node x is labeled by $dir(x)$ and the vector of states that each of the deterministic automata reach after reading x. Note that $\tau_\beta(x)[i] \in F_{\beta_i}$ iff x is in the language of β_i. Note also that $\langle V^*, \tau_\beta \rangle$ is a regular tree whose size is exponential in the sum of the lengths of the regular expressions $\beta_1, \ldots, \beta_n$.

Theorem 15. *Given a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and an LTL formula φ, there is a 2NBP $\mathcal{P}$ such that $\mathcal{P}$ accepts $\langle V^*, \tau_\beta \rangle$ iff $R \not\models \varphi$. The automaton $\mathcal{P}$ has $|Q| \cdot (|Q_\alpha| + |Q_\gamma|) \cdot |T| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.*

Proof: As before we use the NBW $\mathcal{S}_{\neg\varphi} = \langle 2^{AP}, W, \eta_{\neg\varphi}, w_0, F \rangle$.
We define $\mathcal{P} = \langle \Sigma, P, \delta, p_0 \alpha \rangle$ as follows.

- $\Sigma = V \times \Pi_{i=1}^n D_{\beta_i}$.
- $P = \{\langle w, q, t_i, s \rangle \mid w \in W,\ q \in Q,\ t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T,$ and $s \in Q_{\alpha_i} \cup Q_{\gamma_i}\}$
  Thus, $\mathcal{P}$ holds in its state a state of $\mathcal{S}_{\neg\varphi}$, a state in Q, the current rewrite rule being applied, and the current state in Q_α or Q_γ. A state $\langle w, q, \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle, s \rangle$ is an action state if s is the initial state of $\mathcal{U}_{\gamma_i}$, that is $s = q^0_{\gamma_i}$. In action states, $\mathcal{P}$ chooses a new rewrite rule $t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle$. Then $\mathcal{P}$ updates the $\mathcal{S}_{\neg\varphi}$ component according to the current location in the tree and moves to $q^0_{\alpha_{i'}}$, the initial state of $\mathcal{U}_{\alpha_{i'}}$. Other states are navigation states. If $s \in Q_{\gamma_i}$ is a state in $\mathcal{U}_{\gamma_i}$ (that is not initial), then $\mathcal{P}$ chooses a direction in the tree, a predecessor of the state in Q_{γ_i} reading the chosen direction, and moves in the chosen direction. If $s \in Q_{\alpha_i}$ is a state of $\mathcal{U}_{\alpha_i}$ then $\mathcal{P}$ moves up the tree (towards the root) while updating the state of $\mathcal{U}_{\alpha_i}$. If $s \in F_{\alpha_i}$ is an accepting state of $\mathcal{U}_{\alpha_i}$ and $\tau(x)[i] \in F_{\beta_i}$ marks the current node x as a member of

the language of β_i then $\mathcal{P}$ moves to some accepting state $s \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$ (recall that initial states and accepting states have no incoming / outgoing edges respectively).

– The transition function δ is defined for every state in P and letter in $\Sigma = V \times \Pi_{i=1}^{n} D_{\beta_i}$ as follows.
  - If $s \in Q_\alpha$ then

$$\delta(\langle w,q,t_i,s\rangle,\sigma) = \left\{ (\uparrow, \langle w,q,t_i,s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle \\ s' \in \eta_{\alpha_i}(s,\sigma[0]) \end{array} \right\} \bigcup \left\{ (\epsilon, \langle w,q,t_i,s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle, \\ s \in F_{\alpha_i},\ s' \in F_{\gamma_i}, \\ \text{and } \sigma[i] \in F_{\beta_i} \end{array} \right\}$$

  - If $s \in Q_\gamma$, then

$$\delta(\langle w,q,t_i,s\rangle,\sigma) = \left\{ (B, \langle w,q,t_i,s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle \\ s \in \eta_{\gamma_i}(s',B) \text{ and } B \in V \end{array} \right\} \bigcup \left\{ (\epsilon, \langle w',q'',t_{i'},s_0\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle, \\ t_{i'} = \langle q,\alpha_{i'},\beta_{i'},\gamma_{i'},q''\rangle, \\ w' \in \eta_{\neg\varphi}(w, L(q,\sigma[0])), \\ s = q^0_{\gamma_i} \text{ and } s_0 = q^0_{\alpha_{i'}} \end{array} \right\}$$

  Thus, when $s \in Q_\alpha$ the 2NBP $\mathcal{P}$ either chooses a successor s' of s and goes up the tree or in case s is the final state of $\mathcal{U}_{\alpha_i}$ and $\sigma[i] \in F_{\beta_i}$ then $\mathcal{P}$ chooses an accepting state $s' \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$.
  When $s \in Q_\gamma$ the 2NBP $\mathcal{P}$ either guesses a direction B and chooses a predecessor s' of s reading B or in case $s = q^0_{\gamma_i}$ is the initial state of $\mathcal{U}_{\gamma_i}$, the automaton $\mathcal{P}$ updates the state of $\mathcal{S}_{\neg\varphi}$, chooses a new rewrite rule $t_{i'} = \langle q,\alpha_{i'},\beta_{i'},\gamma_{i'},q''\rangle$ and moves to $q^0_{\alpha_{i'}}$, the initial state of $\mathcal{U}_{\alpha_{i'}}$.

– $p_0 = \langle w_0, q_0, t, x_0\rangle$, where t is an arbitrary rewrite rule.
  Thus, $\mathcal{P}$ navigates down the tree to the location x_0. There, it chooses a new rewrite rule and updates the state of $\mathcal{S}_{\neg\varphi}$ and the Q component accordingly.

– $\alpha = \{\langle w,q,t_i,s\rangle \mid w \in F,\ q \in Q,\ t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle, \text{ and } s = q^0_{\gamma_i}\}$
  Only action states may be accepting. As initial states have no incoming edges, in an accepting run, every navigation stage is finite.

As before we can show that a trace that violates φ and the rewrite rules used to create this trace can be used to produce a run of $\mathcal{P}$ on $\langle V^*, \tau_\beta\rangle$

Similarly, an accepting run of $\mathcal{P}$ on $\langle V^*, \tau_\beta\rangle$ is used to find a trace in G_R that violates φ. □

We can modify the conversion of 2NBP to ϵABW described in Section 4 for this particular problem. Instead of keeping in the state of the ϵABW a component of the direction of the node $A \in V \cup \{\perp\}$ we keep the letter from Σ (that is, the tuple $\langle A, q_1, \ldots, q_n\rangle \in V \times \Pi_{i=1}^{n} D_{\beta_i}$). When we take a move forward in the guessed direction $B \in V$ we update $\langle A, q_1, \ldots, q_n\rangle$ to $\langle B, \eta_{\beta_1}(q_1,B), \ldots, \eta_{\beta_n}(q_n,B)\rangle$. This way, the state space of the resulting ϵABW does not contain $(\Pi_{i=1}^{n} D_{\beta_i})^2$ but only $\Pi_{i=1}^{n} D_{\beta_i}$.

Combining Theorems 14, 15, and 11, we get the following.

Theorem 16. *The model-checking problem for a rewrite system R and an LTL formula φ is solvable in*

- *time $\|T\|^3 \cdot 2^{O(|\varphi|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|)}$, if R is a pushdown system.*
- *time $\|T\|^3 \cdot 2^{O(|\varphi|+|Q_\beta|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|+|Q_\beta|)}$, if R is a prefix-recognizable system. The problem is EXPTIME-hard in $|Q_\beta|$ even for a fixed formula.*

For pushdown systems (the first setting), our complexity coincides with the one in [EHRS00]. In Appendix C, we prove the EXPTIME lower bound in the second setting by a reduction from the membership problem of a linear space alternating Turing machine. Thus, our upper bounds are tight.

6 Relating Regular Labeling with Prefix-Recognizability

In this section we consider systems with regular labeling. We show first how to extend our approach to handle regular labeling. Both for branching-time and linear-time, the way we adapt our algorithms to handle regular labeling is very similar to the way we handle prefix-recognizability. In the branching-time framework the 2APT guesses a label and sends a copy of the automaton for the regular label to the root to check its guess. In the linear-time framework we include in the labels of the regular tree also data regarding the membership in the languages of the regular labeling. Based on these observations we proceed to show that the two questions are intereducible. We describe a reduction from μ-calculus (resp., LTL) model checking with respect to a prefix-recognizable system with simple labeling function to μ-calculus (resp., LTL) model checking with respect to a pushdown system with regular labeling. We also give reductions in the other direction. We note that we cannot just replace one system by another, but we also have to adjust the μ-calculus (resp., LTL) formula.

6.1 Model-Checking Graphs with Regular Labeling

We start by showing how to extend the construction in Subsection 3.2 to include also regular labeling. In order to apply a transition of the graph automaton $\mathcal{S}$, from configuration (q, x) our 2APT $\mathcal{A}$ has to guess a label $\sigma \in \Sigma$, apply the transition of $\mathcal{S}$ reading σ, and send an additional copy to the root that checks that the guess is correct and that indeed $x \in R_{\sigma,q}$. The changes to the construction in Subsection 3.1 are similar.

Theorem 17. *Given a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, T, L, q_0, x_0 \rangle$ where L is a regular labeling function and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $(V \cup \{\perp\})$-labeled V-trees such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff G_R satisfies $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|Q| \cdot (\|T\| + \|L\|) \cdot |V|)$ states, and its index is the index of $\mathcal{S}$ plus 1.*

Proof: We take the automaton constructed for the case of prefix-recognizable systems with simple labeling $\mathcal{A} = \langle V \cup \{\perp\}, P, \eta, p_0, \alpha \rangle$ and modify slightly its state set P and its transition η.

– $P = P_1 \cup P_2 \cup P_3$ where $P_1 = \{\exists, \forall\} \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma)$ and $P_2 = \{\exists, \forall\} \times T \times Q_\beta$ are just like in the previous proof and $P_3 = \bigcup_{\sigma \in \Sigma} \bigcup_{q \in Q} Q_{\sigma,q}$ includes all the states of the automata for the regular expressions appearing in L.
– The definition of $apply_T$ does not change and so does the transition of all navigation states. In the transition of action states, we include a disjunction that guesses the correct labeling. For a state $(d, w, q, t_i, s) \in P_1$ such that $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle$ and $s = q^0_{\gamma_i}$ we have

$$\eta((d, w, q, t_i, s), A) = \bigvee_{\sigma \in \Sigma} \Big(q^0_{\sigma,q} \wedge apply_T(\delta(w, \sigma), t_i, s) \Big).$$

For a state $s \in Q_{\sigma,q}$ and letter $A \in V \cup \{\bot\}$ we have

$$\eta(s, A) = \begin{cases} \bigvee_{s' \in \rho_{\sigma,q}(s,A)} (\uparrow, s') & \text{if } A \neq \bot \\ \mathbf{true} & \text{if } A = \bot \text{ and } s \in F_{\sigma,q} \\ \mathbf{false} & \text{if } A = \bot \text{ and } s \notin F_{\sigma,q} \end{cases}$$

□

Theorem 18. *The model-checking problem for a pushdown or a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0\rangle$ with a regular labeling L and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F\rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V| + \|L\| \cdot |V|$ and k is the index of $\mathcal{S}$.*

We show how to extend the construction in Subsection 5.2 to include also regular labeling. We add to the label of every node in the tree V^* also the states of the deterministic automata that recognize the reverse of the languages of the regular expressions of the labels. The navigation through the V-tree proceeds as before, and whenever the 2NBP needs to know the label of the current configuration (that is, in action states, when it has to update the state of $\mathcal{S}_{\neg\varphi}$), it consults the labels of the tree.

