Logics for Context-Free Hyperproperties

Sarah Winter
Université Paris Cité, CNRS, IRIF, Paris, France

Martin Zimmermann
Aalborg University, Denmark

Abstract

We introduce a novel logic for the specification of context-free hyperproperties, which capture, e.g., the flow of information in security-critical recursive systems. Intuitively, the logic extends visibly pushdown automata by quantification over traces, just like HyperLTL, the most important logic for regular hyperproperties, extends LTL by quantification over traces. Using a game-based approach, we show that model-checking is decidable for formulas with a single quantifier alternation, provided the stack height of the visibly pushdown automaton only depends on the traces bound to the variables of the first quantifier block. A single quantifier alternation suffices to express many information-flow properties studied in the literature. Complementarily, we show that model-checking is undecidable for formulas with a single quantifier alternation, if the stack behavior of the visibly pushdown automaton may depend on the second quantifier block. This also implies that model-checking is undecidable for almost all fragments with more than one quantifier alternation.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Verification by model checking; Theory of computation → Automata over infinite objects

Keywords and phrases Hyperproperties, model-checking, context-free languages

Digital Object Identifier 10.4230/LIPIcs.MFCS.2026.54

Related Version *Full Version*: `https://arxiv.org/abs/2605.04657` [23]

Funding *Martin Zimmermann*: Partly supported by the project "Hyperlogics: Expressiveness, Monitorability and Tools (H.-Lo)" of the Icelandic Research Fund, project no. 2612260-051.

1 Introduction

The specification and verification of security-critical systems involves reasoning about the flow of information, which requires simultaneous analysis of multiple execution traces of the system. Clarkson and Schneider [7] coined the term *hyperproperties* for such properties, which are formally sets of sets of traces. A system satisfies a hyperproperty if its set of traces is an element of the hyperproperty. This should be contrasted with classical trace properties, which are sets of traces. A system satisfies a trace property if its set of traces is a subset of the trace property.

Temporal logics are an attractive specification language both for trace and hyperproperties. Arguably the most important logic for trace properties is Linear Temporal Logic (LTL) [17] while the most important logic for hyperproperties is HyperLTL [6], which extends LTL with trace quantification. For example, the formula

$$\varphi_{\mathrm{GNI}} = \forall \pi.\ \forall \pi'.\ \exists \pi''.\ \mathbf{G}\left(\bigwedge\nolimits_{\mathtt{p} \in L_{\mathrm{in}} \cup L_{\mathrm{out}}} \mathtt{p}_{\pi} \leftrightarrow \mathtt{p}_{\pi''}\right) \wedge \mathbf{G}\left(\bigwedge\nolimits_{\mathtt{p} \in H_{\mathrm{in}}} \mathtt{p}_{\pi'} \leftrightarrow \mathtt{p}_{\pi''}\right)$$

expresses generalized noninterference [15]: for all pairs π and π' of traces there is a trace π'' that agrees with the low-security inputs (propositions in L_{in}) and low-security outputs (propositions in L_{out}) of π and the high security inputs (propositions in H_{in}) of π'. Intuitively, it is satisfied if every input-output behavior observable by a low-security user is compatible with

any sequence of high-security inputs, i.e., the low security behavior does not leak information about the high-security inputs. Many information-flow properties from the literature, e.g., generalized noninterference, can be expressed with a single quantifier alternation [6].

Model-checking of finite-state systems against LTL and HyperLTL specifications is PSPACE-complete [20] and TOWER-complete [19, 14], respectively. However, finite-state systems are rather restrictive, as they, for example, cannot model the call-stack of a recursive program. Similarly, both LTL and HyperLTL are restricted to ω-regular properties. Hence, it is natural to investigate whether more general system models and/or more expressive specification languages retain a decidable model-checking problem.

Pushdown systems (pushdown automata (PDA) without an acceptance condition) and context-free languages play a central role in these extensions: Pushdown systems naturally model recursive systems with finite data (and induce in general infinite, but finitely represented, configuration graphs) while context-free languages generalize ω-regular languages with, e.g., abilities to reason about the evolution of the call stack of a program. Model-checking pushdown systems against LTL is EXPTIME-complete [4] while the undecidable universality problem for context-free languages can easily be reduced to model-checking finite-state systems against context-free specifications.

However, by considering fragments of context-free languages, decidability can be regained. Probably the most important fragment here are the visibly pushdown languages [1] (see also [16] for earlier work on input-driven pushdown automata), where the input letter being processed determines how the stack height of a pushdown automaton evolves. This allows to synchronize runs of different visibly pushdown automata (VPA) on the same word and yields, e.g., much better closure properties and better algorithmic properties. Model-checking visibly pushdown systems against visibly pushdown specifications is EXPTIME-complete [1].

After the introduction of HyperLTL, several generalizations have been considered:

- Pommellet and Touili showed that model-checking pushdown systems and visibly pushdown systems against HyperLTL specifications is undecidable, even for formulas with $\exists\exists$ quantifier prefix [18]. On the other hand, they present incomplete methods based on over- and under-approximations and show how these can be used to check security policies.
- Frenkel and Sheinvald introduced hypergrammars [10], which extend context-free grammars (over finite words) with trace quantification, just like HyperLTL extends LTL with trace quantification. Their membership problem can be seen as model-checking a regular language against a context-free hyperproperty. It can be solved in exponential time for formulas with $\exists^*$ quantifier prefix, but is undecidable for $\forall^*$ formulas.
- HyperCTL* extends the branching-time logic CTL* by trace quantification and contains HyperLTL. Model-checking finite-state systems against HyperCTL* is also TOWER-complete [6]. Bajwa et al. introduced stack-aware HyperCTL* [2], in which HyperCTL* formulas are only allowed to relate traces which have the same call-stack access pattern. Thus, all traces under consideration when evaluating a formula are synchronized. Model-checking pushdown systems against stack-aware HyperCTL* is also TOWER-complete [2].
- Gutsfeld et al. presented mumbling H_μ [11], a hyperlogic for asynchronous hyperproperties based on the linear-time μ-calculus and show that, under synchronization assumptions, model-checking of visibly pushdown systems against mumbling H_μ is decidable.

Our Contribution. Inspired by the work of Frenkel and Sheinvald, we introduce HyperPDA and HyperVPA, which extend ω-PDA and ω-VPA by quantification over (infinite) traces, which yields very natural logics for context-free hyperproperties. Thus, the formula $\forall\pi_0.\ \exists\pi_1.\ \mathcal{A}$, where $\mathcal{A}$ is an ω-PDA over $\Sigma \times \Sigma$ is satisfied by a system $\mathfrak{T}$ if for

every trace t_0 of $\mathfrak{T}$, there is a trace t_1 of $\mathfrak{T}$ such that the pair (t_0, t_1) is in $L(\mathcal{A})$. For example, stack-aware noninference and stack-aware observational determinism [2] can be expressed in HyperVPA, as the trace quantifiers range only over traces whose stack behavior is synchronized, which can be captured by ω-VPA.

In its full generality, model-checking finite-state systems against HyperPDA specifications is undecidable (even for formulas with a single universal quantifier), as one can easily capture universality of ω-PDA. On the positive side, the $\exists^*$ fragment can be model-checked in exponential time, as it can be reduced to the nonemptiness problem for ω-PDA.

These results highlight once again that general pushdown automata are too expressive in the context of model-checking hyperproperties. Thus, our main focus is on HyperVPA, in particular on formulas with one quantifier alternation. Recall that ω-VPA are controlled by the input. We prove that model-checking finite-state systems against HyperVPA formulas is decidable for formulas of the form $\forall^+\exists^*.\ \mathcal{A}$ and $\exists^+\forall^*.\ \mathcal{A}$ if the control only depends on the letters of the traces quantified in the first quantifier block, but is undecidable if the control only depends on the letters of the traces quantified in the second quantifier block. The latter result also implies that model-checking is undecidable for almost all quantifier fragments with more than one quantifier alternation. The only case we leave open is for formulas with more than one quantifier alternation when control depends on the first block. Thus, we exhibit an almost complete picture of the decidability border for visibly context-free hyperproperties.

Our decidability result is proven by extending the game-based characterization of $\forall^*\exists^*$ HyperLTL model-checking using prophecies [3] (see also [8, 21, 22]) to context-free specifications. In our setting, we employ prophecies recognized by ω-VPA and construct a visibly pushdown game, which can be solved effectively [13]. Both the prophecies and the game construction rely on the fact that the first quantifier block controls the behavior of $\mathcal{A}$. Note that our decidable classes contain in particular stack-aware noninference and stack-aware observational determinism [2].

All proofs omitted due to space restrictions can be found in the full version [23].

2 Preliminaries

We denote the set of nonnegative integers by $\mathbb{N}$.

Traces and Transition Systems. An alphabet is a nonempty finite set. The sets of finite and infinite words over an alphabet Σ are denoted by Σ^* and Σ^ω, respectively. The length of a finite or infinite word w is denoted by $|w| \in \mathbb{N} \cup \{\infty\}$. For a word w of length at least n and i, i' with $0 \leq i \leq i' < n$, we write $w[i, i']$ for the infix of w starting at position i and ending at position i' (both included). Given k infinite words $w_0, \ldots, w_{k-1}$, let their merge (also known as zip), which is an infinite word over Σ^k, be defined as

$$\mathrm{mrg}(w_0, \ldots, w_{k-1}) = \begin{pmatrix} w_0(0) \\ \vdots \\ w_{k-1}(0) \end{pmatrix} \begin{pmatrix} w_0(1) \\ \vdots \\ w_{k-1}(1) \end{pmatrix} \begin{pmatrix} w_0(2) \\ \vdots \\ w_{k-1}(2) \end{pmatrix} \cdots .$$

We define $\mathrm{mrg}(w_0, \ldots, w_{k-1})$ for finite words $w_0, \ldots, w_{k-1}$ of the same length analogously.

