Formula Synthesis in Propositional Dynamic Logic with Shuffle

Sophie Pinchinat[1], **Sasha Rubin**[2], **François Schwarzentruber**[1]

[1]IRISA/Univ Rennes, France
[2]The university of Sydney, Australia
sophie.pinchinat@irisa.fr, sasha.rubin@sydney.edu.au, francois.schwarzentruber@ens-rennes.fr

Abstract

We introduce the formula-synthesis problem for Propositional Dynamic Logic with Shuffle (PDL$^{||}$). This problem, which generalises the model-checking problem againsts PDL$^{||}$ is the following: given a finite transition system and a regular term-grammar that generates (possibly infinitely many) PDL$^{||}$ formulas, find a formula generated by the grammar that is true in the structure (or return that there is none). We prove that the problem is undecidable in general, but add certain restrictions on the input structure or on the input grammar to yield decidability. In particular, we prove that (1) if the grammar only generates formulas in PDL (without shuffle), then the problem is EXPTIME-complete, and a further restriction to linear grammars is PSPACE-complete, and a further restriction to non-recursive grammars is NP-complete, and (2) if one restricts the input structure to have only simple paths then the problem is in 2-EXPTIME. This work is motivated by and opens up connections to other forms of synthesis from hierarchical descriptions, including HTN problems in Planning and Attack-tree Synthesis problems in Security.

Introduction

Dynamic Logics, such as Propositional Dynamic Logic (PDL) and its extension PDL$^{||}$ with the shuffle operator $||$, were introduced to formally reason about the dynamics of possibly nondeterministic programs (Fischer and Ladner 1979; Abrahamson 1980; Göller 2008). In these logics, formulas are interpreted in a graph-like structure, called a *state-transition system*, whose vertices are states and edges are labeled by actions, reflecting program instruction effects. For instance, the PDL formula $\langle (a_1)^*; a_2\rangle p$ expresses that some execution of the nondeterministic program "`do` a_1 `some number of times, and then do` a_2" results in a state of the state-transition system that satisfies the atomic property p.

In this work, we introduce the *formula-synthesis problem for* PDL$^{||}$: given a finite state-transition system and a fintely-presented mechanism that generates (possibly infinitely many) PDL$^{||}$ formulas, find a generated formula that is true in the state-transition system (or return that there is none).

The mechanism we use to specify the set of formulas are *regular term-grammars*: these are context-free grammars that generate syntactic terms over a ranked alphabet, such as logical formulas where the alphabet contains logical operators (for instance, the conjunction symbol has rank 2), and that have nice closure properties we will make use of.

Our motivation for using regular-tree grammars stems from their ability to provide a *hierarchical* description of the input set of PDL$^{||}$ formulas, and more specifically of the program modalities occurring in the formulas, thus enabling to relate the formula-synthesis problem for PDL$^{||}$ to other synthesis problems where hierarchical features are essential. We discuss such connections in the related work section at the end of the paper. In particular, we show that the formula-synthesis problem for PDL$^{||}$ is a natural abstraction of certain aspects of Hierarchical Task Network problems in Planning (Georgievski and Aiello 2015) and Attack-tree Synthesis in Security (Wideł et al. 2019). For instance, the decomposition patterns of compound tasks into sequences of actions one finds in the framework of Hierarchical Task Networks for planning scenarios can be simulated by grammar rules on the program modalities in PDL$^{||}$, while the state-transition system can represent the planning domain. Thus in our setting, term-grammars replace methods in task-networks, and the formula-synthesis problem for PDL$^{||}$ replaces the Hierarchical Task Network problem.

We remark that the model-checking problem against PDL$^{||}$ is a special case of the formula synthesis problem for PDL$^{||}$ in which the grammar generates a single formula.

Technically, our contributions are as follows:

1. We provide a crisp formalization of the formula synthesis problem for Propositional Dynamic Logic with Shuffle, which we call SYNTHPDL$^{||}$.
2. We prove that SYNTHPDL$^{||}$ is undecidable by reducing the intersection problem for context-free word-grammars, known to be undecidable, see, e.g., (Hopcroft, Motwani, and Ullman 2003). This approach was used earlier for a similar undecidability result in HTN planning (Erol, Hendler, and Nau 1996).

   This motivates the study of special cases of the problem, where restrictions are taken over the input structure or the input grammar. More precisely, we establish that:

3. If one restricts to PDL input grammars (i.e., without shuffle), then this subproblem, called SYNTHPDL, is EXPTIME-complete. The proof proceeds by building a term-grammar that generates exactly the PDL formulas that are true in the input transition system. For the lower bound we reduce from the non-universality problem for non-deterministic tree-automata.
4. If one restricts the input systems to have finitely many paths, then SYNTHPDL$^{||}$ is in 2-EXPTIME, while the lower bound is left open. The proof proceeds by building a term-grammar that generates exactly the PDL$^{||}$ formulas that are true in the input transition system.
5. If one restricts to linear PDL input grammars (each rule derives a term with at most one non-terminal symbol), then SYNTHPDL is PSPACE-complete.
6. If one restricts to non-recursive PDL grammars (such a grammar generates finitely many formulas), then SYNTHPDL$^{||}$ is in EXPSPACE and SYNTHPDL is in PSPACE – the lower bound is still an open question.
7. If one restricts to PDL grammars that are linear and non-recursive, then SYNTHPDL is NP-complete.

Preliminary Notions

In this section, we recall useful notions in formal language theory and fix our notations.

Binary relations We use standard notations for binary relations over set D. Given two binary relations R_1 and R_2, we let $R_1 \circ R_2 := \{(u,v) |$ there exists $w (u,w) \in R_1$ and $(w,v) \in R_2\}$, and $R^* := \bigcup_{n \geq 0} R^n$ with R^n defined as follows: $R^0 = \{(u,u) : u \in D\}$, $R^{n+1} = R^n \circ R$ for $n > 0$.

Finite-word languages Let $\Sigma = \{\ell, \ell', \ell_1, \ldots\}$ be a finite alphabet. A *word* over Σ is a finite sequence $\alpha = \ell_1 \ell_2 \ldots \ell_n$ of elements of Σ. We denote by ϵ the empty word. We recall classic formal language operations. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages over Σ. We define the *concatenation* of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ by $\mathcal{L}_1\mathcal{L}_2 := \{\alpha\beta \,|\, \alpha \in \mathcal{L}_1$ and $\beta \in \mathcal{L}_2\}$. Next, we recall the *shuffle* of two languages as the language obtained by shuffling (or *interleaving*) each word of $\mathcal{L}_1$ with each word of $\mathcal{L}_2$. Formally, the *shuffle* $\alpha \,||\, \beta$ of two words α and β is defined by induction as follows: $\epsilon \,||\, \alpha = \alpha \,||\, \epsilon = \{\alpha\}$ and $(\ell\alpha) \,||(\ell'\beta) = \ell(\alpha \,||\, \ell'\beta) \cup \ell'(\ell\alpha \,||\, \beta)$. For example, $ab \,||\, cd \ni abcd, acbd, cadb$, etc. The *shuffle* of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ is then defined by $\mathcal{L}_1 \,||\, \mathcal{L}_2 := \bigcup_{\alpha \in \mathcal{L}_1, \beta \in \mathcal{L}_2} \alpha \,||\, \beta$.

