International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-025-00801-5

GENERAL

Special Issue: TACAS 2023

AutoHyper: leveraging language inclusion checking for hyperproperty model-checking

Raven Beutner[1] · Bernd Finkbeiner[1]

Accepted: 4 April 2025

Abstract
Hyperproperties are system properties that relate multiple execution traces, and naturally occur, e.g., in information-flow control, knowledge, robustness, mutation testing, path planning, and causality checking. HyperLTL is a temporal logic that can express complex temporal hyperproperties by extending LTL with quantification over execution traces. Thus far, complete model-checking tools for HyperLTL have been limited to alternation-free formulas, i.e., formulas that use only universal or only existential trace quantification. In this paper, we present `AutoHyper`, an explicit-state automata-based model checker for HyperLTL that is complete for formulas with an arbitrary quantifier prefix. On the theoretical side, we show how language inclusion checks between ω-automata can be integrated into HyperLTL verification. On the practical side, this allows `AutoHyper` to leverage a range of existing inclusion-checking tools for hyperproperty verification. We further extend our model-checking algorithm to support HyperLTL modulo theories, i.e., formulas where the atomic formulas consist of first-order formulas instead of Boolean atomic propositions. We show how we can model-check such formulas effectively by tracking partially evaluated first-order formulas within an automaton. We evaluate `AutoHyper` on a broad set of benchmarks drawn from different areas in the literature and compare it with existing (incomplete) methods for HyperLTL verification.

Keywords HyperLTL · Language inclusion · Automata · Completeness · HyperLTL modulo theories

1 Introduction

In 2008, Clarkson and Schneider [23] coined the term *hyperproperties* for the rich class of requirements that relate multiple computation traces in a system. Hyperproperties are of increasing importance as they naturally occur, e.g., in information-flow control [48], robustness [19], (common) knowledge [15, 20], linearizability [40, 42], path planning [49], mutation testing [34], and counterfactual causality checking [27]. As a concrete example, assume that we are given a model of a system over a set of Boolean variables (aka. atomic propositions) $O \uplus L \uplus H$ consisting of (public) outputs O, low-security inputs L, and high-security inputs H. We want to verify that the low-security observations on the system (i.e., the public outputs O and low-security inputs L) do not allow any conclusions about the high-security input. Such an information-flow policy cannot be expressed as a trace property in, e.g., linear-time temporal logic (LTL); we need to relate multiple executions to observe how different inputs impact the output.

✉ R. Beutner
raven.beutner@cispa.de

B. Finkbeiner
finkbeiner@cispa.de

1 CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

HyperLTL A prominent logic to express temporal hyperproperties is HyperLTL, which extends LTL with explicit quantification over execution traces [24]. Using HyperLTL, we can, for example, express a simple information-flow policy – resembling observational determinism (OD) [50] – as follows:

$$\forall \pi_1 . \forall \pi_2 . \Big(\Box \bigwedge_{a \in L} a_{\pi_1} \leftrightarrow a_{\pi_2} \Big) \rightarrow \Big(\Box \bigwedge_{a \in O} a_{\pi_1} \leftrightarrow a_{\pi_2} \Big) \quad \text{(OD)}$$

This formula states that any *pair* of execution traces π_1, π_2, with identical low-security input, produces the same sequence of outputs. Phrased differently, the output of the system is deterministic with respect to the low-security input.

In many cases, (OD) is too strict; in particular, in systems where the output can be influenced by nondeterminism within the system. Instead, we can consider a weaker information-flow policy, called generalized noninterference

Published online: 06 May 2025

(GNI) [46]. Informally, GNI requires that the low-security observations are *independent* of the high-security inputs, i.e., every possible sequence of low-security observations should be compatible with every possible sequence of high-security inputs. We can, again, express this in HyperLTL as follows

$$\forall \pi_1. \forall \pi_2. \exists \pi_3. \square \Big(\bigwedge_{a \in L \cup O} a_{\pi_1} \leftrightarrow a_{\pi_3} \Big) \land \square \Big(\bigwedge_{a \in H} a_{\pi_2} \leftrightarrow a_{\pi_3} \Big). \tag{GNI}$$

That is, for *any* pair of traces π_1, π_2, *some* trace π_3 combines the low-security observations on π_1 with the high-security input on π_2.

Model checking HyperLTL We are interested in the model-checking (MC) problem of HyperLTL, i.e., deciding whether a given (finite-state) transition system satisfies a given HyperLTL formula. For HyperLTL, the structure of the quantifier prefix directly impacts the complexity of this problem, both in theory and practice. For alternation-free formulas, i.e., formulas that only use quantifiers of a single type, such as (OD), verification is reducible to the verification of an LTL trace property on a self-composition of the system [5]. This reduction has, for example, been implemented in `MCHyper` [37], a tool that can model check (alternation-free) HyperLTL formulas in systems of considerable size (circuits with thousands of latches).

Verification is much more challenging for properties involving quantifier alternations (such as (GNI)). While model-checking algorithms supporting full HyperLTL exist (see [24, 37]), they have not been implemented yet. Instead, over the years, a number of approximate approaches to the verification of such properties in practice have been made: Finkbeiner et al. [37] and D'Argenio et al. [29] manually strengthen properties with quantifier alternation into properties that are alternation-free and can be checked by `MCHyper`. Coenen et al. [26] instantiate existential quantification in a $\forall^*\exists^*$ property (i.e., a property involving an arbitrary number of universal quantifiers followed by an arbitrary number of existential quantifiers, such as (GNI)) with an explicit (user-provided) strategy, thus reducing to the verification of an alternation-free formula. Alternatively, the strategy that resolves existential quantification can be automatically synthesized [6, 11]. Hsu et al. [42] study a bounded model-checking (BMC) approach to HyperLTL by employing QBF solving. Section 3 provides a more extensive overview of existing approaches.

While all these verification tools can verify (or refute) interesting properties, they all suffer from the same limitation: they are *incomplete*. That is, for all the tools above, we can come up with model-checking instances where they fail, not necessarily due to resource constraints but because of inherent limitations in the underlying algorithm. For example, many of the benchmarks used to evaluate the BMC approach [42] do not admit a strategy to resolve existential quantification. Conversely, many of the properties verified by Coenen et al. [26] (including, e.g., (GNI)) cannot be verified using Hsu et al.'s [42] BMC encoding.

`AutoHyper` In this paper, we present `AutoHyper`, a HyperLTL model-checking tool. Our tool checks a hyperproperty against a finite-state transition system by iteratively eliminating trace quantification using automata complementations, thereby reducing verification to the emptiness check of an automaton [37]. Importantly – and unlike previous tools for HyperLTL verification – `AutoHyper` can cope with (and is *complete* for) arbitrary HyperLTL formulas. Model checking using `AutoHyper` does not require manual effort (such as writing an explicit strategy in `MCHyper` [26]), nor does a user need to worry if the given property is supported. `AutoHyper` thus provides a *push-button* model-checking experience for HyperLTL.[1]

Language inclusion checks HyperLTL model-checking is provably hard: each quantifier alteration in the quantifier prefix of a formula leads (in the worst case) to an exponential blowup in MC complexity (both in the size of the system and the size of the formula) [24, 48]. In the basic algorithm implemented in `AutoHyper` (based on [37]), this complexity is reflected in an automaton complementation for each quantifier alternation. We show that for the *outermost* quantifier alternation, we can avoid a full automaton complementation and instead reduce to a language inclusion check on Büchi automata (i.e., the question of whether the language of some Büchi automaton $\mathcal{A}$ is contained in the language of an automaton $\mathcal{B}$). While language inclusion checks between Büchi automata follow the same theoretical complexity, they admit many approaches that scale well in practice. In practice, this enables `AutoHyper` to resort to a range of mature language inclusion checkers, including `spot` [33], `RABIT` [25], `BAIT` [31], and `FORKLIFT` [32]. In particular, for formulas with at most one quantifier alternation (which covers many properties of interest; see Sect. 9), `AutoHyper` can delegate the model-checking complexity almost entirely to an external inclusion checker.

HyperLTL modulo theories In many cases, the variables of a (finite-state) system are not Boolean but come from richer domains (e.g., the set of integers). We extend HyperLTL by including first-order formulas (modulo some fixed background theory $\mathfrak{T}$) as atomic expressions,

[1] The name of `AutoHyper` is derived from the fact that it is both **Auto**mata-based and fully **Auto**matic.

called HyperLTL$_{\mathfrak{T}}$. That is, instead of accessing traces at the level of atomic propositions [24], we can reason about relational formulas over complex data types. For example, $\forall \pi_1.\exists \pi_2.\Box(x_{\pi_2} \geq z_{\pi_1}) \land \Diamond y_{\pi_2}$ states that for any trace π_1, there exists some trace π_2 such that on π_2 the value of the (integer-valued) variable x is (globally) greater than or equal to the value of z on π_1, and the (Boolean-valued) variable y is eventually set to *true* on π_2. We show how we can extend the automata-based model-checking framework to support relational first-order formulas as atomic formulas by tracking partially evaluated formulas within an automaton.

Evaluation We evaluate `AutoHyper` on a broad range of benchmarks taken from the literature and compare it with existing (incomplete) tools. We observe that – at least on the currently available benchmarks – explicit-state model-checking as implemented in `AutoHyper` performs on-par (and frequently outperforms) symbolic methods such as BMC [42]. Our benchmarks stem from various areas within computer science, so `AutoHyper` should, thanks to its push-button functionality, completeness, and ease of use, be a valuable addition to many areas.

Apart from using `AutoHyper` as a practical MC tool, we can also use it as a complete baseline to systematically evaluate existing (incomplete) methods. For example, while it is known that replacing existential quantification with a strategy (see [6, 26]) is incomplete, it was, thus far, unknown if this incompleteness occurs frequently. We use `AutoHyper` to obtain a ground truth and evaluate the strategy-based verification approach in terms of its effectiveness (i.e., how many instances it can verify despite being incomplete) and efficiency.

Structure The remainder of this paper is structured as follows. In Sect. 2, we introduce HyperLTL, and Sect. 3 recaps existing approaches for the verification of HyperLTL formulas. We present our automata-based model-checking algorithm and our language-inclusion-based optimization in Sect. 4. In Sect. 5, we extend HyperLTL with a background theory (called HyperLTL$_{\mathfrak{T}}$) and modify our model-checking algorithm in Sect. 6. We present technical background on `AutoHyper` in Sect. 7. In Sect. 8, we evaluate `AutoHyper` on a set of benchmarks from the literature and compare it with the bounded model checker `HyperQB` [42]. Finally, in Sect. 9, we use `AutoHyper` for a detailed analysis of (and comparison with) strategy-based HyperLTL verification [6, 26].

This paper is an extended version of a preliminary conference version [8]. Compared to the conference version, we simplify the technical sections by using nondeterministic and universal automata, extend our model-checking algorithm to support HyperLTL modulo theories (see Sect. 6), and discuss additional optimizations such as bisimulation-based preprocessing. Additionally, we evaluate `AutoHyper` on a more extensive set of benchmarks, showcasing the practical impact of our extensions and optimizations.

2 HyperLTL and transition systems

We first present the standard variant of HyperLTL that accesses traces at the level of (Boolean-valued) atomic propositions [24] and its semantics over Boolean-valued transition systems. In Sect. 5, we then extend HyperLTL to support relational expressions.