A *transition system* $\mathfrak{T} = (V, E, V_I, \lambda)$ consists of a finite set V of vertices, a set $E \subseteq V \times V$ of (directed) edges, a nonempty set $V_I \subseteq V$ of initial vertices, and a labelling $\lambda \colon V \to \Sigma$ of the vertices by labels from some alphabet Σ. We assume that every vertex has at least one outgoing edge. For $v \in V$, we denote by $\mathrm{Succ}(v)$ the set of its successors. A *path* ρ in $\mathfrak{T}$ is an infinite sequence $\rho = v_0 v_1 v_2 \cdots$ of vertices with $v_0 \in V_I$ and $v_{n+1} \in \mathrm{Succ}(v_n)$ for every

$n \geq 0$. Every path ρ induces its *trace*, the ω-word $\lambda(\rho) = \lambda(v_0)\lambda(v_1)\lambda(v_2)\cdots \in \Sigma^\omega$. The *language* of $\mathfrak{T}$ is $L(\mathfrak{T}) = \{\lambda(\rho) \mid \rho \text{ is a path of } \mathfrak{T}\}$. For a nonempty $V' \subseteq V$, we write $\mathfrak{T}_{V'}$ to denote the transition system (V, E, V', λ) obtained from $\mathfrak{T}$ by making V' the set of initial states, and use $\mathfrak{T}_v$ as shorthand for $\mathfrak{T}_{\{v\}}$ for $v \in V$.

Pushdown Automata. An *ω-pushdown automaton* (ω-PDA for short) $\mathcal{A} = (Q, \Sigma, \Gamma, q_I, \Delta, F)$ consists of a finite set Q of states with the initial state $q_I \in Q$, an input alphabet Σ, a stack alphabet Γ, a transition relation Δ to be specified, and a set $F \subseteq Q$ of accepting states. For notational convenience, we define $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\perp = \Gamma \cup \{\perp\}$, where $\perp \notin \Gamma$ is a designated stack bottom symbol. Then, the transition relation Δ is a subset of $Q \times \Gamma_\perp \times \Sigma_\varepsilon \times Q \times \Gamma_\perp^{\leq 2}$ that we require to neither write nor delete the stack bottom symbol from the stack: If $(q, \perp, a, q', \gamma) \in \Delta$, then $\gamma \in \perp \cdot (\Gamma \cup \{\varepsilon\})$, and if $(q, X, a, q', \gamma) \in \Delta$ for $X \in \Gamma$, then $\gamma \in \Gamma^{\leq 2}$. Given a transition $\tau = (q, X, a, q', \gamma)$ let $\ell(\tau) = a \in \Sigma_\varepsilon$. We say that τ is an $\ell(\tau)$-transition and that τ is a Σ-transition, if $\ell(\tau) \in \Sigma$. For a finite or infinite sequence ρ over Δ, $\ell(\rho)$ is defined by applying ℓ homomorphically to every transition.

A stack content is a finite word in $\perp\Gamma^*$ (i.e., the top of the stack is at the end) and a configuration $c = (q, \gamma)$ of $\mathcal{A}$ consists of a state $q \in Q$ and a stack content γ. The stack height of c is $\mathrm{sh}(c) = |\gamma| - 1$. The initial configuration is $(q_I, \perp)$.

A transition $\tau = (q, X, a, q', \gamma') \in \Delta$ is enabled in a configuration c if $c = (q, \gamma X)$ for some $\gamma \in \Gamma_\perp^*$. In this case, we write $(q, \gamma X) \xrightarrow{\tau} (q', \gamma\gamma')$. A run of $\mathcal{A}$ is a finite or infinite sequence $r = c_0\tau_0c_1\tau_1c_2\tau_2\cdots$ of configurations and transitions with $c_n \xrightarrow{\tau_n} c_{n+1}$ for every n. A finite run is required to end with a configuration. A run is initial if it starts in the initial configuration. An infinite run $r = c_0\tau_0c_1\tau_1c_2\tau_2\cdots$ is a run of $\mathcal{A}$ on $w \in \Sigma^\omega$, if $w = \ell(\tau_0\tau_1\tau_2\cdots)$ (this implies that ρ contains infinitely many Σ-transitions). We say that r is accepting if there are infinitely n such that the state of c_n is in F, i.e., we consider Büchi acceptance. The language $L(\mathcal{A})$ recognized by an ω-PDA $\mathcal{A}$ contains all $w \in \Sigma^\omega$ such that $\mathcal{A}$ has an accepting run on w.

Visibly pushdown automata are defined with respect to a partition $(\Sigma_c, \Sigma_r, \Sigma_s)$ of the input alphabet into calls (letters in Σ_c), returns (letters in Σ_r), and skips (letters in Σ_s) and have to satisfy the following conditions:

- A letter $a \in \Sigma_c$ is only processed by transitions of the form (q, X, a, q', XY) with $X \in \Gamma_\perp$, i.e., some stack symbol Y is pushed onto the stack.
- A letter $a \in \Sigma_r$ is only processed by transitions of the form $(q, X, a, q', \varepsilon)$ with $X \neq \perp$ or the form $(q, \perp, a, q', \perp)$, i.e., the topmost stack symbol is removed, or if the stack is empty, it is left unchanged.
- A letter $a \in \Sigma_s$ is only processed by transitions of the form (q, X, a, q', X) with $X \in \Gamma_\perp$, i.e., the stack is left unchanged.
- There are no ε-transitions.

Note that we allow, w.l.o.g., the automata to access the top stack symbol during calls and skips (see [1, Section 2.1]). An ω-PDA is a *visibly* ω-PDA (ω-VPA for short), if there is a partition of its input alphabet into $(\Sigma_c, \Sigma_r, \Sigma_s)$ satisfying the conditions above.

3 Logics for Context-Free Hyperproperties

In this section, we introduce our logics and present some preliminary results.

HyperPDA. Let $\mathcal{V} = \{\pi_0, \pi_1, \pi_2, \ldots\}$ be *the* set of trace variables. A formula of HyperPDA has the form $\varphi = Q_0\pi_0.\ Q_1\pi_1.\ \ldots Q_{k-1}\pi_{k-1}.\ \mathcal{A}$ for some $k \geq 1$ where each Q_i is either an existential or universal quantifier and where $\mathcal{A}$ is an ω-PDA over an alphabet of the form Σ^k.

We call $\mathcal{A}$ the automaton of φ and k the arity of both φ and $\mathcal{A}$. As usual, we classify formulas of HyperPDA by their quantifier alternations. Let $n \geq 1$. Σ_n (Π_n) contains all formulas with $n-1$ quantifier alternations beginning with an existential (universal) quantifier.

The semantics of HyperPDA is defined with respect to a trace assignment, a partial mapping $\Pi\colon \mathcal{V} \to \Sigma^\omega$. The assignment with empty domain is denoted by $\Pi_\emptyset$. Given a trace assignment Π, a variable π, and a trace $t \in \Sigma^\omega$ we denote by $\Pi[\pi \to t]$ the assignment that coincides with Π everywhere but at π, which is mapped to t.

For sets $T \subseteq \Sigma^\omega$ of traces and trace assignments Π we define

- $(T, \Pi) \models \exists \pi_j.\ \varphi$ if there exists a trace $t \in T$ such that $(T, \Pi[\pi_j \to t]) \models \varphi$,
- $(T, \Pi) \models \forall \pi_j.\ \varphi$ if for all traces $t \in T$: $(T, \Pi[\pi_j \to t]) \models \varphi$, and
- $(T, \Pi) \models \mathcal{A}$ if $\mathcal{A}$ accepts $\mathrm{mrg}(\Pi(\pi_0), \ldots, \Pi(\pi_{k-1}))$, where k is the arity of $\mathcal{A}$.

We say that T satisfies a formula φ if $(T, \Pi_\emptyset) \models \varphi$. In this case, we write $T \models \varphi$ and say that T is a model of φ. A transition system $\mathfrak{T}$ satisfies φ, written $\mathfrak{T} \models \varphi$, if $L(\mathfrak{T}) \models \varphi$.

▶ Remark 1. HyperPDA subsumes HyperLTL, as quantifier-free HyperLTL formulas can be translated into Büchi automata (which are ω-PDA that do not use their stack), as they are (essentially) LTL formulas. Hence, all lower bounds for HyperLTL apply to HyperPDA, e.g., HyperPDA satisfiability is Σ^1_1-hard [9] and we conjecture that the problem is Σ^1_1-complete.

The HyperPDA model-checking problem asks, given a transition system $\mathfrak{T}$ and a HyperPDA formula φ, whether $\mathfrak{T} \models \varphi$. The model-checking problem for fragments Σ_n or Π_n is defined by restricting the input formulas to the fragment. The following result follows from emptiness of ω-PDA being decidable respectively universality being undecidable, where the lower bound for Σ_1 is obtained by a reduction from the intersection problem for DFA [12].

▶ **Theorem 2.**

1. *HyperPDA model-checking for Σ_1 formulas is in ExpTime and PSpace-hard.*
2. *HyperPDA model-checking for Π_1 formulas is undecidable.*

As our undecidability result holds even for formulas with a single universal quantifier, we obtain undecidability for any quantifier-fragment that allows universal quantifiers.

