Propositional Dynamic Logic with Recursive Programs

Christof Loeding, Carsten Lutz, Olivier Serre

Propositional Dynamic Logic with Recursive Programs ⋆

Christof Löding

RWTH Aachen, Germany

Carsten Lutz

Dresden University of Technology, Germany

Olivier Serre ⋆⋆

LIAFA,CNRS & Université Paris VII, France

Abstract

We extend the propositional dynamic logic PDL of Fischer and Ladner with a restricted kind of recursive programs using the formalism of visibly pushdown automata (Alur, Madhusudan 2004). We show that the satisfiability problem for this extension remains decidable, generalising known decidability results for extensions of PDL by non-regular programs. Our decision procedure establishes a 2-ExpTime upper complexity bound, and we prove a matching lower bound that applies already to rather weak extensions of PDL with non-regular programs. Thus, we also show that such extensions tend to be more complex than standard PDL.

Key words: Propositional Dynamic Logic, Visibly Pushdown Automata

1 Introduction

Propositional Dynamic Logic (PDL) is a modal logic that was introduced by Fischer and Ladner in [6] to capture the behaviour of programs, see also

⋆ This research has been partially supported by the European Community Research Training Network "Games and Automata for Synthesis and Validation" (Games).
⋆⋆Most of this work was done when the author was a postdoctoral researcher at RWTH Aachen.

the surveys [9,14] and the monograph [10]. The models for PDL formulas are transition systems whose edges are labelled with atomic programs and whose states are labelled with atomic propositions. Formulas and programs are inductively (and mutually) defined from atomic propositions and programs. Formulas are closed by the standard Boolean operations, and for each program α and each formula φ, $\langle\alpha\rangle\varphi$ is a formula meaning that there is an execution of program α that ends in a state where φ holds. A program is a regular language (represented by a regular expression or a finite automaton) over the set of atomic programs and tests (which correspond to formulas).

While PDL is a suitable tool to specify properties of finite state programs, it quickly reaches its limits when considering recursive programs. A simple example for such a program (taken from [10]) is:

```
proc V { if p then  {a; call V; b} else return }
```

The executions of this program can be described by the set $\{(p?a)^i(\neg p)?b^i \mid i \geq 0\}$, which is not regular and, as implied by results in [8], thus cannot be captured by standard PDL.

To overcome this weakness, in the 1980ies an investigation of non-regular versions of PDL was initiated. Besides regular programs, these versions also include context-free ones, or even go beyond that. This investigation has shown decidability of PDL with context-free languages to be much more fragile than decidability of standard PDL. It is rather easy to observe that allowing all context-free languages as programs leads to undecidability because one can easily encode the equivalence problem for context-free languages in terms of satisfiability.

In [11] it is shown that even adding only a single context-free program can lead to undecidability: the logic PDL+$a^\#ba^\#$ is undecidable, where PDL+L stands for the extension of PDL which allows only the language L as single new program, and $a^\#ba^\#$ is a short notation for the language $\{a^nba^n \mid n \in \mathbb{N}\}$. A further result from the same article shows that adding the two languages $a^\#b^\#$ and $b^\#a^\#$ to PDL also leads to undecidability. In view of these results, it is rather surprising that PDL+$a^\#b^\#$ is decidable, as shown in [13]. These results raise the question of the exact borderline between decidable and undecidable extensions of PDL with context-free (or even more general) programs.

A step into this direction was made in [12], where a whole class of context-free languages is identified which can be (simultaneously) added as programs to PDL without losing decidability. This class consists of languages that can be accepted by a rather restricted kind of pushdown automaton, called simple minded. The behaviour of such automata is completely determined by the current input symbol, i.e., the input symbol determines the next control state, the stack operation to be performed, and the symbol to be pushed in

case of a push operation. This result has been generalised in [7] to a larger class of pushdown automata, called semi-simple minded, where only the stack operation and the symbol to be pushed is determined by the input symbol, but the next control state is not. The resulting class of languages covers all context-free languages that are known to retain decidability when added as a program to PDL.

The contribution of the current paper is to advance the study of PDL with context-free programs in two directions. First, we follow the research agenda set out by Harel and others in [11,12,7], who propose to identify as large as possible classes of context-free languages that can be used in PDL without losing decidability. Specifically, we consider the class of visibly pushdown languages [3], i.e., languages that can be accepted by a visibly pushdown automaton (VPA). These automata generalise semi-simple minded automata in that only the operation to be performed is determined by the input symbol, but (the next control state and) the symbol to be pushed is not. For example, the language $\{a^n a^m b_1^m a^\ell b_2^\ell b_1^n \mid n, m, \ell \geq 0\}$ can be accepted by a VPA, but not by a semi-simple minded automaton.

We define a version of PDL that we call *recursive PDL*, and which includes all visibly pushdown languages as programs. Recursive PDL is also more general than existing extensions of PDL with non-regular programs in that it allows the use of test operators inside context-free programs. Our main result is that satisfiability in recursive PDL is decidable in 2-EXPTIME, and that this upper bound is preserved when further extending recursive PDL with a Δ operator which allows to describe infinite computations.

Our second contribution is to analyse the exact computational complexity of PDL with context-free languages. We consider a specific class of parenthesis languages and show that for every language L from this class, the extension PDL+L with the single language L is 2-EXPTIME-hard. This result covers many natural languages such as $a^\# b^\#$ and several of its variations. It is also easy to adapt our lower bound to some non-context free languages such as $a^\# b^\# c^\#$. Since $a^\# b^\#$ can be accepted by a VPA, we also obtain 2-EXPTIME-completeness of recursive PDL.

The remainder of the paper is organised as follows. In Section 2, we give basic definitions and results regarding PDL and visibly pushdown automata, and we define recursive PDL. In Section 3, we discuss the relation of recursive PDL to previously defined extensions of PDL and other logics that allow to capture the behaviour of recursive programs. We also give some examples of useful properties that can be expressed in recursive PDL. Decidability of satisfiability in recursive PDL is shown in Section 4, and Section 5 deals with the lower bound for the complexity. Finally, we extend the results to infinite computations in Section 6.

2 Definitions

In this section, we first define propositional dynamic logic (PDL) using regular programs. Then we introduce visibly pushdown automata and use them to extend PDL with more powerful programs.

2.1 Propositional Dynamic Logic

Formulas of propositional dynamic logic [6] are interpreted over transition systems whose edges are labelled with atomic programs or actions and whose states are labelled with atomic propositions. Hence, we fix a set $\mathbb{P}$ of *atomic propositions* and a set Π of *atomic programs*. The set of *formulas* and the set of *programs* are defined inductively as follows:

(1) $\top$ is a formula.
(2) Every atomic proposition is a formula.
(3) If φ_1 and φ_2 are formulas, then so are $\neg\varphi_1$ and $\varphi_1 \wedge \varphi_2$.
(4) If φ is a formula, then $\varphi?$ is a *test*. The set of tests is denoted by Test.
(5) If α is a program and φ is a formula, then $\langle\alpha\rangle\varphi$ is a formula. Such a formula will be called a *diamond formula*. The negation of a diamond formula will be called a *box formula*. The standard abbreviation for such formulas is $[\alpha]\varphi \equiv \neg\langle\alpha\rangle\neg\varphi$.
(6) A regular expression over $\Pi \cup \text{Test}$ is a program.

In this definition, we refer to standard regular expressions α built from single letters using concatenation, union, and Kleene-star. By $L(\alpha)$ we denote the set of words defined by the regular expression α.

PDL formulas are interpreted over structures $M = (S, R, \nu)$ where S is a set of states, $R : \Pi \to 2^{S\times S}$ is a transition relation, and $\nu : S \to 2^{\mathbb{P}}$ assigns truth values to each atomic proposition in $\mathbb{P}$ for each state in S. In the following, we extend the relation R to all programs and tests, and in parallel define when a formula φ is satisfied in a state s of the structure M, denoted as usual by $M, s \models \varphi$:

- $R(\varphi?) = \{(s, s) \mid M, s \models \varphi\}$ for a test $\varphi?$.
- $R(\alpha)$ for a program α contains the pairs (s, s') for which there are
  - a word $w = w_1 \cdots w_m \in L(\alpha)$ (with $w_i \in \Pi \cup \text{Test}$), and
  - states $s_0, \ldots, s_m \in S$ with $s = s_0$, $s' = s_m$, and $(s_{i-1}, s_i) \in R(w_i)$ for all $1 \leq i \leq m$.
- $M, s \models \varphi_1 \wedge \varphi_2$ if $M, s \models \varphi_1$ and $M, s \models \varphi_2$.
- $M, s \models \neg\varphi$ if $M, s \models \varphi$ does not hold.
- $M, s \models \langle\alpha\rangle\varphi$ if there exists a state s' such that $(s, s') \in R(\alpha)$ and $M, s' \models \varphi$.

A formula φ is *satisfiable* if there is a structure M and a state s such that $M, s \models \varphi$. The *satisfiability problem* is to determine, given a formula φ, whether it is satisfiable.

To show decidability of the satisfiability problem we use tree structures as defined in the following. Let $\Pi = \{a_0, \ldots, a_{n-1}\}$ be a finite set of atomic programs. A *tree structure* for Π is a structure $M = (S, R, \nu)$ such that for some $k \in \mathbb{N}$

- $S \subseteq [k]^*$ is a non-empty, prefix closed set (with $[k] = \{0, \ldots, k-1\}$), and
- $R(a_\ell) = \{(x, xd) \in S \times S \mid x \in [k]^* \text{ and } \ell = d \mod n\}$.

For $x \in [k]^*$ and $d \in [k]$ we call xd the d-successor of x. The second item in the above definition simply states that the relations for the atomic programs are obtained by taking the number of the successor modulo n.

2.2 Visibly Pushdown Automata

In the following, we introduce a subclass of pushdown automata and consider the logic obtained when replacing regular expressions by this kind of automata for defining programs in PDL.

A pushdown automaton is called visibly pushdown automaton [3], if the type of operation that is performed on the stack, i.e. push, skip, or pop, only depends on the input symbol. For such an automaton one can partition the input alphabet into three sets, consisting of the symbols that induce a push, a skip, or a pop, respectively. In [2] these automata are used to solve verification problems for recursive state machines. In this setting pushes correspond to calls of procedures, skips correspond to internal actions, and pops correspond to returns from procedures. This is where the notation used in the following arises from.

A *pushdown alphabet* is a tuple $\widetilde{A} = \langle A_c, A_r, A_{\text{int}} \rangle$ consisting of three disjoint finite alphabets that can be interpreted as a finite set of *calls* (A_c), a finite set of *returns* (A_r), and a finite set of *internal actions* (A_{int}). For any such $\widetilde{A}$, let $A = A_c \cup A_r \cup A_{\text{int}}$.

A *visibly pushdown automaton* (VPA) over $\widetilde{A}$ is a tuple $\mathcal{A} = (Q, \Gamma, Q_{\text{in}}, \delta, F)$ where Q is a finite set of states, $Q_{\text{in}} \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, Γ is a finite stack alphabet that contains a special *bottom-of-stack symbol* $\triangleleft$ and $\delta \subseteq (Q \times A_c \times Q \times (\Gamma \setminus \{\triangleleft\})) \cup (Q \times A_r \times \Gamma \times Q) \cup (Q \times A_{\text{int}} \times Q)$ is the transition relation.

