Propositional Dynamic Logic with Recursive Programs

Christof Löding[1] and Olivier Serre[2] *

[1] RWTH Aachen, Germany
[2] LIAFA, Université Paris VII & CNRS, 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.

1 Introduction

Propositional Dynamic Logic (PDL) is a modal logic that was introduced by Fischer and Ladner in [5] to capture the behaviour of programs. 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 of a finite automaton) over the set of atomic programs and tests (where tests correspond to formulas). As shown in [5], satisfiability is decidable for PDL. Proofs for this result usually rely on model-theoretic properties of PDL, e.g., the small model property and the tree model property.

In order to capture more complex programs, several extensions of PDL have been considered. One can allow new programs operators as, e.g., *converse* [14], or consider an intersection operator on programs to express concurrency properties. Recently, Lutz has shown that PDL with both intersection and converse is decidable [11], extending a difficult result from Danecki [4]. One of the main difficulties when considering such extensions is that they do no longer have the tree model property.

Other extensions use non-regular programs to capture recursive behaviours [8, 9, 6]. Consider the following recursive program [7]:

* Supported by the European Community Research Training Network "Games and Automata for Synthesis and Validation" (GAMES). Most of this work was done when the second author was a postdoctoral researcher at RWTH Aachen.

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

The set of 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 hence cannot be captured by standard PDL. To overcome this weakness various extensions of PDL by sets of non-regular programs have been considered.

For a class $\mathcal{C}$ of languages over the set of atomic programs as letters, one can distinguish the weak and the strong extension ([6]) of PDL by programs in $\mathcal{C}$. The weak extension allows formulas $\langle\alpha\rangle\varphi$ where α is (a placeholder for) some language from $\mathcal{C}$, whereas in the strong extension the set of atomic programs is augmented by symbols for the languages from $\mathcal{C}$ (which can then be used in the regular expressions). For the weak extension of PDL by the class of context-free languages satisfiability is easily seen to be undecidable. Nevertheless, several strong extensions by single context-free languages are known to be decidable, e.g., the one by $\{a^ib^i \mid i \geq 0\}$, even if they no longer have the finite model property. Surprisingly, the weak extension of PDL with $\{a^iba^i \mid i \geq 0\}$ leads to undecidability. In order to establish the borderline between decidable and undecidable extensions of PDL by non-regular languages, restrictions on pushdown automata (and hence subclasses of context-free languages) have been considered: the largest class of languages, for which the strong extension of PDL is decidable, is the one of deterministic semi simple-minded languages considered in [6]. A pushdown automaton is semi simple-minded if the input letter determines whether to execute a pop, no stack operation, or a push according to a fixed partition of the input alphabet. In case of a push also the stack symbol to be pushed is fixed by the input letter.

A related but stronger subclass of context-free languages that has recently been defined in [3] is the class of visibly pushdown languages. The definition of visibly pushdown automata (VPAs) is the same as for semi simple-minded pushdown automata except that the stack symbol that is pushed may also depend on the current control state and not only on the input letter. It is not difficult to see that this additional freedom indeed allows to define more languages.

We define *recursive PDL* by replacing regular expressions as the formalism to define programs in PDL by VPAs. This extension of PDL with VPAs is more general in two senses than the strong extension of PDL with (semi) simple-minded pushdown automata as considered in [8] and [6]: the model of VPA is more expressive than the model of (semi) simple-minded pushdown automaton, and we do not just augment the set of atomic programs by placeholders for VPA languages (as in the strong extension) but use VPAs as formalism for defining programs. This second property allows not just to use VPAs over atomic programs but also to use tests inside the VPAs.

Our main result is that satisfiability for recursive PDL is decidable in doubly exponential time. To our knowledge, this captures all previous known results concerning decidable extensions of PDL using context-free languages. Futhermore, our proofs are simpler and less technical than the ones in [8, 6] because we can use general resutls and constructions from the theory of VPAs.

The remainder of the paper is organised as follows. In Section 2 we give the basic definitions and results concerning PDL and VPAs, and we define recursive PDL. In Section 3 we prove the decidability of the satisfiability problem for recursive PDL, and in Section 4 we extend the results to infinite computations.

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.

Formulas of propositional dynamic logic [5] 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*.
(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.

