AutoHyper: Explicit-State Model Checking for HyperLTL

Raven Beutner(✉) and Bernd Finkbeiner

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{raven.beutner,finkbeiner}@cispa.de

Abstract. HyperLTL is a temporal logic that can express hyperproperties, i.e., properties that relate multiple execution traces of a system. Such properties are becoming increasingly important and naturally occur, e.g., in information-flow control, robustness, mutation testing, path planning, and causality checking. 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. Properties involving quantifier alternations could only be handled in an incomplete way, i.e., the verification might fail even though the property holds. In this paper, we present `AutoHyper`, an explicit-state automata-based model checker that supports full HyperLTL and is complete for properties with arbitrary quantifier alternations. We show that language inclusion checks can be integrated into HyperLTL verification, which allows `AutoHyper` to benefit from a range of existing inclusion-checking tools. 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.

1 Introduction

Hyperproperties [16] are system properties that relate multiple executions of a system. Such properties are of increasing importance as they naturally occur, e.g., in information-flow control [36], robustness [22], linearizability [30,31], path planning [39], mutation testing [27], and causality checking [18]. A prominent logic to express hyperproperties is HyperLTL, which extends linear-time temporal logic (LTL) with explicit trace quantification [15]. HyperLTL can, for instance, express generalized non-interference (GNI) [34], stating that the high-security input of a system does not influence the observable output.

$$\forall \pi. \forall \pi'. \exists \pi''. \Box \Big(\bigwedge_{a \in H} a_\pi \leftrightarrow a_{\pi''} \Big) \land \Box \Big(\bigwedge_{a \in L \cup O} a_{\pi'} \leftrightarrow a_{\pi''} \Big) \qquad \text{(GNI)}$$

Here, H is a set of high-security input, L is a set of low-security inputs, and O is a set of low-security outputs. The formula states that for any traces π, π' there exists a third trace π'' that agrees with the high-security inputs of π and with the low-security inputs and outputs of π'. Any observation made by a low-security attacker is thus compatible with every possible high-security input.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 145–163, 2023.
https://doi.org/10.1007/978-3-031-30823-9_8

We are interested in the model checking (MC) problem of HyperLTL, i.e., whether a given (finite-state) system satisfies a given property. For HyperLTL, the structure of the quantifier prefix directly impacts the complexity of this problem. For alternation-free formulas (i.e., formulas that only use quantifiers of a single type), verification is well understood and is reducible to the verification of a trace property on a self-composition of the system [3]. This reduction has, for example, been implemented in `MCHyper` [29], 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 from above). While MC algorithms supporting full HyperLTL exist (see [15,29]), they have not been implemented yet. Instead, over the years, a number of approaches to the verification of such properties in practice have been made: Finkbeiner et al. [29] and D'Argenio et al. [22] manually strengthen properties with quantifier alternation into properties that are alternation-free and can be checked by `MCHyper`. Coenen et al. [19] 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 [7]. Hsu et al. [31] present a bounded model checking (BMC) approach for HyperLTL that is implemented in `HyperQube`. See Section 4 for more details.

While all these verification tools can verify (or refute) interesting properties, they all suffer from the same fundamental limitation: they are *incomplete*. That is, for all the tools above, we can come up with verification instances where they fail, not because of resource constraints but because of inherent limitations in the underlying verification algorithm. Moreover, such instances are not rare events but are encountered regularly in practice. For example, many of the benchmarks used to evaluate `HyperQube` (by Hsu et al. [31]) do not admit a strategy to resolve existential quantification. Conversely, many of the properties verified by Coenen et al. [19] (such as GNI) cannot be verified using BMC [31].

AutoHyper. In this paper, we present `AutoHyper`, a model checker for HyperLTL. Our tool checks a hyperproperty by iteratively eliminating trace quantification using automata-complementations, thereby reducing verification to the emptiness check of an automaton [29]. Importantly – and different from previous tools for HyperLTL verification such as `MCHyper` [29,19] and `HyperQube` [31] – `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` [19]), nor does a user need to worry if the given property can even be verified with a given method. `AutoHyper` thus provides a "push-button" model checking experience for HyperLTL.[1]

[1] The name of `AutoHyper` is derived from the fact that it is both **Auto**mata-based and **Auto**matic (i.e., it is complete and does not require any user intervention).

To improve AutoHyper's efficiency, we make the (theoretical) observation that we can often avoid explicit automaton complementation and instead reduce to a language inclusion check on Büchi automata (cf. Proposition 1). On the practical side, this enables AutoHyper to resort to a range of mature language inclusion checkers, including spot [26], RABIT [17], BAIT [25], and FORKLIFT [24].