Formally, let $\{R_1, \ldots, R_n\}$ denote the set of regular expressions R_i such that there exist some state $q \in Q$ and proposition $p \in AP$ with $R_i = R_{p,q}$. Let $\mathcal{D}_{R_i} = \langle V, D_{R_i}, \eta_{R_i}, q^0_{R_i}, F_{R_i}\rangle$ be the deterministic automaton for the reverse of the language of R_i. For a word $x \in V^*$, we denote by $\eta_{R_i}(x)$ the unique state that $\mathcal{D}_{R_i}$ reaches after reading the word x^R. Let $\Sigma = V \times \Pi_{1 \leq i \leq n} D_{R_i}$. For a letter $\sigma \in \Sigma$ let $\sigma[i]$, for $i \in \{0, \ldots, n\}$, denote the i-th element of σ. Let $\langle V^*, \tau_L\rangle$ be the Σ-labeled V-tree such that $\tau_L(\epsilon) = \langle \bot, q^0_{R_1}, \ldots, q^0_{R_n}\rangle$ and for every node $A \cdot x \in V^+$ we have $\tau_L(A \cdot x) = \langle A, \eta_{R_1}(A \cdot x), \ldots, \eta_{R_n}(A \cdot x)\rangle$. The 2NBP $\mathcal{P}$ reads $\langle V^*, \tau_L\rangle$. Note that if the state space of $\mathcal{P}$ indicates that the current state of the rewrite system is q and $\mathcal{P}$ reads the node x, then for every atomic proposition p, we have that $p \in L(q, x)$ iff $\tau_L(x)[i] \in F_{R_i}$, where i is such that $R_i = R_{p,q}$. In action states, $\mathcal{P}$ needs to update the state of $\mathcal{S}_{\neg\varphi}$, which reads the label of the current configuration. Based on its current state and τ_L, the 2NBP $\mathcal{P}$ knows the letter with which $\mathcal{S}_{\neg\varphi}$ proceeds.

If we want to handle a prefix-recognizable system with regular labeling we have to label the nodes of the tree V^* by both the deterministic automata for regular expressions β_i and the deterministic automata for regular expressions $R_{p,q}$. Let $\langle V^*, \tau_{\beta,L}\rangle$ be the composition of $\langle V^*, \tau_\beta\rangle$ with $\langle V^*, \tau_L\rangle$. Note that $\langle V^*, \tau_L\rangle$ and $\langle V^*, \tau_{\beta,L}\rangle$ are regular, with $\|\tau_L\| = 2^{O(\|L\|)}$ and $\|\tau_{\beta,L}\| = 2^{O(|Q_\beta| + \|L\|)}$.

Theorem 19. *Given a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ where L is a regular labeling and an LTL formula φ, there is a 2NBP $\mathcal{S}$ such that $\mathcal{S}$ accepts $\langle V^*, \tau_{\beta,L} \rangle$ iff $R \not\models \varphi$. The automaton $\mathcal{S}$ has $|Q| \cdot (|Q_\alpha| + |Q_\gamma|) \cdot \|T\| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.*

Note that Theorem 19 differs from Theorem 15 only in the labeled tree whose membership is checked. Combining Theorems 19 and 11, we get the following.

Theorem 20. *The model-checking problem for a prefix-recognizable system R with regular labeling L and an LTL formula φ is solvable in time $\|T\|^3 \cdot 2^{O(|\varphi|+|Q_\beta|+\|L\|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|+|Q_\beta|+\|L\|)}$.*

For pushdown systems with regular labeling an alternative algorithm is given in Theorem 1. This, together with the lower bound in [EKS01], implies EXPTIME-hardness in terms of $\|L\|$. Thus, our upper bound is tight.

6.2 Prefix-Recognizable to Regular Labeling

We reduce μ-calculus (resp., LTL) model checking of prefix-recognizable systems to μ-calculus (resp., LTL) model checking of pushdown systems with regular labeling. Given a prefix-recognizable system we describe a pushdown system with regular labeling that is used in both reductions. We then explain how to adjust the μ-calculus or LTL formula.

Theorem 21. *Given a prefix-recognizable system $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0 \rangle$, a graph automaton $\mathcal{S}$, and an LTL formula φ, there is a pushdown system $R' = \langle 2^{AP'}$, V, Q', L', T', q'_0, $x_0 \rangle$ with a regular labeling function, a graph automaton $\mathcal{S}'$, and an LTL formula φ', such that $R \models \mathcal{S}$ iff $R' \models \mathcal{S}'$ and $R \models \varphi$ iff $R' \models \varphi'$. Furthermore, $|Q'| = |Q| \times |T| \times (|Q_\alpha| + |Q_\gamma|)$, $\|T'\| = O(\|T\|)$, $\|L\| = |Q_\beta|$, $|\mathcal{S}'| = O(|\mathcal{S}|)$, the index of $\mathcal{S}'$ equals the index of $\mathcal{S}$ plus one, and $|\varphi'| = O(|\varphi|)$. The reduction is computable in logarithmic space.*

The idea is to add to the configurations of R labels that would enable the pushdown system to simulate transitions of the prefix-recognizable system. Recall that in order to apply the rewrite rule $\langle q, \alpha, \beta, \gamma, q' \rangle$ from configuration (q, x), the prefix-recognizable system has to find a partition $y \cdot z$ of x such that the prefix y is a word in α and the suffix z is a word in β. It then replaces y by a word $y' \in \gamma$. The pushdown system can remove the prefix y letter by letter, guess whether the remaining suffix z is a word in β, and add y' letter by letter. In order to check the validity of guesses, the system marks every configuration where it guesses that the remaining suffix is a word in β. It then consults the regular labeling function in order to single out traces in which a wrong guess is made. For that, we add a new proposition, not_wrong, which holds in a configuration iff it is not the case that pushdown system guesses that the suffix z is in the language of some regular expression r and the guess turns out to be incorrect. The pushdown system also marks the configurations where it finishes handling some rewrite rule. For that, we add a new proposition, $ch\text{-}rule$, which is true only when the system finishes handling some rewrite rule and starts handling another.

The pushdown system R' has four modes of operation when it simulates a transition that follows a rewrite rule $\langle q, \alpha, \beta, \gamma, q' \rangle$. In *delete* mode, R' deletes letters from the store x while emulating a run of $\mathcal{U}_{\alpha_i}$. Delete mode starts from the initial state of $\mathcal{U}_{\alpha_i}$, from which R' proceeds until it reaches a final state of $\mathcal{U}_{\alpha_i}$. Once the final state of $\mathcal{U}_{\alpha_i}$ is reached, R' transitions to *change-direction* mode, where it does not change the store and just moves to a final state of $\mathcal{U}_{\gamma_i}$, and transitions to *write* mode. In write mode, R' guesses letters in V and emulates the run of $\mathcal{U}_{\gamma_i}$ on them backward, while adding them to the store. From the initial state of $\mathcal{U}_{\gamma_i}$ the pushdown system R' transitions to *change-rule* mode, where it chooses a new rewrite rule $\langle q', \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q''\rangle$ and transitions to delete mode. Note that if delete mode starts in configuration (q, x) it cannot last indefinitely. Indeed, the pushdown system can remove only finitely many letters from the store. On the other hand, since the store is unbounded, write mode can last forever. Hence, traces along which $\mathit{ch\text{-}rule}$ occurs only finitely often should be singled out.

Singling out of traces is done by the automaton $\mathcal{S}'$ and the formula φ' which restrict attention to traces in which $\mathit{not\_wrong}$ is always asserted and $\mathit{ch\text{-}rule}$ is asserted infinitely often.

Formally, R' has the following components.

– $AP' = AP \cup \{\mathit{not\_wrong}, \mathit{ch\text{-}rule}\}$.
– $Q' = Q \times T \times (\{\mathit{ch\text{-}dir}, \mathit{ch\text{-}rule}\} \cup Q_\alpha \cup Q_\gamma)$. A state $\langle q, t, s\rangle \in Q'$ maintains the state $q \in Q$ and the rewrite rule t currently being applied. the third element s indicates the mode of R'. Change-direction and change-rule modes are indicated by a marker. In delete and write modes, R' also maintains the current state of $\mathcal{U}_\alpha$ and $\mathcal{U}_\gamma$.
– For every proposition $p \in AP$, we have $p \in L'(q, x)$ iff $p \in L(q, x)$. We now describe the regular expression for the propositions $\mathit{ch\text{-}rule}$ and $\mathit{not\_wrong}$. The proposition $\mathit{ch\text{-}rule}$ holds in all the configuration in which the system is in change-rule mode. Thus, for every $q \in Q$ and $t \in T$, we have $R_{\langle q,t,\mathit{ch\text{-}rule}\rangle,\mathit{ch\text{-}rule}} = V^*$ and $R_{\langle q,t,\zeta\rangle,\mathit{ch\text{-}rule}} = \emptyset$ for $\zeta \neq \mathit{ch\text{-}rule}$. The proposition $\mathit{not\_wrong}$ holds in configurations in which we are not in change-direction mode, or configuration in which we are in change-direction mode and the store is in β, thus changing direction is possible in the configuration. Formally, for every $q \in Q$ and $t = \langle q', \alpha, \beta, \gamma, q\rangle \in T$, we have $R_{\langle q,t,\mathit{ch\text{-}dir}\rangle,\mathit{not\_wrong}} = \beta$ and $R_{\langle q,t,\zeta\rangle,\mathit{not\_wrong}} = V^*$ for $\zeta \neq \mathit{ch\text{-}dir}$.
– $q'_0 = \langle q_0, t, \mathit{ch\text{-}rule}\rangle$ for some arbitrary rewrite rule t.

The transition function of R' includes four types of transitions according to the four operation modes. In change-direction mode, in configuration $(\langle q, t, \mathit{ch\text{-}dir}\rangle, x)$ that applies the rewrite rule $t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle$, the system R' does not change x, and moves to a final state $s \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$. In change rule mode, in configuration $(\langle q, t, \mathit{ch\text{-}rule}\rangle, x)$, the system R' does not change x, it chooses a new rewrite rule $t' = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q'\rangle$, changes the Q component to q', and moves to the initial state $q^0_{\alpha_{i'}}$ of $\mathcal{U}_{\alpha_{i'}}$. In delete mode, in configuration $(\langle q, t, s\rangle, x)$, for $t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle$ and $s \in Q_{\alpha_i}$, the system R' proceeds by either removing one letter from x and continuing the run of $\mathcal{U}_{\alpha_i}$, or if $s \in F_{\alpha_i}$ is an accepting state of $\mathcal{U}_{\alpha_i}$ then R' leaves x unchanged, and changes s to $\mathit{ch\text{-}dir}$. In write mode, in configuration $(\langle q, t, s\rangle, x)$, for $t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle$ and $s \in Q_{\gamma_i}$, the system R' proceeds by either extending x with a guessed symbol from V and continuing the run of $\mathcal{U}_{\gamma_i}$ backward using the guessed symbol, or if $s = q^0_{\gamma_i}$, then R' leaves x unchanged and just replaces s by $\mathit{ch\text{-}rule}$. Formally, $T' = T'_{\mathit{ch\text{-}rule}} \cup T'_{\mathit{ch\text{-}dir}} \cup T'_\alpha \cup T'_\gamma$, where

– $T'_{ch\text{-}rule} =$

$$\left\{(\langle q,t,\mathit{ch\text{-}rule}\rangle, A, A, \langle q',t',s\rangle) \;\middle|\; t'=\langle q,\alpha_i,\beta_i,\gamma_i,q'\rangle,\ s=q^{0}_{\alpha_i} \text{ and } A\in V\right\}.$$

– $T'_{ch\text{-}dir} =$

$$\{(\langle q,t,\mathit{ch\text{-}dir}\rangle, A, A, \langle q,t,s\rangle) \mid t=\langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s\in F_{\gamma_i}, \text{ and } A\in V\}.$$

Note that the same letter A is removed from the store and added again. Thus, the store content of the configuration does not change.

– $T'_\alpha =$

$$\left\{(\langle q,t,s\rangle, A, \epsilon, \langle q,t,s'\rangle) \;\middle|\; \begin{array}{l} t=\langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s\in Q_\alpha, \\ s'\in\rho_{\alpha_i}(s,A), \text{ and } A\in V \end{array}\right\} \cup$$
$$\left\{(\langle q,t,s\rangle, A, A, \langle q,t,\mathit{ch\text{-}dir}\rangle) \;\middle|\; \begin{array}{l} t=\langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s\in Q_\alpha, \\ s\in F_{\alpha_i}, \text{ and } A\in V \end{array}\right\}.$$

– $T'_\gamma =$

$$\left\{(\langle q,t,s\rangle, A, AB, \langle q,t,s'\rangle) \;\middle|\; \begin{array}{l} t=\langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s\in Q_\gamma, \\ s\in\rho_{\gamma_i}(s',B), \text{ and } A,B\in V \end{array}\right\} \cup$$
$$\left\{(\langle q,t,s\rangle, A, A, \langle q,t,\mathit{ch\text{-}rule}\rangle) \;\middle|\; \begin{array}{l} t=\langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s\in Q_\gamma, \\ s=q^0_{\gamma_i} \text{ and } A\in V \end{array}\right\}.$$

As final states have no outgoing edges, after a state $\langle q,\langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,s\rangle$ for $s\in F_{\alpha_i}$ we always visit the state $\langle q,t,\mathit{ch\text{-}dir}\rangle$. Recall that initial states have no incoming edges. It follows that we always visit the state $\langle q,t,\mathit{ch\text{-}rule}\rangle$ after visiting a state $\langle q,\langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,q^0_{\gamma_i}\rangle$.