HyperLTL We fix a finite set AP of atomic propositions (APs). HyperLTL [24] extends LTL with explicit quantification over traces, thereby lifting it from a logic expressing trace properties to one expressing hyperproperties. Let $\mathcal{V} = \{\pi, \pi_1, \pi_2, \dots\}$ be a set of trace variables. We define HyperLTL formulas by the following grammar:

$$\psi := a_\pi \mid \neg\psi \mid \psi \land \psi \mid \bigcirc\psi \mid \psi\, \mathcal{U}\, \psi$$
$$\varphi := \exists \pi.\, \varphi \mid \forall \pi.\, \varphi \mid \psi$$

where $\pi \in \mathcal{V}$ is a trace variable, and $a \in AP$ is an atomic proposition. We assume that the formula is closed, i.e., all trace variables used in the LTL body are bound by some quantifier. We use the usual derived Boolean constants and connectives *true*, *false*, $\lor, \rightarrow, \leftrightarrow$ and temporal operators $\Diamond\psi := \mathit{true}\, \mathcal{U}\, \psi$ and $\Box\psi := \neg\Diamond\neg\psi$.

Semantics A trace is an infinite sequence $t \in (2^{AP})^\omega$. The semantics of HyperLTL is given with respect to a trace assignment $\Pi : \mathcal{V} \rightharpoonup (2^{AP})^\omega$, which is a partial mapping from trace variables to traces. For $\pi \in \mathcal{V}$ and $t \in (2^{AP})^\omega$, we write $\Pi[\pi \mapsto t]$ for the trace assignment obtained by updating the value of π to t. Given a set of traces $\mathbb{T} \subseteq (2^{AP})^\omega$, a trace assignment Π, and $i \in \mathbb{N}$, we define:

$$\begin{array}{lll}
\Pi, i \models a_\pi & \text{iff} & a \in \Pi(\pi)(i) \\
\Pi, i \models \neg\psi & \text{iff} & \Pi, i \not\models \psi \\
\Pi, i \models \psi_1 \land \psi_2 & \text{iff} & \Pi, i \models \psi_1 \text{ and } \Pi, i \models \psi_2 \\
\Pi, i \models \bigcirc\psi & \text{iff} & \Pi, i+1 \models \psi \\
\Pi, i \models \psi_1\, \mathcal{U}\, \psi_2 & \text{iff} & \exists j \geq i.\, \Pi, j \models \psi_2 \text{ and} \\
 & & \quad \forall i \leq k < j.\, \Pi, k \models \psi_1 \\
\Pi \models_{\mathbb{T}} \psi & \text{iff} & \Pi, 0 \models \psi \\
\Pi \models_{\mathbb{T}} \exists \pi.\, \varphi & \text{iff} & \exists t \in \mathbb{T}.\, \Pi[\pi \mapsto t] \models_{\mathbb{T}} \varphi \\
\Pi \models_{\mathbb{T}} \forall \pi.\, \varphi & \text{iff} & \forall t \in \mathbb{T}.\, \Pi[\pi \mapsto t] \models_{\mathbb{T}} \varphi
\end{array}$$

The atomic formula a_π holds in the ith step whenever AP a holds in ith step on the trace bound to π (i.e., $a \in \Pi(\pi)(i)$);

Table 1 We group existing verification approaches for HyperLTL based on their strengths. We write ● when the verification method (the row) has the given property (column) and ○ if it does not

	Guaranteed Soundness	Arbitrary Alternations	Symbolic Systems	Fully Automated	NL Complexity	Completeness
Self-Composition [29, 37]	●	○	●	●	●	●
Manual Strengthening [29, 37]	○	○	●	○	●	○
SBV [User-Provided Strategy] [26]	●	○	●	○	●	○
SBV [Strategy Synthesis] [6]	●	○	●	●	○	○
BMC [42]	●	●	●	●	○	○
Explicit-State MC [This work]	●	●	○	●	○	●

Boolean and temporal operators are evaluated as expected; Quantification ranges over traces in $\mathbb{T}$ and binds them to the respective trace variable.

Transition systems We evaluate HyperLTL over finite-state transition systems.

Definition 1
A *transition system* is a tuple $\mathcal{T} = (S, S_0, \kappa, \ell)$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $\kappa \subseteq S \times S$ is a transition relation, and $\ell : S \to 2^{AP}$ is a labeling function. △

A path in $\mathcal{T}$ is an infinite sequence $s_0 s_1 s_2 \cdots \in S^\omega$ such that $s_0 \in S_0$ and $(s_i, s_{i+1}) \in \kappa$ for all $i \in \mathbb{N}$. The associated trace is given by $\ell(s_0)\ell(s_1)\ell(s_2) \cdots \in (2^{AP})^\omega$. We write $\mathit{Traces}(\mathcal{T}) \subseteq (2^{AP})^\omega$ for the set of all traces generated by $\mathcal{T}$. A transition system $\mathcal{T}$ satisfies a HyperLTL formula φ, written $\mathcal{T} \models \varphi$, if $\emptyset \models_{\mathit{Traces}(\mathcal{T})} \varphi$, where $\emptyset$ denotes the empty trace assignment (i.e., the partial function $\mathcal{V} \rightharpoonup (2^{AP})^\omega$ with empty domain).

3 Related work

HyperLTL is among the most studied logics for expressing temporal hyperproperties. A range of problems from different areas in computer science can be expressed as HyperLTL MC problems, including (optimal) path planning [49], mutation testing [34], linearizability [42], robustness [19, 29], information-flow control [48], causality checking [16, 27, 36], and (common) knowledge [15, 20]. Consequently, any model-checking tool for HyperLTL is applicable to computational problems in various application areas and provides a unified solution to many challenging algorithmic problems.

In recent years, different (mostly incomplete) verification tools for HyperLTL have been developed. Table 1 contains an informal overview of their respective strengths and weaknesses. We discuss the approaches below.

Alternation-free HyperLTL Verification of HyperLTL formulas without quantifier alternations can be reduced to checking an LTL property on the self-composition of the system [5, 37]. This approach is very efficient: it applies to systems where the state-space is symbolically presented (e.g., `AIGER` circuits or `NuSMV` models [22]) and follows the same model-checking complexity as LTL, i.e., it is `NL`-complete in the size of the (explicitly-represented) system. `McHyper` [37] implements this self-composition for HyperLTL by utilizing `ABC` [21] for the LTL verification on `AIGER` circuits.

Manual strengthening Although `McHyper` only applies to alternation-free formulas, it can aid in the verification of quantifier alternations. Finkbeiner et al. [37] and D'Argenio et al. [29] use `McHyper` by *manually* strengthening alternating formulas into alternation-free formulas. For example, a $\forall \pi_1 . \exists \pi_2$ formula can be strengthened by replacing the existential quantification with a universal one and, possibly, adding temporal premises on π_2. The soundness of this strengthening must be argued manually and cannot be checked automatically.

Strategy-based verification Instead of using (manual) ad hoc strengthening, Coenen et al. [26] study the idea of using strategies to formalize sound-by-design strengthenings. In this approach – which we refer to as strategy-based verification (SBV) – existential quantification in a $\forall^*\exists^*$ formula is resolved (i.e., strengthened) by using an explicit strategy that governs how the trace is constructed. This strategy is either provided by the user or synthesized automatically. In the former case, model checking reduces to checking an alternation-free formula and can thus handle large systems but requires significant user effort. In the latter case, the method works fully automatically but requires an expensive strategy synthesis [6, 10, 12]. Analogously, the search for a strategy can also be seen as the search for a simulation relation [43]. SBV is incomplete as the strategy resolving existentially quantified traces only observes finite prefixes of the universally quantified traces. Although SBV can be made complete by adding prophecy variables [6], the automatic synthesis of such prophecies is currently limited to very small systems and properties that are temporally safe [6, 14]. We investigate both the performance and incompleteness of SBV in Sect. 9.

Bounded model checking for HyperLTL Hsu et al. [42] propose a bounded model-checking (BMC) procedure for HyperLTL. Similar to BMC for trace properties [18], the system is unfolded up to a fixed depth, and pending obligations beyond that depth are either treated pessimistically (to show the satisfaction of a formula) or optimistically (to show the violation of a formula). While BMC for trace properties reduces to SAT-solving, BMC for hyperproperties naturally reduces to QBF-solving. We compare `AutoHyper` and BMC (in the form of the `HyperQB` tool [42]) in Sect. 8. As usual for bounded methods, BMC for HyperLTL is incomplete. For example, it can never show that a system satisfies a hyperproperty where the LTL body contains an invariant (as, e.g., is the case for (GNI)). In follow-up work, Hsu et al. [43] suggest a simulation-based unrolling, which corresponds directly to strategy-based verification.

4 Automata-based model-checking

Given a fixed transition system $\mathcal{T} = (S, S_0, \kappa, \ell)$ and HyperLTL property $\dot{\varphi}$, we want to determine whether $\mathcal{T} \models \dot{\varphi}$ (we use the dot to refer to the original formula and will later use φ, φ' to refer to subformulas of $\dot{\varphi}$). In this section, we recap the automata-based approach to the model checking of HyperLTL [37]. We further show how language inclusion checks can be incorporated into the model-checking procedure.

4.1 Automata

The idea of automata-based model-checking [37] is to iteratively eliminate quantifiers and thus reduce model-checking to the emptiness check on an automaton.

Definition 2
A nondeterministic Büchi automaton (NBA) (resp., universal co-Büchi automaton, UCA) over some finite alphabet Σ is a tuple $\mathcal{A} = (Q, Q_0, \delta, F)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and $F \subseteq Q$ is a set of accepting (resp., rejecting) states. △

A *run* of $\mathcal{A}$ on a word $u \in \Sigma^\omega$ is an infinite sequence of states $q_0 q_1 q_2 \cdots \in Q^\omega$ such that $q_0 \in Q_0$ and $(q_i, u(i), q_{i+1}) \in \delta$ for every $i \in \mathbb{N}$. A word $u \in \Sigma^\omega$ is accepted by an NBA $\mathcal{A}$ if there exists *some* run on u that visits states in F *infinitely* many times. A word $u \in \Sigma^\omega$ is accepted by a UCA $\mathcal{A}$ if *all* runs on u visit states in F only *finitely* many times. Given an NBA or UCA $\mathcal{A}$, we write $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ for the set of infinite words accepted by $\mathcal{A}$. Note that we can translate NBAs into equivalent UCAs and vice versa with an exponential blowup using, e.g., automata complementation.

4.2 $\mathcal{T}$-Equivalence

The semantics of HyperLTL is based on trace assignments $\Pi : V \to (2^{AP})^\omega$ for some $V \subseteq \mathcal{V}$ (see Sect. 2). To manipulate trace assignments using automata, we *zip* them into infinite traces.

Definition 3
For a trace assignment $\Pi : V \to (2^{AP})^\omega$ (where $V \subseteq \mathcal{V}$ is the domain of the assignment), we define the trace $zip(\Pi) \in (2^{AP \times V})^\omega$ by

$$zip(\Pi)(i) := \Big\{(a, \pi) \mid a \in AP \land \pi \in V \land a \in \Pi(\pi)(i)\Big\}$$

for each $i \in \mathbb{N}$. △

In other words, $(a, \pi) \in AP \times V$ holds on $zip(\Pi)$ in the ith step iff a holds in the ith step on trace $\Pi(\pi)$. Note that zip defines a bijection between trace assignments $\Pi : V \to (2^{AP})^\omega$ and traces in $(2^{AP \times V})^\omega$.