A *configuration* of $\mathcal{A}$ is a pair $(q, \sigma) \in Q \times (\Gamma \setminus \{\triangleleft\})^* \triangleleft$ of a state q and a

stack content σ. Note that the symbol $\triangleleft$ may only appear at the bottom of the stack. We denote the set of all configurations of $\mathcal{A}$ by $Cf(\mathcal{A})$.

For a letter $a \in A$, a configuration (q', σ') is an *a-successor* of (q, σ), denoted by $(q, \sigma) \xrightarrow{a} (q', \sigma')$, if one of the following holds:

- For $a \in A_c$, $\sigma' = \gamma\sigma$ and there is a transition $(q, a, q', \gamma) \in \delta$.
- For $a \in A_{\text{int}}$, $\sigma' = \sigma$ and there is a transition $(q, a, q') \in \delta$.
- For $a \in A_r$, either $\sigma = \gamma\sigma'$ and there is a transition $(q, a, \gamma, q') \in \delta$, or $\sigma = \sigma' = \triangleleft$ and there is a transition $(q, a, \triangleleft, q') \in \delta$.

For a finite word $u = u_0 u_1 \cdots u_n$ in A^* (with $u_i \in A$), a *run* of $\mathcal{A}$ on u is a sequence $\rho = (q_0, \sigma_0)(q_1, \sigma_1) \cdots (q_{n+1}, \sigma_{n+1})$ of configurations with $q_0 \in Q_{in}$, $\sigma_0 = \triangleleft$, and $(q_i, \sigma_i) \xrightarrow{u_i} (q_{i+1}, \sigma_{i+1})$ for every $i \in \{1, \ldots, n\}$. In this situation we also write $(q_0, \sigma_0) \xrightarrow{u} (q_{n+1}, \sigma_{n+1})$.

A word $u \in A^*$ is *accepted* by $\mathcal{A}$ if there is a run of $\mathcal{A}$ on u that ends in a configuration (q, σ) with $q \in F$. The *language* $L(\mathcal{A})$ of a VPA $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$.

Note that allowing VPAs to read return symbols on the empty stack enables VPAs to accept all regular languages. We also point out that it is not a restriction that VPAs do not consider the top stack symbol on internal or call symbols. This can easily be simulated by storing the current top stack symbol in the control state.

As usual, we call a VPA *complete* if for each configuration (q, σ) and each input symbol a there is at least one a-successor of (q, σ). A VPA is *deterministic* if it has a unique initial state q_{in}, and for each input letter and configuration there is at most one successor configuration. For deterministic VPAs we write $\delta(q, a) = (q', \gamma)$ instead of $(q, a, q', \gamma) \in \delta$ if $a \in A_c$, $\delta(q, a, \gamma) = q'$ instead of $(q, a, \gamma, q') \in \delta$ if $a \in A_r$, and $\delta(q, a) = q'$ instead of $(q, a, q') \in \delta$ if $a \in A_{\text{int}}$.

One can easily show that visibly pushdown languages are closed under union and intersection using ordinary product constructions. The closure under complement follows from a more complicated construction for determinisation.

Theorem 1 ([19,3]) *For each* VPA*, there is an equivalent deterministic* VPA *of exponential size.*

We need two extensions of VPAs: to infinite words and to infinite trees. For nondeterministic automata, the extension to infinite words is straightforward ([3]). A run on an infinite input word is a sequence of configurations that satisfies the conditions as given in the definition of runs on finite words. The set F of final states is now interpreted as a set of Büchi states, i.e., an infinite run is accepting if it infinitely often visits configurations with a state from F.

We call such automata *nondeterministic Büchi* VPA*s*. If we do not want to explicitly specify the acceptance condition of a VPA on infinite words, then we call it an ω-VPA.

For deterministic automata, the situation is a bit different. In [3] it is shown that the standard acceptance conditions do not suffice to obtain a deterministic model that is as expressive as nondeterministic Büchi VPAs. We can avoid this problem if we evaluate the acceptance condition on a certain subsequence of the run [15]. This leads to the model of stair parity VPAs.

A *stair parity* VPA over $\widetilde{A}$ is of the form $\mathcal{A} = (Q, \Gamma, Q_{\text{in}}, \delta, \Omega)$ where Q, Γ, Q_{in} and δ are as in VPAs and $\Omega : Q \to \mathbb{N}$ is a priority function. To define acceptance for stair parity VPAs we first have to filter out the relevant positions in a run. Let $\rho = (q_0, \sigma_0)(q_1, \sigma_1) \cdots$ be an infinite run of $\mathcal{A}$. For $i \in \mathbb{N}$ we call (q_i, σ_i) a *step* of ρ if in all successive positions the height of stack does not go below the height of σ_i, i.e., $|\sigma_j| \geq |\sigma_i|$ for all $j \geq i$.

Note that, since the height of the stack at each position solely depends on the input, the set of positions of the steps is the same for different runs on the same input word.

The run $\rho = (q_0, \sigma_0)(q_1, \sigma_1) \cdots$ is accepting if the maximal priority that occurs infinitely often on a step is even, i.e., if

$$\max\{\Omega(q) \mid q = q_i \text{ for infinitely many steps } (q_i, \sigma_i) \text{ of } \rho\}$$

is even. The definition of deterministic stair parity VPA is directly adapted from the definition of deterministic VPA.

Theorem 2 ([15]) *For each nondeterministic Büchi* VPA *there exists a deterministic stair parity* VPA *recognising the same language.*

We also need two very simple acceptance conditions for VPAs on infinite words: reachability and safety. Both conditions are specified by a set F of states. A run of a *reachability* VPA is accepting if some state from F occurs in this run. Dually, a run of a *safety* VPA is accepting if no state from F occurs in the run. Obviously, a complete deterministic safety VPA accepts the complement of the language accepted by the same automaton viewed as a reachability VPA.

If a reachability VPA $\mathcal{A}$ is complete, then the accepted language is of the form $L \cdot A^\omega$ for the language L accepted by $\mathcal{A}$ viewed as a VPA on finite words. Hence, we obtain the following corollary to Theorem 1.

Corollary 3 *For each complete nondeterministic reachability* VPA *there exists an equivalent deterministic reachability* VPA *of exponential size.*

To define visibly pushdown tree automata, we consider *infinite k-ary Σ-labelled trees*, i.e., mappings $t : [k]^* \to \Sigma$. By $\mathcal{T}_{k,\Sigma}$ we denote the set of all infinite k-ary Σ-labelled trees.

The setting on trees that we need is slightly different from the word case: the stack operation performed in a transition of a tree automaton is not determined by the node label but by the direction in the tree. Hence, we assume that $A = [k]$.

A *visibly pushdown tree automaton* (VPTA) over $\widetilde{A}$ (with $A = [k]$) is of the form $\mathcal{A} = (Q, \Sigma, \Gamma, Q_{\text{in}}, \delta, Acc)$ where Q, Γ, Q_{in} are as for VPAs on words, Σ is a node label alphabet, Acc is the acceptance component, and δ is the set of transitions. A transition is of the form (q, b, γ, τ) with $q \in Q$, $b \in \Sigma$, $\gamma \in \Gamma$, and $\tau : [k] \to Q \cup (Q \times \Gamma)$ such that $\tau(d) \in Q$ if $d \in A_{\text{int}} \cup A_r$ and $\tau(d) \in Q \times \Gamma$ if $d \in A_c$. A configuration of $\mathcal{A}$ is defined as before.

For a tree $t : [k]^* \to \Sigma$, a run of $\mathcal{A}$ on t is a mapping $\rho : [k]^* \to Cf(\mathcal{A})$ such that $\rho(\varepsilon) \in Q_{\text{in}} \times \{\triangleleft\}$ is an initial configuration, and for each $x \in [k]^*$ with $\rho(x) = (q, \gamma\sigma)$ there is a transition $(q, t(x), \gamma, \tau) \in \delta$ such that for all $d \in A$:

$$\rho(xd) = \begin{cases} (q', \gamma\sigma) & \text{if } d \in A_{\text{int}} \text{ and } \tau(d) = q', \\ (q', \sigma) & \text{if } d \in A_r,\ \tau(d) = q',\ \text{and } \gamma \in \Gamma \setminus \{\triangleleft\}, \\ (q', \triangleleft) & \text{if } d \in A_r,\ \tau(d) = q', \sigma = \varepsilon,\ \text{and } \gamma = \triangleleft, \\ (q', \gamma'\gamma\sigma) & \text{if } d \in A_c \text{ and } \tau(d) = (q', \gamma'). \end{cases}$$

Intuitively, if the automaton is at a certain node of the input tree, it reads the label of the node and then sends copies of itself to all the successors of the node. Depending on the type of the successor (call, return, or internal action) the automaton performs a push, a pop, or leaves the stack unchanged. Note that, in contrast to VPAs, VPTAs have access to the top stack symbol even for internal and call symbols. We use this definition only for notational convenience. As mentioned above, this does not make any difference for the expressive power.

As for VPAs, we can consider different types of acceptance for VPTAs, e.g., Büchi, parity, or stair parity conditions with the corresponding component substituted for Acc. Then $\mathcal{A}$ accepts an input tree t if there is a run of $\mathcal{A}$ on t such that each path through this run (which is an infinite sequence of configurations) satisfies the acceptance condition. The set of all trees accepted by $\mathcal{A}$ is denoted by $\mathcal{T}(\mathcal{A})$.

Similar to the case of finite automata on infinite trees (cf. [21]), the emptiness test for a VPTA is polynomial time equivalent to the problem of determining the winner in a visibly pushdown game ([15]) with a winning condition corre-

sponding to the acceptance condition of the VPTA. Since solving such games is complete for exponential time (for all the winning conditions considered here), we obtain the following theorem (and also corresponding lower bounds).

Theorem 4 *For a given* VPTA $\mathcal{A}$ *one can decide in exponential time whether* $\mathcal{T}(\mathcal{A})$ *is empty.*

For later use, we need to relate VPAs on words and on trees. For this purpose, we code paths through k-ary Σ-labelled trees by words that can be processed by a VPA.

An infinite path can be uniquely identified with an infinite sequence $d_0 d_1 d_2 \cdots$ with $d_i \in [k]$. Given such a path π and a tree $t : [k]^* \to \Sigma$, we define the infinite word $w_\pi^t \in (\Sigma \times [k])^\omega$ as $w_\pi^t = (t(\varepsilon), d_0)(t(d_0), d_1)(t(d_0 d_1), d_2) \cdots$

The partition of the alphabet $\Sigma \times [k]$ into calls, returns, and internal actions is inherited from the partition of $[k]$.