The automaton $\mathcal{S}'$ adjusts $\mathcal{S}$ to the fact that every transition in R corresponds to multiple transitions in R'. Accordingly, when $\mathcal{S}$ branches universally, infinite navigation stages and states not marked by $\mathit{not\_wrong}$ are allowed. Dually, when $\mathcal{S}$ branches existentially, infinite navigation stages and states not marked by $\mathit{not\_wrong}$ are not allowed.

Formally, let $\mathcal{S}=\langle\Sigma,W,\delta,w_0,F\rangle$. We define, $\mathcal{S}'=\langle\Sigma,W',\delta',w_0,\alpha\rangle$ where

– $W'=W\cup(\{\forall,\exists\}\times W)$ Intuitively, when $\mathcal{S}'$ reads configuration (q,x) and transitions to $\exists w$ it is searching for a successor of (q,x) that is accepted by S^w. The state $\exists w$ navigates to some configuration reachable from (q,x) of R' marked by $\mathit{ch\text{-}rule}$. Dually, when $\mathcal{S}'$ reads configuration (q,x) and transitions to $\forall w$ it is searching for all successors of (q,x) and tries to ensure that they are accepted by S^w. The state $\forall w$ navigates to all configurations reachable from (q,x) of R' marked by $\mathit{ch\text{-}rule}$.
– For every state $w\in W$ and letter $\sigma\in\Sigma$, the transition function δ' is obtained from δ by replacing every atom of the form $\Box w$ by $\Box(\forall w)$ and every atom of the form $\Diamond w$ by $\Diamond(\exists w)$.
For every state $w\in W$ and letter $\sigma\in\Sigma$, we have

$$\delta'(\forall w,\sigma)=\begin{cases}\mathbf{true} & \text{if } \sigma\not\models \mathit{not\_wrong}\\ (\varepsilon,w) & \text{if } \sigma\models \mathit{not\_wrong}\wedge \mathit{ch\text{-}rule}\\ (\Box,\forall w) & \text{if } \sigma\models \mathit{not\_wrong}\wedge\neg \mathit{ch\text{-}rule}\end{cases}$$

$$\delta'(\exists w, \sigma) = \begin{cases} \textbf{false} & \text{if } \sigma \not\models \mathit{not\_wrong} \\ (\varepsilon, w) & \text{if } \sigma \models \mathit{not\_wrong} \wedge \mathit{ch\text{-}rule} \\ (\Diamond, \exists w) & \text{if } \sigma \models \mathit{not\_wrong} \wedge \neg \mathit{ch\text{-}rule} \end{cases}$$

– The set α is obtained from F by including all states in $\{\forall\} \times W$ as the maximal even set and all states in $\{\exists\} \times W$ as the maximal odd set.

Claim. $G_R \models \mathcal{S}$ iff $G_{R'} \models \mathcal{S}'$

Proof: Assume that $G_{R'} \models \mathcal{S}'$. Let $\langle T', r' \rangle$ be an accepting run of $\mathcal{S}'$ on $G_{R'}$. We construct an accepting run $\langle T, r \rangle$ of $\mathcal{S}$ on G_R based on the subtree of nodes in T' labeled by states in W (it follows that these nodes are labeled by configurations with state $\mathit{ch\text{-}rule}$). Formally, we have the following. We have $r'(\varepsilon) = ((q_0, x_0), w_0)$. We add to T the node ε and label it $r(\varepsilon) = ((q_0, x_0), w_0)$. Given a node $z \in T$ labeled by $r(z) = ((q, x), w)$, it follows that there exists a node $z' \in T'$ labeled by $r'(z') = ((q, x), w)$. Let $\{((q_i, x_i), w_i)\}_{i \in I}$ be the labels of the minimal nodes in T' labeled by states in W. We add $|I|$ successors $\{a_i z\}_{i \in I}$ to z in T and label them $r(a_i z) = ((q_i, x_i), w_i)$. From the definition of R' it follows that $\langle T, r \rangle$ is a valid run of $\mathcal{S}$ on G_R. As every infinite path in T corresponds to an infinite path in T' all whose nodes are marked by configurations marked by $\mathit{not\_wrong}$ and infinitely many configurations are marked by $\mathit{ch\text{-}rule}$ it follows that $\langle T, r \rangle$ is an accepting run.

In the other direction, we extend an accepting run tree $\langle T, r \rangle$ of $\mathcal{S}$ on G_R into an accepting run tree of $\mathcal{S}'$ on $G_{R'}$ by adding transitions to $\{\forall, \exists\} \times W$ type states. □

Corollary 1. *Given a prefix-recognizable system R and a graph automaton $\mathcal{S}$ with n states and index k, we can model check $\mathcal{S}$ with respect to R in time exponential in $n \cdot k \cdot \|T\|$.*

Finally, we proceed to the case of an LTL formula φ. The formula φ' is the implication $\varphi'_1 \rightarrow \varphi'_2$ of two formulas. The formula φ'_1 holds in computations of R' that correspond to real computations of R. Thus, $\varphi'_1 = \Box \mathit{not\_wrong} \wedge \Box\Diamond \mathit{ch\text{-}rule}$. Then, φ'_2 adjusts φ to the fact that a single transition in R corresponds to multiple transitions in R'. Formally, $\varphi'_2 = f(\varphi)$, for the function f defined below.

– $f(p) = p$ for a proposition $p \in AP$.
– $f(\neg a) = \neg f(a)$, $f(a \vee b) = f(a) \vee f(b)$, and $f(a \wedge b) = f(a) \wedge f(b)$.
– $f(a \mathcal{U} b) = (\mathit{ch\text{-}rule} \rightarrow f(a)) \mathcal{U} (\mathit{ch\text{-}rule} \wedge f(b))$.
– $f(\bigcirc a) = \bigcirc((\neg \mathit{ch\text{-}rule}) \mathcal{U} (\mathit{ch\text{-}rule} \wedge f(a))$.

Claim. $G_R \models \varphi$ iff $G_{R'} \models \varphi'$

We first need some definitions and notations. We define a partial function g from traces in $G_{R'}$ to traces in G_R. Given a trace π' in $G_{R'}$, if $\pi' \not\models \varphi'_1$ then $g(\pi')$ is undefined. Otherwise, denote $\pi' = (p'_0, w_0), (p'_1, w_1), \ldots$ and

$$g(\pi') = \begin{cases} (p, w_0), g(\pi'_{\geq 1}) & \text{if } p'_0 = \langle p, t, \mathit{ch\text{-}rule} \rangle \\ g(\pi'_{\geq 1}) & \text{if } p'_0 = \langle p, t, \alpha \rangle \text{ and } \alpha \neq \mathit{ch\text{-}rule} \end{cases}$$

Thus, g picks from π' only the configurations marked by $\mathit{ch\text{-}rule}$, it then takes the state from Q that marks those configurations and the store. Furthermore given two traces π' and $g(\pi')$ we define a matching between locations in π' in which the configuration is marked by $\mathit{ch\text{-}rule}$ and the locations in $g(\pi')$. Given a location i in $g(\pi')$ we denote by $ch(i)$ the location in π' of the i-th occurrence of $\mathit{ch\text{-}rule}$ along π'.

Lemma 1. *1. For every trace π' of $G_{R'}$, $g(\pi')$ is either not defined or a valid trace of G_R.*
2. The function g is a bijection between $domain(g)$ and the traces of G_R.
3. For every trace π' of $G_{R'}$ such that $g(\pi')$ is defined, we have $(\pi', ch(i)) \models f(\varphi)$ iff $(g(\pi'), i) \models \varphi$

Proof: 1. Suppose $g(\pi')$ is defined, we have to show that it is a trace of G_R. The first pair in π' is $(\langle q_0, t, \mathit{ch\text{-}rule}\rangle, x_0)$. Hence $g(\pi')$ starts from (q_0, x_0). Assume by induction that the prefix of $g(\pi')$ up to location i is the prefix of some computation in G_R. We show that also the prefix up to location $i{+}1$ is a prefix of a computation. Let $(\langle q, t, \mathit{ch\text{-}rule}\rangle, x)$ be the i-th $\mathit{ch\text{-}rule}$ appearing in π', then the i-th location in $g(\pi')$ is (q, x). The computation of R' chooses some rewrite rule $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q'\rangle \in T$ and moves to state $\langle q', t_i, s\rangle$ where $s = q^0_{\alpha_i}$. It must be the case that a state $\langle q', t_i, \mathit{ch\text{-}dir}\rangle$ appears in the computation of R' after location $ch(i)$. Otherwise, the computation is finite and does not interest us. The system R' can move to a state marked by $\mathit{ch\text{-}dir}$ only from $s \in F_{\alpha_i}$, an accepting state of $\mathcal{U}_{\alpha_i}$. Hence, we conclude that $x = y \cdot z$ where $y \in \alpha_i$. As $\mathit{not\_wrong}$ is asserted everywhere along π' we know that $z \in \beta_i$. Now R' adds a word y' in γ_i to z and reaches state $(\langle q', t', \mathit{ch\text{-}rule}\rangle, y' \cdot z)$. Thus, the transition t is possible also in R and can lead from $(q, y \cdot z)$ to $(q', y' \cdot z)$.

2. It is quite clear that g is an injection. As above, given a trace π in G_R we can construct the trace π' in $G_{R'}$ such that $g(\pi') = \pi$.
3. We prove that $(\pi, i) \models \varphi$ iff $(\pi', ch(i)) \models \varphi$ by induction on the structure of φ.
   – For a boolean combination of formulas the proof is immediate.
   – For a proposition $p \in AP$, it follows from the proof above that if state (q, x) appears in location i in $g(\pi')$ then state $(\langle q, t, \mathit{ch\text{-}rule}\rangle, x)$ appears in location $ch(i)$ in π'. By definition $p \in L(q, x)$ iff $p \in L'(\langle q, t, \mathit{ch\text{-}rule}\rangle, x)$.
   – For a formula $\varphi = \psi_1 \mathcal{U} \psi_2$. Suppose $(g(\pi'), i) \models \varphi$. Then there exists some $j \geq i$ such that $(g(\pi'), j) \models \psi_2$ and for all $i \leq k < j$ we have $(g(\pi'), k) \models \psi_1$. By the induction assumption we have that $(\pi', ch(j)) \models f(\psi_2)$ (and clearly, $(\pi', ch(j)) \models \mathit{ch\text{-}rule}$), and for all $i \leq j < k$ we have $(\pi', ch(k)) \models \psi_1$. Furthermore, as every location marked by $\mathit{ch\text{-}rule}$ is associated by the function ch to some location in $g(\pi')$ all other locations are marked by $\neg \mathit{ch\text{-}rule}$. Hence, $(\pi', ch(i)) \models (\mathit{ch\text{-}rule} \rightarrow f(\psi_1)) \mathcal{U} (f(\psi_2) \wedge \mathit{ch\text{-}rule})$.
     The other direction is similar.
   – For a formula $\varphi = \bigcirc \psi$ the argument resembles the one above for $\mathcal{U}$.

□

We note that for every trace π' and $g(\pi')$ we have that $ch(0) = 0$. Claim 6.2 follows immediately.

If we use this construction in conjunction with Theorem 1, we get an algorithm whose complexity coincides with the one in Theorem 16.

Corollary 2. *Given a prefix-recognizable system R and an LTL formula φ we can model check φ with respect to R in time $O(\|T\|^3)\cdot 2^{O(|Q_\beta|)}\cdot 2^{O(|\varphi|)}$ and space $O(\|T\|^2)\cdot 2^{O(|Q_\beta|)}\cdot 2^{O(|\varphi|)}$.*

Note that for LTL, we change the formula itself while for μ-calculus we change the graph automaton resulting from the formula. Consider the following function from μ-calculus formulas to μ-calculus formulas.