▶ **Corollary 3.** *HyperPDA model-checking is undecidable for all Σ_n with $n > 1$ and all Π_n with $n \geq 1$, i.e., in particular for the full logic.*

Our preliminary results show that a single universal quantifier makes model-checking undecidable, as universality for ω-PDA is undecidable. Thus, it is prudent to study formulas with restricted classes of ω-PDA for which universality is decidable. Thus, we restrict ourselves to ω-VPA, which are closed under all Boolean operations, which implies in particular that universality is decidable.

HyperVPA. HyperVPA is the restriction of HyperPDA to formulas whose automaton is an ω-VPA. For HyperVPA, we can refine the definition of the quantifier fragments Σ_n and Π_n. To this end, let $\mathcal{A}$ be an ω-VPA of arity $k > 0$ and let $I \subseteq \{0, 1, \ldots, k-1\}$. We say that $\mathcal{A}$ is controlled by the indexes in I, if for all letters $(a_0, a_1, \ldots, a_{k-1})$ and $(b_0, b_1, \ldots, b_{k-1})$ of $\mathcal{A}$, $a_i = b_i$ for all $i \in I$ implies that they are both calls, or both returns, or both skips. Intuitively, the type of a letter of $\mathcal{A}$ only depends on the positions in I. The fragments Σ_n and Π_n of HyperVPA are defined as for HyperPDA. Additionally, let $1 \leq j \leq n$. $\Sigma_{n,j}$ and $\Pi_{n,j}$ contain the formulas of HyperVPA in Σ_n and Π_n, respectively, whose automaton is controlled by the set of indexes of the j-th quantifier block.

While HyperVPA does not have a negation operator it is nevertheless closed under negation, since a negation can be pushed over quantifiers and since ω-VPA are closed under complementation [1]. Formally, given a formula $\varphi = Q_0\pi_0.\ Q_1\pi_1.\ \ldots Q_{k-1}\pi_{k-1}.\ \mathcal{A}$ of HyperVPA, we define its negation $\neg\varphi$ as the formula $\overline{Q_0}\pi_0.\ \overline{Q_1}\pi_1.\ \ldots \overline{Q_{k-1}}\pi_{k-1}.\ \overline{\mathcal{A}}$ where $\overline{\exists} = \forall$, $\overline{\forall} = \exists$, and where $\overline{\mathcal{A}}$ denotes an ω-VPA accepting the complement of $L(\mathcal{A})$. Note that $\overline{\mathcal{A}}$ is defined with respect to the same partition of the input alphabet as $\mathcal{A}$. Hence, $\mathcal{A}$ is controlled by some $I \subseteq \{0, 1, \ldots, k-1\}$ if and only if $\overline{\mathcal{A}}$ is controlled by I. Note though that $\overline{\mathcal{A}}$ may be exponentially larger than $\mathcal{A}$ [1].

▶ Remark 4. Let φ be an HyperVPA formula, $n > 0$, and $1 \leq j \leq n$.

1. φ and $\neg\neg\varphi$ are equivalent.
2. φ is in Σ_n if and only if $\neg\varphi$ is in Π_n; and φ is in $\Sigma_{n,j}$ if and only if $\neg\varphi$ is in $\Pi_{n,j}$.
3. $\mathfrak{T} \models \varphi$ if and only if $\mathfrak{T} \not\models \neg\varphi$.

▶ Remark 5. HyperVPA subsumes HyperLTL for the same reason HyperPDA subsumes HyperVPA (see Remark 1). Hence, HyperVPA satisfiability is Σ^1_1-hard and we again conjecture completeness.

The decidability result for the Σ_1-fragment of HyperPDA (Theorem 2.1) carries over to HyperVPA, as it is a fragment of HyperPDA. Similarly, the lower bound carries over, as it does not use the stack of the ω-PDA.

▶ **Corollary 6.** *HyperVPA model-checking for Σ_1 formulas is in ExpTime and PSpace-hard.*

On the other hand, due to closure of ω-VPA under complement, model-checking the Π_1-fragment of HyperVPA is also decidable. Here, the lower bound follows from universality for VPA being ExpTime-complete [1]

▶ **Theorem 7.** *HyperVPA model-checking for Π_1 formulas is ExpTime-complete.*

In the next two sections, we consider the fragments Σ_2, i.e., formulas with quantifier-prefix $\exists^*\forall^*$ and Π_2, i.e., formulas with quantifier-prefix $\forall^*\exists^*$. In Section 4, we show that model-checking is decidable for the fragments $\Sigma_{2,1}$ and $\Pi_{2,1}$, i.e., if the automaton is controlled by the first quantifier block. Dually, in Section 5, we show that model-checking is undecidable for the fragments $\Sigma_{2,2}$ and $\Pi_{2,2}$, i.e., if the automaton is controlled by the second quantifier block. Finally, using these results, we show at the end of Section 5 that (almost all) remaining fragments have an undecidable model-checking problem.

4 Fragments of HyperVPA with Decidable Model-Checking

In this section, we first show that the model-checking problem is decidable for the fragment $\Pi_{2,1}$, i.e., formulas of the form $\forall^*\exists^*.\ \mathcal{A}$ such that the stack height in $\mathcal{A}$ only depends on the universally quantified traces. This then also implies decidability for $\Sigma_{2,1}$.

In the following, we focus on formulas of the form $\forall\pi.\ \exists\pi'.\ \mathcal{A}$ (i.e., with a single variable in each quantifier block) and with variables π and π' instead of π_0 and π_1. Both assumptions simplify our notation in the following proof. The renaming of variables is inconsequential while we comment on how to generalize the proof to general $\forall^*\exists^*$ formulas in Remark 17.

Intuitively, we capture the semantics of $\mathfrak{T} \models \forall\pi.\ \exists\pi'.\ \mathcal{A}$ by a two-player game between Falsifier (constructing a trace t of $\mathfrak{T}$ for π) and Verifier (constructing a trace t' of $\mathfrak{T}$ for π'). Verifier wins if $\mathrm{mrg}(t, t') \in L(\mathcal{A})$. To obtain decidability of the game, the players need to pick their traces in alternation. But this puts Verifier at a disadvantage as she has only access to a prefix of t when determining t', while t' may need to depend on all letters of t.

Assume, e.g., that $\mathcal{A}$ is equivalent to the LTL formula $a_{\pi'} \leftrightarrow \mathbf{F}\, a_\pi$, i.e., Verifier needs to pick an a in the first round if and only if Falsifier plays an a in some round. Verifier does not have a winning strategy, even though $\mathfrak{T}$ may satisfy the formula. However, a single bit of information about Falsifier's move ("will t contain an a?") is sufficient for Verifier to win.

We define a game where Falsifier, in every round, has to make binding predictions about the membership of the suffix of t starting in the current round for a precomputed list of languages (that only depends on $\mathcal{A}$ and $\mathfrak{T}$). Such "prophecies" have previously been applied to HyperLTL model-checking with a single [3] and any number [21] of quantifier alternations. In the context-free setting, the prophecies need to give Verifier also information about the evolution of the stack height (which is fully controlled by Falsifier, i.e., he can indeed make predictions about it).

For example, assume that $\mathcal{A}$ accepts $\mathrm{mrg}(t, t')$ if and only if either

- t has a nonempty prefix that causes the stack height of a run of $\mathcal{A}$ on $\mathrm{mrg}(t, t')$ to reach stack height zero (which only depends on t due to our assumption on control of $\mathcal{A}$) and $t'(0) = t(n)$, where $n > 0$ is the minimal position with this property, or
- t does not have such a prefix and $t'(0) = \#$ for some special symbol $\#$.

This specification does require Verifier not only to have information about the evolution of the stack height (which is under the control of Falsifier) but also about which letter Falsifier is playing when reaching stack height zero for the first time again. In general, one can even construct examples where Verifier needs information about moves at the next time the current stack height (which may possibly be nonzero) is reached again for the first time.

In the prophecies we define later, we will actually require Falsifier to provide even more information: he does not only have to provide information about (certain) future letters, but also with which transition they may be processed.

Our main result is that for every ω-VPA $\mathcal{A}$ and transition system $\mathfrak{T}$, there is a computable list of prophecies so that Verifier wins the game with these prophecies if and only if $\mathfrak{T} \models \forall\pi.\ \exists\pi'.\ \mathcal{A}$, and the resulting game can be solved effectively.

Gale-Stewart Games. A Gale-Stewart game $\mathcal{G}(L)$ is given by an ω-language $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$, its winning condition. It is played between Falsifier and Verifier in rounds $i = 0, 1, 2, \ldots$: In each round, first Falsifier picks a letter $\alpha(i) \in \Sigma_1$, then Verifier picks a letter $\beta(i) \in \Sigma_2$. After ω rounds, the players have constructed an outcome $\mathrm{mrg}(\alpha(0)\alpha(1)\alpha(2)\cdots, \beta(0)\beta(1)\beta(2)\cdots)$ which is winning for Verifier if it is in L. A strategy for Verifier in $\mathcal{G}(L)$ is a mapping $\sigma\colon \Sigma_1^* \to \Sigma_2$. An outcome as above is consistent with σ, if $\beta(i) = \sigma(\alpha(0)\cdots\alpha(i))$ for all i. A strategy σ for Verifier is winning if every outcome that is consistent with σ is in L. Verifier wins $\mathcal{G}(L)$ if she has a winning strategy for $\mathcal{G}(L)$.