Regular tree-grammars We now recall the basic notions (the reader may refer to (Comon et al. 2005, Subsection 2.1, p. 51) for more details). A *ranked alphabet* is a set Σ and associated ranks (arities) of the symbols. The rank of $f \in \Sigma$ is denoted $rk(f)$. The rank of standard function symbols are obvious, e.g., the rank of $\wedge$ is 2, and the rank of $\mathbf{true}$ is 0. Note that we sometimes call symbols of rank 0 *constants*.

The set of *terms*, also known as *trees*, over Σ is the smallest set of expressions satisfying the following: all constants are terms, and if $t_1, \cdots, t_k$ are terms and $rk(f) = k$ then $f(t_1, \cdots, t_k)$ is a term. A *language* over Σ is a set of terms.

A *regular tree-grammar* over Σ is a tuple

$$\mathfrak{G} = (N, \mathbf{S}, \Sigma, \mathcal{R})$$

where:

- N is a finite non-empty set of *non-terminals*,
- $\mathbf{S} \in N$ is the *axiom*,
- Σ, a ranked alphabet, is the set of *terminals*, and
- $\mathcal{R}$ a finite set of *production rules* of the form $\mathbf{X} \to t$ where $\mathbf{X} \in N$ and t is a term over $\Sigma \cup N$ where nonterminals have rank 0.

For terms t, t' over the ranked alphabet $\Sigma \cup N$, write $t \Rightarrow t'$ if t' is the result of replacing a non-terminal $\mathbf{X}$ in t by the right hand side of some rule $\mathbf{X} \to t''$. Write $\Rightarrow^*$ for the reflexive and transitive closure of $\Rightarrow$. The *language generated by* $\mathfrak{G}$ is the set of terms t over Σ such that $\mathbf{S} \Rightarrow^* t$. A set of terms is *regular* if it is the language generated by some regular tree-grammar $\mathfrak{G}$. The *size* of a grammar $\mathfrak{G}$ is the sum of the length of all the rules in $\mathcal{R}$.

We use the following algorithmic facts about regular tree-grammars based on (Comon et al. 2005) that says that the regular-tree grammars are efficiently closed under intersection and non-emptiness testing.

Proposition 1. *1. There is a polynomial time algorithm that given two regular tree-grammars $\mathfrak{G}_1$ and $\mathfrak{G}_2$ builds a regular tree-grammar $\mathfrak{G}$ that generates the intersection of the languages generated by $\mathfrak{G}_1$ and $\mathfrak{G}_2$.*

2. There is a polynomial time algorithm that given a regular tree-grammar $\mathfrak{G}$ decides if the language it generates is non-empty, and in this case returns a term generated by $\mathfrak{G}$.

The Logic PDL$^{||}$

We recall the extension of Propositional Dynamic Logic as considered in (Mayer and Stockmeyer 1996), which offers a way of combining programs by interleaving that we call *shuffle* in this paper.

In the following, we fix a set PROP of atomic propositions and a set ACTIONS of atomic programs (these will be finite sets for all practical purposes). The syntax of the logic PDL$^{||}$ is given as follows:

$$\varphi ::= p \mid \mathbf{true} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid [\rho]\varphi$$
$$\rho ::= a \mid \varphi? \mid \rho_1 + \rho_2 \mid \rho_1; \rho_2 \mid \rho* \mid \rho_1 \,||\, \rho_2$$

where p varies over elements of PROP, and a over elements of ACTIONS. Expressions φ are PDL$^{||}$ *formulas*, and expressions ρ are *programs*.[1] We denote by PDL$^{||}$ the set of PDL$^{||}$ formulas, and by $Prog$ the set of programs. We use usual shorthands for formulas, including $\langle\rho\rangle\varphi = \neg[\rho]\neg\varphi$.

The semantics of PDL$^{||}$ is defined over a *transition-system (TS)* over PROP and ACTIONS, that is a structure $\mathfrak{D} = (D, \lambda, \delta)$ consisting of a finite non-empty set D of *states*, a labeling of states by propositions λ : PROP $\to 2^D$, and a transition relation δ : ACTIONS $\to 2^{D \times \text{ACTIONS} \times D}$ such that $\delta(a)$ only contains elements of the form (u, a, v).

[1] Note that we write $\rho*$ instead of ρ^* since we reserve the superscript for the semantics.

In a TS $\mathfrak{D} = (D, \lambda, \delta)$, a *pure transition* is a triple (u, a, v) in $\delta(a)$ for some $a \in$ ACTIONS. Note that the co-domain of δ is the set of pure transitions. A *stuttering* transition is a pair (u, u) where $u \in D$. A *transition* of $\mathfrak{D}$ is a either a pure or a stuttering transition. We will denote by $\Delta_{\mathfrak{D}}$ the set of transitions of $\mathfrak{D}$. For the set $\Delta_{\mathfrak{D}}$, we will use typical elements $\mathtt{t}, \mathtt{t}', \mathtt{t}_1, \ldots$. We then denote by $\Delta^*_{\mathfrak{D}}$ the set of finite sequences over $\Delta_{\mathfrak{D}}$. Given a sequence $\mathtt{t}_1 \ldots \mathtt{t}_n$, we let $first(\mathtt{t}_1 \ldots \mathtt{t}_n) = u$ if $\mathtt{t}_1 = (u, a, v)$ or $\mathtt{t}_1 = (u, u)$, and we define $last(\mathtt{t}_1 \ldots \mathtt{t}_n)$ in a similar way.

There are particular sequences $\mathtt{t}_1\mathtt{t}_2 \ldots \mathtt{t}_n$ called *paths*, that yield the classic notion of path in a transition system, namely a finite sequence π of transitions $\mathtt{t}_1\mathtt{t}_2 \ldots \mathtt{t}_n$ of the form $\mathtt{t}_i = (u_{i-1}, a_i, u_i) \in \delta(a_i)$ for every $i = 1, \ldots n$; thus, in a path, the end state of a transition is equal to the first state of the next transition (if any) along this sequence. We fix a few notations: we denote by $\Pi_{\mathfrak{D}} \subseteq \Delta^*_{\mathfrak{D}}$ the set of paths of $\mathfrak{D}$, by $\Pi^{(u,v)}_{\mathfrak{D}} \subseteq \Pi_{\mathfrak{D}}$ the set of paths from u to v, and by $\Pi^u_{\mathfrak{D}} \subseteq \Pi_{\mathfrak{D}}$ the set of paths that start from state u.