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 $\langle A_c, A_r, A_{\text{int}} \rangle$ 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 $\perp$ and $\delta \subseteq (Q \times A_c \times Q \times (\Gamma \setminus \{\perp\})) \cup (Q \times A_r \times \Gamma \times Q) \cup (Q \times A_{\text{int}} \times Q)$ is the transition relation.

A *configuration* of $\mathcal{A}$ is a pair $(q, \sigma) \in Q \times (\Gamma \setminus \{\perp\})^* \perp$ of a state q and a *stack content* σ. Note that the symbol $\perp$ 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 $\sigma = \sigma' = \perp$ and there is a transition $(q, a, \perp, q') \in \delta$.

For a finite word $u = a_0 a_1 \cdots a_n$ 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 $q_0 \in Q_{in}$, $\sigma_0 = \perp$, and $(q_i, \sigma_i) \xrightarrow{a_i} (q_{i+1}, \sigma_{i+1})$ 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 $\mathcal{A}$.

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 ([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 a configuration 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 [10]. 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 ([10]). *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 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 1. *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 $\rho(\varepsilon) \in Q_{\text{in}} \times \{\bot\}$ 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 $d \in A$:

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

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.

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. [15]), the emptiness test for a VPTA is polynomial time equivalent to the problem of determining the winner in a visibly pushdown game ([10]) with a winning condition corresponding 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 3. *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:

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

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

Remark 1. For each deterministic ω-VPA $\mathcal{A}$ over $\Sigma \times [k]$ there exists a VPTA with an acceptance condition of the same type accepting $\mathrm{Trees}(L(\mathcal{A}))$.

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 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 $\langle \Pi_c, \Pi_{int} \cup \mathrm{Test}, \Pi_r \rangle$ 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 1. Consider the set of atomic programs given by $\widetilde{\Pi} = \langle \{c_0, c_1\}, \emptyset, \{r_0, r_1\} \rangle$ 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 $\{c_1^k r_1^k \mid k > 0\}$, and
– $\varphi = \langle \mathcal{A} \rangle p_1$ where $\mathcal{A}$ accepts the language $\{((\psi?)c_0)^k r_0^k \mid k > 0\}$.

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 $M, s \models \varphi$.

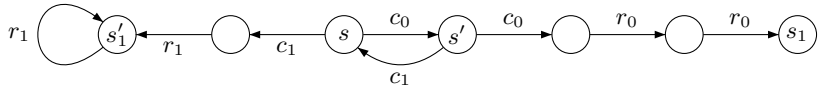

Fig. 1. A model (with s as initial state) for the formula from Example 1

Note that this extension of PDL with VPAs is more general in two senses than the extension of PDL with (semi) simple-minded pushdown automata considered in [8] and [6]. First, VPAs are more expressive than semi simple-minded pushdown automata as, for example, witnessed by the language containing the words of the form $c^n c^m r_1^m c^\ell r_2^\ell r_1^n$ (for c being a call and r_1, r_2 being returns). The proof of this is straightforward because a semi simple-minded pushdown automaton with only one call symbol can only use a single stack symbol.

Second, VPAs can be nested in recursive PDL by using tests as in Example 1. If we view VPAs as descriptions of recursive procedures, then this nesting allows, for example, not only to test properties on entering and exiting a procedure but also to relate these properties to tests that are "launched" inside the execution of the procedure. A simple example for such a formula is $\langle p_1?; c^n; p_2?; r^n; p_3 \rangle \varphi$, where the numbers of calls before, and returns after the test $p_2?$ have to be the same. This is not possible in the extensions of PDL considered in [8, 6], where the non-regular languages that are used as programs are languages over atomic programs only.

3 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 [16] and [8]. 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 adaptions from [8], 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 all atomic propositions from $\mathbb{P}$ 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 1. *A formula of recursive PDL is satisfiable if and only if it has a tree model.*

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 1. *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 $t(x) \neq \bot$, then $\psi \in t(x)$ if and only if $\neg\psi \notin t(x)$ for all $\psi \in \mathrm{cl}(\varphi)$.*
2. *If $\psi_1 \wedge \psi_2 \in \mathrm{cl}(\varphi)$, then $\psi_1 \wedge \psi_2 \in t(x)$ if and only if $\psi_1, \psi_2 \in t(x)$.*
3. (Diamond property) *$\langle\mathcal{A}\rangle\psi \in t(x)$ 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 $\psi \in t(y)$.*
4. (Box property) *$\neg\langle\mathcal{A}\rangle\psi \in t(x)$ if and only if $\psi \notin t(y)$ for all $y \in [k]^*$ 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 3 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 2. *Let φ be a formula of recursive PDL. There is a Hintikka tree for φ if and only if φ has a tree model.*