▶ **Proposition 8** ([13]). *The following problem is 2ExpTime-complete: Given an ω-VPA $\mathcal{A}$ over a product alphabet $\Sigma_1 \times \Sigma_2$, does Verifier win $\mathcal{G}(L(\mathcal{A}))$?*

Löding et al. formally proved that the winner of games played on configuration graphs of visibly pushdown systems (visibly pushdown automata without an acceptance condition) and with winning conditions given by ω-VPA can be determined in doubly-exponential time. Gale-Stewart games with winning conditions given by ω-VPA can be reduced to the games considered by Löding et al. with a linear blowup.

Games with Prophecies. For the remainder of the section, fix a transition system $\mathfrak{T}$ with set V of vertices and a formula $\varphi = \forall\pi.\ \exists\pi'.\ \mathcal{A}$, such that $\mathcal{A}$ is controlled by the first trace. Furthermore, let Σ be the alphabet used to label vertices in $\mathfrak{T}$, i.e., the alphabet of $\mathcal{A}$ is

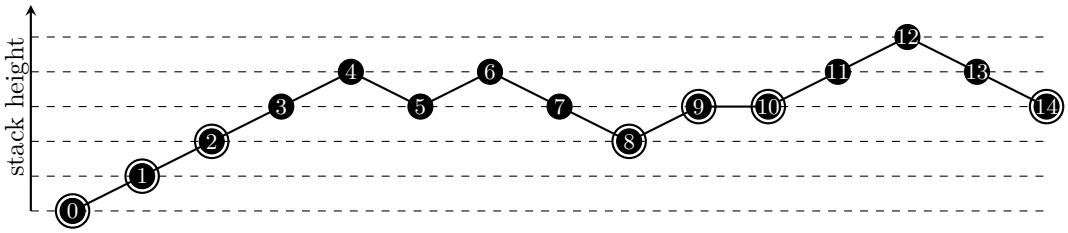

Figure 1 Development of the stack height during a run. Positions that are steps (that is, onward, the stack height never goes below the current stack height) are circled, all other positions lie inside humps. The pair $(2, 7)$ of positions is a call matched by a return. Other such pairs are $(3, 4)$, $(5, 6)$, $(10, 13)$, and $(11, 12)$. Positions 0, 1, and 8 are calls which remain unmatched. The result of an unmatched call is that onward, the current stack height is never reached again. Hence, these positions are proper steps.

$\Sigma \times \Sigma$. As $\mathcal{A}$ is an ω-VPA, $\Sigma \times \Sigma$ is partitioned into $(\Sigma'_c, \Sigma'_r, \Sigma'_s)$. Finally, as $\mathcal{A}$ is controlled by the first component, there is a partition of Σ into $(\Sigma_c, \Sigma_r, \Sigma_s)$ such that $\Sigma'_c = \Sigma_c \times \Sigma$, $\Sigma'_r = \Sigma_r \times \Sigma$, and $\Sigma'_s = \Sigma_s \times \Sigma$. We call $(\Sigma_c, \Sigma_r, \Sigma_s)$ the projected partition of $\mathcal{A}$.

To make the predictions of Falsifier indeed binding, we use so-called prophecy variables that are used by Falsifier to make his predictions and then use the winning condition of the game to ensure he loses when he violates a prediction. Formally, let $\mathcal{P}$ be a finite set of prophecies (to be defined later), i.e., languages over Σ, and let x_P be the prophecy variable associated to $P \in \mathcal{P}$.

Now, let $\Sigma_1 = V \times 2^{\{x_P \mid P \in \mathcal{P}\}}$, let $\Sigma_2 = V$, let $\mathrm{pr}_V(\cdot)$ and $\mathrm{pr}_{\mathcal{P}}(\cdot)$ denote the projections from Σ_1 to V and from Σ_1 to $2^{\{x_P \mid P \in \mathcal{P}\}}$, and let $L(\mathcal{A}, \mathfrak{T}, \mathcal{P})$ be the language

$$\begin{aligned}\{\mathrm{mrg}(\alpha, \beta) \mid \alpha \in \Sigma_1^\omega, \beta \in \Sigma_2^\omega,\ &\text{if } \mathrm{pr}_V(\alpha) \text{ is a path of } \mathfrak{T}, \text{ then } \beta \text{ is a path of } \mathfrak{T} \text{ and,} \\ &[\forall i \in \mathbb{N}.\ \forall P \in \mathcal{P}.\ x_P \in \mathrm{pr}_{\mathcal{P}}(\alpha(i)) \leftrightarrow \lambda(\mathrm{pr}_V(\alpha(i)\alpha(i+1)\alpha(i+2)\cdots)) \in P] \rightarrow \\ &\quad \mathrm{mrg}(\lambda(\mathrm{pr}_V(\alpha)), \lambda(\beta)) \in L(\mathcal{A})\}\end{aligned}$$

expressing that whenever Falsifier's predictions are correct, then the specification expressed by $\mathcal{A}$ must be satisfied. Further, we require Falsifier to actually pick a path in $\mathfrak{T}$. If he does so, then Verifier also needs to pick a path in $\mathfrak{T}$, otherwise she trivially wins.

Prophecies for HyperVPA Model-Checking. In the following, for $\mathfrak{T}$ and $\mathcal{A}$ as fixed above, we present a finite set $\mathcal{P}$ of prophecies such that $\mathfrak{T} \models \forall \pi.\ \exists \pi'.\ \mathcal{A}$ if and only if Verifier wins $\mathcal{G}(L(\mathcal{A}, \mathfrak{T}, \mathcal{P}))$. Further, we show that $L(\mathcal{A}, \mathfrak{T}, \mathcal{P})$ is recognized by an ω-VPA, i.e., the winner of the game can be effectively determined.

We continue by recalling some elementary properties about the evolution of the stack during a run of a pushdown automaton.

▶ Remark 9 (Step, hump, matching). A *step* in a run is a position n such that from there onward, the stack height will never be below the stack height at position n. Every run has infinitely many steps. A step is proper, if its stack height is never reached again. Whenever, a position of a run is not a step, we say the run is in a *hump*. Between two steps of the same stack height, the stack will be build up and down, giving the appearance of a "hump" over time. A call is *matched* if the stack height at the call is reached again. It is then matched with the return that reaches that stack height again for the first time. Otherwise, the call is *unmatched*. Figure 1 shows an illustration.

A central role of our prophecies is to gather information about the development of the stack height during a run. Let us stress again that the stack height during a run is controlled solely by Falsifier, as this enables the approach of model-checking via games with prophecies in the setting of visibly pushdown specifications: The intention of prophecies is that Falsifier has to give Verifier additional information that help her overcome the disadvantage of having to pick t' without knowing t completely. This requires that Falsifier only makes truthful predictions about the future. If Falsifier is not truthful, Verifier wins automatically. Assume that Falsifier makes predictions about the development of the stack height. If Verifier's moves can also influence the stack height, she can actively render Falsifier's predictions false, making her win the game unjustly.

Before introducing the prophecies concerning the development of the stack height, we need some auxiliary notions. Given a finite word w over $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_s$ (recall that $(\Sigma_c, \Sigma_r, \Sigma_s)$ is the projected partition), we define its effect $ef(w) \in \mathbb{Z}$ as $|w|_{\Sigma_c} - |w|_{\Sigma_r}$. Note that this does not necessarily match the stack height of a run processing w, as a return on an empty stack height is like a skip. However, if the effect never gets negative, then it matches.

In the following, we define our prophecies and give some intuition, see also Figure 1. We begin with two auxiliary prophecies.

- $\texttt{Step} = \{t \in \Sigma^\omega \mid ef(t(0)\cdots t(n)) \geq 0 \text{ for all } n \geq 0\}$
  - With this prophecy, we require Falsifier to predict whether the current position is a step (i.e., the current stack top symbol (and everything below) will never be removed).
- $\texttt{ProperStep} = \{t \in \Sigma^\omega \mid ef(t(0)\cdots t(n)) \geq 1 \text{ for all } n \geq 0\}$
  - Here, we require Falsifier to say whether the current position is a *proper step*, i.e., whether it is a step and the current stack height is never reached again. Note that this implies that $t(0)$ can only be processed by a call-transition.

Before we can introduce our main prophecies (which are to be used in conjunction with `Step` and `ProperStep`), we need two more definitions.

▶ **Definition 10** (Fresh run). *Let $r = c_0\tau_0c_1\tau_1c_2\tau_2\cdots$ be a run. We say that r is* fresh *if*

- *$c_0 = (q, \bot A)$ for some $A \neq \bot$ and $\tau_0 = (q, A, a, q', \gamma)$, or*
- *$c_0 = (q, \bot)$ and $\tau_0 = (q, \bot, a, q', \gamma)$,*

i.e., the stack content of c_0 is the smallest one that enables τ_0.

▶ Remark 11. Fix some $t, t' \in \Sigma^\omega$ and let ρ be a run of $\mathcal{A}$ starting in some (not necessarily initial) configuration $(q, \gamma A)$ processing $\mathrm{mrg}(t, t')$. If the effect of $t(0)\cdots t(n)$ is nonnegative for all n, then every configuration reached during ρ has a stack content of the form $\gamma A\gamma'$, i.e., γA is never removed from the stack and the only symbol that is ever "accessed" from the pre-filled part of the stack is the initial top stack symbol A. Thus, one can analyze runs processing $\mathrm{mrg}(t, t')$ by considering fresh runs only. This observation is crucial for the construction of our prophecies.