To obtain the semantics of PDL$^{||}$, the labeling function λ and the transition relation δ are extended by simultaneous induction to $\lambda : \text{PDL}^{||} \to 2^D$ and $\delta : Prog \to 2^{\Delta^*_{\mathfrak{D}}}$ as follows. For programs define:

- $\delta(\rho_1 + \rho_2) = \delta(\rho_1) \cup \delta(\rho_2)$,
- $\delta(\rho_1; \rho_2) = \delta(\rho_1)\delta(\rho_2)$,
- $\delta(\rho*) = (\delta(\rho))^*$,
- $\delta(\rho_1 \,||\, \rho_2) = \delta(\rho_1) \,||\, \delta(\rho_2)$,
- $\delta(\varphi?) = \{(u, u) \in D^2 \mid u \in \lambda(\varphi)\}$,

and for formulas define:

- $\lambda(\mathbf{true}) = D$,
- $\lambda(\neg\varphi) = D \setminus \lambda(\varphi)$,
- $\lambda(\varphi_1 \wedge \varphi_2) = \lambda(\varphi_1) \cap \lambda(\varphi_2)$,
- $\lambda([\rho]\varphi) = \{u \in D \mid \forall \pi \in \Pi^u_{\mathfrak{D}} \cap \delta(\rho), last(\pi) \in \lambda(\varphi)\}$,

Write $\mathfrak{D}, u \models \varphi$ if $u \in \lambda(\varphi)$, and say that a program ρ is *executable* in $\mathfrak{D}$ from u if $\mathfrak{D}, u \models \langle\rho\rangle\mathbf{true}$, namely, according to the semantics above, whenever there exists a path π in $\mathfrak{D}$ starting from u (and whose end state satisfies $\mathbf{true}$).

Remark 1. *If we discard the program operator $||$, we obtain the logic* PDL. *Indeed, one can see that the standard semantics of* PDL *(Fischer and Ladner 1979), and the semantics given here inspired from (Mayer and Stockmeyer 1996) coincide. Recall that the standard semantics of* PDL *is over transition systems (D, λ, δ') where λ :* PROP $\to 2^D$ *as in our semantics, but where δ' is a function* ACTIONS $\to 2^{D\times D}$ *that is extended to $\delta' : Prog \to 2^{D\times D}$ instead of to $Prog \to 2^{\Delta^*_{\mathfrak{D}}}$. We can recover this "finitary" semantics by observing that $\delta'(\rho)$ is the set of pairs $(start(\pi), last(\pi))$ for paths $\pi \in \delta(\rho)$. The important difference between* PDL$^{||}$ *and* PDL *is that while co-domain of δ' is always a finite set, this is not the case of δ in general. This makes decision problems for* PDL$^{||}$ *more difficult than for* PDL.

We now show how to use regular-tree grammars to generate sets of PDL$^{||}$ formulas. The set of PDL$^{||}$ formulas can be viewed as terms over the ranked alphabet

$$\Sigma_{\text{PDL}^{||}} := \text{ACTIONS} \cup \text{PROP} \cup \{\mathbf{true}, *, ?, \neg, ;, +, \wedge, [\,], ||\}$$

with the following arities of symbols: symbols in ACTIONS and PROP and the symbol $\mathbf{true}$ have arity 0, the symbols $*$, ?, and $\neg$ have arity 1, and the remaining ones have arity 2. Throughout the paper, we take the convention that a term of the form $[\,](t, t')$ is rather written $[t]t'$ and that we use infix notations for the binary symbols of the alphabet, e.g. the term $||(t, t')$ is written $t \,||\, t'$, so that terms look like PDL formulas and programs.

Example 1. *Terms that describe arbitrary* PDL$^{||}$ *formulas can be generated by the regular tree-grammar $\mathfrak{G} = (\{\boldsymbol{S}, \boldsymbol{P}\}, \boldsymbol{S}, \Sigma_{\text{PDL}^{||}}, \mathcal{R})$ where the axiom symbol $\boldsymbol{S}$ derives the* PDL$^{||}$ *formulas and the nonterminal $\boldsymbol{P}$ derives the* PDL$^{||}$ *programs, and where the set $\mathcal{R}$ of rules is:*

$$\begin{array}{lll}
\boldsymbol{S} \to p & & \\
\boldsymbol{S} \to \mathbf{true} & \boldsymbol{P} \to a & \boldsymbol{P} \to \boldsymbol{P};\boldsymbol{P} \\
\boldsymbol{S} \to \neg\boldsymbol{S} & \boldsymbol{P} \to \boldsymbol{S}? & \boldsymbol{P} \to \boldsymbol{P}* \\
\boldsymbol{S} \to \boldsymbol{S} \wedge \boldsymbol{S} & \boldsymbol{P} \to \boldsymbol{P} + \boldsymbol{P} & \boldsymbol{P} \to \boldsymbol{P} \,||\, \boldsymbol{P} \\
\boldsymbol{S} \to [\boldsymbol{P}]\boldsymbol{S} & &
\end{array}$$

Example 2. *We provide a tree-grammar that generates all formulas of the form $\langle\rho\rangle\varphi$ where ρ is a* PDL$^{||}$*-program consisting only of sequences of atomic actions, and where φ is a goal set of states, characterized by proposition* `goal`*:*

$$\begin{array}{ll}
\boldsymbol{S} \to \langle\boldsymbol{P}\rangle \mathtt{goal} & \\
\boldsymbol{P} \to a & (a \in \text{ACTIONS}) \\
\boldsymbol{P} \to \boldsymbol{P};\boldsymbol{P} &
\end{array}$$

Recall that $\mathfrak{D}, d \models \langle\rho\rangle\varphi$ means there exists a ρ execution from d that reaches a state satisfying φ.

We now define the formula synthesis problem for PDL$^{||}$.

Definition 1 (SYNTHPDL$^{||}$)**.** *The* PDL$^{||}$-formula synthesis problem, *written* SYNTHPDL$^{||}$ *for short, is: given a finite set* PROP *of atomic propositions, a finite set* ACTIONS *of atomic programs, a TS $\mathfrak{D}$ (over* PROP *and* ACTIONS*), a state $d \in D$, a regular tree-grammar $\mathfrak{G}$ generating a set of* PDL$^{||}$ *formulas, decide if there exists φ generated by $\mathfrak{G}$ such that $\mathfrak{D}, d \models \varphi$, and find one if there is.*

We let SYNTHPDL be the subproblem of SYNTHPDL$^{||}$ where the input regular tree-grammar generates a set of PDL formulas, i.e., formulas without the operator $||$.