– For $p \in AP$ we have $f(p) = \mathit{ch\text{-}rule} \wedge p$.
– $f(\neg a) = \neg f(a)$, $f(a \vee b) = f(a) \vee f(b)$, and $f(a \wedge b) = f(a) \wedge f(b)$.
– $f(\Box a) = \Box \nu X(f(a) \wedge \mathit{ch\text{-}rule} \vee \neg \mathit{not\_wrong} \vee \neg \mathit{ch\text{-}rule} \wedge \Box X)$.
– $f(\Diamond a) = \Diamond \mu X(f(a) \wedge \mathit{ch\text{-}rule} \wedge \mathit{not\_wrong} \vee \neg \mathit{ch\text{-}rule} \wedge \mathit{not\_wrong} \wedge \Diamond X)$.
– $f(\mu X a(X)) = \mu X(\mathit{ch\text{-}rule} \wedge f(a(X)))$.
– $f(\nu X a(X)) = \nu X(\mathit{ch\text{-}rule} \wedge f(a(X)))$.

We claim that $R \models \psi$ iff $R' \models f(\psi)$. However, the alternation depth of $f(\psi)$ my be much larger than that of ψ. For example, $\varphi = \mu X(p \wedge \Box(\neg p \wedge \Box(X \wedge \mu Y(q \vee \Box Y))))$ is alternation free, while $f(\varphi)$ is of alternation depth 3. This kind of transformation is more appropriate with the equational form of μ-calculus where we can declare all the newly added fixpoints as minimal and incur only an increase of 1 in the alternation depth.

We note that since we end up with a pushdown system with regular labeling, it is easy to extend the reduction to start with a prefix-recognizable system with regular labeling. It is left to show the reduction in the other direction.

We can also reduce the problem of μ-calculus (resp., LTL) model checking of pushdown graphs with regular labeling, to the problem of μ-calculus (resp., LTL) model checking of prefix-recognizable graphs. This is summarized in the following two theorems.

Theorem 22. *Given a pushdown system $R = \langle \Sigma, V, Q, T, L, q_0, x_0\rangle$ with a regular labeling function and a graph automaton $\mathcal{S}$, there is a prefix-recognizable system $R' = \langle \Sigma, V, Q', T', L', q_0', x_0\rangle$ with simple labeling and a graph automaton $\mathcal{S}'$ such that $R \models \mathcal{S}$ iff $R' \models \mathcal{S}'$. Furthermore, $|Q'| = |Q|+|\Sigma|$, $|Q'_\alpha|+|Q'_\gamma| = O(\|T\|)$, and $|Q'_\beta| = \|L\|$. The reduction is computable in logarithmic space.*

Theorem 23. *Given a pushdown system $R = \langle 2^{AP}, V, Q, T, L, q_0, x_0\rangle$ with a regular labeling function and an LTL formula φ, there is a prefix-recognizable system $R' = \langle 2^{AP'}, V, Q', T', L', q_0', x_0\rangle$ with simple labeling and an LTL formula φ' such that $R \models \varphi$ iff $R' \models \varphi'$. Furthermore, $|Q'| = O(|Q| \cdot |AP|)$, $|Q'_\alpha| + |Q'_\gamma| = O(\|T\|)$, and $|Q'_\beta| = 2^{\|L\|}$ yet the automata for Q'_β are deterministic. The reduction is computable in polynomial space.*

For the full constructions and proofs we refer the reader to [Pit04].

7 Realizability and Synthesis

In this section we show that the automata-theoretic approach can be used also to solve the realizability and synthesis problems for branching time and linear time specifications of pushdown and prefix-recognizable systems. We start with a definition of the realizability and synthesis problems and then proceed to give algorithms that solve these problems for μ-calculus and LTL.

Given a rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a partition $\{T_1, \ldots, T_m\}$ of T, a *strategy* of R is a function $f : Q \times V^* \to [m]$. The function f restricts the graph G_R so that from a configuration $(q, x) \in Q \times V^*$, only $f(q, x)$ transitions are taken. Formally, R and f together define the graph $G_{R,f} = \langle \Sigma, Q \times V^*, \rho, (q_0, x_0), L \rangle$, where $\rho((q, x), (q', y))$ iff $f(q, x){=}i$ and there exists $t \in T_i$ such that $\rho_t((q, x), (q', y))$. Given R and a specification ψ (either a graph automaton or an LTL formula), we say that a strategy f of R is *winning* for ψ iff $G_{R,f}$ satisfies ψ. Given R and ψ the problem of *realizability* is to determine whether there is a winning strategy of R for ψ. The problem of *synthesis* is then to construct such a strategy.[13] The setting described here corresponds to the case where the system needs to satisfy a specification with respect to environments modeled by a rewrite system. Then, at each state, the system chooses the subset of transitions to proceed with and the environment provides the rules that determine the successors of the state.

Similar to Theorems 7 and 15, we construct automata that solve the realizability problem and provide winning strategies. The idea is simple: a strategy $f : Q \times V^* \to [m]$ can be viewed as a $V \times [m]$-labeled V-tree. Thus, the realizability problem can be viewed as the problem of determining whether we can augment the labels of the tree $\langle V^*, \tau_V \rangle$ by elements in $[m]$, and accept the augmented tree in a run of $\mathcal{A}$ in which whenever $\mathcal{A}$ reads an entry $i \in [m]$, it applies to the transition function of the specification graph automaton only rewrite rules in T_i.

We give the solution to the realizability and synthesis problems for branching-time specifications. Given a rewrite system R and a graph automaton $\mathcal{S}$, we show how to construct a 2APT $\mathcal{A}$ such that the language of $\mathcal{A}$ is not empty iff $\mathcal{S}$ is realizable over R.

Theorem 24. *Given a rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$, a partition $\{T_1,$ …, $T_m\}$ of T, and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $((V \cup \{\perp\}) \times [m])$-labeled V-trees such that $L(\mathcal{A})$ contains exactly all the V-exhaustive trees whose projection on $[m]$ is a winning strategy of R for $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\| \cdot |V|)$ states, and its index is the index of $\mathcal{S}$ (plus 1 for a prefix-recognizable system).*

[13] Note that we define here only memoryless strategies. The strategy depends solely on the current configuration and not on the history of the computations. In general, in order to realize some specifications, strategies that depend on the history of the computation may be required. In order to solve realizability and synthesis for specifications that require memory we have to use a more complex algorithm. In the case of branching time specifications, we have to combine the rewrite system with the graph automaton for the specification and analyze the resulting game. In the case of linear time specifications, we have to combine the rewrite system with a deterministic parity automaton for the specification and analyze the resulting game. In both cases the analysis of the game can be done using 2-way tree automata. In the linear-time framework, the deterministic automaton may be doubly exponential larger than the LTL formula; and the total complexity of this algorithm is triple exponential. For further details and a matching lower bound we refer the reader to [LMS04].

Proof: Unlike Theorem 7 here we use the emptiness problem of 2APT instead of the membership problem. It follows that we have to construct a 2APT that ensures that its input tree is V-exhaustive and that the strategy encoded in the tree is winning. The modification to the construction in the proof of Theorem 7 are simple. Let $\mathcal{A}'$ denote the result of the construction in Theorem 6 or Theorem 7 with the following modification to the function $apply_T$. From action states we allow to proceed only with transitions from T_i, where i is the $[m]$ element of the letter we read. For example, in the case of a pushdown system, we would have for $c \in \Delta$, $w \in W$, $q \in Q$, $A \in V$ and $i \in [m]$ (the new parameter to $apply_T$, which is read from the input tree),

$$apply_T(c, w, q, A, i) = \begin{bmatrix} \langle \varepsilon, (w, q, \varepsilon) \rangle & \text{if } c = \varepsilon \\ \bigwedge_{\langle q,A,y,q' \rangle \in T_i} \langle \uparrow, (w, q', y) \rangle & \text{if } c = \Box \\ \bigvee_{\langle q,A,y,q' \rangle \in T_i} \langle \uparrow, (w, q', y) \rangle & \text{if } c = \Diamond \end{bmatrix}$$

We now construct the automaton $\mathcal{A}'' = \langle (V \cup \{\bot\} \times [m]), (V \cup \{\bot\}), \rho, bot, \{V\} \rangle$ of index 1 (i.e., every valid run is an accepting run) such that for every $A, B \in V \cup \{\bot\}$ and $i \in [m]$ we have

$$\rho(A, (B, i)) = \begin{bmatrix} \bigwedge_{C \in V} (C, C) & \text{if } A = B \\ \mathbf{false} & \text{if } A \neq B \end{bmatrix}$$

It follows that $\mathcal{A}'$ accepts only V-exhaustive trees. Finally, we take $\mathcal{A} = \mathcal{A}' \wedge \mathcal{A}''$ the conjunction of the two automata. $\Box$

Let $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$, let k be the index of $\mathcal{S}$, and let $\Gamma = (V \cup \{\bot\}) \times [m]$. By Theorem 2, we can transform $\mathcal{A}$ to a nondeterministic one-way parity tree automaton $\mathcal{N}$ with $2^{O(nk)}$ states and index $O(nk)$.[14] By [Rab69,Eme85], if $\mathcal{N}$ is nonempty, there exists a Γ-labeled V-tree $\langle V^*, f \rangle$ such that for all $\gamma \in \Gamma$, the set X_γ of nodes $x \in V^*$ for which $f(x) = \gamma$ is a regular set. Moreover, the nonemptiness algorithm of $\mathcal{N}$, which runs in time exponential in nk, can be easily extended to construct, within the same complexity, a deterministic word automaton $\mathcal{U}_\mathcal{A}$ over V such that each state of $\mathcal{U}_\mathcal{A}$ is labeled by a letter $\gamma \in \Gamma$, and for all $x \in V^*$, we have $f(x) = \gamma$ iff the state of $\mathcal{U}_\mathcal{A}$ that is reached by following the word x is labeled by γ. The automaton $\mathcal{U}_\mathcal{A}$ is then the answer to the synthesis problem. Note that since the transitions in $G_{\mathcal{R},f}$ take us from a state $x \in V^*$ to a state $y \in V^*$ such that x is not necessarily the parent of y in the V-tree, an application of the strategy f has to repeatedly run the automaton $\mathcal{U}_\mathcal{A}$ from its initial state resulting in a strategy whose every move is computed in time proportional to the length of the configuration. We can construct a strategy that computes the next step in time proportional to the difference between x and y. This strategy uses a pushdown store. It stores the run of $\mathcal{U}_\mathcal{A}$ on x on the pushdown store. In order compute the strategy in node y, we retain on the pushdown store only the part of the run of $\mathcal{U}_\mathcal{A}$ that relates to the common suffix of x and y. We then continue the run of $\mathcal{U}_\mathcal{A}$ on the prefix of y while storing it on the pushdown store.

[14] Note that the automaton $\mathcal{A}''$ is in fact a 1NPT of index 1. We can improve the efficiency of the algorithm by first converting $\mathcal{A}'$ into a 1NPT and only then combining the result with $\mathcal{A}''$. This would result in $|V|$ being removed from the figure describing the index of $\mathcal{N}$.

The construction described in Theorems 6 and 7 implies that the realizability and synthesis problem is in EXPTIME. Thus, it is not harder than in the satisfiability problem for the μ-calculus, and it matches the known lower bound [FL79]. Formally, we have the following.

Theorem 25. *The realizability and synthesis problems for a pushdown or a prefix-recognizable rewrite system $\mathcal{R} = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$, and k is the index of $\mathcal{S}$.*

By Theorem 4, if the specification is given by a μ-calculus formula ψ, the bound is the same, with $n = |\psi| \cdot |Q| \cdot \|T\| \cdot |V|$, and k being the alternation depth of ψ.

In order to use the above algorithm for realizability of linear-time specifications we cannot use the 'usual' translations of LTL to μ-calculus [Dam94,dAHM01]. The problem is with the fact that these translations are intended to be used in μ-calculus model checking. The translation from LTL to μ-calculus used for model checking [Dam94] cannot be used in the context of realizability [dAHM01]. We have to use a doubly exponential translation intended for realizability [dAHM01], this, however, results in a triple exponential algorithm which is, again, less than optimal.