On the other hand, if the effect of some $t(0)\cdots t(n)$ is negative, then we will only reason about prefixes with a nonnegative effect.

Also, we need to express that we can build an accepting run of the ω-VPA $\mathcal{A}$. Since $\mathcal{A}$ uses Büchi acceptance, an accepting run is a run that visits the set of accepting states infinitely often. However, it is not enough to specify that it is possible in the future to visit infinitely many times an accepting state, a run must actually make progress towards visiting an accepting state in order to be accepting in the limit. Hence, we need to compare runs in terms of visiting an accepting state as soon as possible.

▶ **Definition 12** (FirstAcc(·)). *Given a (possibly finite) run $r = c_0\tau_0c_1\tau_1c_2\tau_2\cdots$ of $\mathcal{A}$, we define* $\mathrm{FirstAcc}(r) = n$ *where n is minimal with c_n being a configuration whose state is accepting. If no such configuration exists, we define* $\mathrm{FirstAcc}(r) = \infty$.

Now, we introduce the main prophecies. The first ones are relevant whenever Falsifier indicates that the current position (in a run) is a step (as indicated by the prophecy `Step`). We dub them "`Step`"-prophecies. Assume Falsifier has already made his move in some round, say to vertex u, i.e., $\lambda(u)$ determines what type (that is, call, skip or return) the transition processing $\lambda(u)$ and Verifier's move in $\mathcal{A}$ has. If the current position is a step, four possibilities can occur: If $\lambda(u)$ induces a call, it is either eventually matched by a return (handled via "`MatchedCallStep`") or never matched by a return (handled via "`UnmatchedCallStep`"). Alternatively, $\lambda(u)$ can induce a skip (handled via "`SkipStep`"). Finally, it is also possible that $\lambda(u)$ induces a return. However, then it must be a return on the empty stack (which induces the same stack behavior as a skip), otherwise the position is clearly not a step. This is handled via "`UnmatchedReturnStep`".

Each prophecy includes a detailed description of its intended use. To denote parameters (for any prophecy) we use $u, v, v_\tau, \ldots$ for vertices of $\mathfrak{T}$, and $\tau, \eta, \ldots$ for transitions of $\mathcal{A}$.

- `MatchedCallStep`$\langle u, v, v_\tau, v_\eta, \tau, \eta\rangle = \{t \in L(\mathfrak{T}_u) \mid$ there exists an infinite path $v_0v_1\cdots$ in $\mathfrak{T}$ and a fresh infinite run $r = c_0\tau_0c_1\tau_1\cdots$ of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(v_0v_1\cdots))$ such that:

  $v_0 = v_\tau \in \mathrm{Succ}(v)$, and $\tau_0 = \tau$, and τ is a call-transition,
  $n \in \mathbb{N}$ is minimal with $\mathrm{sh}(c_0) = \mathrm{sh}(c_{n+1})$ (which we require to exist),
  $v_n = v_\eta$, and $\tau_n = \eta$ (note that η must be a return-transition),
  r is accepting, and
  for all paths ρ' in $\mathfrak{T}_{v'}$ where $v' \in \mathrm{Succ}(v)$ and for all accepting runs r' of $\mathcal{A}$
  on $\mathrm{mrg}(t, \lambda(\rho'))$ of the form $c_0\tau_0'c_1'\tau_1'\cdots$ we have $\mathrm{FirstAcc}(r) \leq \mathrm{FirstAcc}(r')\}$.

  - Falsifier continues his path from u, and Verifier continues her path from $v_\tau \in \mathrm{Succ}(v)$, the traces induced by these continuations are called t and t', respectively.
  - The above starting point reflects that in the game, first Falsifier makes his move (he has moved to u), then it is Verifier's turn, she is in v, and needs to move to some vertex in $\mathrm{Succ}(v)$. This prophecy reflects her possibilities if she chooses to move to $v_\tau \in \mathrm{Succ}(v)$.
  - Say, so far $\mathcal{A}$ has reached q and $A \in \Gamma_\bot$ is the topmost stack symbol (meaning the exact configuration is of the form $(q, \gamma' A)$ for some $\gamma' \in \Gamma_\bot^*$). Note that q and A are not parameters of the prophecy, we just use them for illustrative purposes.
  - We require that Verifier has the possibility to continue her trace such that the run on $\mathrm{mrg}(t, t')$ is accepting using τ as its next transition. The current stack top symbol and everything below will never be removed, so having an accepting run continuing from $(q, \gamma' A)$ is equivalent to having an accepting run from (q, A), i.e., to have a fresh run.
  - As this is a matched call, it is required that its matching return is processed by η (processing the letter $(t(n), \lambda(v_\eta)) \in \Sigma^2$) that happens n transitions later.
  - As explained before the definition of FirstAcc(·), for acceptance it is important to make actual progress towards visiting an accepting state. Here, it is required that r is a run that visits an accepting state as soon as possible among all accepting runs.
  - In Figure 1, this type of prophecy is used at positions 2 and 10.

The above prophecy is central, in a global view, to ensuring Verifier can win the game if the formula is satisfied: It is used when the current position is a step and a call occurs that will be matched by a return eventually, i.e., the position after the matching return is a step again. The prophecy ensures that after the hump between the call and its matching return, Verifier can continue to play in a way that $\mathcal{A}$ can still accept.

The upcoming three prophecies cover the possibilities to reach a step directly after a step without a hump in between, which simplifies their definitions slightly.

- `UnmatchedCallStep`$\langle u, v, v_\tau, \tau\rangle = \{t \in L(\mathfrak{T}_u) \mid$ there exists an infinite path $v_0v_1\cdots$ in $\mathfrak{T}$ and a fresh infinite run $r = c_0\tau_0c_1\tau_1\cdots$ of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(v_0v_1\cdots))$ such that:

  $v_0 = v_\tau \in \mathrm{Succ}(v)$, and $\tau_0 = \tau$, and τ is a call-transition,
  $\mathrm{sh}(c_0) < \mathrm{sh}(c_{i+1})$ for all $i \geq 0$,
  r is accepting, and
  for all paths ρ' in $\mathfrak{T}_{v'}$ where $v' \in \mathrm{Succ}(v)$ and all accepting runs r' of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(\rho'))$ of the form $c_0\tau_0'c_1'\tau_1'\cdots$ we have $\mathrm{FirstAcc}(r) \leq \mathrm{FirstAcc}(r')\}$.

  - This is very similar to the above case, except that the call is unmatched.
  - In Figure 1, this type of prophecy is used in positions 0, 1, and 8.
- `SkipStep`$\langle u, v, v_\tau, \tau\rangle = \{t \in L(\mathfrak{T}_u) \mid$ there exists an infinite path $v_0v_1\cdots$ in $\mathfrak{T}$ and a fresh infinite run $r = c_0\tau_0c_1\tau_1\cdots$ of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(v_0v_1\cdots))$ such that:

  $v_0 = v_\tau$, and $\tau_0 = \tau$, and τ is a skip-transition,
  r is accepting, and
  for all paths ρ' in $\mathfrak{T}_{v'}$ where $v' \in \mathrm{Succ}(v)$ and all accepting runs r' of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(\rho'))$ of the form $c_0\tau_0'c_1'\tau_1'\cdots$ we have $\mathrm{FirstAcc}(r) \leq \mathrm{FirstAcc}(r')\}$.

  - Exactly like `UnmatchedCallStep`$\langle u, v, v_\tau, \tau\rangle$, but the first letter is a skip, not a call.
  - In Figure 1, this type of prophecy is used in position 9.
- `UnmatchedReturnStep`$\langle u, v, v_\tau, \tau\rangle = \{t \in L(\mathfrak{T}_v) \mid$ there exists an infinite path $v_0v_1\cdots$ in $\mathfrak{T}$ and a fresh infinite run $r = c_0\tau_0c_1\tau_1\cdots$ of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(v_0v_1\cdots))$ such that:

  $v_0 = v_\tau, \tau_0 = \tau$, and τ is a return-transition of the form $(q_1, \bot, (\lambda(u), \lambda(v_0)), q_2, \bot)$,
  r is accepting, and
  for all paths ρ' in $\mathfrak{T}_{v'}$ where $v' \in \mathrm{Succ}(v)$ and all accepting runs r' of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(\rho'))$ of the form $c_0\tau_0'c_1'\tau_1'\cdots$ we have $\mathrm{FirstAcc}(r) \leq \mathrm{FirstAcc}(r')\}$.

  - Like "`SkipStep`", but for the special case of returns on the empty stack.

The next three prophecies are relevant whenever the current position is not a step, i.e., the position is inside a hump. We dub them "`Hump`"-prophecies. These prophecies give only information about the future until the stack height goes below the current stack height (which eventually happens, as the position is in a hump, see Remark 9). In a hump, it is possible to encounter calls, which must be matched as unmatched calls always induce a step. This is handled by "`CallHump`". It is furthermore possible to encounter skips, which is handled by "`CallSkip`", or to encounter returns, which is handled by "`ReturnHump`".

The "`Step`"-prophecies take a global view; their purpose is to ensure that in every position that is a step, Verifier can still win taking the whole future into account. In contrast, the "`Hump`"-prophecies take on a more local view; their purpose is to ensure that Verifier is able to handle the current hump.