Evaluation. Using AutoHyper, we extensively study the practical aspects of model checking HyperLTL properties with quantifier alternations. To evaluate the performance of explicit-state model checking, we apply AutoHyper to a broad range of benchmarks taken from the literature and compare it with existing (incomplete) tools. We make the surprising observation that – at least on the currently available benchmarks – explicit-state MC as implemented in AutoHyper performs on-par (and frequently outperforms) symbolic methods such as BMC [31]. 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 (as done by Coenen et al. [19]) is incomplete, it was, thus far, unknown if this incompleteness occurs frequently or is merely a rare phenomenon. 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 Section 2, we introduce HyperLTL. We recap automata-based verification (which we abbreviate ABV) and our new approach utilizing language inclusion checks in Section 3. We discuss alternative verification approaches for HyperLTL in Section 4. In Section 6, we compare different backend solving techniques and study the complexity of HyperLTL MC with multiple quantifier alternations in practice; In Section 7, we evaluate ABV on a set of benchmarks from the literature and compare with the bounded model checker HyperQube [31]; In Section 8 we use AutoHyper for a detailed analysis of (and comparison with) strategy-based verification [19,7].

2 Preliminaries

We fix a set of atomic propositions AP and define $\Sigma := 2^{AP}$. HyperLTL [15] extends LTL with explicit quantification over traces, thereby lifting it from a logic expressing trace properties to one expressing hyperproperties [16]. Let $\mathcal{V}$ 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 $\pi \in \mathcal{V}$ and $a \in AP$.

We assume that the formula is closed, i.e., all trace variables that are used in the body are bound by some quantifier. The semantics of HyperLTL is given with respect to a trace assignment $\Pi : \mathcal{V} \rightharpoonup \Sigma^\omega$ mapping trace variables to traces. For $\pi \in \mathcal{V}$ and $t \in \Sigma^\omega$, we write $\Pi[\pi \mapsto t]$ for the trace assignment obtained by updating the value of π to t. Given a set of traces $\mathbb{T} \subseteq \Sigma^\omega$, a trace assignment Π, and $i \in \mathbb{N}$, we define:

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

$$
\begin{array}{lll}
\Pi \models_{\mathbb{T}} \psi & \text{iff} & \Pi, 0 \models \psi \\
\Pi \models_{\mathbb{T}} \exists\pi.\, \varphi & \text{iff} & \exists t \in \mathbb{T}.\, \Pi[\pi \mapsto t] \models_{\mathbb{T}} \varphi \\
\Pi \models_{\mathbb{T}} \forall\pi.\, \varphi & \text{iff} & \forall t \in \mathbb{T}.\, \Pi[\pi \mapsto t] \models_{\mathbb{T}} \varphi
\end{array}
$$

A *transition system* is a tuple $\mathcal{T} = (S, S_0, \kappa, L)$ where S is a set of states, $S_0 \subseteq S$ is a set of initial states, $\kappa \subseteq S \times S$ is a transition relation, and $L : S \to \Sigma$ is a labeling function. We write $s \xrightarrow{\mathcal{T}} s'$ whenever $(s, s') \in \kappa$. A path is an infinite sequence $s_0 s_1 s_2 \cdots \in S^\omega$, s.t., $s_0 \in S_0$, and $s_i \xrightarrow{\mathcal{T}} s_{i+1}$ for all i. The associated trace is given by $L(s_0)L(s_1)L(s_2)\cdots \in \Sigma^\omega$. We write $\mathit{Traces}(\mathcal{T}) \subseteq \Sigma^\omega$ for the set of all traces generated by $\mathcal{T}$. We say $\mathcal{T}$ satisfies a HyperLTL property φ, written $\mathcal{T} \models \varphi$, if $\emptyset \models_{\mathit{Traces}(\mathcal{T})} \varphi$, where $\emptyset$ denotes the empty trace assignment.

3 Automata-based HyperLTL Model Checking

Given a system $\mathcal{T}$ and HyperLTL property φ, we want to decide whether $\mathcal{T} \models \varphi$. In this section, we recap the automata-based approach to the model checking of HyperLTL [29]. We further show how language inclusion checks can be incorporated into the model checking procedure to make use of a broad collection of mature language inclusion checkers.

3.1 Automata-based Verification

The idea of automata-based verification (ABV) [29] is to iteratively eliminate quantifiers and thus reduce MC to the emptiness check on an automaton. A non-deterministic Büchi automaton (NBA) 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, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, and $F \subseteq Q$ is a set of accepting states. We write $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ for the language of $\mathcal{A}$, i.e., all infinite words that have a run that visits states in F infinitely many times (see, e.g., [2]). For traces $t_1, \ldots, t_n \in \Sigma^\omega$, we write $\mathit{zip}(t_1, \ldots, t_n) \in (\Sigma^n)^\omega$ as the pointwise product, i.e., $\mathit{zip}(t_1, \ldots, t_n)(i) := (t_1(i), \ldots, t_n(i))$.