Note that in case the grammar generates finitely many formulas $\varphi_1, \ldots, \varphi_m$, the formula-synthesis problem reduces to solving the model-checking instance $\mathfrak{D}, d \models \varphi_1 \vee \ldots \vee \varphi_m$. We recall that model-checking against PDL$^{||}$ is PSPACE-complete (Göller 2008, Proposition 2.10), while model-checking against PDL is P-complete (Harel, Kozen, and Tiuryn 2000; Lange 2006).

In the subsequent sections, we analyze SYNTHPDL$^{||}$ and various relevant subproblems, including SYNTHPDL.

Analysis of SYNTHPDL$^{||}$

We present a proof that SYNTHPDL$^{||}$ is undecidable, taking inspiration from the undecidability of the Hierarchical Task Network problem of (Erol, Hendler, and Nau 1996). As a

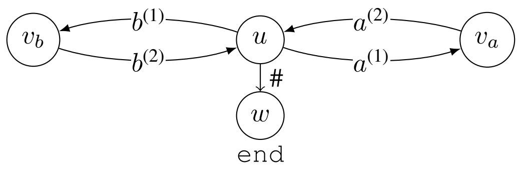

Figure 1: The TS for alphabet $\Gamma = \{a, b\}$.

consequence, the formula-synthesis problem for PDL$^{||}$ does not always reduce to the model-checking problem against PDL$^{||}$.

Theorem 1. SYNTHPDL$^{||}$ *is recursively-enumerable complete (and therefore undecidable), even when restricted to sets of formulas expressing executability of programs with shuffle and concatenation only.*

Proof. The problem is recursively enumerable (RE) since one can enumerate the PDL$^{||}$ formulas φ generated by the input tree-grammar, and for each one effectively check if $\mathfrak{D}, d \models \varphi$.

For RE-hardness, we reduce from the *intersection problem of context-free word grammars (CFGs)* known to be RE-hard, see, e.g., (Hopcroft, Motwani, and Ullman 2003): given two CFGs H_1, H_2 in Chomsky-normal-form with the same terminal alphabet Γ, decide if $L(H_1) \cap L(H_2) \neq \emptyset$.

Recall that a CFG H is in *Chomsky-normal-form* if every rule is of the form $\mathbf{A} \to a$ or $\mathbf{A} \to \mathbf{BC}$, for nonterminals $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and terminal a. Note that, w.l.o.g., we have assumed that the grammars do not generate the empty string, and we can assume that the sets of nonterminals of H_1 and H_2 are disjoint, and that the start symbol of H_i is $\mathbf{Z}_i$.

Given an instance (H_1, H_2) of the intersection problem, we compute an instance $(\mathfrak{D}, d, \mathfrak{G})$ of SYNTHPDL$^{||}$ as follows. The TS $\mathfrak{D} = (D, \lambda, \delta)$ only depends on the alphabet Γ of the CFGs. By letting PROP $:= \{\texttt{end}\}$ and ACTIONS $:= \Gamma^{(1)} \uplus \Gamma^{(2)} \cup \{\#\}$, where $\Gamma^{(1)}$ and $\Gamma^{(2)}$ are disjoint copies of Γ and $\#$ is a fresh symbol, the TS is defined by:

- $D = \{u, w\} \cup \{v_a \mid a \in \Gamma\}$;
- $\lambda(\texttt{end}) = \{w\}$;
- for $a \in \Gamma$, $\delta(a^{(1)}) = \{(u, v_a)\}$ and $\delta(a^{(2)}) = \{(v_a, u)\}$;
- $\delta(\#) = \{(u, w)\}$.

The TS for $\Gamma = \{a, b\}$ is drawn in Figure 1; a typical path from u starts like $(u, a^{(1)}, v_a)$ $(v_a, a^{(2)}, u)$ $(u, b^{(1)}, v_b)$ $(v_b, b^{(2)}, u) \cdots$. The distinguished state d in $\mathfrak{D}$ is u.

Regarding the grammar $\mathfrak{G}$, we first turn word-grammars H_1 and H_2 into tree-grammars $\mathfrak{G}_1$ and $\mathfrak{G}_2$, respectively, which are meant to generate sequential PDL$^{||}$ programs. The tree-grammar $\mathfrak{G}_i$ has:

- terminal symbols $\Sigma_i := \Gamma^{(i)} \uplus \{;\}$ where the symbols in $\Gamma^{(i)}$ have arity 0, and the symbol ";" has arity 2;
- the same non-terminal symbols as H_i;
- every rule obtained by replacing a rule $\mathbf{A} \to a$ in H_i by the rule $\mathbf{A} \to a^{(i)}$, and by replacing a rule $\mathbf{A} \to \mathbf{BC}$ in H_i by the rule $\mathbf{A} \to \mathbf{B}; \mathbf{C}$.

The grammar $\mathfrak{G}$ is obtained by gathering the rules from $\mathfrak{G}_1$ and $\mathfrak{G}_2$, and by adding the rule $\mathbf{S} \to \langle(\mathbf{Z}_1 \,||\, \mathbf{Z}_2); \#\rangle\texttt{end}$, where $\mathbf{S}$ is a fresh axiom symbol.

Then, the grammar $\mathfrak{G}$ generates a formula of the form $\langle((a_1^{(1)}; \ldots; a_k^{(1)}) \,||(b_1^{(2)}; \ldots; b_\ell^{(2)})); \#\rangle\texttt{end}$ if, and only if, $a_1 \ldots a_k \in L(H_1)$ and $b_1 \ldots b_\ell \in L(H_2)$.

Now, $\mathfrak{D}, u \models \langle((a_1^{(1)}; \ldots; a_k^{(1)}) \,||(b_1^{(2)}; \ldots; b_\ell^{(2)})); \#\rangle\texttt{end}$ iff $k = \ell$ and $a_j = b_j$ for all $1 \leq j \leq k$, which entails $a_1 \ldots a_k \in L(H_1) \cap L(H_2)$.