- $\texttt{SkipHump}\langle u, v, v_\tau, v_\eta, \tau, \eta\rangle = \{t \in L(\mathfrak{T}_u) \mid$ there exists a finite path $\rho = v_0 \cdots v_n$ in $\mathfrak{T}$ and a fresh finite run $r = c_0\tau_0c_1\tau_1 \cdots \tau_n c_{n+1}$ of $\mathcal{A}$ on $\mathrm{mrg}(t[0,n], \lambda(\rho))$ such that:

  $v_0 = v_\tau$, and $\tau_0 = \tau$, and τ is a skip-transition,
  $n \in \mathbb{N}$ is minimal with $\mathrm{sh}(c_0) - 1 = \mathrm{sh}(c_{n+1})$ (which we require to exist)$\}$,
  $v_n = v_\eta$, and $\tau_n = \eta$ (note that η must be a return-transition), and
  for all paths $\rho' = v'_0 \cdots v'_n$ in $\mathfrak{T}$ where $v'_0 \in \mathrm{Succ}(v)$ and $v'_n = v_n$ and
    all finite runs r' of $\mathcal{A}$ on $\mathrm{mrg}(t[0,n], \lambda(\rho'))$ of the form $c_0\tau'_0c'_1\tau'_1 \cdots c'_n\tau_n c_{n+1}$
    we have $\mathrm{FirstAcc}(r) \leq \mathrm{FirstAcc}(r')\}$.

  - The current stack top symbol (say, for illustration purposes only, it is A and the exact configuration reached is of the form $(q, \gamma A)$ for some $\gamma \in \Gamma_\bot^*$) will be removed eventually (note that this implies $A \neq \bot$).
  - We are now only interested in what happens until the current stack top symbol is removed (what happens after is handled by some "`MatchedCallStep`"-prophecy).
  - Concretely, we require that Verifier has the possibility that the run on $\mathrm{mrg}(t, t')$ continues from $(q_1, \gamma A)$ using τ (processing $(t(0), \lambda(v_\tau)) \in \Sigma^2$) as its next transition and the current stack top symbol A is removed via η (processing $(t(n), \lambda(v_\eta)) \in \Sigma^2$) after n transitions.
  - The concrete stack content γ below A is irrelevant for the calculation of the run until the current top symbol is removed, thus it suffices to specify that there is such a run starting from (q_1, A) instead, that is, it suffices to consider such a fresh run.
  - As for the other prophecies, since we use Büchi acceptance, we need to make progress towards visiting an accepting state. Hence, the finite run r must be a run that visits an accepting state as soon as possible (if possible at all) compared to all other runs that respect the same conditions on how the next return must happen.

- $\texttt{CallHump}\langle u, v, v_\eta, v_\tau, v_\vartheta, \tau, \eta, \vartheta\rangle = \{t \in L(\mathfrak{T}_u) \mid$ there exists a finite path $\rho = v_0 \cdots v_n$ in $\mathfrak{T}$ and a fresh finite run $r = c_0\tau_0c_1\tau_1 \cdots \tau_n c_{n+1}$ of $\mathcal{A}$ on $\mathrm{mrg}(t[0,n], \lambda(\rho))$ such that:

  $v_0 = v_\tau$, and $\tau_0 = \tau$, and τ is a call-transition,
  $m \in \mathbb{N}$ is minimal with $\mathrm{sh}(c_0) = \mathrm{sh}(c_{m+1})$ (which we require to exist),
  $v_m = v_\eta$, and $\tau_m = \eta$ (note that η must be a return-transition),
  $n \in \mathbb{N}$ is minimal with $\mathrm{sh}(c_0) - 1 = \mathrm{sh}(c_{n+1})$ (which we require to exist),
  $v_n = v_\vartheta$, and $\tau_n = \vartheta$ (note that ϑ must be a return-transition), and
  for all paths $\rho' = v'_0 \cdots v'_n$ in $\mathfrak{T}$ where $v'_0 \in \mathrm{Succ}(v)$ and $v'_n = v_n$
    and all finite runs r' of $\mathcal{A}$ on $\mathrm{mrg}(t[0,n], \lambda(\rho'))$ of the form $c_0\tau'_0c'_1\tau'_1 \cdots c'_n\tau_n c_{n+1}$
    we have $\mathrm{FirstAcc}(r) \leq \mathrm{FirstAcc}(r')\}$.

  - The current letter is a call (obviously matched, as all calls in a hump are). We require that the call is processed by τ (processing $(t(0), \lambda(v_\tau)) \in \Sigma^2$)) and the matching return, m transitions later, processed by η (processing $(t(m), \lambda(v_\eta)) \in \Sigma^2$).
  - Since we are already in hump, the stack symbol that was on top at the beginning will be removed too, after $n - m$ additional transitions, via ϑ (processing $(t(n), \lambda(v_\vartheta)) \in \Sigma^2$).
  - Again, we need to make progress towards visiting an accepting state. As for the previous prophecy, how the topmost stack symbol must be removed has been fixed before, when its matching call has been made. Thus, we are looking for an optimal (in terms of visiting an accepting state as soon as possible) path/run among those that, after n transitions, reach v_ϑ and use ϑ.
  - In Figure 1, this type of prophecy is used in positions 3, 5, and 11.

- $\texttt{ReturnHump}\langle u, v_\eta, \eta\rangle = \{t \in L(\mathfrak{T}_u) \mid \eta$ has the form $(q, A, (\lambda(u), \lambda(v_\eta)), q', \varepsilon)$ for some $q, q' \in Q$, and $A \in \Gamma\ \}$.
  - The topmost stack symbol is removed and it is specified how Verifier is able to do so.
  - This prophecy does not include the condition regarding visiting an accepting state as soon as possible. This is because all returns (that are not returns on the empty stack) have been fixed beforehand when their matching call was made. At that time, progress towards visiting an accepting state has been ensured. This also explains why this prophecy does not have a vertex v (symbolizing the vertex Verifier is in) as parameter, since we do not care that Verifier could also move to some $v' \in \mathrm{Succ}(v)$ with $v' \neq v_\eta \in \mathrm{Succ}(v)$.
  - This type of prophecy is used in all positions where a return happens on a nonempty stack. In Figure 1, that is, positions 4, 6, 7, 12, and 13.

For each of these prophecies, one can construct an ω-VPA accepting it. The only nontrivial aspect here is to ensure that the run r referred to in the definitions of the prophecies satisfies $\mathrm{FirstAcc}(r) \leq \mathrm{FirstAcc}(r')$ for all other runs r', which can be taken care of using projection and complementation. Here, we again rely on the stack height being controlled by the universally quantified variable only.

Recall that a fresh run is a run $r = c_0\tau_0c_1\tau_1c_2\tau_2\cdots$ such that

- $c_0 = (q, \bot A)$ for some $A \neq \bot$ and $\tau_0 = (q, A, a, q', \gamma)$, or
- $c_0 = (q, \bot)$ and $\tau_0 = (q, \bot, a, q', \gamma)$,

Note that c_0 is uniquely determined by τ_0. Hence, a fresh run is uniquely determined by the sequence $\tau_0\tau_1\tau_2\cdots$, as each c_{n+1} is uniquely determined by applying τ_n to c_n. In the following, we will often use this property.

▶ **Lemma 13.** *Each prophecy in $\mathcal{P}$ is recognized by an ω-VPA of at most exponential (in $|\mathcal{A}|$ and $|\mathfrak{T}|$) size. Furthermore, all these ω-VPA use the projected partition of $\mathcal{A}$.*

Proof. Constructing ω-VPA for the auxiliary prophecies `Step` and `ProperStep` is trivial, so let us consider $\texttt{MatchedCallStep}\langle u, v, v_\tau, v_\eta, \tau, \eta\rangle$. We first construct two auxiliary ω-VPA and then use closure properties to obtain an ω-VPA recognizing the prophecy. To this end, first consider the language L_1 of words of the form $\mathrm{mrg}(t, \rho, \tau_0\tau_1\tau_2\cdots)$ satisfying the following properties (cp. the definition of $\texttt{MatchedCallStep}\langle u, v, v_\tau, v_\eta, \tau, \eta\rangle$):

- $t \in L(\mathfrak{T}_u)$,
- $\rho = v_0v_1v_2\cdots$ is a path of $\mathfrak{T}$,
- $\tau_0\tau_1\tau_2\cdots$ is a sequence of transitions of $\mathcal{A}$ that induces a fresh run $c_0\tau_0c_1\tau_1\cdots$ of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(\rho))$ (which is uniquely determined by $\tau_0\tau_1\tau_2\cdots$),
- $v_0 = v_\tau \in \mathrm{Succ}(v)$, and $\tau_0 = \tau$, and τ is a call-transition,
- n is the smallest number such $\mathrm{sh}(c_0) = \mathrm{sh}(c_{n+1})$ (which we require to be well-defined),
- $v_n = v_\eta$ and $\tau_n = \eta$ (note, η is a return-transition), and
- r is accepting.

It expresses all but the last requirement in the definition of $\texttt{MatchedCallStep}\langle u, v, v_\tau, v_\eta, \tau, \eta\rangle$. The last one will be taken care of later.

One can construct an ω-VPA recognizing L_1 using the product of the states of $\mathcal{A}$ (to simulate the run r induced by $\tau_0\tau_1\tau_2\cdots$) and two copies of the vertices of $\mathfrak{T}$ (to check that t is in $L(\mathfrak{T}_u)$ and to guess the path ρ), and using the same stack alphabet as $\mathcal{A}$ (again, to simulate r). Furthermore, it checks all the initial constraints on v_0 and τ_0 using the state space and uses the stack to ensure that the first time the stack height $\mathrm{sh}(c_0)$ is reached, the vertex just processed in ρ is v_η and that the transition used to process it is η. This requires an additional component of the states to keep track of whether that stack height has been

reached already or not. The accepting states are inherited from $\mathcal{A}$ to ensure that the run r of $\mathcal{A}$ induced by $\tau_0\tau_1\tau_2\cdots$ is accepting, and that stack height $\mathrm{sh}(c_0)$ is reached again. Finally, the partition is the one induced by the projected one of $\mathcal{A}$, i.e., it depends only on the first component. We leave the slightly tedious, but straightforward details to the reader.