Alur et al. show that LTL realizability and synthesis can be exponentially reduced to μ-calculus realizability [ATM03]. Given an LTL formula φ, they construct a graph automaton $\mathcal{S}_\varphi$ such that $\mathcal{S}_\varphi$ is realizable over R iff φ is realizable over R. The construction of the graph automaton proceeds as follows. According to Theorem 5, for every LTL formula ψ we can construct an NBW N_ψ such that $L(N_\psi) = L(\psi)$. We construct an NBW $N_{\neg\varphi} = \langle \Sigma, W, \eta, w_0, F \rangle$ from $\neg\varphi$. We then construct the graph automaton $\mathcal{S}_\varphi = \langle \Sigma, W, \rho, w_0, \{F, W\} \rangle$ where $\rho(w, \sigma) = \bigwedge_{w' \in \eta(w,\sigma)} \Box w'$ and the parity condition $\{F, W\}$ is equivalent to the co-Büchi condition F. It follows that $\mathcal{S}_\varphi$ is a universal automaton and has a unique run over every trace. Alur et al. show that the fact that $\mathcal{S}_\varphi$ has a unique run over every trace makes it adequate for solving the realizability of φ [ATM03]. The resulting algorithm is exponential in the rewrite system and doubly exponential in the LTL formula. As synthesis of LTL formulas with respect to finite-state environments is already 2EXPTIME-hard [PR89], this algorithm is optimal. Note that realizability with respect to LTL specifications is exponential in the system already for pushdown systems and exponential in all components of the system for prefix-recognizable systems.

8 Discussion

The automata-theoretic approach has long been thought to be inapplicable for effective reasoning about infinite-state systems. We showed that infinite-state systems for which decidability is known can be described by finite-state automata, and therefore, the states and transitions of such systems can be viewed as nodes in an infinite tree and transitions between states can be expressed by finite-state automata. As a result, automata-theoretic techniques can be used to reason about such systems. In particular, we showed that various problems related to the analysis of such systems can be reduced to the membership or emptiness problems for alternating two-way tree automata. Our framework achieves the same complexity bounds of known model-checking algorithms and gives the first solution to model-checking LTL with respect to prefix-recognizable systems. In [PV04] we show how to extend it also to global model checking. In [Cac03,PV04] the scope of automata-theoretic reasoning is extended beyond prefix-recognizable systems.

We have shown that the problems of model checking with respect to pushdown systems with regular labeling and model checking with respect to prefix-recognizable systems are intimately related. We give reductions between model checking of pushdown systems with regular labeling and model checking of prefix-recognizable systems with simple labeling.

The automata-theoretic approach offers several extensions to the model checking setting. The systems we want to reason about are often augmented with *fairness constraints*. Like state properties, we can define a *regular fairness constraint* by a regular expression α, where a computation of the labeled transition graph is fair iff it contains infinitely many states in α (this corresponds to weak fairness; other types of fairness can be defined similarly). It is easy to extend our model-checking algorithm to handle fairness (that is, let the path quantification in the specification range only on fair paths[15]). In the branching-time framework, the automaton $\mathcal{A}$ can guess whether the state currently visited is in α, and then simulate the word automaton $\mathcal{U}_\alpha$ upwards, hoping to visit an accepting state when the root is reached. When $\mathcal{A}$ checks an existential property, it has to make sure that the property is satisfied along a fair path, and it is therefore required to visit infinitely many states in α. When $\mathcal{A}$ checks a universal property, it may guess that a path it follows is not fair, in which case $\mathcal{A}$ eventually always send copies that simulate the automaton for $\neg\alpha$. In the linear-time framework, we add the automata for the fairness constraints to the tree whose membership is checked. The guessed path violating the property must visit infinitely many fair states. The complexity of the model-checking algorithm stays the same.

Another extension is the treatment of μ-calculus specifications with *backwards modalities*. While forward modalities express weakest precondition, backward modalities express strongest postcondition, and they are very useful for reasoning about the past [LPZ85]. In order to adjust graph automata to backward reasoning, we add to Δ the "directions" $\Diamond^-$ and $\Box^-$. This enables the graph automata to move to predecessors of the current state. More formally, if a graph automaton reads a state x of the input graph, then fulfilling an atom $\Diamond^- t$ requires $\mathcal{S}$ to send a copy in state t to some predecessor of x, and dually for $\Box^- t$. Theorem 4 can then be extended to μ-calculus formulas and graph automata with both forward and backward modalities [Var98]. Extending our solution to graph automata with backward modalities is simple. Consider a configuration $(q, x) \in Q \times V^*$ in a prefix-recognizable graph. The predecessors of (q, x) are configurations $(q'y)$ for which there is a rule $\langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \in T$ and partitions $x' \cdot z$ and $y' \cdot z$, of x and y, respectively, such that x' is accepted by $\mathcal{U}_{\gamma_i}$, z is accepted by $\mathcal{U}_{\beta_i}$, and y' is accepted by $\mathcal{U}_{\alpha_i}$. Hence, we can define a mapping T^- such that $\langle q, \gamma, \beta, \alpha, q'\rangle \in T^-$ iff $\langle q, \alpha, \beta, \gamma, q\rangle \in T$, and handle atoms $\Diamond^- t$ and $\Box^- t$ exactly as we handle $\Diamond t$ and $\Box t$,

[15] The exact semantics of *fair graph automata* as well as *fair μ-calculus* is not straightforward, as they enable cycles in which we switch between existential and universal modalities. To make our point here, it is simpler to assume in the branching-time framework, say, graph automata that correspond to CTL* formulas.

only that for them we apply the rewrite rules in T^- rather than these in T. The complexity of the model-checking algorithm stays the same. Note that the simple solution relies on the fact that the structure of the rewrite rules in a prefix-recognizable rewrite system is symmetric (that is, switching α and γ results in a well-structured rule), which is not the case for pushdown systems[16].

Recently, Alur et al. suggested the logic CARET, that can specify non-regular properties [AEM04]. Our algorithm generalizes to CARET specifications as well. Alur et al. show how to combine the specification with a pushdown system in a way that enables the application of our techniques. The logic CARET is tailored for use in conjunction with pushdown systems. It is not clear how to modify CARET in order to apply to prefix-recognizable systems. Other researchers have used the versatility of the automata-theoretic framework for reasoning about infinite-state systems. Cachat shows how to model check μ-calculus specifications with respect to high order pushdown graphs [Cac03]. Gimbert shows how to solve games over pushdown graphs where the winning conditions are combinations of parity and unboundedness [Gim03].

References

[ACM06] R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Proc 18th Int. Conf. on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2006.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proc. 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2725 of *Lecture Notes in Computer Science*, pages 67–79. Springer, 2004.

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symp. on Foundations of Computer Science*, pages 100–109, 1997.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. 36th ACM Symp. on Theory of Computing*. ACM, ACM press, 2004.

[AM06] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Proc. Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2006.

[ATM03] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for infinite games on recursive game graphs. In *Proc 15th Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science. Springer, 2003.

[BC04] M. Bojanczyk and T. Colcombet. Tree-walking automata cannot be determinized. In *Proc. 31st Int. Colloq. on Automata, Languages, and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 246–256. Springer, 2004.

[BC05] M. Bojanczyk and T. Colcombet. Tree-walking automata do not recognize all regular languages. In *Proc. 37th ACM Symp. on Theory of Computing*. ACM press, 2005.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conf. on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[16] Note that this does not mean we cannot model check specifications with backwards modalities in pushdown systems. It just means that doing so involves rewrite rules that are no longer pushdown. Indeed, a rule $\langle q, A, x, q'\rangle \in T$ in a pushdown system corresponds to the rule $\langle q, A, V^*, x, q'\rangle \in T$ in a prefix-recognizable system, inducing the rule $\langle q', x, V^*, A, q\rangle \in T^{-1}$.

[BLM01] P. Biesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer, 2001.

[BLS06] V. Bárány, C. Löding, and O. Serre. Regularity problem for visibly pushdown languages. In *Proc. 23rd Symp. on Theoretical Aspects of Computer Science*, volume 3884 of *Lecture Notes in Computer Science*, pages 420–431. Springer, 2006.

[BMP05] L. Bozzelli, A. Murano, and A. Peron. Pushdown module checking. In *Proc. 12th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning*, Lecture Notes in Artificial Intelligence, pages 504–518. Springer, 2005.

[Boz06] L. Bozzelli. Complexity results on branching-time pushdown model checking. In *Proc. 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2006.

[BQ96] O. Burkart and Y.-M. Quemener. Model checking of infinite graphs defined by graph grammars. In *Proc. 1st Int. workshop on verification of infinite states systems*, volume 6 of *ENTCS*, page 15. Elsevier, 1996.

[BR00] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. 7th Int. SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.

[BR01] T. Ball and S. Rajamani. The SLAM toolkit. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[BS92] O. Burkart and B. Steffen. Model checking for context-free processes. In *3rd Int. Conf. on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 1992.

[BS95] O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic J. Comut.*, 2:89–125, 1995.

[BSW03] A.-J. Bouquet, O. Serre, and I. Walukiewicz. Pushdown games with unboundedness and regular conditions. In *Proc. 23rd Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2003.

[BTP06] L. Bozzelli, S. La Torre, and A. Peron. Verification of well-formed communicating recursive state machines. In *Proc. 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 412–426. Springer, 2006.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congress on Logic, Method, and Philosophy of Science. 1960*, pages 1–12. Stanford University Press, 1962.

[Bur97] O. Burkart. Model checking rationally restricted right closures of recognizable graphs. In F. Moller, editor, *Proc. 2nd Int. workshop on verification of infinite states systems*, 1997.

[Cac03] T. Cachat. Higher order pushdown automata, the caucal hierarchy of graphs and parity games. In *Proc. 30th Int. Colloq. on Automata, Languages, and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 556–569. Springer, 2003.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. 23rd Int. Colloq. on Automata, Languages, and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 1996.

[Cau03] D. Caucal. On infinite transition graphs having a decidable monadic theory. *Theoretical Computer Science*, 290(1):79–115, 2003.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languagues and Systems*, 8(2):244–263, 1986.

[CFF$^+$01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer, 2001.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, 1981.

[CW02] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proc. 9th ACM conference on Computer and Communications Security*, pages 235–244. ACM, 2002.

[CW03] A. Carayol and S. Wöhrle. The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *Proc. 23rd Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *Lecture Notes in Computer Science*, pages 112–123. Springer, 2003.

[dAHM01] L. de Alfaro, T.A. Henzinger, and R. Majumdar. From verification to control: dynamic programs for omega-regular objectives. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, pages 279–290. IEEE Computer Society Press, 2001.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[DW99] M. Dickhfer and T. Wilke. Timed alternating tree automata: the automata-theoretic solution to the TCTL model checking problem. In *Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 1999.

[EE05] J. Esparza and K. Etessami. Verifying probabilistic procedural programs. In *Proc. 24th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2005.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc 12th Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[EHvB99] J. Engelfriet, H.J. Hoggeboom, and J.-P van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 328–337, 1988.

[EJ91] E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 368–377, 1991.

[EJS93] E.A. Emerson, C. Jutla, and A.P. Sistla. On model-checking for fragments of μ-calculus. In *Proc 5th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 1993.

[EKS01] J. Esparza, A. Kucera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. 18th Symp. on Theoretical Aspects of Computer Science*, volume 2215 of *Lecture Notes in Computer Science*, pages 316–339. Springer, 2001.

[EKS06] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. In *Proc. 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2006.

[EL86] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pages 267–278, 1986.

[Eme85] E.A. Emerson. Automata, tableaux, and temporal logics. In *Proc. Workshop on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 79–87. Springer, 1985.

[Eme97] E.A. Emerson. Model checking and the μ-calculus. In N. Immerman and Ph.G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, pages 185–214. American Mathematical Society, 1997.

[ES01] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2001.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Science*, 18:194–211, 1979.

[Gim03] H. Gimbert. Explosion and parity games over context-free graphs. Technical Report 2003-015, Liafa, CNRS, Paris University 7, 2003.

[GO03] P. Gastin and D. Oddoux. Ltl with past and two-way very-weak alternating automata. In *28th Int. Symp. on Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 439–448. Springer, 2003.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[HHK96] R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *Proc 8th Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 1996.

[HHWT95] T.A. Henzinger, P.-H Ho, and H. Wong-Toi. A user guide to HYTECH. In *Proc. 1st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer, 1995.

[HKV96] T.A. Henzinger, O. Kupferman, and M.Y. Vardi. A space-efficient on-the-fly algorithm for real-time model checking. In *7th Int. Conf. on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 514–529. Springer, 1996.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[JSWR06] S. Jha, S. Schwoon, H. Wang, and T. Reps. Weighted pushdown systems and trust-management systems. In *Proc. 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, page 126. Springer, 2006.

[JW95] D. Janin and I. Walukiewicz. Automata for the modal μ-calculus and related results. In *20th Int. Symp. on Mathematical Foundations of Computer Science*, volume 969 of *Lecture Notes in Computer Science*, pages 552–562. Springer, 1995.

[KG06] V. Kahlon and A. Gupta. An automata theoretic approach for model checking threads for ltl properties. In *Proc. 21st IEEE Symp. on Logic in Computer Science*, pages 101–110. IEEE, IEEE press, 2006.