For a language $L \subseteq (\Sigma \times [k])^\omega$ we define the corresponding language of trees that contains exactly those trees for which all codings of paths are in L:

$$\text{Trees}(L) = \{t \in \mathcal{T}_{k,\Sigma} \mid w_\pi^t \in L \text{ for all paths } \pi\}.$$

If L is accepted by some deterministic ω-VPA $\mathcal{A}$, then one can easily define a VPTA accepting Trees(L) by simulating $\mathcal{A}$ on each path.

Remark 5 For each deterministic ω-VPA $\mathcal{A}$ over $\Sigma \times [k]$ there exists a VPTA with an acceptance condition of the same type accepting Trees($L(\mathcal{A})$).

A class of automata (nested tree automata) similar to our VPTAs has been defined in [4]. These automata are not equipped with a stack but the input tree is enriched by edges connecting a call to its matching return, and the automaton can look backwards along these edges to determine in which state it was when being at the call matching with the current return. Nevertheless, the two models are equivalent in the sense that a set of trees can be accepted by a VPTA iff the corresponding set of trees enriched with the matching edges can be accepted by a nested tree automaton.

2.3 Recursive PDL

The formalism of recursive PDL is obtained by replacing regular expressions (as the formalism to define programs) by VPAs. For this purpose we assume that the set of atomic programs is given as a pushdown alphabet $\widetilde{\Pi} = \langle \Pi_c, \Pi_{int}, \Pi_r \rangle$ of calls Π_c, internal actions Π_{int}, and returns Π_r as required

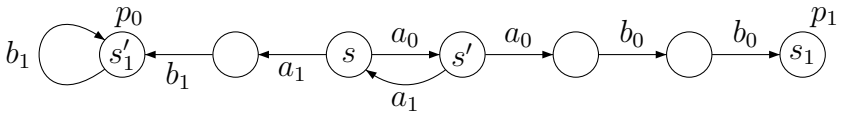

Fig. 1. A model (with s as initial state) for the formula from Example 6

for VPAs. The set of formulas of recursive PDL is defined in the same way as for PDL. To define the set of programs we replace (6) from the syntax definition of PDL by

(6′) A VPA $\mathcal{A}$ over $\langle \Pi_c, \Pi_{int} \cup \text{Test}, \Pi_r \rangle$ is a program.

So we replace regular expressions or finite automata by VPAs, where tests are treated as internal actions. Note that an atomic program a may be seen as a singleton $\{a\}$ and thus as a visibly pushdown language. Therefore, we will always assume that all diamond formulas are of the form $\langle \mathcal{A} \rangle \varphi$ for some VPA $\mathcal{A}$.

The definition of the semantics does not change. The only difference is that in the extension of the relation R to programs we now refer to the language defined by VPAs instead of regular expressions.

Example 6 Consider the set of atomic programs $\widetilde{\Pi} = \langle \{a_0, a_1\}, \emptyset, \{b_0, b_1\} \rangle$ and the set $\mathbb{P} = \{p_0, p_1\}$ of atomic propositions. Let

- $\psi = \langle \mathcal{B} \rangle p_0$ where $\mathcal{B}$ accepts the language $\{a_1^k b_1^k \mid k > 0\}$, and
- $\varphi = \langle \mathcal{A} \rangle p_1$ where $\mathcal{A}$ accepts the language $\{((\psi?)a_0)^k b_0^k \mid k > 0\}$.

For the structure M, as depicted in Figure 1 with $p_1 \in \nu(s_1)$ and $p_0 \in \nu(s'_1)$, we have $(s, s'_1) \in R(\mathcal{B})$ and $(s', s'_1) \in R(\mathcal{B})$. Since $p_0 \in \nu(s'_1)$, we obtain $M, s \models \psi$ and $M, s' \models \psi$. Thus, $(s, s), (s', s') \in R(\psi?)$ and therefore $(s, s_1) \in R(\mathcal{A})$. Since $p_1 \in \nu(s_1)$ we finally obtain that $M, s \models \varphi$.

3 Expressiveness

In this section we compare recursive PDL to other formalisms and give some more examples. We discuss what properties can be expressed in our logic and where the differences to other formalisms are.

3.1 Comparison to previous extensions of PDL

Since VPAs can accept all regular languages, recursive PDL contains standard PDL as a fragment. In [8], there is a simple proof of the fact that the extension of PDL with any single non-regular program is more expressive than standard PDL. Hence, recursive PDL is more expressive than standard PDL.

The introduction already gave a brief discussion of the relationship between recursive PDL, PDL with simple minded pushdown automata (SM-PDL) [12], and PDL with semi-simple minded pushdown automata (SSM-PDL) [10]. In the following, we add some details. The main observation about simple minded automata is that they are very weak. For example, the language $a^{2\#}b^{2\#}$ cannot be accepted by such an automaton because the next state is determined by the input, and thus states cannot be used to count modulo 2. In fact, for a fixed input alphabet there are only finitely many such automata because the number of (useful) states is bounded by the size of the alphabet.

Semi-simple minded automata are less restricted. In fact, they can be seen as VPAs where the stack can only be used to store the input call symbols. We have already mentioned that the language $\{a^n a^m b_1^m a^\ell b_2^\ell b_1^n \mid n, m, \ell \geq 0\}$ can be accepted by a VPA, but not by a semi-simple minded automaton. A proof of the latter is straightforward using the facts that the symbol a needs to be associated with stack pushes, b_1 and b_2 need to be associated with pops from the stack, and a semi-simple minded automaton with only one push symbol can only use a single stack symbol (which does not suffice for this language). Unfortunately, the simple proof of [8] that any extension of PDL with a non-regular language is more expressive than PDL cannot easily be adapted to show that PDL with VPAs is more expressive than SSM-PDL. Still, it seems that there is no formula in SSM-PDL that is equivalent to $\langle \mathcal{A} \rangle p$, where $\mathcal{A}$ is a VPA accepting the above language. We do not, however, have a formal proof of this statement.

Another notable difference between recursive PDL and both SM-PDL and SSM-PDL as defined in [12,10] is that we allow to incorporate tests into the visibly pushdown programs. If we view a VPA as the descriptions of a recursive procedure, then this nesting allows, for example, to formulate pre- and post-conditions for these procedures that relate to events happening during the procedure's execution. A simple example is the formula $\langle a^\#; p_0?; b^\# \rangle p_1$, which states that it is possible to execute $a^\# b^\#$ such that p_0 was true after the $a^\#$ part and p_1 holds after the whole execution finished. It is not clear how to express such a property without using tests inside a context-free language. In the next subsection, we show how to use tests inside programs to express multiple return conditions.

3.2 Comparison to VP-μ

In [1], the authors define a new modal fixpoint logic, the *visibly pushdown μ-calculus* (VP-μ), as an extension of the modal μ-calculus to reason about the behaviour of recursive programs. The models for this logic are trees where the procedure calls and returns are made visible as edge labels. With every state s in a VP-μ model, one associates the set of its matching returns, i.e., the states reached when leaving the context of s (where the context of s consists of all states belonging to the same procedure execution as s). Then a state together with its matching return states is a summary. Roughly, the main difference between VP-μ and the standard μ-calculus is that, in VP-μ, the fixpoints are interpreted over sets of summaries instead of sets of states as in standard μ-calculus.

VP-μ is a powerful logic that allows to express a number of natural properties of recursive programs which cannot be captured by logics such as the modal μ-calculus. In [1], the authors show that model-checking formulas of VP-μ against pushdown models is decidable in ExpTime, but satisfiability is undecidable. Interestingly, the example properties presented in [1] to illustrate the expressive power of VP-μ can also be expressed in recursive PDL or its extension recursive Δ-PDL, which both have a decidable satisfiability problem. Here we review some of these examples.

Reachability (resp. local reachability). There is a path to some state where a property φ holds and this state is reached before exiting (resp. in) the current context. This is easily expressed by the formula $\langle L \rangle \varphi$ where L consists of all words that do not contain unmatched returns (resp. of all well-matched words).

Termination. Every call is eventually matched. As recursive PDL does not allow to talk about infinite computations, termination cannot be expressed. However, it can easily be expressed in recursive Δ-PDL, the extension to infinite computations developed in Section 6. In fact, it suffices to require that there is no infinite path containing an unmatched call. This can easily be tested by a Vpa on infinite words. We return to this example in Section 6, where a concrete such Vpa is given.

Multiple return conditions. Formulate a property that holds after the return of a procedure and that is conditional w.r.t. events happening during the execution of the procedure. Obviously, this is similar to the example $\langle a^{\#}; p_0?; b^{\#} \rangle p_1$ given in the previous subsection. We may use a Vpa to find the return point of the procedure and a test to check for events that happen during the procedure's execution. We give more details in the following example.

Example 7 Assume that the atomic program a indicates a call to a procedure

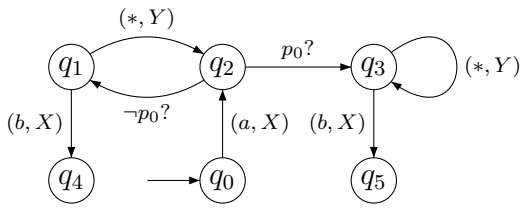

Fig. 2. VPA for the example on multiple return conditions

P, and b indicates a return from P. We want to specify that for each call of P, the following holds: if property p_0 holds at some point before the call to P returns, then p_1 should hold when the call to P returns. Otherwise, p_1 should not hold on the return. We express this by the formula

$$[\Pi^*](([\mathcal{A}_1]p_1) \wedge ([\mathcal{A}_2]\neg p_1)$$

where $\mathcal{A}_1$ and $\mathcal{A}_2$ both have the transition structure depicted in Figure 2, q_5 is the final state of $\mathcal{A}_1$, and q_4 is the final state of $\mathcal{A}_2$. A transition label (a, X) means that on reading a the automaton pushes X onto the stack. Similarly, (b, X) means that the automaton can read b and pop X from the stack. The transition label $(*, Y)$ means that each input symbol can be read, and for calls Y is put onto the stack, for returns Y is taken from the stack.

The automaton starts by reading a, standing for a call of procedure P, pushing an X onto the stack. After that it will store Y on the stack for each call. As long as p_0 does not hold, the automaton will alternate between the states q_1 to q_2. If p_0 holds, the automaton memorises this by moving to q_3. If b is read when X is the top stack symbol, then the automaton knows that the initial call a returns now. Depending on whether p_0 was true in the meantime, it moves to state q_4 or q_5.

4 Satisfiability for Recursive PDL

In this section, we show that the satisfiability problem for recursive PDL is decidable. The idea for the satisfiability test is the same as in [22] and [12]. One first shows that each recursive PDL formula φ has a tree model. In these tree models one labels each node s with all subformulas of φ that are true in s. Such trees are called Hintikka trees. Then one constructs a tree automaton that accepts the Hintikka trees of φ and checks this automaton for emptiness. When starting with a PDL formula one obtains a Büchi tree automaton. Since we use VPAs for the definition of programs we will end up with a visibly pushdown tree automaton.

The following definitions and propositions concerning Hintikka trees are simple adaptations from [12], we just recall them here for completeness.