Conversely, from a word $a_1 \ldots a_k \in L(H_1) \cap L(H_2)$, since for $i = 1, 2$, $\mathbf{Z}_i$ derives $a_i^{(1)} \ldots a_i^{(1)}$, axiom $\mathbf{S}$ derives the formula $\varphi_\cap := \langle((a_1^{(1)}; \ldots; a_k^{(1)}) \,||(a_1^{(2)}; \ldots; a_k^{(2)})); \#\rangle\texttt{end}$. Moreover, the sequence of transitions $(u, a_1^{(1)}, v_{a_1})$ $(v_{a_1}, a_1^{(2)}, u) \cdots (u, a_k^{(1)}, v_{a_k})(v_{a_k}, a_k^{(2)}, u)(u, \#, w)$ is a path in $\mathfrak{D}$ that witnesses the statement $\mathfrak{D}, u \models \varphi_\cap$. □

We make some observations about the SYNTHPDL$^{||}$ instances constructed in the proof of RE-hardness. First, the formulas make use of a single shuffle operator. A natural question is to discard it and study the subproblem SYNTHPDL. Second, the finite TS (Figure 1) has infinitely many paths. But what if we only consider DAG[2]-like input transition systems, while keeping the set of candidate formulas infinite? Third, we may consider restrictions on the input grammar, in addition to forbidding the shuffle operator. We explore all of these in the next section.

Decidability

In this section, we explore a broad spectrum of assumptions on the inputs that make our formula-synthesis problem decidable.

PDL Grammars

We consider the case where the input grammar generates PDL formulas only, i.e., there is no occurrence of $||$ in any rule. Recall that SYNTHPDL is this subproblem of SYNTHPDL$^{||}$.

Theorem 2. SYNTHPDL *is* EXPTIME*-complete.*

We establish the upper bound in Proposition 3 by designing a tree-grammar for the set of all PDL formulas satisfied by state d in $\mathfrak{D}$ (preliminary Proposition 2). Our ability to effectively build such a grammar heavily relies on the PDL co-domain-finite semantics δ' for programs (see Remark 1). For the lower bound we reduce from the non-universality of nondeterministic word-automata (NFW) in Proposition 4.

Proposition 2. *For every transition system $\mathfrak{D} = (D, \lambda, \delta)$ and every state $d \in D$, we can build a regular tree-grammar $\mathfrak{G}_{\mathfrak{D},d}$ of size exponential in $\mathfrak{D}$ generating exactly the* PDL *formulas φ such $\mathfrak{D}, d \models \varphi$.*

Proof. Intuitively, the grammar mimicks the inductive definition of λ and of the program semantics δ' of Remark 1. We therefore need a nonterminal $\mathbf{X}$ for each $X \subseteq D$ and a

[2]Directed acyclic graphs, i.e. there is no cycle $(u_1, a_1, u_2), \ldots, (u_{k-1}, a_{k-1}, u_k) \in \lambda$ wth $u_k = u_1$.

nonterminal $\mathbf{Y}$ for each binary relation $Y \subseteq D \times D$, as well as production rules that ensure:

$$\mathbf{X} \text{ derives } \varphi \text{ iff } \lambda(\varphi) = X \quad (1a)$$

$$\mathbf{Y} \text{ derives } \rho \text{ iff } \delta'(\rho) = Y \quad (1b)$$

We therefore let $N_{Form} := \{\mathbf{X} \mid X \subseteq D\}$ and $N_{Prog} := \{\mathbf{Y} \mid Y \subseteq 2^{D\times D}\}$, and $\mathfrak{G}_{\mathfrak{D},d} = (N_{Form} \uplus N_{Prog} \uplus \{\mathbf{S}\}, \mathbf{S}, \Sigma_{\text{PDL}}, \mathcal{R})$ where $\mathcal{R}$ consists of the rules $\mathbf{S} \to \mathbf{X}$ for every $\mathbf{X} \in N_{Form}$ containing d, and of all the following rules, for each $X \subseteq D$ and each $Y \subseteq D \times D$:

$\mathbf{X} \to \mathbf{true}$	
$\mathbf{X} \to p$	if $X = \lambda(p)$
$\mathbf{X} \to \neg\mathbf{X}_1$	if $X = D \setminus X_1$
$\mathbf{X} \to \mathbf{X}_1 \wedge \mathbf{X}_2$	if $X = X_1 \cap X_2$
$\mathbf{X} \to [\mathbf{Y}]\mathbf{X}_1$	if $X = \{u \mid \forall v.(u,v) \in Y \text{ implies } v \in X_1\}$
$\mathbf{Y} \to a$	if $Y = \delta(a)$
$\mathbf{Y} \to \mathbf{X}?$	if $Y = \{(u,u) \mid u \in X\}$
$\mathbf{Y} \to \mathbf{Y}_1 + \mathbf{Y}_2$	if $Y = Y_1 \cup Y_2$
$\mathbf{Y} \to \mathbf{Y}_1; \mathbf{Y}_2$	if $Y = Y_1 \circ Y_2$
$\mathbf{Y} \to \mathbf{Y}_1 *$	if $Y = Y_1^*$

A straightforward induction (on formulas and programs) shows that (1a) and (1b) hold. From those it is immediate that $\mathbf{S}$ derives exactly the formulas φ such that $\mathfrak{D}, d \models \varphi$. We just give some of the cases.

- $\mathbf{X}$ derives p iff $\lambda(p) = X$.
- $\mathbf{X}$ derives $\varphi_1 \wedge \varphi_2$ iff $X = X_1 \cap X_2$ where $\mathbf{X}_i$ derives φ_i for $i = 1, 2$ iff $X = X_1 \cap X_2$ and (IH) $X_i = \lambda(\varphi_i)$ for $i = 1, 2$ iff $\lambda(\varphi_1 \wedge \varphi_2) = \lambda(\varphi_1) \cap \lambda(\varphi_2) = X$.
- $\mathbf{X}$ derives $[\rho]\varphi$ iff $X = \{u \mid (u,v) \in Y \text{ implies } u \in X_1\}$ and $\mathbf{Y}$ derives ρ and $\mathbf{X}_1$ derives φ iff $X = \{u \mid (u,v) \in Y \text{ implies } v \in X_1\}$ and (IH) $Y = \delta'(\rho)$ and (IH) $X_1 = \lambda(\varphi)$ iff $X = \lambda([\rho]\varphi)$.
- $\mathbf{Y}$ derives $\rho_1 + \rho_2$ iff $Y = Y_1 \cup Y_2$ and $\mathbf{Y}_i$ derives ρ_i for $i = 1, 2$ iff $Y = Y_1 \cup Y_2$ and (IH) $\delta'(\rho_i) = Y_i$ for $i = 1, 2$ iff $Y = \delta'(\rho_1) \cup \delta'(\rho_2) = \delta'(\rho_1 + \rho_2)$.