[KG07] V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *Proc. 34th ACM Symp. on Principles of Programming Languages*, 2007.

[KGS06] V. Kahlon, A. Gupta, and N. Sinha. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *Proc 18th Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science, pages 286–299. Springer, 2006.

[KIG05] V. Kahlon, F. Ivanĉić, and A. Gupta. Reasoning about threads communicating via locks. In *Proc 17th Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science, pages 505–518. Springer, 2005.

[KNUW05] T. Knapik, D. Niwinski, P. Urzczyn, and I. Walukiewicz. Unsafe grammars and panic automata. In *Proc. 32nd Int. Colloq. on Automata, Languages, and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 1450–1461. Springer, 2005.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP95] O. Kupferman and A. Pnueli. Once and for all. In *Proc. 10th IEEE Symp. on Logic in Computer Science*, pages 25–35, 1995.

[KPV02] O. Kupferman, N. Piterman, and M.Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *Proc 14th Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 371–385. Springer, 2002.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[KV98] O. Kupferman and M.Y. Vardi. Modular model checking. In *Proc. Compositionality Workshop*, volume 1536 of *Lecture Notes in Computer Science*, pages 381–401. Springer, 1998.

[KV99] O. Kupferman and M.Y. Vardi. Robust satisfaction. In *10th Int. Conf. on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 383–398. Springer, 1999.

[KV00a] O. Kupferman and M.Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proc 12th Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2000.

[KV00b] O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, 2000.

[KV01] O. Kupferman and M.Y. Vardi. On bounded specifications. In *Proc. 8th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 24–38. Springer, 2001.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[LMS04] C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *Proc. 24th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 408–420. Springer, 2004.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, 1985.

[LPY97] K. G. Larsen, P. Petterson, and W. Yi. UPPAAL: Status & developments. In *Proc 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 456–459. Springer, 1997.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[MS85] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[MS87] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[Nev02] F. Neven. Automata, logic, and XML. In *Proc. 11th Annual Conf. of the European Association for Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2002.

[Ong06] L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Proc. 21st IEEE Symp. on Logic in Computer Science*, pages 81–90. IEEE, IEEE press, 2006.

[Pit04] N. Piterman. *Verification of Infinite-State Systems*. PhD thesis, Weizmann Institute of Science, 2004.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, 1989.

[PV03] N. Piterman and M.Y. Vardi. From bidirectionality to alternation. *Theoretical Computer Science*, 295(1–3):295–321, 2003.

[PV04] N. Piterman and M. Vardi. Global model-checking of infinite-state systems. In *Proc 16th Int. Conf. on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 387–400. Springer, 2004.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 8th ACM Symp. on Principles of Programming Languages*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[Sch02] S. Schwoon. *Model-checking pushdown systems*. PhD thesis, Technische Universität München, 2002.

[Ser06] O. Serre. Parity games played on transition graphs of one-counter processes. In *Proc. 9th Int. Conf. on Foundations of Software Science and Computation Structures*, volume 3921 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2006.

[SSE06] D. Suwimonteerabuth, S. Schwoon, and Javier Esparza. Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In *4th Int. Symp. on Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science. Springer, 2006.

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th Int. Colloq. on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer, Berlin, 1998.

[VB00] W. Visser and H. Barringer. Practical CTL* model checking: Should SPIN be extended? *Software Tools for Technology Transfer*, 2(4):350–365, 2000.

[VW86a] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pages 332–344, 1986.

[VW86b] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and Systems Science*, 32(2):182–221, 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Proc 8th Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1996.

[Wal00] I. Walukiewicz. Model checking ctl properties of pushdown systems. In *Proc. 20th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lecture Notes in Computer Science*, pages 127–138. Springer, 2000.

[Wal02] I. Walukiewicz. Monadic second-order logic on tree-like structures. *Theoretical Computer Science*, 275(1-2):311–346, 2002.

[Wil99] T. Wilke. CTL$^+$ is exponentially more succinct than CTL. In *Proc. 19th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 1999.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 185–194, 1983.

[WW96] B. Willems and P. Wolper. Partial-order methods for model checking: From linear time to branching time. In *Proc. 11th IEEE Symp. on Logic in Computer Science*, pages 294–303, 1996.

A Proof of Claim 4.3

The proof of the claim is essentially equivalent to the same proof in [PV03].

Claim. $L(\mathcal{A}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau\rangle \in L(\mathcal{P})$.

Proof: We prove that $\langle \Upsilon^*, \tau\rangle \in L(\mathcal{P})$ implies $L(\mathcal{A}) \neq \emptyset$. Let $r = (p_0, w_0) \cdot (p_1, w_1) \cdot (p_2, w_2) \cdots$ be an accepting run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau\rangle$. We add the annotation of the locations in the run $(p_0, w_0, 0) \cdot (p_1, w_1, 1) \cdot (p_2, w_2, 2) \cdots$. We construct the run $\langle T', r'\rangle$ of $\mathcal{A}$. For every node $x \in T'$, if x is labeled by a singleton state we add a tag to x some triplet from the run r. If x is labeled by a pair state we add two tags to x, two triplets from the run r. The labeling and the tagging conform to the following.

– Given a node x labeled by state (p, d, α) and tagged by the triplet (p', w, i) from r, we build r' so that $p = p'$ and $d = \rho_\tau(w)$. Furthermore all triplets in r whose third element is greater than i have their second element greater or equal to w (Υ^* is ordered according to the lexical order on the reverse of the words).
– Given a node x labeled by state (q, p, d, α) and tagged by the triplets (q', w, i) and (p', w', j) from r, we build r' so that $q = q'$, $p = p'$, $w = w'$, $d = \rho_\tau(w)$, and $i < j$. Furthermore all triplets in r whose third element k is between i and j, have their second element greater or equal to w. Also, if $j > i + 1$ then $w_{j-1} = \upsilon \cdot w_j$ for some $\upsilon \in \Upsilon$.

Construct the run tree $\langle T', r'\rangle$ of $\mathcal{A}$ as follows. Label the root of T' by $(p_0, d^0_\tau, \bot)$ and tag it by $(p_0, \varepsilon, 0)$. Given a node $x \in T'$ labeled by (p, d, α) tagged by (p, w, i). Let (p_j, w_j, j) be the minimal $j > i$ such that $w_j = w$. If $j = i + 1$ then add one son to x, label it $(p_j, d, \bot)$ and tag it (p_j, w, j). If $j > i + 1$, then $w_{j-1} = \upsilon \cdot w_i$ for some $\upsilon \in \Upsilon$ and we add two sons to x, label them (p, p_j, d, β) and (p_j, d, β). We tag (p_i, p_j, d, β) by (p, w, i) and (p_j, w, j), and tag (p_j, d, β) by (p_j, w, j), β is $\top$ if there is a visit to F between locations i and j in r. If there is no other visit to w then $w_{i+1} = \upsilon \cdot w$ for some $\upsilon \in \Upsilon$. We add one son to x and label it $(p_{i+1}, \rho_\tau(d, \upsilon), \bot)$ and tag it $(p_{i+1}, \upsilon \cdot w, i+1)$. Obviously the labeling and the tagging conform to the assumption.

Given a node x labeled by a state (p, q, d, α) and tagged by (p, w, i) and (q, w, j). Let (p_k, w, k) be the first visit to w between i and j. If $k = i + 1$ then add one son to x, label it $(p_k, q, d, f_\alpha(p_k, q))$, and tag it by (p_k, w, k) and (q, w, j). If $k > i + 1$ then add two sons to x and label them $(p, p_k, d, f_{\beta_1}(p, p_k))$ and $(p_k, q, d, f_{\beta_2}(p_k, q))$ where β_1, β_2 are determined according to the visits to F between i and j. We tag the state $(p, p_k, d, f_{\beta_1}(p, p_k))$ by (p, w, i) and (p_k, w, k) and tag $(p_k, t, d, f_{\beta_2}(p_k, q))$ by (p_k, w, k) and (q, w', j).

If there is no visit to w between i and j it must be the case that all triplets in r between i and j have the same suffix $\upsilon \cdot w$ for some $\upsilon \in \Upsilon$ (otherwise w is visited). We add a son to x labeled $(p_{i+1}, q_{j-1}, \rho_\tau(d, \upsilon), f_\alpha(p', q'))$ and tagged by $(p_{i+1}, \upsilon \cdot$

$w, i+1)$ and $(p_{j-1}, \upsilon \cdot w, j-1)$. We are ensured that $p_{j-1} \in C_q^{L_\tau(\rho_\tau(d,\upsilon))}$ as $(\uparrow, p_j) \in \delta(p_{j-1}, \tau(\upsilon \cdot w))$.

In the other direction, given an accepting run $\langle T', r' \rangle$ of $\mathcal{A}$ we use the recursive algorithm in Figure 1 to construct a run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau \rangle$.

A node $x \cdot a$ in T' is *advancing* if the transition from x to $x \cdot a$ results from an atom $(1, r'(x{\cdot}a))$ that appears in $\eta(r'(x))$. An advancing node that is the immediate successor of a singleton state satisfies the disjunct $\bigvee_{\upsilon \in \Upsilon} \bigvee_{(\upsilon,p') \in \delta(p,L(d))} (1, (p', \rho_\tau(d,\upsilon), \perp))$ in η. We tag this node by the letter υ that was used to satisfy the transition. Similarly, an advancing node that is the immediate successor of a pair state satisfies the disjunct $\bigvee_{\upsilon \in \Upsilon} \bigvee_{\langle \upsilon,p' \rangle \in \delta(p_1, L_\tau(d))} \bigvee_{p'' \in C_{p_2}^{L_\tau(d)}} (1, (p', p'', \rho_\tau(d,\upsilon), f_\alpha(p', p'')))$ in η. We tag this node by the letter υ that was used to satisfy the transition. We use these tags in order to build the run of $\mathcal{P}$. When handling advancing nodes we update the location on the tree Υ^* according to the tag. For an advancing node x we denote by $tag(x)$ the letter in Υ that tags it. A node is *non advancing* if the transition from x to $x \cdot a$ results from an atom $(0, r'(x \cdot a))$ that appears in $\eta(r'(x))$.

The function **build_run** uses the variable w to hold the location in the tree $\langle \Upsilon^*, \tau \rangle$. Working on a singleton (p, d, α) the variable add_l is used to determine whether p was already added to the run. Working on a pair (p, q, d, α) the variable add_l is used to determine whether p was already added to the run and the variable add_r is used to determine whether q was already added to the run.

The intuition behind the algorithm is quite simple. We start with a node x labeled by a singleton (p, d, α). If the node is advancing we update w by $tag(x)$. Now we add p to r (if needed). The case where x has one son matches a transition of the form $(\Delta, p') \in \delta(p, L_\tau(d))$. In this case we move to handle the son of x and clearly p' has to be added to the run r. In case $\Delta = \varepsilon$ the son of x is non advancing and p' reads the same location w. Otherwise, w is updated by Δ and p' reads $\Delta \cdot w$. The case where x has two sons matches a guess that there is another visit to w. Thus, the computation splits into two sons (p, q, d, β) and (q, d, β). Both sons are non advancing. The state p was already added to r and q is added to r only in the first son.

With a node x labeled by a pair (p, q, d, α), the situation is similar. The case where x has one non advancing son matches a transition of the form $(\epsilon, s') \in \delta(p, A)$. Then we move to the son. The state p' is added to r but q is not. The case where x has two non advancing sons matches a split to (p, p', d, α_1) and (p', q, d, α_2). Only p' is added to r as p and q are added by the current call to build_run or by an earlier call to build_run. The case where x has one advancing son matches the move to the state $(p', q', \rho_\tau(d,\upsilon), \alpha)$ and checking that $q' \in C_q^{L_\tau(\rho_\tau(d,\upsilon))}$. Both p' and q' are added to r and **handle_**C_q handles the sequence of ε transitions that connects q' to q.

It is quite simple to see that the resulting run is a valid and accepting run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau \rangle$.