From now on, we identify a formula φ with the formula $\neg\neg\varphi$. For each formula φ in recursive PDL, we define its closure $\mathrm{cl}(\varphi)$ as the minimal set satisfying the following:

- $\varphi \in \mathrm{cl}(\varphi)$.
- If $\varphi_1 \wedge \varphi_2 \in \mathrm{cl}(\varphi)$, then $\varphi_1, \varphi_2 \in \mathrm{cl}(\varphi)$.
- If $\psi \in \mathrm{cl}(\varphi)$, then $\neg\psi \in \mathrm{cl}(\varphi)$.
- If $\langle\mathcal{A}\rangle\psi \in \mathrm{cl}(\varphi)$, then $\psi \in \mathrm{cl}(\varphi)$. Additionally, if $\psi'?$ is an internal action in $\mathcal{A}$, then $\psi' \in \mathrm{cl}(\varphi)$.

Note that the size of $\mathrm{cl}(\varphi)$ is linear in the size of φ. By $\mathrm{cl}_\diamond(\varphi)$ we denote the set of all diamond formulas from $\mathrm{cl}(\varphi)$.

We now fix a formula φ of recursive PDL containing n atomic programs $a_0, \ldots, a_{n-1}$. Furthermore, we assume w.l.o.g. that $\mathbb{P}$ contains only those atomic propositions that are used in φ.

A tree structure $M = (S, R, \nu)$ is a *tree model* for φ if $M, \varepsilon \models \varphi$. As for PDL formulas, one can show that if a recursive PDL formula has a model then it has a tree model.

Proposition 8 *A formula of recursive PDL is satisfiable if and only if it has a tree model.*

Proof. It is clear that if φ has a tree model, then φ has a model.

Suppose that $M, s_0 \models \varphi$ for some structure $M = (S, R, \nu)$. Define $r := 2^{|\mathrm{cl}(\varphi)|}$ and let $C_0, \ldots, C_{r-1}$ be an enumeration of the subsets of $\mathrm{cl}(\varphi)$. Recall that n is the number of atomic programs used in φ. For $k := n \cdot r$ we define a mapping $\Phi : [k]^* \to 2^S$ by induction on the length of words in $[k]^*$. First, let $\Phi(\varepsilon) = \{s_0\}$. Assume that Φ is already defined for $x \in [k]^*$ and let $d = jn + \ell$ with $0 \leq j < r$ and $0 \leq \ell < n$. If $\Phi(x) = \emptyset$, then we let $\Phi(xd) = \emptyset$. If $\Phi(x) = S_x$ is not empty, then we set:

$$\Phi(xd) = \{s' \in S \mid \exists s \in S_x,\ (s, s') \in R(a_\ell) \text{ and } C_j = \{\psi \in \mathrm{cl}(\varphi) \mid M, s' \models \psi\}\}.$$

Intuitively, $\Phi(xd)$ describes all states in M that are reachable from a state in $\Phi(x)$ with program a_ℓ and that satisfy exactly the formulas in C_j.

Consider now the structure $M' = (S', R', \nu')$ where $S' = \{x \mid \Phi(x) \neq \emptyset\}$,

$R'(a_\ell) = \{(x, xd) \in S' \times S' \mid \ell = d \mod n\}$, and $\nu'(x) = \nu(s)$ for some $s \in \Phi(x)$ (which is well defined as all $s \in \Phi(x)$ satisfy the same formulas from $\mathrm{cl}(\varphi)$, hence the same atomic propositions). It is easy to see that M' is a tree structure for φ. Furthermore, for $\psi \in \mathrm{cl}(\varphi)$, one can show by induction on the structure of ψ that $M', x \models \psi$ iff $M, s \models \psi$ for all $s \in \Phi(x)$. For diamond formulas one can proceed by induction on the length of the witnessing path without referring to the formalism used to define the programs. Applied to $x = \varepsilon$ and $\psi = \varphi$ we obtain $M', \varepsilon \models \varphi$. $\square$

We now define the notion of *Hintikka tree*. For this purpose we define the alphabet $\Sigma_\varphi = 2^{\mathrm{cl}(\varphi)} \cup \{\bot\}$, where $\bot$ is some symbol used to label nodes that do not have to be considered. Note that this use of $\bot$ is not at all related to the bottom-of-stack symbol used for VPAs.

Definition 9 A *Hintikka tree* for a formula φ of recursive PDL with atomic programs $a_0, \ldots, a_{n-1}$ is a k-ary tree $t : [k]^* \to \Sigma_\varphi$ with $k \geq n$ such that $\varphi \in t(\varepsilon)$, and for all elements $x \in [k]^*$:

(1) If $t(x) = \bot$, then $t(xd) = \bot$ for all $d \in [k]$.
(2) If $t(x) \neq \bot$, then $\psi \in t(x)$ if and only if $\neg\psi \notin t(x)$ for all $\psi \in \mathrm{cl}(\varphi)$.
(3) If $t(x) \neq \bot$ and $\psi_1 \wedge \psi_2 \in \mathrm{cl}(\varphi)$, then $\psi_1 \wedge \psi_2 \in t(x)$ if and only if $\psi_1, \psi_2 \in t(x)$.
(4) *(Diamond property)* If $t(x) \neq \bot$, then $\langle\mathcal{A}\rangle\psi \in t(x)$ if and only if there exists an $\mathcal{A}$-path (to be defined below) from x to y in t for some $y \in [k]^*$ such that $t(y) \neq \bot$ and $\psi \in t(y)$.
(5) *(Box property)* If $t(x) \neq \bot$, then $\neg\langle\mathcal{A}\rangle\psi \in t(x)$ if and only if $\psi \notin t(y)$ for all $y \in [k]^*$ with $t(y) \neq \bot$ for which there is an $\mathcal{A}$-path from x to y.

An *$\mathcal{A}$-path* from a node x to a node y is a sequence $x_0, \ldots, x_m$ of nodes with $x_0 = x$ and $x_m = y$ such that there is a word $w = w_1 \cdots w_m \in L(\mathcal{A})$ and the following holds for all $i = 1, \ldots, m$:

- If $w_i = \psi'?$ for some formula ψ', then $x_i = x_{i-1}$ and $\psi' \in t(x_{i-1})$.
- If $w_i = a_\ell$ for some atomic program a_ℓ, then $x_i = x_{i-1}d$ for some d with $\ell = d \mod n$.

The $\mathcal{A}$-path required in 4 of the previous definition is also called a *witnessing path* for $\langle\mathcal{A}\rangle\psi$.

It is not difficult to see that Hintikka trees for φ are obtained from tree models of φ by annotating each node with the set of formulas that are satisfied in this node.

Proposition 10 *Let φ be a formula of recursive PDL. There is a Hintikka tree for φ if and only if φ has a tree model.*

Proof. If $M = (S, R, \nu)$ is a tree model for φ, then we obtain a Hintikka tree for φ as follows: for an element $x \in [k]^* \setminus S$ take $t(x) = \bot$, and for an element $x \in S$, take $t(x) = \{\psi \in \mathrm{cl}(\varphi) \mid M, x \models \psi\}$. Noting that $M, \varepsilon \models \varphi$ and using the definition of the semantics of recursive PDL one shows by induction that t is indeed a Hintikka tree for φ.

Starting from a Hintikka tree t for φ we construct a tree model $M = (S, R, \nu)$ for φ as follows. The set of states is $S = \{x \in [k]^* \mid t(x) \neq \bot\}$, the transition relation R is such that for all atomic programs a_ℓ, $R(a_\ell) = \{(s, sd) \mid \ell = d \mod n \text{ and } sd \in S\}$, and for all $s \in S$, $\nu(s) = \{p \in \mathbb{P} \mid p \in t(s)\}$. In order to show that $M, \varepsilon \models \varphi$, one shows by induction on the structure of the formula that for all $\psi \in \mathrm{cl}(\varphi)$ and all states $s \in S$ that $\psi \in t(s)$ if and only if $M, s \models \psi$. The base case where $\psi \in \mathbb{P}$ comes from the definition of ν while the inductive step follows from the definition of Hintikka tree. $\square$

Our goal is to build a tree automaton that accepts Hintikka trees for φ. Such an automaton has to verify for each node x with a diamond formula $\langle \mathcal{A} \rangle \psi$ in $t(x)$ that there is an $\mathcal{A}$-path starting from x to some node y. Such paths may overlap and the tree automaton would have to keep track of which VPAs to simulate in order to check the diamond property for several nodes. To simplify this task we show that it is always possible to find a Hintikka tree where the paths witnessing the diamond properties are (edge) disjoint. Such Hintikka trees are called *unique diamond path Hintikka trees* in [12]. In the definition from [12] it is possible that for a diamond formula $\langle \mathcal{A} \rangle \psi$ that is in $t(x)$ the witnessing path contains a node y such that $\langle \mathcal{A} \rangle \psi$ is also in $t(y)$. Then the witnessing path for this second occurrence of the diamond formula might overlap the witnessing path for the first occurrence. In our definition we also avoid this problem.

Definition 11 A *unique diamond path Hintikka tree* for φ is a Hintikka tree t for φ that satisfies the following additional condition: there exists a mapping $\rho : [k]^* \to (\mathrm{cl}_\diamond(\varphi) \times [k]^*) \cup \{\bot\}$, such that for all $x \in [k]^*$: If $\langle \mathcal{A} \rangle \psi \in t(x)$ then, for some witnessing $\mathcal{A}$-path $x_0, \ldots, x_m$ (starting in x), we have $\rho(x_i) = (\langle \mathcal{A} \rangle \psi, x)$ for all $1 \leq i \leq m$.

Any Hintikka tree can be transformed into a unique diamond path Hintikka tree by increasing the number of descendants of each node such that there is a separate branch for each formula when needed. The branching degree resulting from this increase of descendants can be bounded as stated in the following proposition, where r denotes the number of diamond formulas in $\mathrm{cl}(\varphi)$ and n the number of atomic programs.

Proposition 12 *Let φ be a formula of recursive PDL. There is a Hintikka tree for φ if and only if there is a k-ary unique diamond path Hintikka tree for φ with $k = 2^{|\mathrm{cl}(\varphi)|} \cdot n \cdot 2r$.*

Proof. Consider a Hintikka tree $t : [k]^* \to \Sigma_\varphi$ for φ. Let $k' = 2rk$, where $r = |\mathrm{cl}_\diamond(\varphi)|$, and fix an ordering $\psi_0, \ldots, \psi_{r-1}$ on the diamond formulas from $\mathrm{cl}_\diamond(\varphi)$. We define a mapping $\eta : [k']^* \to [k]^*$ that associates to each element from $[k']^*$ an element of $[k]^*$ by induction on the length of words in $[k']^*$ as follows: $\eta(\varepsilon) = \varepsilon$ and $\eta(xd') = \eta(x)d$ where $d = d' \mod k$. We finally define $t' : [k']^* \to \Sigma_\varphi$ by $t'(x) = t(\eta(x))$. It is not difficult to show that t' is a Hintikka tree.