Instead, we focus on the last requirement, i.e., that there is no other accepting run r' starting in c_0 and processing $\mathrm{mrg}(t, \lambda(\rho'))$ for some path ρ' starting in some $v_0' \in \mathrm{Succ}(v)$ such that $\mathrm{FirstAcc}(r') < \mathrm{FirstAcc}(r)$. To this end, consider the language L_2 of words of the form $\mathrm{mrg}(t, \rho, \tau_0\tau_1\tau_2\cdots, \rho', \tau_0'\tau_1'\tau_2'\cdots)$ satisfying the following properties:

- $\rho = v_0v_1v_2\cdots$ is a path of $\mathfrak{T}$,
- $\rho' = v_0'v_1'v_2'\cdots$ is a path of $\mathfrak{T}$ starting in some $v_0' \in \mathrm{Succ}(v)$,
- $\tau_0\tau_1\tau_2$ is a sequence of transitions of $\mathcal{A}$ that induces a (not necessarily accepting) fresh run $c_0\tau_0c_1\tau_1\cdots$ of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(\rho))$ (which is uniquely determined by $\tau_0\tau_1\tau_2\cdots$),
- $\tau_0'\tau_1'\tau_2'$ is a sequence of transitions of $\mathcal{A}$ that induces an accepting fresh run $c_0\tau_0'c_1'\tau_1'\cdots$ of $\mathcal{A}$ on $\mathrm{mrg}(t, \lambda(\rho'))$ (which is uniquely determined by $\tau_0'\tau_1'\tau_2'\cdots$), and
- $\mathrm{FirstAcc}(r') < \mathrm{FirstAcc}(r)$.

An ω-VPA recognizing L_2 can be constructed using the product of two copies of the state space of $\mathcal{A}$ (to simulate the runs r and r') and two copies of the set of vertices of $\mathfrak{T}$ (to check that ρ and ρ' are indeed paths of $\mathfrak{T}$), and using the product of two copies of the stack alphabet of $\mathcal{A}$ (to simulate the runs r and r'). Here, we rely on the fact that $\mathcal{A}$ is controlled by the letters of t only, i.e., r and r' always have the same stack height. Again, the automaton checks the initial constraint on v_0' using its state space as well as an additional component of the state space to reject when $\mathrm{FirstAcc}(r') \geq \mathrm{FirstAcc}(r)$. The accepting states are inherited from the second copy of the states of $\mathcal{A}$ to ensure that r' is accepting. Again, the partition is the one induced by the projected one of $\mathcal{A}$, i.e., it depends only on the first component. And again, we leave the details to the reader.

Now, when projecting away the last two components of L_2 and then complementing the resulting ω-VPA we have eliminated all $\mathrm{mrg}(t, \rho, \tau_0\tau_1\tau_2\cdots)$ where the run r induced by $\tau_0\tau_1\tau_2\cdots$ is not among the ones reaching F as soon as possible. Call the resulting language L_2'. Hence, $\texttt{MatchedCallStep}\langle u, v, v_\tau, v_\eta, \tau, \eta\rangle$ is the language obtained by taking the intersection of L_1 and L_2' and then projecting away the last two components (representing ρ and $\tau_0\tau_1\tau_2\cdots$). Note that both projections we have used here are actually renamings (projections that preserve the partition into calls, returns, and skips). Hence, as ω-VPA are closed under, complementation, intersection, and renaming [1], we obtain an ω-VPA for $\texttt{MatchedCallStep}\langle u, v, v_\tau, v_\eta, \tau, \eta\rangle$, which can can be shown to have at most exponential size in $|\mathcal{A}|$ and $|\mathfrak{T}|$, as the only "expensive" operation is the complementation.

Using a similar approach, one can construct ω-VPA of exponential size (in $|\mathcal{A}|$ and $|\mathfrak{T}|$) recognizing the prophecies $\texttt{UnmatchedCallStep}\langle u, v, v_\tau, \tau\rangle$, $\texttt{SkipStep}\langle u, v, v_\tau, \tau\rangle$, and $\texttt{UnmatchedReturnStep}\langle u, v, v_\tau, \tau\rangle$. Also, the constructions can be adapted for the prophecies $\texttt{SkipHump}\langle u, v, v_\tau, v_\eta, \tau, \eta\rangle$ and $\texttt{CallHump}\langle u, v, v_\eta, v_\tau, v_\vartheta, \tau, \eta, \vartheta\rangle$: here, we are only interested in finite runs r and r' induced by the sequences $\tau_0\tau_1\tau_2\cdots$ and $\tau_0'\tau_1'\tau_2'\cdots$ until the first position where the stack height is strictly smaller than that of c_0. Note that this position only depends on t. Hence, we again obtain ω-VPA of exponential size in $|\mathcal{A}|$ and $|\mathfrak{T}|$.

Finally, the prophecy $\texttt{ReturnHump}\langle u, v_\eta, \eta\rangle$ only refers to the first letter of the input t and is is therefore trivially recognized by an ω-VPA with two states. ◀

Using the ω-VPA for the prophecies and closure properties of these automata, one can show that the winning condition of our game is also accepted by an ω-VPA.

▶ **Lemma 14.** *The winning condition $L(\mathcal{A}, \mathfrak{T}, \mathcal{P})$ is recognized by an ω-VPA of triply-exponential size (in $|\mathcal{A}|$ and $|\mathfrak{T}|$).*

Proof. In the following, all quantities (e.g., automata sizes) are measured in $|\mathcal{A}|$ and $|\mathfrak{T}|$.

There is a polynomial number of prophecies in $\mathcal{P}$. Due to Lemma 13, for every $P \in \mathcal{P}$, there is an exponentially-sized ω-VPA $\mathcal{A}_P$ recognizing P, which uses the input alphabet Σ and the projected partition of $\mathcal{A}$. The alphabet of $\mathcal{A}_P$ can be extended to the input alphabet $(V \times 2^{\{x_P | P \in \mathcal{P}\}}) \times V$ so that an input letter $((v, S), v')$ is projected to $\lambda(v)$. The resulting ω-VPA $\mathcal{A}'_P$ has the same size as $\mathcal{A}_P$. Similarly, $\mathcal{A}$ can be extended to the input alphabet $(V \times 2^{\{x_P | P \in \mathcal{P}\}}) \times V$, obtaining the ω-VPA $\mathcal{A}'$. All these extended automata use the same partition that only depends on the vertex in the first component (picked by Falsifier).

These $\mathcal{A}'_P$ can be combined into an alternating ω-VPA $\mathcal{A}_{\text{prem}}$ (see [5] for formal definitions) of exponential size recognizing

$$\forall i \in \mathbb{N}.\ \forall P \in \mathcal{P}.\ x_P \in \text{pr}_{\mathcal{P}}(\alpha(i)) \leftrightarrow \lambda(\text{pr}_V(\alpha(i)\alpha(i+1)\alpha(i+2)\cdots)) \in P.$$

Intuitively, the alternating automaton reads, for each input letter it processes, which prophecies P are currently predicted to be true (using the component $2^{\{x_P | P \in \mathcal{P}\}}$ of the input alphabet) and then spawns a fresh copy of each $\mathcal{A}'_P$ so that P is predicted to be true and spawns a copy of the dual automaton of each $\mathcal{A}'_P$ so that P is predicted to be false. The dual automaton of $\mathcal{A}'_P$ accepts the complement language of $L(\mathcal{A}'_P)$ w.r.t. the alphabet $(V \times 2^{\{x_P | P \in \mathcal{P}\}}) \times V$, but again ignores all inputs but the first vertex (picked by Falsifier). Hence, it checks that the prophecy P is indeed violated. One needs to take care that each copy spawned at stack height h treats returns that decrease the stack height below h as returns on the empty stack. We leave the straightforward details to the reader.

Now, we complement $\mathcal{A}_{\text{prem}}$ by dualizing it (without size increase) and take the disjunction with $\mathcal{A}'$, and a component that checks that if the sequence of vertices picked by Falsifier is a path in $\mathfrak{T}$, then the sequence of vertices picked by Verifier is a path in $\mathfrak{T}$ as well. Altogether, this yields an exponentially sized alternating ω-VPA recognizing $L(\mathcal{A}, \mathfrak{T}, \mathcal{P})$. This can be turned into a triply-exponential equivalent ω-VPA [5] recognizing the winning condition. ◀

Finally, one can show that the game with prophecies in $\mathcal{P}$ captures the model-checking problem. One direction of the equivalence is rather straightforward: a winning strategy for Verifier in $\mathcal{G}(L(\mathcal{A}, \mathfrak{T}, \mathcal{P}))$ can be turned into a Skolem function for π' witnessing that $\mathfrak{T} \models \forall \pi.\ \exists \pi'.\ \mathcal{A}$: given a trace t of $\mathfrak{T}$ for π, consider the play where Falsifier picks the vertices of a path with trace t and where he always picks the prophecies correctly. Then, the winning strategy yields a t' so that $\text{mrg}(t, t')$ are accepted by $\mathcal{A}$.