$\square$

B Lower bound on Emptiness of 2NBP

We give the full details of the construction in the proof of Theorem 13 Formally, $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$ where

```
**build_run** $(x, r'(x) = (p, d, \alpha), w, add_l, add_r)$
  if (advancing($x$))
    $w := tag(x) \cdot w$;
  if ($add_l$)
    $r := r \cdot (w, p)$;
  if ($x$ has one son $x \cdot a$)
    build_run $(x \cdot a, r'(x \cdot a), w, 1, 0)$
  if ($x$ has two sons $x \cdot a$ and $x \cdot b$)
    build_run $(x \cdot a, r'(x \cdot a), w, 0, 1)$
    build_run $(x \cdot b, r'(x \cdot b), w, 0, 0)$

**handle_$C_q$** $(r'(x) = (p', p, d, \alpha), q, w)$
  Let $t_0, \ldots, t_n \in P^+$ be the sequence of
  $\varepsilon$-transitions connecting $p$ to $q$
    $r := r \cdot (w, t_1), \cdots, (w, t_{n-1})$
```

```
**build_run** $(x, r'(x) = (p, q, d, \alpha), w, add_l, add_r)$
  if (advancing($x$))
    $w := tag(x) \cdot w$;
  if ($add_l$)
    $r := r \cdot (w, p)$;
  if ($x$ has one non advancing son $x \cdot a$)
    build_run $(x \cdot a, r'(x \cdot a), w, 1, 0)$
  if ($x$ has two sons $x \cdot a$ and $x \cdot b$)
    build_run $(x \cdot a, r'(x \cdot a), w, 0, 1)$
    build_run $(x \cdot b, r'(x \cdot b), w, 0, 0)$
  if ($x$ has one advancing son $x \cdot a$)
    build_run $(x \cdot a, r'(x \cdot a), w, 1, 1)$
    handle_$C_q$ $(r'(x \cdot a), q, tag(x \cdot a) \cdot w)$
  if ($add_r$)
    $r := r \cdot (w, q)$;
```

Fig. 1. Converting a run of A into a run of P

– $\Sigma = \{0, 1, \bot\} \times (\{\sharp\} \cup \Gamma \cup (S \times \Gamma))$.
  Thus, the letters are pairs consisting of a direction and either a $\sharp$, a tape symbol of M, or a tape symbol of M marked by a state of M.
– $P = F \cup B \cup I \cup \{acc\}$ where F is the set of forward states, B is the set of backward states, and I is the set of states that check that the tree starts from the initial configuration of M. All three sets are defined formally below. The state acc is an accepting sink.
– $F = \{acc\}$.

The transition function δ and the initial state p_0 are described below.

We start with forward mode. In forward mode, every state is flagged by either l or r, signaling whether the next configuration to be checked is the left successor or the right successor of the current configuration. The 2NBP starts by memorizing the current location it is checking and the environment of this location (that is for checking location i, memorize the letters in locations $i-1$, i, and $i+1$). For checking the left (resp. right) successor it continues $f(n) - i$ steps in direction 0 then it progresses one step in direction 0 (resp. 1) and then takes i steps in direction 0. Finally, it checks that the letter it is reading is indeed the $next_l$ (resp. $next_r$) successor of the memorized environment. It then goes $f(n) - 1$ steps back, increases the location that it is currently checking and memorizes the environment of the new location. It continues zigzagging between the two configurations until completing the entire configuration and then it starts checking the next.

Thus, the forward states are $F = \{f\} \times \{l, r\} \times [f(n)] \times V^3 \times [f(n)] \times \{x, v\} \times \{0, 1, \bot\}$. Every state is flagged by f and either r or l (next configuration to be checked is either right or left successor). Then we have the current location $i \in [f(n)]$ we are trying to check, the environment $(\sigma, \sigma', \sigma'') \in V^3$ of this location. Then a counter for advancing $f(n)$ steps. Finally, we have x for *still-checking* and v for *checked* (and going backward to the next letter). We also memorize the direction we went to in order to check that every node is labeled by its direction (thus, we have 0 or 1 for forward moves and $\perp$ for backward moves).

The transition of these states is as follows.

– For $0 \leq i \leq f(n)$ and $0 \leq j < f(n)$ we have
$\delta(\langle f, d, i, \sigma, \sigma', \sigma'', j, x, \varDelta\rangle, \langle \varDelta, \sigma'''\rangle) =$
$$\begin{cases} \{(1, \langle f, d, i, \sigma, \sigma', \sigma'', j+1, x, 1\rangle\} & \text{if } i+j = f(n) \text{ and } d = r \\ \{(0, \langle f, d, i, \sigma, \sigma', \sigma'', j+1, x, 0\rangle\} & \text{otherwise} \end{cases}$$
Continue going forward while increasing the counter. If reached the end of configuration and next configuration is the right configuration go in direction 1. Otherwise go in direction 0.
– For $0 \leq i \leq f(n)$ we have
$\delta(\langle f, d, i, \sigma, \sigma', \sigma'', f(n), x, \varDelta\rangle, \langle \varDelta, \sigma'''\rangle) =$
$$\begin{cases} \emptyset & \text{if } \sigma''' \neq next_d(\sigma, \sigma', \sigma'') \\ \{(\uparrow, \langle f, d, (i+1)_{f(n)}, \sigma', \sigma'', \perp, f(n)-1, v, \perp\rangle)\} & \text{if } \sigma''' = next_d(\sigma, \sigma', \sigma'') \end{cases}$$
If σ''' is not the $next_d$ letter, then abort. Otherwise, change the mode to v and start going back. Push σ' and σ'' to the first two memory locations and empty the third memory location.
– For $0 \leq i \leq f(n)$ and $1 < j \leq f(n)$ we have
$\delta(\langle f, d, i, \sigma, \sigma', \perp, j, v, \perp\rangle, \langle \varDelta, \sigma''\rangle) = \{(\uparrow, \langle f, d, i, \sigma, \sigma', \perp, j-1, v, \perp\rangle)\}$.
Continue going backward while updating the counter.
– For $0 \leq i \leq f(n)$ we have
$\delta(\langle f, d, i, \sigma, \sigma', \perp, 1, v, \perp\rangle, \langle \varDelta, \sigma''\rangle) =$
$$\begin{cases} \{(\uparrow, \langle b_\forall, \perp, x\rangle)\} & \text{if } \sigma'' \in F_a \times \varGamma \\ \{(\uparrow, \langle b_\exists, \perp, x\rangle)\} & \text{if } \sigma'' \in F_r \times \varGamma \\ \{(\epsilon, \langle f, d, i, \sigma, \sigma', \sigma'', 0, x, \perp\rangle)\} & \text{otherwise} \end{cases}$$
Stop going backward. If the configuration that is checked is either accepting or rejecting go to backward mode (recall that the configuration is already verified as the correct successor of the previous configuration). Otherwise memorize the third letter of the environment and initialize the counter to 0.
– $\delta(\langle f, d, 0, \sharp, \sharp, \perp, 0, x, \varDelta\rangle, \langle \varDelta, \sigma\rangle) = \{(0, \langle f, d, 0, \sharp, \sharp, \sigma, 1, x, 0\rangle)\}$
This is the first forward state after backward mode and after the initial phase. It starts checking the first letter of the configuration. The 2NBP already knows that the letter it has to check is $\sharp$, it memorizes the current letter (the third letter of the environment) and moves forward while updating the counter.
Note that also the first letter is marked as $\sharp$, this is because when checking location 0 of a configuration we are only checking that the length of the configuration is $f(n)+1$ and that after $f(n)+1$ locations there is another $\sharp$.

Backward mode (either universal or existential) is again flagged by l or r, signaling whether the last configuration the 2NBP saw was the left or right successor. Backward mode starts in a node labeled by a state of M. As the 2NBP goes backward, whenever it passes a $\sharp$ it memorizes its direction. When the 2NBP gets again to a letter that is marked with a state of M, if the memorized direction is l and the type of the state the 2NBP is reading matches the type of backward mode (universal state of M and backward universal or existential state of M and backward existential) then the 2NBP continues going up until the $\sharp$, then it moves to forward mode again (marked by r). Otherwise (i.e. if the memorized direction is r or the type of the state the 2NBP is reading does not match the type of backward mode) then the 2NBP stays in backward mode, when it passes the next $\sharp$ it memorizes the current direction, and goes on moving backward. When returning to the root in backward existential mode, this means that the 2NBP is trying to find a new pruning tree. As no such pruning tree exists the 2NBP rejects. When returning to the root in backward universal mode, this means that all universal choices of the currently explored pruning tree were checked and found accepting. Thus, the pruning tree is accepting and the 2NBP accepts.

The set of backward states is $B = \{b_\forall, b_\exists\} \times \{l, r, \bot\} \times \{x, v\}$. Every state is flagged by $\forall$ (for universal) or $\exists$ (for existential) and by either l or r (the last configuration seen is left successor or right successor, or $\bot$ for unknown). Finally, every state is flagged by either x or v. A state marked by v means that the 2NBP is about to move to forward mode and that it is just going backward until the $\sharp$.

The transition of backward states is as follows.

– $\delta(\langle b_\forall, d, x\rangle, \langle \varDelta, \sigma\rangle) = \begin{bmatrix} \{(\uparrow, \langle b_\forall, l, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \varDelta = 0 \\ \{(\uparrow, \langle b_\forall, r, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \varDelta = 1 \\ \{(\epsilon, acc)\} & \text{if } \varDelta = \bot \\ \{(\uparrow, \langle b_\forall, l, v\rangle)\} & \text{if } \sigma \in S_u \times \varGamma \text{ and } d = l \\ \{(\uparrow, \langle b_\forall, d, x\rangle)\} & \text{otherwise} \end{bmatrix}$

In backward universal mode reading a $\sharp$ we memorize its direction. If reading the root, we accept. If reading a universal state of M and the last configuration was the left successor then change the x to v. Otherwise, just keep going backward.

– $\delta(\langle b_\exists, d, x\rangle, \langle \varDelta, \sigma\rangle) = \begin{bmatrix} \{(\uparrow, \langle b_\exists, l, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \varDelta = 0 \\ \{(\uparrow, \langle b_\exists, r, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \varDelta = 1 \\ \emptyset & \text{if } \varDelta = \bot \\ \{(\uparrow, \langle b_\exists, l, v\rangle)\} & \text{if } \sigma \in S_e \times \varGamma \text{ and } d = l \\ \{(\uparrow, \langle b_\forall, d, x\rangle)\} & \text{otherwise} \end{bmatrix}$

In backward existential mode reading a $\sharp$ we memorize its direction. If reading the root, we reject. If reading an existential state of M and the last configuration was the left successor then change x to v. Otherwise, just keep going backward.

– $\delta(\langle b, l, v\rangle, \langle \varDelta, \sigma\rangle) = \begin{bmatrix} \{(\uparrow, \langle b, l, v\rangle)\} & \text{if } \sigma \neq \sharp \\ \{(\varepsilon, \langle f, r, 0, \sharp, \sharp, \bot, 0, x, 0\rangle)\} & \text{if } \sigma = \sharp \end{bmatrix}$

In backward mode marked by v we go backward until we read $\sharp$. When reading $\sharp$ we return to forward mode. The next configuration to be checked is the right successor. The location we are checking is location 0, thus the letter before is not interesting and is filled by $\sharp$. The counter is initialized to 0.

Finally, the set I of 'initial' states makes sure that the first configuration in the tree is indeed $\sharp \cdot (s_0, b) \cdot b^{f(n)-1}$. When finished checking the first configuration $\mathcal{S}$ returns to the node 0 and moves to forward mode.

Formally, $I = \{i\} \times [f(n)] \times \{x, v\}$ with transition as follows.

– $\delta(\langle i,0,x\rangle,\langle \Delta,\sigma\rangle) = \begin{bmatrix} \{(0,\langle i,1,x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \Delta = \bot \\ \emptyset & \text{otherwise} \end{bmatrix}$
Make sure that the root is labeled by $\langle \bot, \sharp\rangle$.
– $\delta(\langle i,1,x\rangle,\langle \Delta,\sigma\rangle) = \begin{bmatrix} \{(0,\langle i,2,x\rangle)\} & \text{if } \sigma = (s_0,b) \text{ and } \Delta = 0 \\ \emptyset & \text{otherwise} \end{bmatrix}$
Make sure that the first letter is (s_0, b).
– For $1 < j < f(n)$ we have

$$\delta(\langle i,j,x\rangle,\langle \Delta,\sigma\rangle) = \begin{bmatrix} \{(0,\langle i,j+1,x\rangle)\} & \text{if } \sigma = b \text{ and } \Delta = 0 \\ \emptyset & \text{otherwise} \end{bmatrix}$$

Make sure that all other letters are b.
– $\delta(\langle i,f(n),x\rangle,\langle \Delta,\sigma\rangle) = \begin{bmatrix} \{(\uparrow,\langle i,f(n)-1,v\rangle)\} & \text{if } \sigma = b \text{ and } \Delta = 0 \\ \emptyset & \text{otherwise} \end{bmatrix}$
Make sure that the last letter is b. The first configuration is correct, start going back to node 0. Change x to v.
– For $2 < j < f(n)$ we have $\delta(\langle i,j,v\rangle,\langle \Delta,\sigma\rangle) = \{(\uparrow,\langle i,j-1,v\rangle)\}$
Continue going backward while updating the counter.
– $\delta(\langle i,2,v\rangle,\langle 0,\sigma\rangle) = \{(\uparrow,\langle f,l,0,\sharp,\sharp,\bot,0,x,0\rangle)\}$.
Finished checking the first configuration. Go up to node 0 in the first state of forward mode.