In order to show that t' is a unique diamond path Hintikka tree, we have to define the mapping $\rho : [k']^* \to (\mathrm{cl}_\diamond(\varphi) \times [k']^*) \cup \{\bot\}$. The idea for this mapping is the following. Assume that ψ_j is in $t(x)$ for some $x \in [k]^*$ and that the last vertex in the witnessing path for ψ_j is obtained from x by appending $d_0, d_1, \ldots, d_m$. In t' we have $\psi_j \in t'(x')$ for all x' with $\eta(x') = x$. The witnessing path in t' is then obtained by appending $d'_0, d'_1, \ldots, d'_m$ to x with $d'_0 = d_0 + (r + j)k$ and $d'_i = d_i + jk$ for $i \in \{1, \ldots, m\}$.

Formally, we define ρ inductively on the length of the nodes. We set $\rho(\varepsilon) = \bot$. Assume that $\rho(x')$ is already defined for $x' \in [k']^*$. Let $d \in \{0, \ldots, k-1\}$ and $j \in \{0, \ldots, r-1\}$. Now we define for $d' = d + jk$

$$\rho(x'd') = \begin{cases} \rho(x') \text{ if } \rho(x') = (\psi_j, y) \text{ for some } y \in [k']^*, \\ \bot \text{ otherwise,} \end{cases}$$

and for $d' = d + (r + j)k$

$$\rho(x'd') = \begin{cases} (\psi_j, x') \text{ if } \psi_j \in t'(x'), \\ \bot \text{ otherwise.} \end{cases}$$

Using the idea sketched above on how to obtain the witnessing paths in t' from those in t, one can show by induction on the length of $\mathcal{A}$-paths that ρ satisfies the conditions for t' being a unique diamond path Hintikka tree.

The arity of t' is $2rk$ where k can be chosen equal to $2^{|\mathrm{cl}(\varphi)|} \cdot n$ as shown by the proof of Proposition 10. □

4.2 From Recursive PDL to Tree Automata

We now show how to build a Büchi VPTA accepting exactly the k-ary unique diamond path Hintikka trees for φ. Together with Theorem 4 one obtains decidability of the satisfiability problem for recursive PDL formulas.

So from now on we are interested in trees from $\mathcal{T}_{k,\Sigma_\varphi}$. Further, note that each $d \in [k]$ is associated in a natural way to an atomic program in the definition

of $\mathcal{A}$-path, namely to a_ℓ if $\ell = d \mod n$. This directly induces a partition of $[k]$ into calls, returns, and internal actions.

The construction of the VPTA follows the same lines as in [12]. We first build three visibly pushdown tree automata. The first automaton is called the *local automaton* and accepts all trees satisfying the first two items of Definition 9. The second automaton called *box automaton* accepts all trees satisfying the box property (see Definition 9). The third automaton called *diamond automaton* accepts all trees satisfying the diamond property (see Definition 9) and the condition of Definition 11.

The intersection of the languages accepted by these three automata defines exactly the set of k-ary unique diamond path Hintikka trees for φ. As visibly pushdown tree languages are closed under intersection, a nondeterministic visibly pushdown tree automaton recognising the desired language can be constructed.

Local automaton.

The local automaton is easily constructed as a two-state finite tree automaton equipped with a safety condition. The automaton checks for all nodes x in the tree whether $t(x)$ satisfies the first two conditions of Definition 9. If in some node one of these two conditions is violated, the automaton goes to its rejecting state, otherwise it stays in the initial state.

Lemma 13 *There is a finite tree automaton with a safety acceptance condition and two states that accepts the trees that satisfy the first two properties of Definition 9.*

Box automaton.

We now construct a VPTA accepting those trees from $\mathcal{T}_{k,\Sigma_\varphi}$ that satisfy the box property from Definition 9. First note that the box property is a condition on the paths through the tree. This means we can define a language $L_{\text{box}} \subseteq (\Sigma_\varphi \times [k])^\omega$ such that $T_{\text{box}} = \text{Trees}(L_{\text{box}})$, where T_{box} denotes the set of all trees satisfying the box property. We now define L_{box} and then show that it can be accepted by a deterministic safety VPA.

For each word $w \in (\Sigma_\varphi \times [k])^\omega$ there exists a tree $t \in \mathcal{T}_{k,\Sigma_\varphi}$ and a path π such that $w = w^t_\pi$. Then w is in L_{box} if this t satisfies the box property on π: for all $x \in \pi$, $\neg\langle\mathcal{A}\rangle\psi \in t(x)$ if and only if $\psi \notin t(y)$ for all $y \in \pi$ for which there is an $\mathcal{A}$-path from x to y.

It is not difficult to see that $t \in \mathcal{T}_{k,\Sigma_\varphi}$ indeed satisfies the box property if and

only if all its paths are in L_{box}. Hence, by Remark 5, to construct a VPTA for T_{box} it is sufficient to construct a deterministic VPA for L_{box}.

Lemma 14 *There is a deterministic safety* VPA *of size exponential in the size of φ that accepts L_{box}.*

Proof. Let $\psi_1, \ldots, \psi_m$ be an enumeration of all box formulas $\psi_i = \neg\langle\mathcal{A}_i\rangle\varphi_i \in \text{cl}(\varphi)$. We show how to construct a visibly pushdown automaton for the complement $\overline{L}_{\text{box}}$ of L_{box}, and we conclude using closure of visibly pushdown languages under complementation.

First note that $\overline{L}_{\text{box}} = \bigcup_{i=1}^{m} \overline{L}_i$, where $\overline{L}_i$ is the set of all words describing a path that violates the box condition for ψ_i. For every i, $\overline{L}_i$ is accepted by a VPA $\mathcal{B}_i$ equipped with a reachability condition as follows.

For an input word $w = (C_0, d_0)(C_1, d_1)\cdots$ with $C_j \in \Sigma_\varphi$ and $d_j \in [k]$ the VPA $\mathcal{B}_i$ guesses a segment $(C_j, d_j)\cdots(C_{j'}, d_{j'})$ with $\psi_i \in C_j$ and $\varphi_i \in C_{j'}$, and verifies that it corresponds to an $\mathcal{A}_i$-path. This is realised as follows:

- Before guessing the initial position j of the segment, $\mathcal{B}_i$ stores a special symbol $\sharp$ on the stack. On guessing j it enters a state indicating that the simulation of $\mathcal{A}_i$ starts.
- In the simulation phase, on reading a letter (C, d), $\mathcal{B}_i$ can simulate a sequence of transitions of $\mathcal{A}_i$ consisting of tests and ending with the atomic program a_ℓ corresponding to d, i.e., with $\ell = d \mod n$. So, a change of configuration in $\mathcal{A}_i$ on reading a word of the form $\chi_1?\cdots\chi_r?a_\ell$ is performed in $\mathcal{B}_i$ in a single transition on (C, d) if $\chi_1, \ldots, \chi_r$ are in $C \neq \bot$. This is possible since tests are handled as internal actions in $\mathcal{A}_i$ and thus only induce a change of the control state.

  In this simulation, whenever $\mathcal{B}_i$ sees $\sharp$ as top stack symbol, it treats it as the bottom-of-stack symbol $\triangleleft$ is handled in $\mathcal{A}_i$.
- Finally, if $\mathcal{B}_i$ reads (C, d) with $\varphi_i \in C$, and there is a (possibly empty) sequence $\chi_1?\cdots\chi_r?$ of tests leading to an accepting state in $\mathcal{A}_i$ where $\chi_1, \ldots, \chi_r$ are in C, then $\mathcal{B}_i$ can move to its accepting state on reading (C, d). Once $\mathcal{B}_i$ has reached its accepting state it remains there forever.

Note that the size of $\mathcal{B}_i$ is linear in the size of $\mathcal{A}_i$. Furthermore, $\mathcal{B}_i$ can be constructed such that it is complete because every run that reaches an accepting state never stops.

Taking the union of these VPAs one obtains a reachability VPA $\mathcal{B}$ for $\overline{L}_{\text{box}}$. Determinising and then complementing $\mathcal{B}$ (see Corollary 3) yields a safety VPA for L_{box} that is of size exponential in $\mathcal{B}$ and thus also exponential in the size of φ. □

Applying Remark 5 we directly get the following result.

Lemma 15 *There is a safety* VPTA *of size exponential in the size of* φ *that accepts* T_{box}.

Diamond automaton.

We give an informal description of the diamond automaton. This automaton is designed to accept trees that satisfy both the diamond condition and the one of Definition 11.

The control state of the diamond automaton stores the following informations:

- A diamond formula $\langle \mathcal{A} \rangle \psi$ currently checked or $\bot$ if nothing is checked.
- If some diamond formula $\langle \mathcal{A} \rangle \psi$ is being checked, a control state of $\mathcal{A}$ is stored (and stack information from $\mathcal{A}$ will be encoded in the stack of the diamond automaton).

At the beginning no formula is checked. The diamond automaton reads the labelling $t(x)$ of the current node x. If it contains some diamond formula, it will go for each of these formulas in a different branch of the tree where it checks this formula. If the automaton was already checking for a diamond formula, it keeps looking for its validation by choosing yet another branch. As the tree should satisfy the unique diamond path property, a validation of the diamond formulas can be found in this way.

When checking for a diamond formula $\langle \mathcal{A} \rangle \psi$, the automaton performs a simulation of $\mathcal{A}$ on the path it guesses. A sequence of tests read by $\mathcal{A}$ followed by some atomic program is simulated in a single transition of the VPTA. For this it stores in its control state the current state q of $\mathcal{A}$ in the simulation and uses its stack to mimic the one of $\mathcal{A}$. Assume that in $\mathcal{A}$ a sequence of the following form is possible: $(q, \gamma) \xrightarrow{\chi_1?} (q_1, \gamma) \xrightarrow{\chi_2?} \cdots \xrightarrow{\chi_m?} (q_m, \gamma) \xrightarrow{a_\ell} (q', \sigma)$, where γ denotes the top stack symbol and σ is the new top of the stack, depending on the type of a_ℓ, i.e., $\sigma = \varepsilon$ for a return, $\sigma = \gamma$ for an internal action, and $\sigma = \gamma'\gamma$ for a call and some γ' from the stack alphabet of $\mathcal{A}$. Then the VPTA on reading a node label $t(x)$ that contains $\chi_1, \ldots, \chi_m$ can update the state q of $\mathcal{A}$ to q' when proceeding to a d-successor with $\ell = d \mod k$.

To keep track of the level of the stack where the simulation of $\mathcal{A}$ started, the first symbol pushed onto the stack after starting the simulation of $\mathcal{A}$ is marked by $\sharp$. If this symbol is popped later, then it is recorded in the state of the VPTA that the simulation is at the bottom of the stack, i.e., $\mathcal{A}$-transitions are simulated as if $\triangleleft$ would be the top stack symbol. If a symbol is pushed, it is again marked by $\sharp$.

The simulation ends if the current node label $t(x)$ contains ψ and from the current state q of the $\mathcal{A}$-simulation a final state of $\mathcal{A}$ is reachable by a (possibly empty) sequence of tests such that the corresponding formulas are included in $t(x)$. In this case the VPTA signals this successful simulation in the next transition by setting a special flag in all successor states. This flag also defines the acceptance condition. If the flag is set infinitely often on each path, then the input is accepted. For this to work we also set the flag if no simulation is performed. This acceptance condition is of Büchi type and hence we have the following result.