Our iterative algorithm is now based on the idea that using the bijection created by zip, we can use automata to summarize trace assignments that satisfy subformulas of $\dot{\varphi}$. We formalize this using the concept of $\mathcal{T}$-equivalence [37]:

Definition 4
Let φ be a HyperLTL formula with free trace variables $V \subseteq \mathcal{V}$. An automaton $\mathcal{A}$ over alphabet $2^{AP \times V}$ is $\mathcal{T}$*-equivalent to* φ if for every trace assignments $\Pi : V \to (2^{AP})^\omega$, we have $\Pi \models_{Traces(\mathcal{T})} \varphi$ if and only if $zip(\Pi) \in \mathcal{L}(\mathcal{A})$. △

In other words, $\mathcal{A}$ accepts exactly the zippings of traces that constitute a satisfying trace assignment for φ.

4.3 Product construction

To check if $\mathcal{T} \models \dot{\varphi}$, we inductively construct an automaton $\mathcal{A}_\varphi$ that is $\mathcal{T}$-equivalent to φ for each subformula φ of $\dot{\varphi}$. For the (quantifier-free) LTL body of $\dot{\varphi}$, we can construct this automaton via a standard LTL-to-NBA construction [3, 37]. We then iteratively eliminate quantifiers by computing the product with the given system $\mathcal{T}$ (to eliminate trace quantifiers).

- **Case** $\varphi' = \exists \pi. \varphi$**:** We are given an inductively constructed automaton $\mathcal{A}_\varphi = (Q, Q_0, \delta, F)$ over alphabet $2^{AP \times (V \uplus \{\pi\})}$ for some $V \subseteq \mathcal{V}$ that is $\mathcal{T}$-equivalent to φ. We ensure that $\mathcal{A}_\varphi$ is an NBA (by possibly translating a UCA into an

NBA via complementation) and define the NBA $\mathcal{A}_{\varphi'}$ over alphabet $2^{AP \times V}$ as

$$\mathcal{A}_{\varphi'} := (S \times Q, S_0 \times Q_0, \delta', S \times F),$$

where $\delta' \subseteq (S \times Q) \times 2^{AP \times V} \times (S \times Q)$ is defined as

$$\delta' := \Big\{ \big((s,q), \sigma, (s',q')\big) \mid (s,s') \in \kappa \wedge \\ \Big(q, \sigma \uplus \big\{(a,\pi) \mid a \in \ell(s)\big\}, q'\Big) \in \delta \Big\}.$$

Intuitively, $\mathcal{A}_{\varphi'}$ reads the zipping of a trace assignment $V \to (2^{AP})^\omega$, guesses a trace in $\mathcal{T}$ for trace variable π, and simulates the zipping of the extended trace assignment $(V \uplus \{\pi\}) \to (2^{AP})^\omega$ in $\mathcal{A}_\varphi$ by adding the π-indexed guessed trace $(\big\{(a,\pi) \mid a \in \ell(s)\big\})$ to each letter $\sigma \in 2^{AP \times V}$.

– **Case** $\varphi' = \forall \pi. \varphi$**:** We are given an automaton $\mathcal{A}_\varphi$ over alphabet $2^{AP \times (V \uplus \{\pi\})}$ that is $\mathcal{T}$-equivalent to φ. We ensure that this automaton is a UCA (by possibly translating an NBA into a UCA via complementation) and define $\mathcal{A}_{\varphi'}$ as the UCA over $2^{AP \times V}$ that is syntactically identical to the NBA constructed in the previous case.

Proposition 1
For every subformula φ of $\dot{\varphi}$, the automaton $\mathcal{A}_\varphi$ is $\mathcal{T}$-equivalent to φ.

As the final formula $\dot{\varphi}$ is closed (i.e., contains no free variables), we eventually obtain a $\mathcal{T}$-equivalent automaton $\mathcal{A}_{\dot{\varphi}}$ over the singleton alphabet $2^{AP \times \emptyset} = 2^\emptyset$. By the definition of $\mathcal{T}$-equivalence, we then have $\mathcal{T} \models \dot{\varphi}$ iff $\emptyset \models_{Traces(\mathcal{T})} \dot{\varphi}$ iff $zip(\emptyset) \in \mathcal{L}(\mathcal{A}_{\dot{\varphi}})$. Checking if $\mathcal{T} \models \dot{\varphi}$ thus reduces to a word containment check for $zip(\emptyset)$ in $\mathcal{A}_{\dot{\varphi}}$ (or, equivalently, as the alphabet of $\mathcal{A}_{\dot{\varphi}}$ is a singleton set, we can check if $\mathcal{L}(\mathcal{A}_{\dot{\varphi}})$ is nonempty).

4.4 Leveraging language inclusion

The algorithm outlined above requires one complementation for each quantifier alternation in the HyperLTL formula. While we cannot avoid the theoretical cost of this complementation (see [24, 48]), we can reduce it to a problem that is more tamable in practice: *language inclusion*.

For a transition system $\mathcal{T}$ and a finite set of trace variables $V \subseteq \mathcal{V}$, we define an NBA $\mathcal{A}_{\mathcal{T},V}$ over alphabet $2^{AP \times V}$ that accepts the AP evaluations of all possible trace combinations in $\mathcal{T}$. That is, for any trace assignment $\Pi : V \to (2^{AP})^\omega$, we have that $zip(\Pi) \in \mathcal{L}(\mathcal{A}_{\mathcal{T},V})$ iff $\Pi(\pi) \in Traces(\mathcal{T})$ for all $\pi \in V$. We can easily construct $\mathcal{A}_{\mathcal{T},V}$ by construction of a $|V|$-fold self-composition of $\mathcal{T}$ [5]:

Definition 5
Define the NBA $\mathcal{A}_{\mathcal{T},V}$ over alphabet $2^{AP \times V}$ as $\mathcal{A}_{\mathcal{T},V} := (Q, Q_0, \delta, F)$, where

– $Q := (V \to S)$, i.e., the state-space consists of all functions $V \to S$ (which is a finite set as V and S are both finite). Each state $q \in Q$ thus tracks a state $q(\pi) \in S$ for all trace variables $\pi \in V$;
– $Q_0 := \big\{q \in Q \mid \forall \pi \in V. q(\pi) \in S_0\big\}$, i.e., the initial states are all state combinations where each trace $\pi \in V$ starts in an initial state of $\mathcal{T}$;
– $\delta \subseteq Q \times 2^{AP \times V} \times Q$ is defined as

$$\delta := \Big\{ (q, \sigma, q') \mid \forall \pi \in V. \big(q(\pi), q'(\pi)\big) \in \kappa \wedge \\ \sigma = \big\{(a,\pi) \mid a \in AP, \pi \in V, a \in \ell\big(q(\pi)\big)\big\} \Big\},$$

i.e., there is a transition from q to q' if for each $\pi \in V$, the state tracked for π is updated according to some transition in $\mathcal{T}$ (i.e., $\big(q(\pi), q'(\pi)\big) \in \kappa$). The unique label of this transition contains exactly those indexed APs (a, π) where a holds in the state assigned to π (i.e., $a \in \ell\big(q(\pi)\big)$);
– $F := Q$, i.e., we mark all states as accepting. △

We can now state a formal connection between language inclusion and HyperLTL model-checking.

Proposition 2
Let $\dot{\varphi} = \forall \pi_1. \ldots \forall \pi_n. \varphi$ be a HyperLTL formula (where φ may contain additional trace quantifiers), and let $\mathcal{A}_\varphi$ be an automaton over $2^{AP \times \{\pi_1, \ldots, \pi_n\}}$ that is $\mathcal{T}$-equivalent to φ. Then $\mathcal{T} \models \dot{\varphi}$ if and only if $\mathcal{L}(\mathcal{A}_{\mathcal{T},\{\pi_1,\ldots,\pi_n\}}) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$.

Proof
For the first direction, assume that $\mathcal{L}(\mathcal{A}_{\mathcal{T},\{\pi_1,\ldots,\pi_n\}}) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$. We need to show that $\mathcal{T} \models \forall \pi_1. \ldots \forall \pi_n. \varphi$. To this end, let $t_1, \ldots, t_n \in Traces(\mathcal{T})$ be arbitrary traces, and define the trace assignment $\Pi : \{\pi_1, \ldots, \pi_n\} \to (2^{AP})^\omega$ by $\Pi = [\pi_1 \mapsto t_1, \ldots, \pi_n \mapsto t_n]$. We need to show that $\Pi \models_{Traces(\mathcal{T})} \varphi$. By the definition of $\mathcal{A}_{\mathcal{T},\{\pi_1,\ldots,\pi_n\}}$ we know that $zip(\Pi) \in \mathcal{L}(\mathcal{A}_{\mathcal{T},\{\pi_1,\ldots,\pi_n\}})$. As the inclusion holds, we get that $zip(\Pi) \in \mathcal{L}(\mathcal{A}_\varphi)$. By the $\mathcal{T}$-equivalence of $\mathcal{A}_\varphi$ this implies $\Pi \models_{Traces(\mathcal{T})} \varphi$, as required.

For the reverse, we assume that $\mathcal{T} \models \dot{\varphi}$. To show $\mathcal{L}(\mathcal{A}_{\mathcal{T},\{\pi_1,\ldots,\pi_n\}}) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$, let $t \in \mathcal{L}(\mathcal{A}_{\mathcal{T},\{\pi_1,\ldots,\pi_n\}})$ be arbitrary. By the definition of $\mathcal{A}_{\mathcal{T},\{\pi_1,\ldots,\pi_n\}}$ we get that there exist traces $t_1, \ldots, t_n \in Traces(\mathcal{T})$ such that $t = zip([\pi_1 \mapsto t_1, \ldots, \pi_n \mapsto t_n])$. As $\mathcal{T} \models \forall \pi_1. \ldots \forall \pi_n. \varphi$, we get $[\pi_1 \mapsto t_1, \ldots, \pi_n \mapsto t_n] \models_{Traces(\mathcal{T})} \varphi$. By the $\mathcal{T}$-equivalence this implies $t = zip([\pi_1 \mapsto t_1, \ldots, \pi_n \mapsto t_n]) \in \mathcal{L}(\mathcal{A}_\varphi)$, as required. □

We can use Proposition 2 to avoid a complementation for the outermost quantifier alternation. For example, assume that $\dot{\varphi} = \forall \pi_1. \forall \pi_2. \exists \pi_3. \psi$, where ψ is the quantifier-free LTL body. Using the product construction (Sect. 4.3), we obtain an NBA $\mathcal{A}_{\exists \pi_3.\psi}$ that is $\mathcal{T}$-equivalent to $\exists \pi_3.\psi$. Notably, we can construct $\mathcal{A}_{\exists \pi_3.\psi}$ in linear time in the size of $\mathcal{T}$. By Proposition 2 we then have $\mathcal{T} \models \dot{\varphi}$ iff $\mathcal{L}(\mathcal{A}_{\mathcal{T},\{\pi_1,\pi_2\}}) \subseteq \mathcal{L}(\mathcal{A}_{\exists \pi_3.\psi})$.

Note that complementation and subsequent emptiness check is a theoretically optimal method to check for language inclusion. Proposition 2 thus offers no asymptotic advantages over complementation-based model-checking (see Sect. 4.3). In *practice*, constructing an explicit complemented automaton is often unnecessary as language inclusion or noninclusion might be witnessed without a complete complementation [25, 31–33, 45]. This makes Proposition 2 relevant to the present work and the performance of `AutoHyper`.

5 HyperLTL modulo theories

The HyperLTL variant we have studied so far accesses the traces at the level of atomic propositions, which we can think of as Boolean variables within a system. However, in practice, many systems use complex non-Boolean data, which requires costly preprocessing (we discuss this later in Sect. 6.1). In this section, we extend HyperLTL with atomic formulas from a first-order background theory, called HyperLTL$_\mathfrak{T}$, which allows us to express complex relations between datatypes beyond Booleans. Similar to extensions of LTL and LTL$_f$ with theories [1, 30, 38, 39, 44], we allow (quantifier-free) first-order formulas as atomic expressions; in our case, over variables indexed by trace variables.