The remaining cases are similar. □

Proposition 3. SYNTHPDL *is decidable, and can be solved in exponential time in the size of the input transition system and polynomial time in the size of the input grammar.*

Proof. We take advantage of the effective construction of $\mathfrak{G}_{\mathfrak{D},d}$ (Proposition 2) to design an algorithm for solving SYNTHPDL.

1. Build the grammar $\mathfrak{G}_{\mathfrak{D},d}$ from the proof of Proposition 2 in time exponential in $|\mathfrak{D}|$.
2. Build a grammar $\mathfrak{G}'$ for the intersection of the languages generated by $\mathfrak{G}$ and $\mathfrak{G}_{\mathfrak{D},d}$ (Proposition 1), of size $poly(|\mathfrak{G}|) \times poly(|\mathfrak{G}_{\mathfrak{D},d}|)$.
3. Decide in polynomial time in $|\mathfrak{G}'|$ (Proposition 1) whether or not the language generated by $\mathfrak{G}'$ is empty. If it is empty, then return "No solution". Otherwise, extract and return a formula generated by $\mathfrak{G}'$.

This completes the proof. □

By reducing the non-universality problem for *(bottom up) nondeterministic finite tree automata (NFT)*, (known to be EXPTIME-complete, see (Löding 2012, p. 91)) to SYNTHPDL, we obtained the following EXPTIME-hardness.

Proposition 4. SYNTHPDL *is* EXPTIME-*hard.*

Since SYNTHPDL is decidable, and because the proof of the undecidabilty of SYNTHPDL$^{||}$ makes use of a single occurrence of the shuffle operator in the generated formulas, there is little hope to find a decidable subproblem in between SYNTHPDL and SYNTHPDL$^{||}$ based on a restriction of the input grammar. Therefore, in the next section, we explore an assumption on the input transition-system side.

DAG-like Transition Systems

Restricting to DAG-like TS amounts to considering TS with finitely many paths, that we call here *finitely-many-path* TS; observe that the TS of Figure 1 is not finitely-many-path.

In this setting, the semantics of programs are bounded, i.e., the image $\delta(\rho)$ of every program ρ is a finite set of sequences, each sequence being a subsequence of one of the finitely many paths in the TS. On this basis, one can effectively build a regular tree-grammar that generates the PDL$^{||}$ formulas true at some fixed $\mathfrak{D}, d$ (see Proposition 5), and thus design an algorithm similar to the one for PDL grammars.

Theorem 3. SYNTHPDL$^{||}$ *restricted to finitely-many-path transition systems is in* 2-EXPTIME.

The remainder of the section is dedicated to the proof that is similar to the one of Proposition 2, but has to exhibit an image-finiteness for the program semantics.

Proposition 5. *Given a transition system $\mathfrak{D} = (D, \lambda, \delta)$ with finitely many paths, and some state $d \in D$, one can effectively build a regular tree grammar $\mathfrak{G}_{\mathfrak{D},d}$ that generates exactly all the* PDL$^{||}$ *formulas φ such that $\mathfrak{D}, d \models \varphi$.*

Proof. Fix a transition system $\mathfrak{D} = (D, \lambda, \delta)$ with finitely many paths, and some state $d \in D$.

Recall that the set of transitions of $\mathfrak{D}$ (pure and stuterring transitions) is denoted by $\Delta_{\mathfrak{D}}$, and that the set of paths $\Pi_{\mathfrak{D}} \subseteq \Delta^*_{\mathfrak{D}}$. Given two transitions $\mathtt{t}, \mathtt{t}' \in \Delta_{\mathfrak{D}}$, we let $\mathtt{t} < \mathtt{t}'$ hold whenever along some path $\pi \in \Pi_{\mathfrak{D}}$, $last(\mathtt{t})$ preceeds $first(\mathtt{t}')$. Observe that $\leq$ is a partial order since $\mathfrak{D}$ is a DAG.

We let $\Delta^{\leq}_{\mathfrak{D}} \subseteq \Delta^*_{\mathfrak{D}}$ be the finite subset of linearly $<$-ordered sequences of transitions. Clearly, every path and every stuttering transition belong to $\Delta^{\leq}_{\mathfrak{D}}$.

We define the grammar $\mathfrak{G}_{\mathfrak{D},d} = (N, \mathbf{S}, \Sigma_{\text{PDL}^{||}}, \mathcal{R})$ that intuitively mimicks the inductive definition of λ and δ.

$N = \{\mathbf{S}\} \uplus N_{Form} \uplus N_{Pairs} \uplus N_{Prog}$ where:

1. N_{Form} contains a non-terminal $\mathbf{X}$ for each $X \subseteq D$;
2. N_{Pairs} contains a non-terminal $\mathbf{Y}$ for each $Y \subseteq D \times D$;
3. N_{Prog} contains a non-terminal $\mathbf{Z}$ for each $Z \subseteq \Delta^{\leq}_{\mathfrak{D}}$.

In the spirit of the proof of Proposition 2, it is possible to construct in time $O(2^{2^{poly(|\mathfrak{D})|}})$ a set of rules $\mathcal{R}$, so that the resulting grammar has size $O(2^{2^{poly(|\mathfrak{D}|)}})$ and satisfies the following lemma.

The constructed grammar garanties the three properties of the following lemma, similar to the properties (1a) and (1b) in Proposition 2.

Lemma 1. *For each* $\boldsymbol{X} \in N_{Form}, \boldsymbol{Y} \in N_{Pairs}, \boldsymbol{Z} \in N_{Prog}$,

$$\boldsymbol{X} \textit{ derives } \varphi \textit{ iff } \lambda(\varphi) = X \quad (2a)$$

$$\boldsymbol{Y} \textit{ derives } \rho \textit{ iff } Y = \{(u, v) \in D^2 \mid \Pi_{\mathfrak{D}}^{(u,v)} \cap \delta(\rho) \neq \emptyset\} \quad (2b)$$

$$\boldsymbol{Z} \textit{ derives } \rho \textit{ iff } Z = \delta(\rho) \cap \Delta_{\mathfrak{D}}^{\leq} \quad (2c)$$

It is clear that Equation (2a) entails Proposition 5 since by definition of the single rule from **S**, it is immediate that **S** derives φ iff $\mathfrak{D}, d \models \varphi$.

Lemma 1 can be proved by induction over the formulas and the programs. ☐

Now, we conclude the proof of Theorem 3 by designing an algorithm along the same lines as the one for Proposition 2. Because the constuction of the grammar $\mathfrak{G}_{\mathfrak{D},d}$ requires $poly(2^{2^{poly(|\mathfrak{D}|)}})$-time, this algorithm runs in double-exponential time, which concludes the proof.

Theorem 3 shows that keeping the shuffle operator is possible. We currently are not aware of another restriction on the input transition systems that would work. We therefore return to restrictions on the input grammars, disallowing the use of the $||$ operator.