Lemma 16 *There is a Büchi* VPTA *of size* $\mathcal{O}(|\varphi|)$ *that accepts those trees from* $\mathcal{T}_{k,\Sigma_\varphi}$ *that satisfy the diamond property and the condition of Definition 11.*

Now, consider the automaton obtained by taking the product of the local automaton, the box automaton, and the diamond automaton. The combination of two safety conditions and one Büchi condition can easily be transformed into a single Büchi condition.

Lemma 17 *There is a Büchi* VPTA *of size exponential in the size of* φ *that accepts the* k*-ary unique diamond path Hintikka trees for* φ*.*

Using Theorem 4 we deduce the decidability of the satisfiability problem for recursive PDL formulas.

Theorem 18 *Given a recursive PDL formula, one can decide in doubly exponential time whether it is satisfiable.*

In the next section we establish a matching lower bound for this complexity.

5 Lower Bound

Our aim is to establish lower bounds for extensions of PDL with recursive programs. We start with a 2-EXPTIME lower bound for PDL+$a^\# b^\#$, i.e., PDL extended with the single program $a^\# b^\#$, where the new program cannot even be used to build up complex programs via regular expressions. The proof is by a reduction of the word problem for exponentially space-bounded alternating Turing machines. In a second step, we give a sketch of how to generalise this lower bound to a whole class of context-free languages.

An *Alternating Turing Machine (ATM)* is of the form $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \Delta)$. The set of *states* $Q = Q_\exists \uplus Q_\forall \uplus \{q_a\} \uplus \{q_r\}$ consists of *existential states* in

$Q_\exists$, *universal states* in $Q_\forall$, an *accepting state* q_a, and a *rejecting state* q_r; Σ is the *input alphabet* and Γ the *work alphabet* containing a *blank symbol* $\Box$ and satisfying $\Sigma \subseteq \Gamma$; $q_0 \in Q_\exists \cup Q_\forall$ is the *starting* state; and the *transition relation* Δ is of the form

$$\Delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}.$$

We write $\Delta(q, \sigma)$ to denote $\{(q', \sigma', M) \mid (q, \sigma, q', \sigma', M) \in \Delta\}$.

A *configuration* of an ATM is a word wqw' with $w, w' \in \Gamma^*$ and $q \in Q$. The intended meaning is that the one-side infinite tape contains the word ww' with only blanks behind it, the machine is in state q, and the head is on the symbol just after w. The *successor configurations* of a configuration wqw' are defined in the usual way in terms of the transition relation Δ. A *halting configuration* is of the form wqw' with $q \in \{q_a, q_r\}$.

A *computation* of an ATM $\mathcal{M}$ on a word w is a (finite or infinite) sequence of configurations $K_0, K_1, \ldots$ such that $K_0 = q_0 w$ and K_{i+1} is a successor configuration of K_i for all $i \geq 0$. The ATMs considered in the following have only *finite* computations on any input. Since this case is simpler than the general one, we define acceptance for ATMs with finite computations, only. Let $\mathcal{M}$ be such an ATM. A halting configuration is *accepting* iff it is of the form $wq_a w'$. For other configurations $K = wqw'$, acceptance depends on q: if $q \in Q_\exists$, then K is accepting iff at least one successor configuration is accepting; if $q \in Q_\forall$, then K is accepting iff all successor configurations are accepting. Finally, the ATM $\mathcal{M}$ with starting state q_0 *accepts* the input w iff the *initial configuration* $q_0 w$ is accepting. We use $L(\mathcal{M})$ to denote the language accepted by $\mathcal{M}$.

There is an exponentially space bounded ATM $\mathcal{M}$ whose word problem is 2-ExpTime-hard and we may assume that the length of every computation path of $\mathcal{M}$ on $w \in \Sigma^n$ is bounded by 2^{2^n}, and all the configurations wqw' in such computation paths satisfy $|ww'| \leq 2^n$, see [5]. We may also assume w.l.o.g. that $\mathcal{M}$ never attempts to move left on the left-most tape cell. Let $w = \sigma_0 \cdots \sigma_{n-1} \in \Sigma^*$ be an input to $\mathcal{M}$. We construct a formula $\varphi_{\mathcal{M},w}$ of PDL+$a^\# b^\#$ such that $w \in L(\mathcal{M})$ iff $\varphi_{\mathcal{M},w}$ is satisfiable.

In models of $\varphi_{\mathcal{M},w}$, each state represents a tape cell, and a path of 2^n states connected by the program a is used to represent a configuration. The program a is also used to connect each configuration to its successor configurations. Already in PDL, it is possible to formulate most properties of ATMs, e.g. that each tape cell is labelled with exactly one symbol, and that the initial configuration has the expected shape. What cannot be done using a PDL formula of only polynomial length is to access the tape cell of the consecutive configuration that has the same position as the current cell (since it is 2^n steps away). This is necessary to describe a step made by the ATM and also to state

that tape cells which are not underneath the head do not change. It is here that we use the program $a^{\#}b^{\#}$. The idea is to attach to each element a path of 2^n states connected by the program b. The states on this path only serve an auxiliary purpose and do not correspond to tape cells. Then the program $a^{\#}b^{\#}$ will bring us from a tape cell to the end of the auxiliary chain that is attached to the corresponding tape cell in consecutive configurations. As we shall see, this suffices to finish the reduction.

Summing up, in $\varphi_{\mathcal{M},w}$ we use the atomic programs a and b and the set of atomic propositions $\{t\} \cup \Gamma \cup Q \cup \{c_0, \ldots, c_{n-1}\} \cup X$, where $X := \{m_{q,\sigma,M} \mid q \in Q, \sigma \in \Gamma, M \in \{L, R\}\}$. The propositions have the following meaning:

- t is used to identify the states that represent tape cells (as opposed to being on the auxiliary chains);
- $\sigma \in \Gamma$ is true at a state s if the tape cell represented by s is labelled with σ in the current configuration;
- $q \in Q$ is true at s if the head of $\mathcal{M}$ is on s in the current configuration and the machine is in state q;
- $m_{q,\sigma,M}$ is true at s if the head was on s in the previous configuration and the machine reached the current configuration by switching to state q, writing σ, and moving in direction M;
- $c_{n-1}, \ldots, c_0$ describe a counter C in binary coding for counting the length of configurations and of auxiliary b-paths.

We now assemble the reduction formula $\varphi_{\mathcal{M},w}$, starting with the auxiliary paths and the behaviour of the counter C. On states satisfying t, it counts modulo 2^n along the program a. On states satisfying $\neg t$, it counts modulo 2^n along the program b. We first define the following abbreviation for incrementing the counter C when travelling along a program $\alpha \in \{a, b\}$:

$$\begin{aligned}\mathsf{Inc}(\alpha) := \bigwedge_{k=0}^{n-1} \Big(\bigwedge_{j=0}^{k-1} c_j \to (c_k \to [\alpha]\neg c_k) \wedge (\neg c_k \to [\alpha] c_k) \Big) \wedge \\ \bigwedge_{k=0}^{n-1} \Big(\bigvee_{j=0}^{k-1} \neg c_j \to (c_k \to [\alpha] c_k) \wedge (\neg c_k \to [\alpha]\neg c_k) \Big)\end{aligned}$$

We will also use the abbreviation $(C = 2^n - 1)$ for $c_0 \wedge \cdots \wedge c_{n-1}$, $(C < 2^n - 1)$ for $\neg(C = 2^n - 1)$, and $(C = 0)$ for $\neg c_0 \wedge \cdots \wedge \neg c_{n-1}$. Now we establish auxiliary paths and the behaviour of the counter C:

$$\begin{aligned}\varphi_1 := [U]\Big(t \to (\mathsf{Inc}(a) \wedge \langle b \rangle \neg t \wedge [b](C = 0)) \wedge \\ (\neg t \wedge (C < 2^n - 1)) \to (\langle b \rangle \neg t \wedge \mathsf{Inc}(b)) \wedge \\ (\neg t \wedge (C = 2^n - 1)) \to ([b]\bot) \Big)\end{aligned}$$

where $[U]\psi$ abbreviates $[(a \cup b)^*]\psi$ and $\bot$ is logical falsity, both here and in what follows.

We now formulate some general requirements on ATMs: every tape cell is labelled with exactly one symbol from Γ and never with two different states, and no configuration has more than one cell marked with the tape head.

$$\varphi_2 := [U]\Big(\bigvee_{\sigma\in\Gamma} \sigma \ \wedge \bigwedge_{\sigma,\sigma'\in\Gamma,\sigma\neq\sigma'} \neg(\sigma \wedge \sigma') \ \wedge \bigwedge_{q,q'\in Q,q\neq q'} \neg(q \wedge q') \wedge \bigvee_{q\in Q} q \rightarrow [((C < 2^n - 1)?; a)^*] \bigwedge_{q\in Q} \neg q \Big)$$

Next, we set up the initial configuration. Recall that $w = \sigma_0 \cdots \sigma_{n-1}$ is the input and $\Box$ the blank symbol.

$$\varphi_3 := (C = 0) \wedge q_0 \wedge \bigwedge_{i<n} [a^i]\sigma_i \wedge [a^n; ((C < 2^n - 1)?; a)^*]\,\Box$$

As explained above, we can use the program $a^\# b^\#$ to travel from a tape cell to the end of the auxiliary path that starts at the corresponding tape cell in consecutive configurations. To propagate information between the cell and the auxiliary path, it is useful to state that they interpret the relevant propositional letters in the same way.

$$\varphi_4 := [U] \bigwedge_{p\in Q\cup\Gamma\cup X} \Big((p \rightarrow [b]p) \wedge (\langle b\rangle p \rightarrow p)\Big)$$

Given φ_4, we can easily state that tape cells that are not underneath the tape head do not change when $\mathcal{M}$ makes a transition:

$$\varphi_5 := \bigwedge_{\sigma\in\Gamma} \Big((\bigwedge_{q\in Q} \neg q \wedge \sigma) \ \rightarrow \ [a^\# b^\#]([b]\bot \rightarrow \sigma)\Big)$$

Observe that we indeed reach the auxiliary path of the corresponding cell in the successor configuration because (i) using $a^\# b^\#$, we travel as many a's as b's; (ii) the auxiliary paths have exactly length 2^n due to φ_1; and (iii) we use $[b]\bot$ to ensure that we reach the end of the auxiliary path.