5.1 First-order theories and formulas

First-order theories A *first-order signature* is a pair $\mathfrak{S} = (\mathfrak{F}, \mathfrak{P})$, where $\mathfrak{F}$ is a set of function symbols, and $\mathfrak{P}$ is a set of predicate symbols. Each $f \in \mathfrak{F}$ and $P \in \mathfrak{P}$ has an associated arity. We consider all functions and predicate symbols to be interpreted, and, for simplicity, consider a theory as consisting of a single such interpretation [4]. Formally, a *theory* for signature $(\mathfrak{F}, \mathfrak{P})$ is a pair $\mathfrak{T} = (\mathbb{V}, \mathcal{I})$, where $\mathbb{V}$ is a set of values (called the universe), and $\mathcal{I}$ maps each function symbol $f \in \mathfrak{F}$ of arity n to a total function $f^\mathcal{I} : \mathbb{V}^n \to \mathbb{V}$ and each predicate symbol $P \in \mathfrak{P}$ of arity n to a relation $P^\mathcal{I} \subseteq \mathbb{V}^n$. In the following, we assume some fixed signature $\mathfrak{S}$ and theory $\mathfrak{T}$.

Terms and formulas We assume that $\mathcal{X}$ is a set of variables and define terms inductively as $t := x \mid f(t_1, \ldots, t_n)$, where $x \in \mathcal{X}$ is a variable, $f \in \mathfrak{F}$ is a function symbol of arity n, and $t_1, \ldots, t_n$ are terms. Likewise, quantifier-free formulas are inductively defined as $\theta := \neg\theta \mid \theta \wedge \theta \mid P(t_1, \ldots, t_n)$, where $P \in \mathfrak{P}$ is a predicate symbol with arity n, and $t_1, \ldots, t_n$ are terms. We write $\mathfrak{S}_\mathcal{X}$ for the set of all formulas over variables $\mathcal{X}$ (within the fixed signature $\mathfrak{S}$). A variable evaluation $\Delta : \mathcal{X} \to \mathbb{V}$ satisfies a formula $\theta \in \mathfrak{S}_\mathcal{X}$ in theory $\mathfrak{T}$, written $\Delta \models_\mathfrak{T} \theta$, if θ evaluates to *true* (defined as expected). We assume that for every variable-free formula $\theta \in \mathfrak{S}_\emptyset$, we can decide if $\emptyset \models_\mathfrak{T} \theta$ (where $\emptyset$ denotes the variable evaluation $\emptyset \to \mathbb{V}$ with empty domain).

Substitution A key technique we will exploit in our model-checking algorithm is the ability to substitute (a subset of) variables with concrete values. Given a formula $\theta \in \mathfrak{S}_\mathcal{X}$ over a set of variables $\mathcal{X}$, a subset of variables $\mathcal{Y} \subseteq \mathcal{X}$, and a variable assignment $\Delta : \mathcal{Y} \to \mathbb{V}$, we say that a formula θ' over variables $\mathcal{X} \setminus \mathcal{Y}$ is a Δ-*substitution of* θ if for any assignment $\Delta' : (\mathcal{X} \setminus \mathcal{Y}) \to \mathbb{V}$, we have $\Delta' \models_\mathfrak{T} \theta'$ iff $\Delta \uplus \Delta' \models_\mathfrak{T} \theta$. Note that Δ-substitutions are defined purely semantically and there may exist many (or no) Δ-substitutions of a given formula. If the first-order signature contains constants (i.e., nullary functions) for all values in $\mathbb{V}$, then we can construct at least one Δ-substitution of θ by simply replacing each variable $y \in \mathcal{Y}$ with the constant that corresponds to value $\Delta(y)$.

Example 1
Consider $\theta := (x > y + 5z) \in \mathfrak{S}_{\{x,y,z\}}$ over linear-integer arithmetic (LIA) and $\Delta := [y \mapsto 6]$. A possible Δ-substitution of θ is $x > 6 + 5z$. $\triangle$

We assume that for all Δ and θ, we can effectively compute at least one Δ-substitution of θ and denote it by $\theta \circ \Delta$.

5.2 HyperLTL$_\mathfrak{T}$

We now define HyperLTL$_\mathfrak{T}$ as an extension of HyperLTL that allows relational formulas from signature $\mathfrak{S}$ as atomic structures which are evaluated according to theory $\mathfrak{T}$. We, again, assume that $\mathcal{V} = \{\pi, \pi_1, \ldots\}$ is a set of trace variables and fix a set of system variables $\mathcal{X}$. For any $\pi \in \mathcal{V}$, we define $\mathcal{X}_\pi := \{x_\pi \mid x \in \mathcal{X}\}$ as a set of indexed system variables. For a set of trace variables $V \subseteq \mathcal{V}$, we abbreviate $\mathcal{X}_V := \bigcup_{\pi \in V} \mathcal{X}_\pi$. HyperLTL$_\mathfrak{T}$ formulas are then defined as follows:

$$\psi := \theta \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi$$
$$\varphi := \exists \pi.\, \varphi \mid \forall \pi.\, \varphi \mid \psi$$

where $\pi \in \mathcal{V}$ is a trace variable, and $\theta \in \mathfrak{S}_{\mathcal{X}_\mathcal{V}}$ is a formula over $\mathcal{X}_\mathcal{V}$, i.e., over all possible indexed variables. Notably, compared to plain HyperLTL (Sect. 2), we do not use trace-variable-indexed atomic propositions as atomic formulas, but rather arbitrary first-order formulas over (trace-variable-indexed) system variables. We assume that the formula is closed, i.e., for all formulas θ used in ψ and any variable x_π used in θ, the trace variable π is bound by some trace quantifier.

Semantics We evaluate a HyperLTL$_\mathfrak{T}$ formula over traces in $(\mathcal{X} \to \mathbb{V})^\omega$, i.e., each position in the trace defines a value for each system variable. A trace assignment $\Pi : \mathcal{V} \rightharpoonup (\mathcal{X} \to \mathbb{V})^\omega$ now maps trace variables to traces. Given a trace assignment $\Pi : V \to (\mathcal{X} \to \mathbb{V})^\omega$ with domain $V \subseteq \mathcal{V}$ and position $i \in \mathbb{N}$, we define the extended variable evaluation $\Pi_{(i)} : \mathcal{X}_V \to \mathbb{V}$ by $\Pi_{(i)}(x_\pi) := \Pi(\pi)(i)(x)$, i.e., the value of indexed variable x_π is defined as the value of variable x on the trace bound to π in the ith step (see [7, 13]). Given a set of traces $\mathbb{T} \subseteq (\mathcal{X} \to \mathbb{V})^\omega$, a trace assignment Π, and $i \in \mathbb{N}$, we define:

$$
\begin{array}{lll}
\Pi, i \models \theta & \text{iff} & \Pi_{(i)} \models_\mathfrak{T} \theta \\
\Pi, i \models \neg\psi & \text{iff} & \Pi, i \not\models \psi \\
\Pi, i \models \psi_1 \land \psi_2 & \text{iff} & \Pi, i \models \psi_1 \text{ and } \Pi, i \models \psi_2 \\
\Pi, i \models \bigcirc \psi & \text{iff} & \Pi, i+1 \models \psi \\
\Pi, i \models \psi_1 \mathcal{U} \psi_2 & \text{iff} & \exists j \geq i. \Pi, j \models \psi_2 \text{ and} \\
& & \quad \forall i \leq k < j. \Pi, k \models \psi_1 \\
\Pi \models_\mathbb{T} \psi & \text{iff} & \Pi, 0 \models \psi \\
\Pi \models_\mathbb{T} \exists \pi. \varphi & \text{iff} & \exists t \in \mathbb{T}. \Pi[\pi \mapsto t] \models_\mathbb{T} \varphi \\
\Pi \models_\mathbb{T} \forall \pi. \varphi & \text{iff} & \forall t \in \mathbb{T}. \Pi[\pi \mapsto t] \models_\mathbb{T} \varphi
\end{array}
$$

Temporal operators, Boolean operators, and quantifiers are evaluated as before. Whenever we evaluate a first-order formula θ in the ith step, we evaluate θ under variable evaluation $\Pi_{(i)}$ in the fixed theory $\mathfrak{T}$.

5.3 Extended transition systems

The full potential of HyperLTL$_\mathfrak{T}$ comes from its ability to directly reason about non-Boolean variables. For example, when checking hyperproperties on NuSMV models (see Sect. 8), we naturally obtain a finite-state system where variables take values within the respective NuSMV domain.

Definition 6
An *extended transition system* is a tuple $\mathcal{T} = (S, S_0, \kappa, \ell)$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $\kappa \subseteq S \times S$ is a transition relation, and $\ell : S \to (\mathcal{X} \to \mathbb{V})$ maps each state to a variable assignment over $\mathcal{X}$. △

As expected, an extended transition system generates a set of traces $\mathit{Traces}(\mathcal{T}) \subseteq (\mathcal{X} \to \mathbb{V})^\omega$. An extended transition system $\mathcal{T}$ satisfies φ, written $\mathcal{T} \models \varphi$, if $\emptyset \models_{\mathit{Traces}(\mathcal{T})} \varphi$, where $\emptyset$ denotes the empty trace assignment.

Remark 1
We can view "standard" transition systems (see Definition 1) as a special case of extended transition systems by viewing each $a \in AP$ as a system variable that takes Boolean values. In particular, if we assume that $\mathbb{V} = \mathbb{B}$ (i.e., all variables are Boolean-valued), then each trace in $(2^{AP})^\omega$ corresponds exactly to a trace in $(AP \to \mathbb{V})^\omega$. △

6 Automata-based model-checking for HyperLTL$_\mathfrak{T}$

In this section, we modify the automata-based model-checking algorithm from Sect. 4 to natively support the first-order theory atoms. In the following, we assume that $\mathcal{T} = (S, S_0, \kappa, \ell)$ is a fixed extended transition system and that $\dot\varphi$ is a fixed HyperLTL$_\mathfrak{T}$ formula.

6.1 A naïve approach: unfolding

A naïve approach to model-check HyperLTL$_\mathfrak{T}$ (which we used in the initial version of `AutoHyper` presented in [8]) is to eliminate all theory atoms by unfolding it for all possible values. For example, assume that $\mathfrak{T}$ is the theory of linear integer arithmetic (LIA), x is an integer-valued system variable in the system, and we want to model-check the HyperLTL$_\mathfrak{T}$ formula

$$\forall \pi_1. \exists \pi_2. \Box (x_{\pi_1} = x_{\pi_2}).$$

To check the above formula, we could simply replace the relational first-order formula with a disjunction over all possible values of x, i.e.,

$$\forall \pi_1. \exists \pi_2. \Box \Bigg(\bigvee_{v \in U} \Big((x_{\pi_1} = v) \land (x_{\pi_2} = v) \Big) \Bigg),$$

where

$$U := \big\{ \ell(s)(x) \mid s \in S \big\} \subseteq \mathbb{Z}$$

is the finite set of possible values assigned to x in the *finite-state* extended transition system $\mathcal{T}$. This unfolding ensures that each first-order atom (in the above example, $x_{\pi_j} = v$ for $v \in U$ and $j \in \{1, 2\}$) refers to a unique trace variable. We can therefore replace each expression $x_{\pi_j} = v$ with a fresh atomic proposition and add this atomic proposition to all states in the system where $x = v$, thus reducing to the verification of a "standard" HyperLTL formula (over APs) on a "standard" (AP-based) transition system (Sect. 2).