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 [8]. In the definition from [8] 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 2. *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 3. *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 $k = 2^{|\mathrm{cl}(\varphi)|} \cdot n \cdot 2r$.*

We now show how to build a Büchi VPTA accepting exactly the k-ary unique diamond path Hintikka trees for φ. Together with Theorem 3 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 [8]. 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 1. The second automaton called *box automaton* accepts all trees satisfying the box property (see Definition 1). The third automaton called *diamond automaton* accepts all trees satisfying the diamond property (see Definition 1) and the condition of Definition 2.

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 1. 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 1. *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 1.*

Box automaton. We now construct a VPTA accepting those trees from $\mathcal{T}_{k,\Sigma_\varphi}$ that satisfy the box property from Definition 1. 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 1, to construct a VPTA for T_{box} it is sufficient to construct a deterministic VPA for L_{box}.

Lemma 2. *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 $\bot$ 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 1) 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 1 we directly get the following result.

Lemma 3. *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 2.

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 $\perp$ 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 4. *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 2.*

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 5. *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 3 we deduce the decidability of the satisfiability problem for recursive PDL formulas.

Theorem 4. *Given a recursive PDL formula, one can decide in doubly exponential time whether it is satisfiable.*

We leave open the question whether this complexity is optimal. A singly exponential lower bound directly follows from the one for standard PDL [5].

4 Extension to Infinite Computations

In [14] 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 $s \in R_\omega(\mathcal{A})$.

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*. One easily shows that (adapted versions of) Propositions 1, 2, and 3 still hold.

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 6. *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.*

Finally, one has to check emptiness for a stair parity VPTA, which can be done in exponential time (Theorem 3).

Theorem 5. *Given a recursive Δ-PDL formula, one can decide in doubly exponential time whether it is satisfiable.*

Again, we leave open the question whether this complexity is optimal.

5 Conclusion

Using visibly pushdown automata we have defined recursive PDL as an extension of regular PDL that allows to capture the behaviour of recursive programs. The result on the satisfiability of this logic subsumes all known decidable extensions of PDL with context-free programs. Comparisons of recursive PDL with μ-calculus using relational fixed points and with visibly pushdown μ-calculus would be interesting. The first one [13] allows to capture the example from the introduction using the formula $\mu R.((p?; a; R; b) \cup (\neg p)?)$ (for a binary relation symbol R), while the second one [1] embeds in the modal μ-calculus the formalism of visibly pushdown automata. Another possible direction for future research is to combine visibly pushdown automata with the game logic of Parikh [12].

References

1. R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *Proceedings of POPL'06*. To appear.
2. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004.
3. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of STOC'04*, pages 202–211. ACM, 2004.
4. R. Danecki. Nondeterministic propositional dynamic logic with intersection is decidable. In *Proceedings of the 5th Symposium on Computation Theory*, volume 208 of *LNCS*, pages 34–53. Springer, 1984.
5. M.J. Fischer and R.E. Ladner. Propositional dyncamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.
6. D. Harel and M. Kaminsky. Strengthened results on nonregular PDL. Technical Report MCS99-13, Weizmann Institute of Science, 1999.
7. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.
8. D. Harel and D. Raz. Deciding properties of nonregular programs. *SIAM Journal on Computing*, 22(4):857–874, 1993.
9. D. Harel and E. Singerman. More on nonregular PDL: Expressive power, finite models, fibonacci programs. In *ISTCS: 3rd Israeli Symposium on the Theory of Computing and Systems*, 1995.
10. C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *Proceedings of FST&TCS'04*, volume 3328 of *LNCS*, pages 408–420. Springer, 2004.
11. C. Lutz. PDL with intersection and converse is decidable. In *Proceedings of CSL'05*, volume 3634 of *LNCS*, pages 413–427. Springer, 2005.
12. R. Parikh. The logic of games an its applications. *Annals of discrete mathematics*, 24:111–140, 1985.
13. D. Park. Finiteness is μ-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.
14. R. Streett. Propositional dynamic logic of looping and converse is elementary decidable. *Information and Control*, 54:121–141, 1982.
15. W. Thomas. Languages, automata, and logic. In *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer, 1997.
16. M. Vardi and P. Wolper. Automata-theoretic techniques for modal logic of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.