We now encode $\mathcal{M}$'s transition function δ. This is done using the propositional letters from X (which are of the form $m_{q,\sigma,M}$). First, we propagate the transition of $\mathcal{M}$ to the successor configurations.

$$\varphi_6 := [U]\Big(\bigwedge_{q\in Q_\exists,\sigma\in\Gamma} \Big(q \wedge \sigma \ \rightarrow \bigvee_{(p,\sigma',M)\in\delta(q,\sigma)} \langle a^\# b^\#\rangle([b]\bot \wedge m_{p,\sigma',M})\Big) \wedge \bigwedge_{q\in Q_\forall,\sigma\in\Gamma} \Big(q \wedge \sigma \ \rightarrow \bigwedge_{(p,\sigma',M)\in\delta(q,\sigma)} \langle a^\# b^\#\rangle([b]\bot \wedge m_{p,\sigma',M})\Big)$$

The following formula then actually implements the transition, relying on φ_4.

$$\varphi_7 := [U]\Big(\bigwedge_{q\in Q,\sigma\in\Gamma,M\in\{L,R\}} (m_{q,\sigma,M} \to \sigma) \wedge \bigwedge_{q\in Q,\sigma\in\Gamma} (\langle a\rangle m_{q,\sigma,L}) \to q \wedge \bigwedge_{q\in Q,\sigma\in\Gamma} (m_{q,\sigma,R} \to [a]q) \Big)$$

It remains to describe acceptance of the machine. Since all computation paths of $\mathcal{M}$ are finite, it suffices to require that the state q_r never appears:

$$\varphi_8 := [U]\neg q_r$$

Altogether, we obtain the formula $\varphi_{\mathcal{M},w} := \bigwedge_{1\le i\le 8} \varphi_i$. It is not difficult to verify that $w \in L(\mathcal{M})$ iff $\varphi_{\mathcal{M},w}$ is satisfiable. Since $|\varphi_{\mathcal{M},w}|$ is polynomial in n, together with Theorem 18, we get the following result.

Theorem 19 *Satisfiability in PDL+$a^\# b^\#$ is 2-*ExpTime*-complete.*

The lower bound presented above is easily adapted to other non-regular languages. Take, for example, $a^\# b^\# c^\#$: we can simply use auxiliary chains consisting of 2^n b's followed by 2^n c's. In the following, we generalise our lower bound to a whole class of (context-free) languages.

A *context-free grammar* $G = (N, T, S_0, P)$ consists of a set of *nonterminal symbols* N, *terminal symbols* T, a *start symbol* $S_0 \in N$, and a set P of productions of the form $A \to w$, where $A \in N$ and $w \in (N \cup T)^*$. We deviate slightly from this usual presentation and allow w to be from $(N \cup T \cup \mathcal{R})^*$, where $\mathcal{R}$ denotes the set of all regular languages over T in some unspecified (but fixed) representation.

A *parenthesis grammar* is a context-free grammar G such that

(1) there are two (distinct) *parenthesis symbols* $a, b \in T$, and
(2) all productions are of the form $A \to awb$, where w does not contain a and b, and all regular languages occurring in w are over $T \setminus \{a, b\}$.

In this definition, a and b function only as placeholders rather than as concrete symbols: any symbols satisfying the required conditions can serve as parenthesis symbols. A *parenthesis language* is a language generated by a parenthesis grammar.

Parenthesis languages where introduced by McNaughton in [16] as a class

of context-free grammars for which the equivalence problem is decidable.[1] It is not hard to see that all parenthesis languages can be accepted by a VPA, and are thus captured by Theorem 18. For the lower bound, we only consider parenthesis grammars that generate a non-empty language and in which the start symbol can reach itself, i.e., we have $S_0 \vdash^* wS_0w'$ for some $w, w' \in (N \cup T)^*$. We call such a parenthesis grammar (and the language generated by it) *simple*. Clearly, every simple parenthesis language is infinite.

Theorem 20 *Let L be a simple parenthesis language. Then satisfiability in PDL+L is 2-*EXPTIME*-complete.*

Proof. (sketch) The upper bound follows from Theorem 18 and the lower bound is obtained by adapting the lower bound for PDL+$a^\# b^\#$ given above. We only give a sketch of the latter. It is not difficult to see that for every simple parenthesis language L with parenthesis symbols a and b, there are $v, w, x \in T^*$ (which may contain a and b) such that

$$(av)^i awb\,(xb)^i \in L \text{ for all } i \geq 0.$$

We may thus adapt the above reduction as follows. We make each element of $T \setminus \{a, b\}$ an additional atomic program. Between every two consecutive tape cells (i.e. states satisfying t in the original reduction), we insert a v-path. Auxiliary paths are modified in two ways. First, we insert a w-path at the beginning; and second, we insert an x-path between every two consecutive states.

If a and b occur in v, w, x, then some additional care has to be taken. In particular, we have to make sure that occurrences of a in v and w are not misinterpreted as a step to the next tape cell and occurrences of b in w and x are not counted by the counter C when determining the length of auxiliary paths. This can be easily done using some additional atomic propositions as markers.

Then, the program $a^\# b^\#$ is replaced with L. It should we obvious that in this way, we reach the end of the auxiliary chains emerging from the corresponding tape cell in consecutive configurations as desired. We have to argue that we do not reach any undesired states. Since (already in the original reduction) we use $[b]\bot$ to make sure that we indeed have reached the end of an auxiliary chain, it actually suffices to ensure that we do not reach the end of any other auxiliary chain. Suppose that we do reach such an undesired chain ending. This means that $(av)^i awb\,(xb)^j \in L$ for some $i, j \geq 0$ with $i \neq j$. But that's impossible because every word contained in a parenthesis language has

[1] However, McNaughton does not allow regular languages on the right-hand-side of production rules.

(i) balanced parentheses and (ii) an opening bracket as first symbol and the corresponding closing bracket as last symbol. □

It is not clear how to generalise the proof of Theorem 20 from simple parenthesis languages to infinite parenthesis languages. For the latter, we can only infer the existence of $u, v, w, x, y \in T^*$ such that

$$au(av)^i awb\,(xb)^i yb \in L \text{ for all } i \geq 0,$$

and the au prefix seems difficult to handle unless we enrich our language with a converse constructor on atomic programs.

We note that Theorem 20 implies Theorem 19 and additionally captures a lot of other languages such as $a^{2n}b^{2n}$ and $a^n c^* b^n$.

6 Extension to Infinite Computations

In [20] an extension of PDL with a construct $\Delta\alpha$ for building formulas from programs α is considered. The meaning of such a formula is that the program α can be repeated infinitely often. The resulting logic is called Δ-PDL. In this section we extend recursive PDL by a similar construct $\Delta\mathcal{A}$ for Büchi VPAs $\mathcal{A}$ over atomic programs and tests. The meaning of such a formula is that there exists a path that is accepted by $\mathcal{A}$.

For the formal definition we introduce the notion of ω-program and add to the syntax rules of recursive PDL the following clauses:

- A Büchi VPA $\mathcal{A}$ over $\langle \Pi_c, \Pi_{int} \cup \text{Test}, \Pi_r \rangle$ is an ω-program.
- If $\mathcal{A}$ is an ω-program, then $\Delta\mathcal{A}$ is a formula.

This extension is called recursive Δ-PDL. For the semantics we only give the definitions for the new constructs. Each ω-program defines a unary relation R_ω and the corresponding Δ-formulas hold at those states of the structure that are in R_ω:

- $s \in R_\omega(\mathcal{A})$ if and only if there is an infinite word $w = w_0 w_1 w_2 \cdots \in L(\mathcal{A})$ (with $w_i \in \Pi \cup \text{Test}$) and a sequence $s_0, s_1, s_2, \ldots$ of states of the structure such that $s = s_0$ and $(s_i, s_{i+1}) \in R(w_i)$ for all $i \geq 0$.
- $M, s \models \Delta\mathcal{A}$ if and only if $s \in R_\omega(\mathcal{A})$.

To give an example we come back to the property of termination described in Section 3.

Example 21 For simplicity we consider only two atomic programs, a being

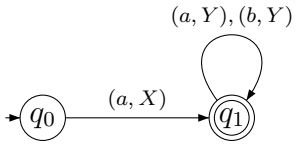

Fig. 3. VPA for Example 21.

a call and b being a return. We want to express that each call eventually returns. For this, we use the automaton $\mathcal{A}$ from Figure 3 that accepts all infinite sequences that start with a call that does not have a matching return. The desired formula is then $\varphi_{\text{ter}} = [\Pi^*](\neg\Delta\mathcal{A})$.

The definition of Hintikka tree extends in a straightforward way by adding the natural properties for formulas $\Delta\mathcal{A}$ and $\neg\Delta\mathcal{A}$. In the following, we call these properties Δ-property and $\neg\Delta$-property. The notion of unique diamond path Hintikka tree has to be extended by also requiring *unique Δ-paths*.

Definition 22 A *Hintikka tree* for a formula φ of recursive Δ-PDL with atomic programs $a_0, \ldots, a_{n-1}$ is a k-ary tree $t : [k]^* \to \Sigma_\varphi$ with $k \geq n$ such that $\varphi \in t(\varepsilon)$, and for all elements $x \in [k]^*$ properties (1)–(4) of Definition 9 are satisfied together with the two additional ones:

(5) *(Δ-property)* $\Delta\mathcal{A} \in t(x)$ if and only if there exists an $\mathcal{A}$-path (to be defined below) from x in t.
(6) *($\neg\Delta$-property)* $\neg\Delta\mathcal{A} \in t(x)$ if and only if there is no $\mathcal{A}$-path from x in t.

For a Büchi VPA $\mathcal{A}$, an *$\mathcal{A}$-path* from a node x is an infinite sequence $x_0, x_1, x_2, \ldots$ of nodes with $x_0 = x$ such that there is a word $w = w_1 w_2 w_3 \cdots \in L(\mathcal{A})$ and the following holds for all $i \geq 1$:

- If $w_i = \psi'?$ for some formula ψ', then $x_i = x_{i-1}$ and $\psi' \in t(x_{i-1})$.
- If $w_i = a_\ell$ for some atomic program a_ℓ, then $x_i = x_{i-1}d$ for some d with $\ell = d \mod n$.

We define the closure $\text{cl}(\varphi)$ of a Δ-formula as for recursive PDL formulas and we denote by $\text{cl}_\Delta(\varphi)$ the set of all Δ-formulas from $\text{cl}(\varphi)$. An Hintikka tree satisfies the *unique Δ-path property* if it fulfils the following additional condition: there exists a mapping $\rho : [k]^* \to (\text{cl}_\Delta(\varphi) \times [k]^*) \cup \{\bot\}$, such that for all $x \in [k]^*$: If $\Delta\mathcal{A}\psi \in t(x)$ then, for some witnessing $\mathcal{A}$-path $x_0, x_1, x_2, \ldots$ (starting in x), we have $\rho(x_i) = (\langle\mathcal{A}\rangle\psi, x)$ for all $i \geq 1$.

The unique diamond path property for Hintikka trees for formulas in recursive Δ-PDL is the same as the one given for recursive PDL in Definition 11.

One easily shows that (adapted versions of) Propositions 8, 10, and 12 still hold.

Proposition 23 *Let φ be a formula of recursive Δ-PDL. Then the following*

are equivalent.

(1) Formula φ is satisfiable.
(2) Formula φ has a tree model.
(3) There is a Hintikka tree for φ.
(4) There is a k-ary Hintikka tree for φ with both the unique diamond path property and the unique Δ-path property, where $k = 2^{|\mathrm{cl}(\varphi)|} \cdot n \cdot 2r$ with $r = \max(|\mathrm{cl}_\diamond(\varphi)|, |\mathrm{cl}_\Delta(\varphi)|)$ and n the number of atomic programs.