Such a naïve unfolding is clearly inefficient. Firstly, it creates a HyperLTL formula with many atomic propositions, which complicates the LTL-to-NBA conversion (the first step in the model-checking process). Secondly, it unfolds the formula for every possible value, even for combinations that might not occur during the product construction. Instead, we will adopt the algorithm from Sect. 4 to unfold relational atoms lazily (on demand).

6.2 $\mathcal{T}$-Equivalence

As before, the basic idea of our model-checking approach is to summarize trace assignments within an automaton. However, in the semantics of HyperLTL$_{\mathfrak{T}}$, we use trace assignments $\Pi : V \rightarrow (\mathcal{X} \rightarrow \mathbb{V})^\omega$, which we cannot effectively zip using a finite alphabet. Instead, of tracking the precise value for all variables (as we have done in Definition 3), we will only track a suitable (finite) set of first-order formulas. In the following, we write $A \subseteq_{fin} B$ if $A \subseteq B$ and $|A| < \infty$ (i.e., A is finite).

Definition 7
Let $\Pi : V \rightarrow (\mathcal{X} \rightarrow \mathbb{V})^\omega$ be a trace assignment with domain $V \subseteq \mathcal{V}$, and let $\Theta \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_V}$ be a finite set of formulas over $\mathcal{X}_V$. We define the trace $zip_\Theta(\Pi) \in (2^\Theta)^\omega$ by

$$zip_\Theta(\Pi)(i) := \left\{\theta \in \Theta \mid \Pi_{(i)} \models_{\mathfrak{T}} \theta\right\}$$

for every $i \in \mathbb{N}$. △

Example 2
Consider the following trace assignment and set of formulas

$$\Pi := \begin{bmatrix} \pi_1 : \Big(\big[x \mapsto 2, y \mapsto \mathit{false}\big]\big[x \mapsto 1, y \mapsto \mathit{true}\big]\Big)^\omega \\ \pi_2 : \Big(\big[x \mapsto 1, y \mapsto \mathit{true}\big]\Big)^\omega \end{bmatrix},$$

$$\Theta := \left\{x_{\pi_1} > x_{\pi_2}, y_{\pi_1}\right\}.$$

Then $zip_\Theta(\Pi) = \Big(\left\{x_{\pi_1} > x_{\pi_2}\right\}\left\{y_{\pi_1}\right\}\Big)^\omega \in (2^\Theta)^\omega$. △

Remark 2
Similar to Remark 1, we can view Definition 7 as a generalization of the zipping for AP-based trace assignments (Definition 3). We define $\Theta \subseteq_{fin} \mathfrak{S}_{AP_V}$ by $\Theta := \{a_\pi \mid a \in AP, \pi \in V\}$, i.e., we track all Boolean-valued variables (aka. APs) on all traces in V (see Remark 1). Then each trace from $(2^\Theta)^\omega$ corresponds directly to a trace from $(2^{AP\times V})^\omega$. In this setting, the zipping of the trace assignment $\Pi : V \rightarrow (AP \rightarrow \mathbb{B})^\omega$ (via the zipping construction from Definition 7) results in a trace from $(2^\Theta)^\omega$ that corresponds to the zipped trace from $(2^{AP\times V})^\omega$ obtained via Definition 3 on the "equivalent" assignment $V \rightarrow (2^{AP})^\omega$. △

By tracking expressions within the automaton we can reestablish the notation of $\mathcal{T}$-equivalence.

Definition 8
Let φ be a HyperLTL$_{\mathfrak{T}}$ formula with free trace variables $V \subseteq \mathcal{V}$, and let $\Theta \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_V}$ be a finite set of formulas. An automaton $\mathcal{A}$ over alphabet 2^Θ is *$\mathcal{T}$-equivalent to* φ if for every trace assignments $\Pi : V \rightarrow (\mathcal{X} \rightarrow \mathbb{V})^\omega$, we have $\Pi \models_{Traces(\mathcal{T})} \varphi$ if and only $zip_\Theta(\Pi) \in \mathcal{L}(\mathcal{A})$. △

6.3 Automaton alphabet

With our generalized version of $\mathcal{T}$-equivalence fixed, we can now modify our model-checking algorithm to support HyperLTL$_{\mathfrak{T}}$. From a technical standpoint, the main challenge is to formalize how the alphabet of the automata changes during the product construction. In the purely Boolean (AP-based) setting in Sect. 4.3, we could immediately fix a subset of the atomic propositions: During the product construction for trace variable π, the alphabet of the automaton changes from $2^{AP\times(V \uplus \{\pi\})}$ to $2^{AP\times V}$; we fix all letters in $AP \times \{\pi\}$. In the current setting, atomic formulas are relational (i.e., refer to variables from multiple trace variables), so we cannot reduce the alphabet by fixing Boolean values for (some of) the variables. Say the current alphabet of the automaton (prior to the product construction) is 2^Θ with $\Theta \subseteq \mathfrak{S}_{\mathcal{X}_{V\uplus\{\pi\}}}$. Given the current state $s \in S$ of the system, we can fix values for all variables in $\mathcal{X}_{\{\pi\}}$, but in general this does not suffice to fully evaluate a formula $\theta \in \Theta$. Instead, our approach is based on the idea of *substituting* all variables in $\mathcal{X}_{\{\pi\}}$, resulting in a simpler formula from $\mathfrak{S}_{\mathcal{X}_V}$. For example, if the automaton prior to the product construction tracks a formula $x_{\pi_1} = x_{\pi_2}$ (from $\mathfrak{S}_{\mathcal{X}_{\{\pi_1,\pi_2\}}}$) and during the product for π_2, the variable x has value 4 in the current state, then we need to continue to track formula $x_{\pi_1} = 4$ (from $\mathfrak{S}_{\mathcal{X}_{\{\pi_1\}}}$). Consequently, the size of the automaton's alphabet might *increase*. For example, in the above example, we need to consider all possible values for x. Note that this substitution happens *during* the product construction and only on value combinations that are relevant (which is in sharp contrast to the naïve unfolding in Sect. 6.1).

To formalize the alphabet of the automaton, we need to reason about all possible substantiations that can occur within the system:

Definition 9
For any system state $s \in S$ and trace variable π, we define the variable assignment $\Delta_{\pi,s} : \mathcal{X}_\pi \rightarrow \mathbb{V}$ by $\Delta_{\pi,s} := \big[x_\pi \mapsto \ell(s)(x)\big]_{x\in\mathcal{X}}$. Given a set of formulas $\Theta \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_{V\uplus\{\pi\}}}$ for some $V \subseteq \mathcal{V}$, we define $\Theta_\pi \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_V}$ by $\Theta_\pi := \big\{\theta \circ \Delta_{\pi,s} \mid \theta \in \Theta \land s \in S\big\}$. △

In other words, $\Delta_{\pi,s}$ assigns each variable $x_\pi \in \mathcal{X}_\pi$ the value of x in state s. Using the family of all possible installations (i.e., $\{\Delta_{\pi,s}\}_{s\in S}$), we can then formally define how the alphabet of the product automaton changes: We take all formulas in Θ and consider all possible substitutions with evaluations from $\{\Delta_{\pi,s}\}_{s\in S}$.

6.4 Product construction

To check if $\mathcal{T} \models \varphi$, we, similar to Sect. 4.3, inductively construct an automaton $\mathcal{A}_\varphi$ that is $\mathcal{T}$-equivalent to φ for each subformula φ of $\dot{\varphi}$. Initially, the LTL body of $\dot{\varphi}$ uses finitely many first-order formulas (say $\Theta \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_V}$), so we can use a standard LTL-to-NBA translation to obtain an NBA over alphabet 2^Θ. We then iteratively eliminate quantifiers by computing the product with $\mathcal{T}$:

- **Case** $\varphi' = \exists \pi. \varphi$**:** We are given an inductively constructed automaton $\mathcal{A}_\varphi = (Q, Q_0, \delta, F)$ that is $\mathcal{T}$-equivalent to φ. Note that the alphabet of $\mathcal{A}_\varphi$ is 2^Θ for some set of formulas $\Theta \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_{V \uplus \{\pi\}}}$ for some $V \subseteq \mathcal{V}$. We ensure that $\mathcal{A}_\varphi$ is an NBA and define the NBA $\mathcal{A}_{\varphi'}$ over alphabet 2^{Θ_π} (cf. Definition 9) as

$$\mathcal{A}_{\varphi'} := (S \times Q, S_0 \times Q_0, \delta', S \times F),$$

where $\delta' \subseteq (S \times Q) \times 2^{\Theta_\pi} \times (S \times Q)$ is defined as

$$\delta' := \Big\{ \big((s, q), \sigma, (s', q')\big) \mid (s, s') \in \kappa \land \\ \Big(q, \big\{\theta \in \Theta \mid (\theta \circ \Delta_{\pi,s}) \in \sigma\big\}, q'\Big) \in \delta \Big\}.$$

Intuitively, $\mathcal{A}_{\varphi'}$ guesses a path in $\mathcal{T}$ and uses this path to fix the variables in $\mathcal{X}_\pi$ by substituting them within all formulas $\theta \in \Theta$. For example, consider state (s, q) of $\mathcal{A}_{\varphi'}$, and assume the system $\mathcal{T}$ has a transition $(s, s') \in \kappa$ and $\mathcal{A}_\varphi$ has an edge from q to q' that can be taken iff formula $\theta \in \Theta$ holds, i.e., $(q, \dot{\sigma}, q') \in \delta$ (where $\dot{\sigma} \subseteq \Theta$) iff $\theta \in \dot{\sigma}$. In this case, state (s, q) of $\mathcal{A}_{\varphi'}$ has an outgoing edge to (s', q') that can be taken iff $\theta \circ \Delta_{\pi,s}$ holds, i.e., $\big((s, q), \sigma, (s', q')\big) \in \delta'$ (where $\sigma \subseteq \Theta_\pi$) iff $\theta \circ \Delta_{\pi,s} \in \sigma$.
- **Case** $\varphi' = \forall \pi. \varphi$**:** We are given an automaton $\mathcal{A}_\varphi$ over alphabet 2^Θ that is $\mathcal{T}$-equivalent to φ. We ensure that this automaton is a UCA and define automaton $\mathcal{A}_{\varphi'}$ over alphabet 2^{Θ_π} as the UCA that is syntactically identical to the NBA constructed in the previous case.

Proposition 3
For every subformula φ of $\dot{\varphi}$, the automaton $\mathcal{A}_\varphi$ is $\mathcal{T}$-equivalent to φ.

The final automaton $\mathcal{A}_{\dot{\varphi}}$ has alphabet 2^Θ for some $\Theta \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_\emptyset}$ and $\mathcal{T} \models \dot{\varphi}$ holds iff $\emptyset \models_{Traces(\mathcal{T})} \dot{\varphi}$ iff $zip_\Theta(\emptyset) \in \mathcal{L}(\mathcal{A}_{\dot{\varphi}})$ (see Definition 8). As $\mathcal{X}_\emptyset = \emptyset$, all formulas used in the alphabet are variable-free. We thus get that $zip_\Theta(\emptyset) = \big(\{\theta \in \Theta \mid \emptyset \models_{\mathfrak{T}} \theta\}\big)^\omega$, so checking if $zip_\Theta(\emptyset) \in \mathcal{L}(\mathcal{A}_{\dot{\varphi}})$ reduces to a word containment check.

6.5 Leveraging language inclusion

As expected, we can generalize our language-inclusion-based optimization (Sect. 4.4) to the more general setting of HyperLTL$_{\mathfrak{T}}$.