Linear PDL Grammars

Inspired from the definition of linear context-free grammars (Salomaa 1987, Section 5 p. 44), we say that a regular tree grammar is *linear* whenever for every rule there is at most one non-terminal occurring in the right-hand expression of this rule. For example, the grammar in Theorem 1 is not linear.

Theorem 4. SYNTHPDL *for linear grammars is* PSPACE-*complete.*

Proof. First, we establish that its PSPACE-hardness already holds for the following fixed linear grammar $\mathfrak{G}$:

$$\mathbf{S} \to p \quad \mathbf{S} \to [0]\mathbf{S} \quad \mathbf{S} \to [1]\mathbf{S}$$

Indeed, we can reduce from the PSPACE-complete problem of the *non-universality of nondeterministic finite word automata* (NFW), known to be PSPACE-complete (Aho, Hopcroft, and Ullman 1974, Section 10.6, p. 395). Intuitively, this reduction takes the TS as the very NFW where proposition p holds of every non final state, while the grammar can generate some formula of the form $[a_0][a_1]\cdots[a_k]p$, thus showing that word $a_0a_1\cdots a_k$ is rejected by the NFW.

Regarding the PSPACE-membership, there is a non-deterministic algorithm that runs in polynomial space, and that generates a formula φ from the grammar $\mathfrak{G}$, and simultaneously verifies that $\mathfrak{D}, d \models \varphi$. The idea is to iteratively guess the rule to apply to the current unique non-terminal, and to guess its semantics. ☐

Non-Recursive Grammars

A regular tree grammar is *non-recursive* (Nederhof and Satta 2002) when every nonterminal occurs at most once in every derivation; as a consequence a non-recursive grammar generates a finite language. Note that the grammar of the RE-hardness of SYNTHPDL$^{||}$ (Theorem 1) is recursive.

Theorem 5. SYNTHPDL *with non-recursive grammars is in* PSPACE.

Proof. We design an alternating algorithm that runs in polynomial time (recall APTIME = PSPACE). The algorithm is based on the idea given in the proof of Theorem 4 with the difference that there may be several non-terminal symbols in the right-hand side of a rule. Rules to apply are guessed existentially, as well as the semantics X', Y', etc. of the symbols in the right-hand side of the choosen rule. Then we universally guess on which non-terminal symbol we keep on for next rule in the derivation. Since the grammar is non-recursive, there is a natural well-founded partial order between non-terminals, which makes the algorithm run in polynomial-time; and there are at most a linear number of recursive calls. ☐

We exploit the approach above for SYNTHPDL$^{||}$.

Theorem 6. SYNTHPDL$^{||}$ *with non-recursive grammars is in* EXPSPACE.

Proof. Due to non-recursiveness, there are finitely many derivable formulas, in fact at most an exponential number in the size of the grammar $\mathfrak{G}$. Also, each formula is of size $O(2^{poly(|\mathfrak{G}|)})$. We first generate all derivable formulas, and for each of these formulas φ, we check $\mathfrak{D}, d \models \varphi$, which is in PSPACE (Göller 2008). As formula φ is of size $O(2^{poly(|\mathfrak{G}|)})$, each check requires an exponential amount of memory, which concludes. ☐

Linear and Non-Recursive PDL Grammars

Finally, combining the last two restrictions we get:

Theorem 7. *The* PDL *synthesis problem restricted to non-recursive linear grammars is* NP-*complete.*

Proof. For NP-membership, note that a linear non-recursive grammar $\mathfrak{G}$ only generates terms of size in $O(|\mathfrak{G}|)$. Indeed, in each derivation step, we add $O(1)$ symbols, and since there are at most a linear number of derivations, a formula generated by the grammar has has size at most $O(|\mathfrak{G}|)$. The NP procedure guesses a derivation of some φ, and checks $\mathfrak{D}, d \models \varphi$ (which can be done in P).

For NP-hardness, we reduce from 3SAT by converting an input 3CNF ϕ into a pair $(\mathfrak{D}, u)$ and a grammar $\mathfrak{G}$, so that ϕ is satisfiable iff $\mathfrak{G}$ generates some φ with $\mathfrak{D}, u \models \varphi$. Intuitively, the reduction relies on the satisfiability game for ϕ where initially Player 0 chooses a valuation ν, next Player 1 picks a clause, and Player 1 to win the play if the picked clause satisfies the valuation. The generated formulas are of the form $\langle valuation\rangle[clause]\texttt{sat}$. The grammar's role for the program is to generate all possible valuations Player 0 may select, and the choice of Player 2 as well as the satisfaction check are reflected in the TS. □

Discussion and Related Work

The formula synthesis problem seems to be new. A remotely related work concerns a schema for randomly generating Description Logic formulas (Hladik 2005). More closely related work concerns other mathematical settings, which have similar features to PDL$^{||}$, and are concerned with synthesising objects from hierarchical descriptions. We now discuss two of these: HTN Planning and Attack-tree synthesis.

A *task problem* in Hierarchical Task Network (HTN) Planning (Bacchus 2001) involves task-network methods that decompose compound tasks into a combination of subtasks, just as tree-grammar rules for a PDL$^{||}$ program decompose programs in subprograms. It can be established that if we restrict to the propositional setting of HTN and if all methods are linearly ordered task decomposition (L) or unordered task decomposition (U), also known as *TOTD* and *UTD* respectively in the literature (see (Georgievski and Aiello 2015, Section 2.2.1)), then the following claim holds.

Claim 1. *The task problem on* HTNP$^{\mathrm{L+U}}$ *instances, i.e., propositional instance only consisting of* L *and* U *methods, polynomially reduces to* SYNTHPDL$^{||}$.

Intuitively, the tree-grammar in this reduction generates only formulas of the $\langle\rho\rangle\mathbf{true}$ where the grammar rules for ρ reflect the task decomposition methods of the HTNP$^{\mathrm{L+U}}$ instance: linearly ordered and unordered task decompositions are respectively captured by program operators ; and $||$.