The (much) harder task is to show that the prophecies give Verifier enough information about the trace t Falsifier is constructing during a play to construct a "winning" t' without having full access to t. Intuitively, she always has a move so that from the resulting configuration, she can still win, while also infinitely often ensuring that an accepting state is indeed visited. Essentially, the parameters (in particular, the v_τ and the v_η) of carefully selected prophecies yield such moves.

▶ **Lemma 15.** *Let $\mathcal{P}$ be the set of prophecies introduced above. Verifier wins $\mathcal{G}(L(\mathcal{A}, \mathfrak{T}, \mathcal{P}))$ if and only if $\mathfrak{T} \models \forall \pi.\ \exists \pi'.\ \mathcal{A}$.*

Thus, combining Lemma 14, Lemma 15, and Proposition 8, we obtain our main result about HyperVPA model-checking.

▶ **Theorem 16.** *HyperVPA model-checking for $\Pi_{2,1}$ formulas is in 5ExpTime.*

▶ Remark 17. Recall that we restricted ourselves to formulas of the form $\forall\pi.\ \exists\pi'.\ \mathcal{A}$, i.e., with a single variable in each quantifier block. To generalize the construction to arbitrary $\Pi_{2,1}$ formulas (say with k universal quantifiers and k' existential ones), the prophecies are languages of k-tuples of traces and their definition existentially quantifies k' paths. So, the parameters u, v, v_τ, and v_η are replaced by vectors of vertices of length k or k', respectively.

Applying Remark 4, we also obtain decidability for the dual fragment of $\Pi_{2,1}$.

▶ **Corollary 18.** *HyperVPA model-checking for $\Sigma_{2,1}$ formulas is in 5ExpTime.*

5 Fragments of HyperVPA with Undecidable Model-Checking

In this section, we consider the model-checking problem for the fragment $\Pi_{2,2}$, i.e., formulas of the form $\forall^*\exists^*.\ \mathcal{A}$ such that the stack height in $\mathcal{A}$ only depends on the existentially quantified traces. Here, one can encode the undecidable universality problem for pushdown automata by letting the universal quantifier range over inputs w and the existential one over runs r, i.e., $\mathcal{A}$ checks that r is indeed an accepting run on w by simulating r. Hence, the stack of $\mathcal{A}$ is indeed controlled by the existentially quantified variable.

▶ **Theorem 19.** *HyperVPA model-checking for $\Pi_{2,2}$ formulas is undecidable.*

Again, by applying Remark 4 we also obtain undecidability for the dual fragment $\Sigma_{2,2}$. Further, as $\Sigma_{2,2}$ or $\Pi_{2,2}$ can be embedded in each $\Sigma_{n,j}$ and each $\Pi_{n,j}$ with $n > 2$ and $j > 1$ (using dummy variables), model-checking for these fragments is undecidable as well.

▶ **Corollary 20.** *Let $n > 1$ and $1 < j \leq n$. Then, HyperVPA model-checking for $\Sigma_{n,j}$ and $\Pi_{n,j}$ formulas is undecidable.*

Hence, we have settled the (un)decidability of all fragments but $\Sigma_{n,1}$ and $\Pi_{n,1}$ for $n > 2$.

6 Conclusion

We have introduced HyperPDA and HyperVPA to specify context-free hyperproperties, which extend ω-PDA and ω-VPA, respectively, by quantification over traces, just like HyperLTL extends LTL by trace quantification. Not surprisingly, HyperPDA model-checking is undecidable for all quantifier-fragments but Σ_1 (i.e., $\exists^*$ formulas), as the undecidable universality problem for PDA can be encoded using formulas of the form $\forall\pi_0.\ \mathcal{A}$.

For HyperVPA, the situation is better. Here, model-checking for $\Pi_{2,1}$ and $\Sigma_{2,1}$ formulas (i.e., $\forall^*\exists^*$ and $\exists^*\forall^*$ formulas where the partition into calls, returns, and skips only depends on the letters of the traces quantified in the first quantifier block) is decidable. Thus, one quantifier alternation can be handled under some mild assumptions, which covers many hyperproperties from the literature. We complemented this by showing that model-checking for $\Pi_{2,2}$ and $\Sigma_{2,2}$ (i.e., one quantifier alternation, but the partition depends only on the second block of quantifiers) is undecidable. These results also immediately imply undecidability for all $\Sigma_{n,j}$ and $\Pi_{n,j}$ for $n > 2$ and $j > 1$.

This leaves only the fragments $\Sigma_{n,1}$ and $\Pi_{n,1}$ for $n > 2$. Our undecidability proof crucially depends on the partition depending on a non-first block of quantifiers. On the other hand, the game-based characterization of HyperLTL model-checking using prophecies can be applied to arbitrary quantifier prefixes [21], at the price of requiring imperfect information games. We are currently investigating whether this approach can be lifted to $\Sigma_{n,1}$ and $\Pi_{n,1}$ relying on imperfect information games with visibly pushdown winning conditions.

References

1 Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In László Babai, editor, *STOC 2004*, pages 202–211. ACM, 2004. doi:10.1145/1007352.1007390.

2 Ali Bajwa, Minjian Zhang, Rohit Chadha, and Mahesh Viswanathan. Stack-aware hyperproperties. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *TACAS 2023, Part I*, LNCS, pages 308–325. Springer, 2023. doi:10.1007/978-3-031-30823-9_16.

3 Raven Beutner and Bernd Finkbeiner. Prophecy variables for hyperproperty verification. In *CSF 2022*, pages 471–485, , 2022. IEEE. doi:10.1109/CSF54842.2022.9919658.

4 Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR 1997*, LNCS, pages 135–150. Springer, 1997. doi:10.1007/3-540-63141-0_10.

5 Laura Bozzelli. Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007*, LNCS, pages 476–491. Springer, 2007. doi:10.1007/978-3-540-74407-8_32.

6 Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *POST 2014*, volume 8414 of *LNCS*, pages 265–284, , 2014. Springer. doi:10.1007/978-3-642-54792-8_15.

7 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

8 Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In Isil Dillig and Serdar Tasiran, editors, *CAV 2019, Part I*, volume 11561 of *LNCS*, pages 121–139, , 2019. Springer. doi:10.1007/978-3-030-25540-4_7.

9 Marie Fortin, Louwe B. Kuijer, Patrick Totzke, and Martin Zimmermann. HyperLTL satisfiability is highly undecidable, HyperCTL* is even harder. *Log. Methods Comput. Sci.*, 21(1):3, 2025. doi:10.46298/LMCS-21(1:3)2025.

10 Hadar Frenkel and Sarai Sheinvald. Realizable and context-free hyperlanguages. In Pierre Ganty and Dario Della Monica, editors, *GandALF 2022*, EPTCS, pages 114–130, 2022. doi:10.4204/EPTCS.370.8.

11 Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Deciding asynchronous hyperproperties for recursive programs. *Proc. ACM Program. Lang.*, 8(POPL):33–60, 2024. doi:10.1145/3632844.

12 Dexter Kozen. Lower bounds for natural proof systems. In *FOCS 1977*, pages 254–266. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.16.

13 Christof Löding, P. Madhusudan, and Olivier Serre. Visibly pushdown games. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004*, volume 3328 of *LNCS*, pages 408–420. Springer, 2004. doi:10.1007/978-3-540-30538-5_34.

14 Corto Mascle and Martin Zimmermann. The keys to decidable HyperLTL satisfiability: Small models or very simple formulas. In Maribel Fernández and Anca Muscholl, editors, *CSL 2020*, volume 152 of *LIPIcs*, pages 29:1–29:16, , 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2020.29.

15 Daryl McCullough. Noninterference and the composability of security properties. In *SSP 1988*, pages 177–186, , 1988. IEEE Computer Society. doi:10.1109/SECPRI.1988.8110.

16 Kurt Mehlhorn. Pebbling mountain ranges and its application of DCFL-recognition. In J. W. de Bakker and Jan van Leeuwen, editors, *ICALP 1980*, LNCS, pages 422–435. Springer, 1980. doi:10.1007/3-540-10003-2_89.

17 Amir Pnueli. The temporal logic of programs. In *FOCS 1977*, pages 46–57, , October 1977. IEEE. doi:10.1109/SFCS.1977.32.

18 Adrien Pommellet and Tayssir Touili. Model-checking HyperLTL for pushdown systems. In María-del-Mar Gallardo and Pedro Merino, editors, *SPIN 2018*, LNCS, pages 133–152. Springer, 2018. doi:10.1007/978-3-319-94111-0_8.

19 Markus N. Rabe. *A temporal logic approach to information-flow control*. PhD thesis, Saarland University, 2016. URL: http://scidok.sulb.uni-saarland.de/volltexte/2016/6387/.

20 A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985. doi:10.1145/3828.3837.

21 Sarah Winter and Martin Zimmermann. Prophecies all the way: Game-based model-checking for HyperQPTL beyond $\forall^*\exists^*$. In Patricia Bouyer and Jaco van de Pol, editors, *CONCUR 2025*, volume 348 of *LIPIcs*, pages 37:1–37:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.CONCUR.2025.37.

22 Sarah Winter and Martin Zimmermann. Tracy, traces, and transducers: computable counterexamples and explanations for HyperLTL model-checking. *Acta Informatica*, 62(3):31, 2025. doi:10.1007/S00236-025-00499-7.

23 Sarah Winter and Martin Zimmermann. Logics for context-free hyperproperties. *arXiv*, 2605.04657, 2026. doi:10.48550/arXiv.2605.04657.