For a transition system $\mathcal{T}$, a finite set of trace variable $V \subseteq \mathcal{V}$, and a finite set of formulas $\Theta \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_V}$, we define an NBA $\mathcal{A}_{\mathcal{T},V,\Theta}$ over alphabet 2^Θ that accepts the Θ-evaluations of all possible trace combinations in $\mathcal{T}$, i.e., $\mathcal{L}(\mathcal{A}_{\mathcal{T},V,\Theta}) = \Big\{zip_\Theta(\Pi) \mid \Pi : V \to Traces(\mathcal{T})\Big\}$. We can construct $\mathcal{A}_{\mathcal{T},V,\Theta}$ by building a $|V|$-fold self-composition of $\mathcal{T}$ similar to Definition 5:

Definition 10
Define the NBA $\mathcal{A}_{\mathcal{T},V,\Theta}$ over alphabet 2^Θ as $\mathcal{A}_{\mathcal{T},V,\Theta} := (Q, Q_0, \delta, F)$, where

- $Q := (V \to S)$;
- $Q_0 := \big\{q \in Q \mid \forall \pi \in V. q(\pi) \in S_0\big\}$;
- $\delta \subseteq Q \times 2^\Theta \times Q$ is defined by

$$\delta := \Big\{ (q, \sigma, q') \mid \forall \pi \in V. \big(q(\pi), q'(\pi)\big) \in \kappa \land \\ \sigma = \Big\{\theta \in \Theta \mid \big[x_\pi \mapsto \ell(q(\pi))(x)\big]_{x_\pi \in \mathcal{X}_V} \models_{\mathfrak{T}} \theta\Big\}\Big\},$$

i.e., the unique label of all transitions leaving q contains exactly those formulas $\theta \in \Theta$ that hold in the combined states in q;
- $F := Q$. △

Proposition 4
Let $\dot{\varphi} = \forall \pi_1. \ldots \forall \pi_n. \varphi$ be a HyperLTL formula (where φ may contain additional trace quantifiers), and let $\mathcal{A}_\varphi$ be an automaton over alphabet 2^Θ (for some $\Theta \subseteq_{fin} \mathfrak{S}_{\mathcal{X}_{\{\pi_1,\ldots,\pi_n\}}}$) that is $\mathcal{T}$-equivalent to φ. Then $\mathcal{T} \models \dot{\varphi}$ if and only if $\mathcal{L}(\mathcal{A}_{\mathcal{T},\{\pi_1,\ldots,\pi_n\},\Theta}) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$.

7 AutoHyper: tool overview

`AutoHyper` is written in `F#` and implements the model-checking approach for HyperLTL$_{\mathfrak{T}}$ described in Sect. 6. If desired by the user, `AutoHyper` leverages the language-inclusion-based optimization. `AutoHyper` uses `spot` [33] for LTL-to-NBA translations and automata complementations. We always ensure that the model-checked formula is of the form $\forall \pi_1 \ldots \forall \pi_n. \varphi$, so language inclusion checks are applicable (see Proposition 2); If the checked formula starts with existential quantification (so Proposition 2 does not apply), then we check the negated formula. The newest version of `AutoHyper` adds support for propositional quantification within the formula (formally, `AutoHyper` supports a more expressive logic called HyperQPTL [48]) by integrating the insights from [9]. To check inclusion between NBAs, `AutoHyper` uses `spot` default inclusion check (which is based on determinization), `RABIT` [25] (which is based on a Ramsey-based approach with heavy use of simulations), BAIT [31], FORKLIFT [32] (both based on well-quasiorders), and spot's internal implementation of the well-quasiorder-based inclusion checking algorithm (the same as used in FORKLIFT [32]) (which we refer to as FORQ). AutoHyper is designed such that communication with external automata tools is based on the HANOI [2] and BA automaton formats. New (or updated) tools that improve on fundamental automata operations, such as complementation and inclusion checks, can thus be integrated easily. Internally, AutoHyper operates on automata with a symbolic alphabet (similar to the HANOI automaton format). We use spot's -split-edges option to convert an automaton with symbolic alphabet to an explicit-alphabet representation as needed for BA-format-based tools (RABIT, BAIT, and FORKLIFT).

All experiments in this paper were conducted on a MacBook Pro with an M1 Pro CPU and 32 GB of memory. We used spot version 2.12; RABIT version 2.4.5; BAIT commit 369e1a4; and FORKLIFT commit 5d519e3.

Input formats AutoHyper supports both explicit-state extended transition systems with Boolean and integer-based variables (given in a HANOI-like [2] input format) and symbolic systems that are internally converted to an explicit-state representation. The support for symbolic systems includes symbolic models written in a fragment of the NuSMV input language [22], which we internally convert to an extended transition system over the variables in the NuSMV model.

Witness computation AutoHyper supports the computation of witness traces. That is, if a HyperLTL formula of the form $\exists \pi_1 \ldots \exists \pi_n . \varphi$ holds on a given system (where φ is any HyperLTL formula, possibly containing additional quantifiers), then AutoHyper can produce n lasso-shaped paths in the system that serve as witnesses for $\pi_1, \ldots, \pi_n$. Likewise, if a HyperLTL formula of the form $\forall \pi_1 \ldots \forall \pi_n . \varphi$ does *not* hold on a given system, then AutoHyper can compute concrete counterexample paths for $\pi_1, \ldots, \pi_n$.

Optimizations AutoHyper further implements various optimizations compared to the presentation in Sect. 6. During the product construction, we build the alphabet of the new automaton lazily, i.e., we only start tracking a formula if it is actually used in the product construction. Additionally, we simplify the partially evaluated formulas as much as possible and eliminate them altogether once all variables are instantiated. This helps reduce the number of atomic propositions tracked in the automaton. Moreover, AutoHyper supports various preprocessing steps. Most notably, unless otherwise specified, AutoHyper computes a bisimulation quotient of each system (with respect to those variables that are actually needed to evaluate the HyperLTL$_{\mathfrak{T}}$ formula). In our experience, this reduces the size of the system in many instances and leads to faster model-checking.

8 Evaluation

In this section, we challenge AutoHyper with complex model-checking problems found in the literature.

8.1 Explicit model checking of symbolic systems

We evaluate AutoHyper on challenging symbolic NuSMV models [22]. The set of benchmarks we use was created by Hsu et al. [42] to evaluate the BMC tool HyperQB. For further details on the benchmarks, we refer to Hsu et al. [42].

We model-check each instance using both HyperQB [41] and AutoHyper and depict the results in Table 2. For AutoHyper, we report the model-checking time using different inclusion-checking tools (RABIT, BAIT, FORKLIFT, spot's FORQ implementation, and spot's default inclusion-checking algorithm). Alternation-free formulas can be checked without any inclusion check, so the runtime of AutoHyper does not depend on the provided inclusion checker. For comparison, we additionally use the old version of AutoHyper [8], which unfolds all formulas as discussed in Sect. 6.1 and uses spot for inclusion checking ($t^{unfold}_{\texttt{spot}}$). For HyperQB, we use the unrolling semantics and unrolling depth listed in [42, Table 2] and the HyperQB repository.

Analysis We can draw a few conclusions from the verification results. Firstly, the use of HyperLTL$_{\mathfrak{T}}$-specific algorithms clearly improves verification. Compared to an unfolding over all possible values, i.e., column $t^{unfold}_{\texttt{spot}}$,– treating the first-order formulas from first principles allows for smaller alphabets and more efficient solving. Secondly, the flexibility of using multiple automata inclusion checkers is evident. We observe that spot's default algorithm performs generally best, but in some instances the FORQ-based algorithm is more efficient. Our results also point to some surprising findings: In some cases, FORKLIFT is faster than spot's implementation of the same FORQ-based algorithm, despite the fact that the latter supports symbolic alphabets. Lastly, and perhaps most importantly, we can compare the performance of AutoHyper with HyperQB. Here we see that the additional optimizations of AutoHyper (e.g., the support for HyperLTL$_{\mathfrak{T}}$ and prepossessing) allow us to verify all instances faster than the BMC approach of HyperQB, even on larger instances such as the SNARK example.

8.2 Hyperproperties for path planning

As a second set of benchmarks, we use planning problems for robots encoded into HyperLTL as proposed by Wang et al. [49]. For example, the synthesis of a shortest path can

Table 2 We evaluate `HyperQB` and `AutoHyper` on the benchmarks from [42]. We list the quantifier structure (Q^*), the sizes of the system(s), and the verification result (✓ if the property holds and ✗ if it is violated). For `HyperQB`, we provide the unrolling bound k and the verification time t. For `AutoHyper`, we report the verification time when using `RABIT`, `BAIT`, `FORKLIFT`, `spot`'s `FORQ` implementation, and `spot`'s default algorithm ($t_{\texttt{BAIT}}$, $t_{\texttt{RABIT}}$, $t_{\texttt{FORKLIFT}}$, $t^{\texttt{FORQ}}_{\texttt{spot}}$, and $t_{\texttt{spot}}$, respectively). Times are given in seconds. The timeout (TO) is set to 2 minutes.

				`HyperQB` [42]		`AutoHyper`					
Instance	**Q***	**Size**	**Result**	**k**	**t**	$t^{unfold}_{\texttt{spot}}$	$t_{\texttt{RABIT}}$	$t_{\texttt{BAIT}}$	$t_{\texttt{FORKLIFT}}$	$t^{\texttt{FORQ}}_{\texttt{spot}}$	$t_{\texttt{spot}}$
Bakery3 + φ_{S1}	∃∃	167	✗	10	1.1	0.5			**0.4**		
Bakery3 + φ_{S2}	∀∃	167	✗	10	1.1	0.5	0.9	0.6	0.6	**0.4**	**0.4**
Bakery3 + φ_{S3}	∃∀	167	✗	10	1.2	**0.6**	0.8	0.8	0.8	36.4	**0.6**
Bakery3 + φ_{sym1}	∀∃	167	✗	10	1.1	3.8	0.5	0.5	0.5	0.5	**0.4**
Bakery3 + φ_{sym2}	∀∃	167	✗	10	1.1	1.2	0.5	0.6	0.5	**0.4**	**0.4**
Bakery5 + φ_{sym1}	∀∃	996	✗	10	6.4	118.5	TO	TO	TO	**1.5**	**1.5**
Bakery5 + φ_{sym2}	∀∃	996	✗	10	6.4	4.6	4.4	2.1	1.3	**0.8**	**0.8**
SNARK	∀∃	4914/548	✗	18	75.1	16.5	TO	47.8	41.1	TO	**15.9**
3-Thread$_{correct}$	∀∃	64	✓	50	1.3	**0.3**	0.9	0.9	0.8	**0.3**	**0.3**
3-Thread$_{incorrect}$	∀∃	368	✗	50	5.5	2.2	TO	TO	TO	**1.1**	1.6
NRP$_{correct}$	∃∀	55	✓	15	1.0	**0.3**	0.5	0.5	0.4	0.5	**0.3**
NRP$_{incorrect}$	∃∀	54	✓	15	0.8	0.6	3.1	1.7	1.6	**0.5**	0.6
Mutant	∃∀	32	✓	10	0.8	0.4	0.6	0.6	0.5	**0.3**	**0.3**
CoTerm	∀∀	53/28	✓	102	1.4	**0.3**			**0.3**		
DeniabilitySmall	∀∃∃	240	✓	10	5.8	45.0	1.6	1.3	1.4	**1.2**	2.5
BufferOD	∀∀	876	✗	10	2.6	0.6			**0.5**		
BufferSchedOD	∀∀	228	✓	10	1.2	0.8			**0.6**		
BufferSchedNI	∀∃	228	✓	10	1.2	2.9	2.9	3.5	2.7	**0.4**	2.8
NIExp + TINI	∀∃	876	✓	10	2.2	TO	6.3	6.3	6.8	**1.8**	TO
NIExp + TSNI	∀∃	876	✓	10	2.2	TO	6.2	6.3	6.6	**1.7**	TO
KSafety	∀∀	150	✓	64	2.7	TO			**0.6**		
MapSythEx	∃∀∀∃∃	16/10/7	✓	5	0.6	**0.4**	0.5	0.5	0.5	**0.4**	**0.4**
TeamLTL1	∃∃∀	65	✗	10	1.6	**0.4**	0.5	**0.4**	0.5	TO	**0.4**
TeamLTL2	∃∃∀	257	✗	10	10.4	0.7	0.8	0.8	0.9	TO	**0.6**
NonDet1	∀∃	6	✗	5	0.6	0.4	0.4	0.4	0.5	**0.3**	**0.3**
NonDet2	∀∃	33	✗	5	0.6	TO	TO	53.8	52.4	**0.3**	TO
CSRF	∀∀	3434	✗	10	14.7	TO			**0.8**		
Bank	∀∀	648	✗	15	5.7	TO			**0.5**		
ATM	∀∀	1314	✗	15	6.1	TO			**2.0**		

be phrased as a ∃∀ property that states that there exists a path to the goal such that all alternative paths to the goal take at least as long. Wang et al. [49] check the resulting HyperLTL property by encoding it in first-order logic. Although not competitive with state-of-the-art planning tools, HyperLTL allows us to express a broad range of problems (shortest path, path robustness, etc.) in a very general way. Hsu et al. [42] observe that the QBF encoding implemented in `HyperQB` outperforms the SMT-based approach by Wang et al. [49]. We depict the results in Table 3.[2] It is evident that