A careful inspection of the proof in (Erol, Hendler, and Nau 1996) of the undecidability of the task problem for arbitrary HTN instances reveals that undecidability already holds for the HTNP$^{\mathrm{L+U}}$ instances. Thus, our undecidability result for SYNTHPDL$^{||}$ (Theorem 1) becomes a mere corollary of Claim 1; and indeed the direct proof proposed in this paper borrows ideas from the one in (Erol, Hendler, and Nau 1996). Claim 1 also makes it possible to connect our complexity results for SYNTHPDL$^{||}$ with the ones in (Erol, Hendler, and Nau 1996) for HTN planning. However, it should be noticed that in the HTN setting, the planning domain is symbolic, while the input transition system in SYNTHPDL$^{||}$ is explicit. This, and the fact that task-network methods in HTN planning may rely on arbitrary partial orders between subtasks, renders the comparison of our complexity results with the ones of (Erol, Hendler, and Nau 1996; Alford, Bercher, and Aha 2015) tricky, but opens clear avenues for future work to explore deeper connections between the two frameworks. Besides, the inability of SYNTHPDL$^{||}$ programs to describe linearizations of arbitrary partial orders prevents a natural reduction from the (propositional) HTN problem to SYNTHPDL$^{||}$, which calls for an extension of the latter.

Attack Tree Synthesis

The second notable work that concerns hierarchical decomposition is the *attack tree synthesis problem* in Security. Attack trees (AT) are aknowledged models for reasoning in Risk Analysis (Schneier 1999). Their automated generation (synthesis) is considered as a Holy Grail by security experts so as to avoid their tedious and error-prone manual design (Vigo, Nielson, and Nielson 2014). Although much less studied than the task problem, there are few works that tackle this tricky challenge; see (Wideł et al. 2019) for the survey on formal methods for AT, which include (Gadyatskaya et al. 2017; Pinchinat, Schwarzentruber, and Lê Cong 2020). In these approaches, one can recast the addressed problems as subproblems of SYNTHPDL$^{||}$. In particular in the latter contribution, the notion of *library* is in essence a set of grammatical rules for PDL$^{||}$-like programs, with tight correspondance between `OR`, `SAND` and `AND` node types in AT with the $+$, ; and $||$ program operators in SYNTHPDL$^{||}$, respectively.

Claim 2. *The attack tree synthesis problem of (Pinchinat, Schwarzentruber, and Lê Cong 2020) polynomially reduces to* SYNTHPDL$^{||}$.

Intuitively, the reduction provides input grammars that generate only formulas of the form $\langle\rho\rangle$`goal`, similarly to HTN planning. In (Pinchinat, Schwarzentruber, and Lê Cong 2020) the input transition system consists of a single path, yielding an NP-complete complexity (by a dynamic-programming-based approach), and the case of a DAG-like transition system was left open. We have proved that this generalisation also polynomially reduces to SYNTHPDL$^{||}$. Due to lack of space, this reduction will be available in an extended version of this paper. It is another exciting avenue for future work to exploit our results about the synthesis problem to shed light on general forms of the attack tree synthesis problem.

Future Work

Besides the future research foreseen in the related work, many tasks remain.

First, although we showed how to lower the complexity of the problem for various restrictions, we left open the lower bounds in a number of cases.

Second, we observed that the shuffle operator threatens decidability since, in general, it yields an image-infinite program semantics. We exploited this to exhibit decidable cases of the general problem, and note here that this is a useful heuristic for the quest of finding other decidable cases.

Finally, from a broader perspective, it would be interesting to study the formula-synthesis problem for other logics, and to find applications of this generalisation of model-checking. In particular, we are inclined to consider the case of Game Logic (Pauly and Parikh 2003) for attack tree synthesis in an adverserial setting.

References

Abrahamson, K. R. 1980. *Decidability and expressiveness of logics of processes*. University of Washington.

Aho, A. V.; Hopcroft, J. E.; and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley. ISBN 0-201-00029-6.

Alford, R.; Bercher, P.; and Aha, D. 2015. Tight bounds for HTN planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, 7–15.

Bacchus, F. 2001. AIPS 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *Ai magazine*, 22(3): 47–47.

Comon, H.; Dauchet, M.; Gilleron, R.; Jacquemard, F.; Lugiez, D.; Löding, C.; Tison, S.; and Tommasi, M. 2005. *Tree automata techniques and applications*. https://hal.inria.fr/hal-03367725

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Ann. Math. Artif. Intell.*, 18(1): 69–93.

Fischer, M. J.; and Ladner, R. E. 1979. Propositional Dynamic Logic of Regular Programs. *J. Comput. Syst. Sci.*, 18(2): 194–211.

Gadyatskaya, O.; Jhawar, R.; Mauw, S.; Trujillo-Rasua, R.; and Willemse, T. A. C. 2017. Refinement-Aware Generation of Attack Trees. In *STM*, volume 10547 of *LNCS*, 164–179. Springer.

Georgievski, I.; and Aiello, M. 2015. HTN planning: Overview, comparison, and beyond. *Artif. Intell.*, 222: 124–156.

Göller, S. 2008. *Computational complexity of propositional dynamic logics*. Ph.D. thesis, University of Leipzig.

Harel, D.; Kozen, D.; and Tiuryn, J. 2000. *Dynamic Logic*. MIT Press.

Hladik, J. 2005. A Generator for Description Logic Formulas. In *Description Logics*.

Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2003. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley. ISBN 978-0-321-21029-6.

Lange, M. 2006. Model checking propositional dynamic logic with all extras. *Journal of Applied Logic*, 4(1): 39–49.

Löding, C. 2012. Basics on Tree Automata. In D'Souza, D.; and Shankar, P., eds., *Modern Applications of Automata Theory*, volume 2 of *IISc Research Monographs Series*, 79–110. World Scientific.

Mayer, A. J.; and Stockmeyer, L. J. 1996. The complexity of PDL with interleaving. *Theoretical Computer Science*, 161(1-2): 109–122.

Nederhof, M.-J.; and Satta, G. 2002. Parsing non-recursive CFGs. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 112–119.

Pauly, M.; and Parikh, R. 2003. Game logic-an overview. *Studia Logica*, 75(2): 165–182.

Pinchinat, S.; Schwarzentruber, F.; and Lê Cong, S. 2020. Library-Based Attack Tree Synthesis. In III, H. E.; and Gadyatskaya, O., eds., *Graphical Models for Security - 7th International Workshop, GraMSec 2020, Boston, MA, USA, June 22, 2020 Revised Selected Papers*, volume 12419 of *Lecture Notes in Computer Science*, 24–44. Springer.

Salomaa, A. 1987. *Formal languages*. Computer science classics. Academic Press. ISBN 978-0-12-615750-5.

Schneier, B. 1999. Attack trees. *Dr. Dobb's journal*, 24(12): 21–29.

Vigo, R.; Nielson, F.; and Nielson, H. R. 2014. Automated generation of attack trees. In *2014 IEEE 27th Computer Security Foundations Symposium*, 337–350. IEEE.

Wideł, W.; Audinot, M.; Fila, B.; and Pinchinat, S. 2019. Beyond 2014: Formal Methods for Attack Tree–based Security Modeling. *ACM Computing Surveys (CSUR)*, 52(4): 1–36.