Let $\mathcal{T} = (S, S_0, \kappa, L)$ be a fixed transition system and let $\dot{\varphi}$ be some fixed closed HyperLTL formula (we use the dot to refer to the original formula and use φ, φ' to refer to subformulas of $\dot{\varphi}$). For some subformula φ that contains free trace variables $\pi_1, \dots, \pi_n$, we say an NBA $\mathcal{A}$ over Σ^n is $\mathcal{T}$-*equivalent* to φ, if for all traces $t_1, \dots, t_n$ it holds that $[\pi_1 \mapsto t_1, \dots, \pi_n \mapsto t_n] \models_{\mathit{Traces}(\mathcal{T})} \varphi$ iff $\mathit{zip}(t_1, \dots, t_n) \in \mathcal{L}(\mathcal{A})$. That is, $\mathcal{A}$ accepts exactly the zippings of traces that constitute a satisfying trace assignment for φ.

To check if $\mathcal{T} \models \dot{\varphi}$, we inductively construct an automation $\mathcal{A}_\varphi$ 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 [29,2]. Now consider some subformula $\varphi' = \exists \pi. \varphi$ where φ' contains free trace variables $\pi_1, \dots, \pi_n$ and so φ contains free trace variables $\pi_1, \dots, \pi_n, \pi$. We are given an inductively constructed NBA $\mathcal{A}_\varphi = (Q, Q_0, \delta, F)$ over Σ^{n+1} that is $\mathcal{T}$-equivalent to φ. We define the automaton $\mathcal{A}_{\varphi'}$ over Σ^n as $\mathcal{A}_{\varphi'} := (S \times Q, S_0 \times Q_0, \delta', S \times F)$ where δ' is defined as

$$\delta'\Big((s, q), \langle l_1, \dots, l_n \rangle\Big) := \Big\{(s', q') \mid s \xrightarrow{\mathcal{T}} s' \;\wedge\; q' \in \delta\big(q, \langle l_1, \dots, l_n, L(s) \rangle\big)\Big\}.$$

Informally, $\mathcal{A}_{\varphi'}$ reads the zippings of traces $t_1, \dots, t_n$ and guesses a trace $t \in \mathit{Traces}(\mathcal{T})$ such that $\mathit{zip}(t_1, \dots, t_n, t) \in \mathcal{L}(\mathcal{A}_\varphi)$. It is easy to see that $\mathcal{A}_{\varphi'}$ is $\mathcal{T}$-equivalent to φ'. To handle universal trace quantification, we consider a formula $\varphi' = \forall \pi. \varphi$ as "$\varphi' = \neg \exists \pi. \neg \varphi$" and combine the construction for existential quantification with an automaton complementation.

Following the inductive construction, we obtain an automaton $\mathcal{A}_{\dot{\varphi}}$ over the singleton alphabet Σ^0 that is $\mathcal{T}$-equivalent to $\dot{\varphi}$. By definition of $\mathcal{T}$-equivalence, $\mathcal{T} \models \dot{\varphi}$ iff $\emptyset \models_{\mathit{Traces}(\mathcal{T})} \dot{\varphi}$ iff $\mathcal{A}_{\dot{\varphi}}$ is non-empty (which we can decide [21]).

3.2 HyperLTL Model Checking by 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 [36,15]), we can reduce to a, in practice, more tamable problem: *language inclusion*.

For a system $\mathcal{T}$, and a natural number $n \in \mathbb{N}$ we define $\mathcal{A}^n_\mathcal{T}$ as an NBA over Σ^n such that for any traces $t_1, \dots, t_n \in \Sigma^\omega$ we have $\mathit{zip}(t_1, \dots, t_n) \in \mathcal{L}(\mathcal{A}^n_\mathcal{T})$ if and only if $t_i \in \mathit{Traces}(\mathcal{T})$ for every $1 \leq i \leq n$. We can construct $\mathcal{A}^n_\mathcal{T}$ by building the n-fold self-composition of $\mathcal{T}$ [3] and convert this to an automaton by moving the labels from states to edges and marking all states as accepting. We can now state a formal connection between language inclusion and HyperLTL MC (a proof can be found in the full version [9]):