[2] In these instances, the systems are already AP-based, so our lazily unfolding does not result in any speed-up or slow-down. Using an explicit unfolding (as in column $t^{unfold}_{\texttt{spot}}$ in Table 2) would result in the same running times listed in column $t_{\texttt{spot}}$.

Table 3 We evaluate HyperQB and AutoHyper on hyperproperties that encode the existence of a shortest path φ_{sp} and robust path φ_{rp}. We give the specification (Spec), the quantifier structure (Q^*), the size of the grid (Grid), and the size of the explicit state space (Size). For HyperQB, we give the unrolling bound k, the file size of the QBF generated (|QBF|), and the verification time t. For AutoHyper, we report the verification time when using RABIT, BAIT, FORKLIFT, spot's FORQ implementation, and spot's default algorithm ($t_{\texttt{BAIT}}$, $t_{\texttt{RABIT}}$, $t_{\texttt{FORKLIFT}}$, $t^{\texttt{FORQ}}_{\texttt{spot}}$, and $t_{\texttt{spot}}$, respectively). Times are given in seconds. The timeout (TO) is set to 5 minutes.

				HyperQB [42]			AutoHyper				
Spec	**Q***	**Grid**	**Size**	**k**	**|QBF|**	**t**	$t_{\texttt{RABIT}}$	$t_{\texttt{BAIT}}$	$t_{\texttt{FORKLIFT}}$	$t^{\texttt{FORQ}}_{\texttt{spot}}$	$t_{\texttt{spot}}$
φ_{sp}	∃∀	10 × 10	146	20	2.1 MB	4.2	0.7	0.6	0.6	TO	**0.5**
		20 × 20	188	40	7.3 MB	132.8	1.1	1.1	1.0	TO	**0.9**
		40 × 40	408	80	-	TO	12.1	12.6	12.1	TO	**12.0**
		60 × 60	404	120	-	TO	39.4	40.8	39.2	TO	**38.7**
φ_{rp}	∃∀	10 × 10	266	20	3.6 MB	4.1	1.1	TO	**0.8**	53.1	1.7
		20 × 20	572	40	20.0 MB	10.5	6.9	43.2	**1.5**	8.0	14.5
		40 × 40	1212	80	117.5 MB	179.2	39.7	TO	**2.9**	TO	152.8
		60 × 60	1852	120	-	TO	TO	TO	**52.6**	TO	TO

AutoHyper outperforms HyperQB, sometimes by orders of magnitude. This is surprising as planning problems (which are essentially reachability problems) on symbolic systems should be advantageous for symbolic methods such as BMC. The large size of the intermediate QBF indicates that a more optimized encoding (perhaps specific to path planning) could improve the performance of BMC on such examples.

8.3 Bounded vs. explicit-state model checking

Bounded model checking has seen remarkable success in the verification of trace properties and frequently scales to systems whose size is well out of scope for explicit-state methods [28]. Similarly, in the context of *alternation-free* hyperproperties, symbolic verification tools such as MCHyper [37] (which internally reduces to the verification of a circuit using ABC [21]) can verify systems that are well beyond the reach of explicit-state methods. In contrast, in the context of model checking for hyperproperties that involve *quantifier alternations*, our findings make a strong case for the use of explicit-state methods (as implemented in AutoHyper):

First, compared to symbolic methods (such as BMC), explicit-state (automata-based) model checking is currently the only method that is *complete*. Although BMC was able to verify or refute all properties in Tables 2 and 3, many instances cannot be solved with the current BMC encoding. As a concrete example, BMC can *never* verify formulas whose body contains simple invariants (such as (GNI)). Thus the most significant advantage of explicit-state MC (as implemented in AutoHyper) is that it is both push-button and complete, i.e., it can, at least in theory, verify or refute all properties.

Second, the performance of AutoHyper seems to be *on-par* with that of BMC and frequently outperforms it (even by several orders of magnitude; see Table 3). We stress that this is despite the fact that for the evaluation of HyperQB, we already fix an unrolling depth and unrolling semantics, thus creating favorable conditions for HyperQB.[3] While BMC for trace properties reduces to SAT solving, BMC of hyperproperties reduces to QBF solving, a problem that is much harder and has seen less support by industry-strength tools. It is therefore unclear whether the advance of modern QBF solvers can improve the performance of hyperproperty BMC, to the same degree that the advance of SAT solvers has stimulated the success of BMC for trace properties. Our findings seem to indicate that, at the moment, QBF solving (often) seems inferior to an explicit (automata-based) solving strategy.

9 Analyzing strategy-based verification

In the previous section, we used AutoHyper to check hyperproperties on instances arising in the literature. In this last section, we demonstrate that AutoHyper also serves as a valuable baseline to evaluate different (possibly incomplete) verification methods. Here we focus on strategy-based verification (SBV), i.e., the idea of automatically synthesizing a strategy that resolves existential quantification in $\forall^*\exists^*$ HyperLTL properties [6, 11, 17, 26].

[3] In Tables 2 and 3, we perform a single QBF query with a fixed unrolling depth k and semantics, i.e., we already know if we want to show satisfaction or violation and the depth needed to show this (as done in [42]). In a classical BMC loop, we would check for satisfaction and violation with an incrementally increasing unrolling depth and thus perform roughly $2k$ many QBF queries, where k is the least bound for which satisfaction or violation can be established (provided that such a bound exists).

9.1 Effectiveness of strategy-based verification

SBV is known to be incomplete [6, 26]. However, due to the previous lack of *complete* tools for verifying $\forall^*\exists^*$ properties, a detailed study into how effective SBV is in practice was impossible on a larger scale (i.e., beyond hand-crafted examples). With AutoHyper, we can, for the first time, rigorously evaluate SBV. We use the SBV implementation from [6], which synthesizes a strategy for the $\exists$-player by translating the formula to a deterministic parity automaton (DPA) [47] and phrases the synthesizes as a parity game.

Random instances To evaluate SBV, we use randomly generated model-checking problems. The advantage of randomly generated instances is twofold. Firstly, it allows for the easy generation of a large set of benchmarks. Secondly, the random generation is parameterized by multiple parameters (such as system size, formula size, etc.), enabling a comprehensive analysis of the exact impact of different parameters on the model checking complexity. We generate random transition systems based on the Erdős–Rényi–Gilbert model [35]. Given a size n and a density parameter $p \in [0, 1]$, we generate a graph with n states, where for every two states s, s', there is a transition $s \longrightarrow s'$ with probability p. We generate random HyperLTL formulas (with a given quantifier prefix) by sampling the LTL matrix using spot's randltl.

Effectiveness We have generated random transition systems and properties of varying sizes and computed a ground truth using AutoHyper. We then performed SBV (recall that SBV can never show that a property does not hold and might fail to establish that it does). We find that for our generated instances, the property holds in 61.1% of the cases, and SBV can verify the property in 60.4% of the cases. Successful verification with SBV is thus possible in many cases, even without the addition of expensive mechanisms such as prophecies [6]. On the other hand, our results show that random generation produces instances (albeit not many) on which SBV fails (so far, examples where SBV fails required careful construction by hand).

9.2 Efficiency of strategy-based verification

After having analyzed the effectiveness of SBV (i.e., how many instances *can* be verified), we turn our attention to the efficiency of SBV. In theory, model checking of $\forall^*\exists^*$ HyperLTL – as implemented in AutoHyper – is EXPSPACE-complete in the specification and PSPACE-complete in the size of the system [24, 48]. Conversely, SBV is 2-EXPTIME-complete in the size of the specification but only PTIME-complete in the size of the system [26]. Consequently, we would expect that AutoHyper fares better on larger specifications and SBV fares better on larger systems (the more important measure in practice).

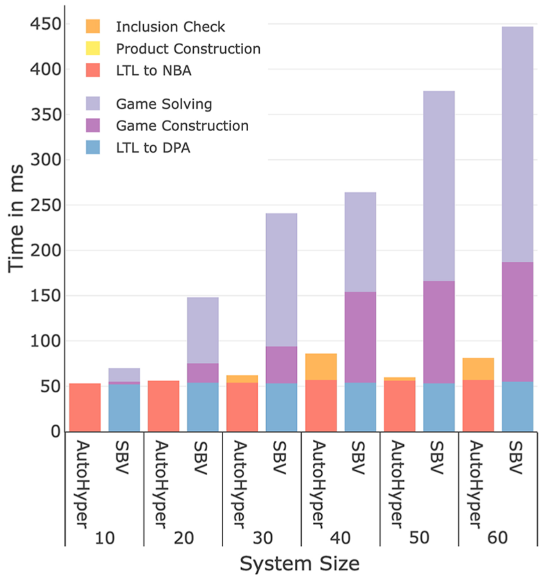

Fig. 1 We compare AutoHyper and strategy-based verification (SBV) [6] on instances of varying system size. We fix the property size to 20. We generate 100 random model-checking instances for each size and take the average over the fastest L instances, where L is the minimum number of instances solved within a 5 s timeout by both methods.

However, our results show that this does not translate into practice (at least using the current implementation of SBV [6]). We compare the running time of AutoHyper (using spot's inclusion checker) and SBV. We break the running time into the three main steps for each method. For AutoHyper, this is the LTL-to-NBA translation, the construction of the product automaton, and the inclusion check. For SBV, it is the LTL-to-DPA translation, the construction of the game, and the game-solving.