Last but not least the initial state is $p_0 = \langle i,0,x\rangle$.

Finally, we analyze the reduction. Given an alternating Turing machine with n states and alphabet of size m we get a 2NBP with $O(n \cdot m)$ states, that reads an alphabet with $O(n \cdot m)$ letters. The 2NBP is actually deterministic. Clearly, the reduction is polynomial.

We note that instead of checking emptiness of $\mathcal{P}$, we can check the membership of some correct encoding of the run tree of M in the language of $\mathcal{P}$. However, the transducer that generates a correct encoding of M is exponential.

C Lower Bound for Linear Time Model-Checking on Prefix-Recognizable Systems

It was shown by [BEM97] that the problem of model-checking an LTL formula with respect to a pushdown graph is EXPTIME-hard in the size of the formula. The problem is polynomial in the size of the pushdown system inducing the graph. Our algorithm for model-checking an LTL formula with respect to a prefix-recognizable graph is exponential both in the size of the formula and in $|Q_\beta|$.

As prefix-recognizable systems are a generalization of pushdown systems the exponential resulting from the formula cannot be improved. We show that also the exponent resulting from Q_β cannot be removed. We use the EXPTIME-hard problem of whether a linear space alternating Turing machine accepts the empty tape [CKS81]. We reduce this question to the problem of model-checking a fixed LTL formula with respect to the graph induced by a prefix-recognizable system with a constant number of states and transitions. Furthermore Q_α and Q_γ depend only on the alphabet of the Turing machine. The component Q_β does 'all the hard work'. Combining this with Theorem 15 we get the following.

Theorem 26. *The problem of linear-time model-checking the graph induced by the prefix-recognizable system $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0\rangle$ is EXPTIME-complete in $|Q_\beta|$.*

Proof: Let $M = \langle \Gamma, S_u, S_e, \mapsto, s_0, F_{acc}, F_{rej}\rangle$ be an alternating linear-space Turing machine. Let $f : \mathbb{N} \to \mathbb{N}$ be the linear function such that M uses $f(n)$ cells in its working tape in order to process an input of length n. In order to make sure that M does not accept the empty tape, we have to check that every legal pruning of the computation tree of M contains one rejecting branch.

Given such an alternating linear-space Turing machine M, we construct a prefix-recognizable system R and an LTL formula φ such that $G_R \models \varphi$ iff M does not accept the empty tape. The system R has a constant number of states and rewrite rules. For every rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q'\rangle$, the languages of the regular expressions α_i and γ_i are subsets of $\Gamma \cup (\{\downarrow\} \times \Gamma) \cup S \cup \{\epsilon\}$. The language of the regular expression β_i, can be encoded by a nondeterministic automaton whose size is linear in n. The LTL formula φ does not depend on the structure of M.

The graph induced by R has one infinite trace. This trace searches for rejecting configurations in all the pruning trees. The trace first explores the left son of every configuration. If it reaches an accepting configuration, the trace backtracks until it reaches a universal configuration for which only the left son was explored. It then goes forward again and explores under the right son of the universal configuration. If the trace returns to the root without finding such a configuration then the currently explored pruning tree is accepting. Once a rejecting configuration is reached, the trace backtracks until it reaches an existential configuration for which only the left son was explored. It then explores under the right son of the existential configuration. In this mode, if the trace backtracks all the way to the root, it means that all pruning trees were checked and that there is no accepting pruning tree for M.

We change slightly the encoding of a configuration by including with the state of M a symbol l or r denoting whether the next explored configuration is the right or left successor. Let $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma \times \{l, r\})$ and let $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)} \cdot \sharp\sigma_1^d \ldots \sigma_{f(n)}^d$ be a configuration of M and its d-successor (where d is either l or r). We also set σ_0 and σ_0^d to $\sharp$. Given σ_{i-1}, σ_i, and σ_{i+1} we know, by the transition relation of M, what σ_i^d should be. In addition the symbol $\sharp$ should repeat exactly every $f(n) + 1$ letters. Let $next : V^3 \to V$ denote our expectation for σ_i^d. Note that whenever the triplet σ_{i-1}, σ_i, and σ_{i+1} does not include the reading head of the Turing machine, it does not matter whether d is l or r. In both cases the expectation for σ_i^d is the same. We set $next(\sigma, \sharp, \sigma') = \sharp$, and

$next(\sigma, \sigma', \sigma'') =$

$$\begin{cases} \sigma' & \text{if } \{\sigma, \sigma', \sigma''\} \subseteq \{\sharp\} \cup \Gamma \\ \sigma' & \text{if } \sigma'' = (s, \gamma, d) \text{ and } (s, \gamma) \to^d (s', \gamma', R) \\ (s', \sigma', d') & \text{if } \sigma'' = (s, \gamma, d),\ (s, \gamma) \to^d (s', \gamma', L), \text{ and } d' \in \{l, r\} \\ \sigma' & \text{if } \sigma = (s, \gamma, d) \text{ and } (s, \gamma) \to^d (s', \gamma', L) \\ (s', \sigma', d') & \text{if } \sigma = (s, \gamma, d),\ (s, \gamma) \to^d (s', \gamma', R), \text{ and } d' \in \{l, r\} \\ \gamma' & \text{if } \sigma' = (s, \gamma, d) \text{ and } (s, \gamma) \to^d (s', \gamma', \alpha) \end{cases}$$

Consistency with $next$ now gives us a necessary condition for a sequence in V^* to encode a branch in the computation tree of M. Note that when $next(\sigma, \sigma', \sigma'') \in S \times \Gamma \times \{l, r\}$ then marking it by both l and r is correct.

The prefix-recognizable system starts from the initial configuration of M. It has two main modes, a *forward* mode and a *backward* mode. In forward mode, the system guesses a new configuration. The configuration is guessed one letter at a time, and this letter should match the functions $next_l$ or $next_r$. If the computation reaches an accepting configuration, this means that the currently explored pruning tree might still be accepting. The system moves to backward mode and remembers that it should explore other universal branches until it finds a rejecting state. In backward universal mode, the system starts backtracking and removes configurations. Once it reaches a universal configuration that is marked by l, it replaces the mark by r, moves to forward mode, and explores the right son. If the root is reached (in backward universal mode), the computation enters a rejecting sink. If in forward mode, the system reaches a rejecting configuration, then the currently explored pruning tree is rejecting. The system moves to backward mode and remembers that it has to explore existential branches that were not explored. Hence, in backward existential mode, the system starts backtracking and removes configurations. Once it reaches an existential configuration that is marked by l, the mark is changed to r and the system returns to forward mode. If the root is reached (in backward existential mode) all pruning trees have been explored and found to be rejecting. Then the system enters an accepting sink. All that the LTL formula has to check is that there exists an infinite computation of the system and that it reaches the accepting sink. Note that the prefix-recognizable system accepts, when the alternating Turing machine rejects and vice versa.

More formally, the LTL formula is $\Diamond reject$ and the rewrite system is $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0 \rangle$, where

- $AP = \{reject\}$
- $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma \times \{l, r\})$
- $Q = \{forward, backward_\exists, backward_\forall, sink_a, sink_r\}$
- $L(q, \alpha) = \begin{bmatrix} \emptyset & \text{if } q \neq sink_a \\ \{reject\} & \text{if } q = sink_a \end{bmatrix}$
- $q_0 = forward$
- $x_0 = b \cdots b \cdot (s_0, b, l) \cdot \sharp$

In order to define the transition relation we use the following languages.

- $L^1_{egal} = \left\{ next(\sigma, \sigma', \sigma'') \cdot V^{f(n)-1} \sigma \cdot \sigma' \cdot \sigma'' \right\}$
  $L^2_{egal} = \left\{ w \in V^{f(n)+1} \mid w \notin V^* \cdot \sharp \cdot V^* \cdot \sharp \cdot V^* \right\}$
  $L^3_{egal} = \left\{ w \in V^{f(n)+1} \mid w \notin V^* \cdot (S \times \Gamma \times \{l, r\}) \cdot V^* \cdot (S \times \Gamma \times \{l, r\}) \cdot V^* \right\}$
  $L_{egal} = (L^1_{egal} \cap L^2_{egal} \cap L^3_{egal}) \cdot V^*$
  Thus, this language contains all words whose suffix of length $f(n) + 1$ contains at most one $\sharp$ and at most one symbol from $S \times \Gamma \times \{l, r\}$ and the last letter is the $next$ correct successor of the previous configuration.
- $A_{ccept} = V \cdot (\{F_{acc}\} \times \Gamma \times \{l, r\}) \cdot V^*$

Thus, this language contains all words whose one before last letter is marked by an accepting state[17].

– $R_{eject} = V \cdot (\{F_{rej}\} \times \Gamma \times \{l, r\}) \cdot V^*$
  Thus, this language contains all words whose one before last letter is marked by a rejecting state.
– $R_{emove}^{S_u \times \{l\}} = V \setminus (S_u \times \Gamma \times \{l\})$
  Thus, this language contains all the letters that are not marked by universal states and the direction l.
– $R_{emove}^{S_e \times \{l\}} = V \setminus (S_e \times \Gamma \times \{l\})$.
  Thus, this language contains all the letters that are not marked by existential states and the direction l.

Clearly the languages L_{egal}, A_{ccept}, and R_{eject} can be accepted by nondeterministic automata whose size is linear in $f(n)$.

The transition relation includes the following rewrite rules:

1. $\langle forward, \{\epsilon\}, L_{egal}, V \setminus (S \times \Gamma \times \{r\}), forward\rangle$ - guess a new letter and put it on the store. States are guessed only with direction l. The fact that L_{egal} is used ensures that the currently guessed configuration (and in particular the previously guessed letter) is the successor of the previous configuration on the store.
2. $\langle forward, \{\epsilon\}, A_{ccept}, \{\epsilon\}, backward_\forall\rangle$ - reached an accepting configuration. Do not change the store and move to backward universal mode.
3. $\langle forward, \{\epsilon\}, R_{eject}, \{\epsilon\}, backward_\exists\rangle$ - reached a rejecting configuration. Do not change the store and move to backward existential mode.
4. $\langle backward_\forall, R_{emove}^{S_u \times \{l\}}, V^*, \{\epsilon\}, backward_\forall\rangle$ - remove one letter that is not in $S_u \times \Gamma \times \{l\}$ from the store.
5. $\langle backward_\forall, S_u \times \Gamma \times \{l\}, V^*, S_u \times \Gamma \times \{r\}, forward\rangle$ - replace the marking l by the marking r and move to forward mode. The state s does not change[18].
6. $\langle backward_\forall, \epsilon, \epsilon, \epsilon, sink_r\rangle$ - when the root is reached in backward universal mode enter the rejecting sink
7. $\langle backward_\exists, R_{emove}^{S_e \times \{l\}}, V^*, \{\epsilon\}, backward_\exists\rangle$ - remove one letter that is not in $S_e \times \Gamma \times \{l\}$ from the store.
8. $\langle backward_\exists, S_e \times \Gamma \times \{l\}, V^*, S_e \times \Gamma \times \{r\}, forward\rangle$ - replace the marking l by the marking r and move to forward mode. The state s does not change.
9. $\langle backward_\exists, \epsilon, \epsilon, \epsilon, sink_a\rangle$ - when the root is reached in backward existential mode enter the accepting sink.
10. $\langle sink_a, \epsilon, \epsilon, \epsilon, sink_a\rangle$ - remain in accepting sink.
11. $\langle sink_r, \epsilon, \epsilon, \epsilon, sink_r\rangle$ - remain in rejecting sink.

□

[17] It is important to use the one before last letter so that the state itself is already checked to be the correct next successor of previous configuration.

[18] Actually, we guess all states in S_u. As we change state into $forward$, the next transition verifies that indeed the state is the same state.