Proof. In the following, we only indicate the differences with the recursive PDL case.

To show equivalence between (1) and (2) one constructs from a model of φ a tree model exactly as for Proposition 8. The only additional step in the proof is to deal with Δ and $\neg\Delta$-formulas. This is easily done by noting that both models have the "same" witnessing paths.

To show equivalence between (2) and (3) one reasons as in the proof of Proposition 10. The only difference is that $\mathrm{cl}(\varphi)$ may now contain $\Delta/\neg\Delta$-formulas, but the formal construction of the Hintikka tree is the same. To prove that it is indeed a Hintikka tree, one follows the same proof and again argues that a witnessing path for some Δ-formula can be found in the Hintikka tree if and only if it exists in the original tree model.

Finally, to show the equivalence between (3) and (4) one considers the same construction as the one in Proposition 12 except that now one sets $r = \max(|\mathrm{cl}_\diamond(\varphi)|, |\mathrm{cl}_\Delta(\varphi)|)$. From a Hintikka tree t one gets a new Hintikka tree t' which has the unique diamond path property (as shown by Proposition 12). To prove that it also enjoys the unique Δ-path property, one reasons exactly as for the unique diamond path property except that $\mathrm{cl}_\diamond(\varphi)$ is now replaced by $\mathrm{cl}_\Delta(\varphi)$ in the construction. □

Then one can construct a VPTA that accepts all trees that have the Δ-property and unique Δ-paths. This construction is similar to the one of the diamond automaton and results in a Büchi VPTA of size linear in the size of the given formula φ.

For the $\neg\Delta$-property one can proceed in a similar way as for the box property. One defines the word language $L_{\neg\Delta}$ corresponding to L_{box} and shows that this language can be accepted by a deterministic VPA. The main difference here is that instead of obtaining a reachability VPA for the complement of $L_{\neg\Delta}$ we obtain a nondeterministic Büchi VPA. Hence, to get a deterministic VPA for $L_{\neg\Delta}$ we have to use a stair parity condition (Theorem 2). All this results in the following lemma.

Lemma 24 *For every recursive Δ-PDL formula φ there is a stair parity* VPTA *of size exponential in the size of φ accepting the unique diamond path and unique Δ-path Hintikka trees of φ.*

Proof. In Section 4 we have already shown how to build a VPTA accepting the unique diamond path Hintikka trees of a given recursive PDL formula φ. Starting now with a recursive Δ-PDL formula, one additionally has to deal with Δ and $\neg\Delta$-formulas.

Let us first explain how to build a Büchi VPTA that accepts all trees that have the Δ-property and unique Δ-path property. The construction is a slight adaptation of the diamond automaton.

The control state of the Δ-automaton stores the following informations:

- A Δ-formula $\Delta\mathcal{A}$ currently checked or $\bot$ if nothing is checked.
- If some formula $\Delta\mathcal{A}$ is being checked, a control state of $\mathcal{A}$ is stored (and stack information from $\mathcal{A}$ will be encoded in the stack of the Δ-automaton).

Initially, no formula is checked. The Δ-automaton reads the labelling $t(x)$ of the current node x. If it contains some Δ-formula, it will go for each of these formulas in a different branch of the tree where it checks this formula. If the automaton was already checking for a Δ-formula, it keeps looking for its validation by choosing yet another branch. As the tree should satisfy the unique Δ-path property, a validation of the Δ-formulas can be found in this way.

The simulation of $\mathcal{A}$ on the path guessed by the Δ-automaton is handled exactly as the simulation in the diamond automaton (Section 4.2). The only difference is that this simulation never ends (we are now checking for a formula involving an ω-program). The Büchi acceptance condition is defined as follows: if no formula is checked, then the Δ-automaton is in a final state, and if some formula $\Delta\mathcal{A}$ is checked, the Δ-automaton goes into a final state if and only if the currently simulated state of automaton $\mathcal{A}$ is final. Then it is easily seen that the Δ-automaton accepts the desired set of trees.

Now, let us consider the case of the $\neg\Delta$-property. As the box property, it is actually a condition on the paths through the tree: one can define a language $L_{\neg\Delta} \subseteq (\Sigma_\varphi \times [k])^\omega$ such that $T_{\neg\Delta} = \text{Trees}(L_{\neg\Delta})$, where $T_{\neg\Delta}$ denotes the set of all trees satisfying the $\neg\Delta$-property. We now define $L_{\neg\Delta}$ and then show that it can be accepted by a deterministic stair parity VPA that we call $\neg\Delta$-*automaton*.

For each word $w \in (\Sigma_\varphi \times [k])^\omega$ there exists a tree $t \in \mathcal{T}_{k,\Sigma_\varphi}$ and a path π such that $w = w^t_\pi$. Then w is in $L_{\neg\Delta}$ if this t satisfies the $\neg\Delta$ property on π: for all

$x \in \pi$, $\neg\Delta\mathcal{A} \in t(x)$ if and only if the suffix of π starting at x is not an $\mathcal{A}$-path.

It is not difficult to see that $t \in \mathcal{T}_{k,\Sigma_\varphi}$ indeed satisfies the $\neg\Delta$-property if and only if all its paths are in $L_{\neg\Delta}$. Hence, by Remark 5, to construct a VPTA for $T_{\neg\Delta}$ it is sufficient to construct a deterministic VPA for $L_{\neg\Delta}$.

Now let $\psi_1, \ldots, \psi_m$ be an enumeration of all $\neg\Delta$-formulas $\psi_i = \neg\Delta\mathcal{A}_i \in \mathrm{cl}(\varphi)$. We show how to construct a visibly pushdown automaton for the complement $\overline{L}_{\neg\Delta}$ of $L_{\neg\Delta}$, and we conclude using closure of visibly pushdown languages under complementation.

First note that $\overline{L}_{\neg\Delta} = \bigcup_{i=1}^{m} \overline{L}_i$, where $\overline{L}_i$ is the set of all words describing a path that violates the $\neg\Delta$-condition for ψ_i. For every i, $\overline{L}_i$ is accepted by a VPA $\mathcal{B}_i$ equipped with a Büchi condition as follows.

For an input word $w = (C_0, d_0)(C_1, d_1)\cdots$ with $C_j \in \Sigma_\varphi$ and $d_j \in [k]$ the VPA $\mathcal{B}_i$ guesses a suffix $(C_j, d_j)(C_{j+1}, d_{j+1})\cdots$ with $\psi_i \in C_j$, and verifies that it corresponds to an $\mathcal{A}_i$-path. The simulation is realised as explained in the proof of Lemma 14, and the only difference is that the simulation never ends as we are dealing now with an ω-program. The Büchi acceptance condition of $\mathcal{B}_i$ is inherited from the one of $\mathcal{A}_i$.

Note that the size of $\mathcal{B}_i$ is linear in the size of $\mathcal{A}_i$. Taking the union of these VPAs one obtains a Büchi VPA $\mathcal{B}$ for $\overline{L}_{\neg\Delta}$. Determinising and then complementing $\mathcal{B}$ yields a stair parity VPA for $L_{\neg\Delta}$ that is of size exponential in $\mathcal{B}$ (note that only determinisation cost an exponential blow up) and thus also exponential in the size of φ.

Now, to conclude the proof, one only needs to consider the automaton obtained by taking the product of the local automaton, the box automaton, the diamond automaton, the Δ-automaton and the $\neg\Delta$-automaton. □

Finally, one has to check emptiness for a stair parity VPTA, which can be done in exponential time (Theorem 4).

Theorem 25 *Given a recursive Δ-PDL formula, one can decide in doubly exponential time whether it is satisfiable.*

As recursive Δ-PDL is an extension of recursive PDL we also have the lower bound.

Corollary 26 *The satisfiability problem for recursive Δ-PDL is 2-*EXPTIME*-complete.*

7 Conclusion

Using visibly pushdown automata we have defined recursive PDL (and recursive Δ-PDL to deal with infinite computations) as an extension of regular PDL that allows to capture the behaviour of recursive programs. The result on satisfiability for these logics subsumes all known decidable extensions of PDL with context-free programs. Further, we have established a 2-ExpTime lower bound for a large class of context-free extensions of PDL.

Our comparison to the logic VP-μ [1] (that combines μ-calculus and visibly pushdown automata) shows that, even though recursive PDL is weaker in expressive power, all specifications of programs presented in [1] can also be captured in recursive PDL or recursive Δ-PDL.

For further research, a more detailed analysis of the expressive power of recursive PDL would be interesting, for example a comparison with μ-calculus using relational fixed points [18]. The latter allows to express the formula $\langle a^{\#}; p_0?; b^{\#}\rangle p_1$ of recursive PDL as $\mu R.((p?; a; R; b) \cup (\neg p)?)$ (for a binary relation symbol R). Another possible direction for future research is to combine visibly pushdown automata with the game logic of Parikh [17].

References

[1] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 153–165. ACM, 2006.

[2] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.

[3] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 2004)*, pages 202–211. ACM, 2004.

[4] R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Proceedings of Computer Aided Verification, 18th International Conference (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2006.

[5] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[6] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[7] D. Harel and M. Kaminsky. Strengthened results on nonregular PDL. Technical Report MCS99-13, Weizmann Institute of Science, Faculty of Mathematics and Computer Science, 1999.

[8] D. Harel and E. Singerman. More on nonregular PDL: Expressive power, finite models, fibonacci programs. In *Proceedings of the 3rd Israeli Symposium on the Theory of Computing and Systems (ISTCS)*, 1995.

[9] D. Harel. Dynamic logic. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic, Volume II*, pages 496–604. D. Reidel Publishers, 1984.

[10] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

[11] D. Harel, A. Pnueli, and J. Stavi. Propositional dynamic logic of nonregular programs. *Journal of Computer and System Sciences*, 26(2):222–243, 1983.

[12] D. Harel and D. Raz. Deciding properties of nonregular programs. *SIAM Journal on Computing*, 22(4):857–874, 1993.

[13] T. Koren and A. Pnueli. There exit decidable context free propositional dynamic logics. In *Proceedings of Logics of Programs, Workshop, Carnegie Mellon University*, pages 290–312, 1983.

[14] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. The MIT Press, 1990.

[15] C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *Proceedings of the 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS 2004)*, volume 3328 of *Lecture Notes in Computer Science*, pages 408–420. Springer, 2004.

[16] R. McNaughton. Parenthesis grammars. *Journal of the ACM*, 14(3):490–500, July 1967.

[17] R. Parikh. The logic of games an its applications. *Annals of discrete mathematics*, 24:111–140, 1985.

[18] D. Park. Finiteness is μ-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.

[19] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proceedings of the 21st Symposium on Principles of Database Systems (PODS'02)*, pages 53–64. ACM, 2002.

[20] R. S. Streett. Propositional dynamic logic of looping and converse is elementary decidable. *Information and Control*, 54:121–141, 1982.

[21] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer, 1997.

[22] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logic of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.