We depict the average cost for varying system sizes in Fig. 1. We observe that SBV performs worse than AutoHyper and, more importantly, scales poorly in the size of the system. This is contrary to the theoretical analysis of automata-based model-checking and SBV. As the detailed breakdown of the running time suggests, the poor performance is due to the costly construction of the game and the time taken to solve the game. An almost identical picture emerges if we compare AutoHyper and SBV relative to the property size (see Fig. 2). Although in this case the results match the theory (i.e., SBV scales worse in the size of the specification), we find that the bottleneck for SBV is again the construction and solving of the parity game. We remark that the SBV engine we used [6] is not optimized and always constructs the full (reachable) game graph. The poor performance of SBV can be attributed to the fact that the size of the game does, in the worst case, scale quadratically in the size of the system (when considering $\forall^1\exists^1$ properties). This is amplified in dense systems (i.e., systems with many transitions), as, with increasing transition density, the size of the parity

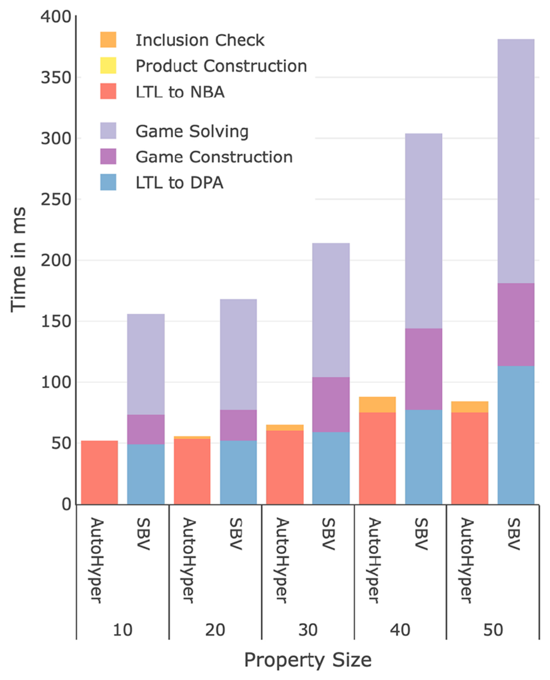

Fig. 2 We compare `AutoHyper` and strategy-based verification (SBV) on instances with varying property size. For each size, we generate 100 random $\forall^1\exists^1$ model-checking instances. The timeout is set to 5 s. We take the average over the fastest L instances, where L is the minimum of the instances solved within the timeout by both methods.

games approaches its worst-case size. In contrast, the heavily optimized inclusion checker (in this case, `spot`) seems to be able to check inclusion very efficiently (despite being exponential in theory). This efficiency of mature language inclusion checkers is what enables `AutoHyper` to achieve remarkable performance that often exceeds that of symbolic methods such as BMC (see Sect. 8) and further strengthens the practical impact of Proposition 2.

10 Conclusion

In this paper, we have presented `AutoHyper`, the first complete model checker for HyperLTL formulas with arbitrary quantifier prefixes. We extended an automata-based verification algorithm for HyperLTL [37] to support first-order formulas modulo theories. Our extension tracks partially evaluated formulas and expands the alphabet on demand. `AutoHyper` integrates language inclusion checks to further improve mode-checking performance for the outermost quantifier alternation. We have demonstrated that `AutoHyper` can check many interesting properties involving quantifier alternations and often outperforms symbolic methods such as BMC.

Acknowledgements This work was supported by the European Research Council (ERC) Grant HYPER (101055412), and by the German Research Foundation (DFG) as part of TRR 248 (389792660).

Funding information Open Access funding enabled and organized by Projekt DEAL.

References

1. Artale, A., Mazzullo, A., Ozaki, A.: Do you need infinite time? In: International Joint Conference on Artificial Intelligence, IJCAI 2019 (2019). https://doi.org/10.24963/IJCAI.2019/210
2. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Kretínský, J., Müller, D., Parker, D., Strejcek, J.: The Hanoi omega-automata format. In: International Conference on Computer Aided Verification, CAV 2015 (2015). https://doi.org/10.1007/978-3-319-21690-4_31
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, 2nd edn. (2021). https://doi.org/10.3233/FAIA201017
5. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6) (2011). https://doi.org/10.1017/S0960129511000193
6. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: Computer Security Foundations Symposium, CSF 2022 (2022). https://doi.org/10.1109/CSF54842.2022.00030
7. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13185-1_17
8. Beutner, R., Finkbeiner, B.: AutoHyper: explicit-state model checking for HyperLTL. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023 (2023). https://doi.org/10.1007/978-3-031-30823-9_8
9. Beutner, R., Finkbeiner, B.: Model checking omega-regular hyperproperties with AutoHyperQ. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2023 (2023). https://doi.org/10.29007/1XJT
10. Beutner, R., Finkbeiner, B.: Hyper strategy logic. In: International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2024 (2024). https://doi.org/10.5555/3635637.3662866
11. Beutner, R., Finkbeiner, B.: Non-deterministic planning for hyperproperty verification. In: International Conference on Automated Planning and Scheduling, ICAPS 2024 (2024). https://doi.org/10.1609/ICAPS.V34I1.31457
12. Beutner, R., Finkbeiner, B.: On alternating-time temporal logic, hyperproperties, and strategy sharing. In: Conference on Artificial Intelligence, AAAI 2024 (2024). https://doi.org/10.1609/AAAI.V38I16.29679
13. Beutner, R., Finkbeiner, B.: Predicate abstraction for hyperliveness verification. Form. Methods Syst. Des. (2025)

14. Beutner, R., Carral, D., Finkbeiner, B., Hofmann, J., Krötzsch, M.: Deciding hyperproperties combined with functional specifications. In: Symposium on Logic in Computer Science, LICS 2022 (2022). https://doi.org/10.1145/3531130.3533369
15. Beutner, R., Finkbeiner, B., Frenkel, H., Metzger, N.: Second-order hyperproperties. In: International Conference on Computer Aided Verification, CAV 2023 (2023). https://doi.org/10.1007/978-3-031-37703-7_15
16. Beutner, R., Finkbeiner, B., Frenkel, H., Siber, J.: Checking and sketching causes on temporal sequences. In: International Symposium on Automated Technology for Verification and Analysis, ATVA 2023 (2023). https://doi.org/10.1007/978-3-031-45332-8_18
17. Beutner, R., Finkbeiner, B., Göbl, A.: Visualizing game-based certificates for hyperproperty verification. In: International Symposium on Formal Methods, FM 2024 (2024). https://doi.org/10.1007/978-3-031-71177-0_5
18. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS 1999 (1999). https://doi.org/10.1007/3-540-49059-0_14
19. Biewer, S., Dimitrova, R., Fries, M., Gazda, M., Heinze, T., Hermanns, H., Mousavi, M.R.: Conformance relations and hyperproperties for doping detection in time and space. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/LMCS-18(1:14)2022
20. Bozzelli, L., Maubert, B., Pinchinat, S.: Unifying hyper and epistemic temporal logics. In: International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2015 (2015). https://doi.org/10.1007/978-3-662-46678-0_11
21. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: International Conference on Computer Aided Verification, CAV 2010 (2010). https://doi.org/10.1007/978-3-642-14295-6_5
22. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: International Conference on Computer Aided Verification, CAV 2002, Copenhagen (2002). https://doi.org/10.1007/3-540-45657-0_29
23. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: IEEE Computer Security Foundations Symposium, CSF 2008 (2008). https://doi.org/10.1109/CSF.2008.7
24. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference on Principles of Security and Trust, POST 2014 (2014). https://doi.org/10.1007/978-3-642-54792-8_15
25. Clemente, L., Mayr, R.: Efficient reduction of nondeterministic automata with application to language inclusion testing. Log. Methods Comput. Sci. **15**(1) (2019). https://doi.org/10.23638/LMCS-15(1:12)2019
26. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: International Conference on Computer Aided Verification, CAV 2019 (2019). https://doi.org/10.1007/978-3-030-25540-4_7
27. Coenen, N., Finkbeiner, B., Frenkel, H., Hahn, C., Metzger, N., Siber, J.: Temporal causality in reactive systems. In: International Symposium on Automated Technology for Verification and Analysis, ATVA 2022 (2022). https://doi.org/10.1007/978-3-031-19992-9_13
28. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of bounded model checking at an industrial setting. In: International Conference on Computer Aided Verification, CAV 2001 (2001). https://doi.org/10.1007/3-540-44585-4_43
29. D'Argenio, P.R., Barthe, G., Biewer, S., Finkbeiner, B., Hermanns, H.: Is your software on dope? Formal analysis of surreptitiously "enhanced" programs. In: European Symposium on Programming, ESOP 2017 (2017). https://doi.org/10.1007/978-3-662-54434-1_4
30. Demri, S., D'Souza, D.: An automata-theoretic approach to constraint LTL. Inf. Comput. **205**(3) (2007). https://doi.org/10.1016/J.IC.2006.09.006
31. Doveri, K., Ganty, P., Parolini, F., Ranzato, F.: Inclusion testing of Büchi automata based on well-quasiorders. In: International Conference on Concurrency Theory, CONCUR 2021 (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.3
32. Doveri, K., Ganty, P., Mazzocchi, N.: FORQ-based language inclusion formal testing. In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13188-2_6
33. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From spot 2.0 to spot 2.10: what's new? In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13188-2_9
34. Fellner, A., Befrouei, M.T., Weissenbacher, G.: Mutation testing with hyperproperties. Softw. Syst. Model. **20**(2) (2021). https://doi.org/10.1007/s10270-020-00850-1
35. Fienberg, S.E.: A brief history of statistical models for network analysis and open challenges. J. Comput. Graph. Stat. **21**(4) (2012)
36. Finkbeiner, B., Siber, J.: Counterfactuals modulo temporal logics. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2023 (2023). https://doi.org/10.29007/QTW7
37. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: International Conference on Computer Aided Verification, CAV 2015 (2015). https://doi.org/10.1007/978-3-319-21690-4_3
38. Finkbeiner, B., Heim, P., Passing, N.: Temporal stream logic modulo theories. In: International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2022 (2022). https://doi.org/10.1007/978-3-030-99253-8_17
39. Geatti, L., Gianola, A., Gigante, N.: Linear temporal logic modulo theories over finite traces. In: International Joint Conference on Artificial Intelligence, IJCAI 2022 (2022). https://doi.org/10.24963/IJCAI.2022/366
40. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990). https://doi.org/10.1145/78969.78972
41. Hsu, T.H.: HyperQB: a QBF-based bounded model checker for hyperproperties (2024). https://doi.org/10.5281/zenodo.11282197
42. Hsu, T., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021 (2021). https://doi.org/10.1007/978-3-030-72016-2_6
43. Hsu, T., Sánchez, C., Sheinvald, S., Bonakdarpour, B.: Efficient loop conditions for bounded model checking hyperproperties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023 (2023). https://doi.org/10.1007/978-3-031-30823-9_4
44. Kontchakov, R., Lutz, C., Wolter, F., Zakharyaschev, M.: Temporalising tableaux. Stud. Log. **76**(1) (2004). https://doi.org/10.1023/B:STUD.0000027468.28935.6D
45. Li, Y., Turrini, A., Sun, X., Zhang, L.: Proving non-inclusion of Büchi automata based on Monte Carlo sampling. In: International Symposium on Automated Technology for Verification and Analysis, ATVA 2020 (2020). https://doi.org/10.1007/978-3-030-59152-6_26
46. McCullough, D.: Noninterference and the composability of security properties. In: Symposium on Security and Privacy, SP 1988 (1988). https://doi.org/10.1109/SECPRI.1988.8110
47. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3) (2007). https://doi.org/10.2168/LMCS-3(3:5)2007

48. Rabe, M.N.: A temporal logic approach to information-flow control. Ph.D. thesis, Saarland University (2016)
49. Wang, Y., Nalluri, S., Pajic, M.: Hyperproperties for robotics: planning via HyperLTL. In: International Conference on Robotics and Automation, ICRA 2020 (2020). https://doi.org/10.1109/ICRA40945.2020.9196874
50. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Computer Security Foundations Workshop, CSFW 2003 (2003). https://doi.org/10.1109/CSFW.2003.1212703