Proposition 1. *Let $\dot{\varphi} = \forall \pi_1 \dots \forall \pi_n . \varphi$ be a HyperLTL formula (where φ may contain additional trace quantifiers) and let $\mathcal{A}_\varphi$ be an automaton over Σ^n that is $\mathcal{T}$-equivalent to φ. Then $\mathcal{T} \models \dot{\varphi}$ if and only if $\mathcal{L}(\mathcal{A}^n_\mathcal{T}) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$.*

We can use Proposition 1 to avoid a complementation for the outermost quantifier alternation. For example, assume $\dot{\varphi} = \forall \pi_1 . \forall \pi_2 . \exists \pi_3 . \psi$ where ψ is quantifier-free. Using the construction from Section 3.1, we obtain an automaton $\mathcal{A}_{\exists \pi_3 . \psi}$

that is $\mathcal{T}$-equivalent to $\exists\pi_3.\psi$ (we can construct $\mathcal{A}_{\exists\pi_3.\psi}$ in linear time in the size of $\mathcal{T}$). By Proposition 1, we then have $\mathcal{T} \models \dot{\varphi}$ iff $\mathcal{L}(\mathcal{A}^2_{\mathcal{T}}) \subseteq \mathcal{L}(\mathcal{A}_{\exists\pi_3.\psi})$.

Note that complementation and subsequent emptiness check is a theoretically optimal method to solve the (PSPACE-complete) language inclusion problem. Proposition 1 thus offers no asymptotic advantages over "standard" ABV in Section 3.1. In *practice* constructing an explicit complemented automaton is often unnecessary as the language inclusion or non-inclusion might be witnessed without a complete complementation [26,25,17,24]. This makes Proposition 1 relevant for the present work and the performance of `AutoHyper`.

4 Related Work and HyperLTL Verification Approaches

HyperLTL [15] is the most studied logic for expressing hyperproperties. A range of problems from different areas in computer science can be expressed as HyperLTL MC problems, including (optimal) path panning [39], mutation testing [27], linearizability [31], robustness [22], information-flow control [36], and causality checking [18], to name only a few. Consequently, any model checking tool for HyperLTL is applicable to many disciples within computer science and provides a unified solution to many challenging algorithmic problems. In recent years, different (mostly incomplete) methods for the verification of HyperLTL have been developed. We discuss them below (see the full version [9] for details).

Automata-based Model Checking. Finkbeiner et al. [29] introduce the automata-based model checking approach as presented in Section 3.1. For alternation-free formulas, the algorithms corresponds to the construction of the self-composition of a system [3] and is implemented in the `MCHyper` tool [29]. `MCHyper` can handle systems of significant size (well beyond the reach of explicit-state methods) but is unable to handle any quantifier alternation (the main motivation for `AutoHyper`). `htltl2mc` [15] is a prototype model checker for HyperLTL_2 (a fragment of HyperLTL with at most one alternation) built on top of `GOAL` [38]. In contrast to `htltl2mc`, `AutoHyper` supports properties with arbitrarily many quantifier alternations and features automata with symbolic alphabets – which is important to handle large systems with many atomic propositions, cf. Footnote 7.

Strategy-based Verification. Coenen et al. [19] verify $\forall^*\exists^*$ properties by instantiating existential quantification with an explicit strategy. This method – which we refer to as strategy-based verification (SBV) – comes in two flavors: either the strategy is provided by the user or the strategy is 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 (and is thus no "push-button" technique). In the latter case, the method works fully automatically [8,7] but requires an expensive strategy synthesis. SBV is incomplete as the strategy resolving existentially quantified traces only observes finite prefixes of the universally quantified traces. While SBV can be made complete by adding prophecy variables [7], the automatic synthesis of such prophecies is currently limited to very small systems and properties that are temporally safe [5]. We investigate both the performance and incompleteness of SBV in Section 8.

Bounded Model Checking. Hsu et al. [31] propose a bounded model checking (BMC) procedure for HyperLTL. Similar to BMC for trace properties [11], 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. 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).[2] We compare `AutoHyper` and BMC (in the form of `HyperQube` [31]) in Section 7.

5 AutoHyper: Tool Overview

`AutoHyper` is written in `F#` and implements the automata-based verification approach described in Section 3.1 and, if desired by the user, makes use of the language-inclusion-based reduction from Section 3.2. `AutoHyper` uses `spot` [26] for LTL-to-NBA translations and automata complementations. To check language inclusion, `AutoHyper` uses `spot` (which is based on determinization), `RABIT` [17] (which is based on a Ramsey-based approach with heavy use of simulations), `BAIT` [25], and `FORKLIFT` [24] (both based on well-quasiorders). `AutoHyper` is designed such that communication with external automata tools is done via established text-based formats (opposed to proprietary APIs), namely the `HANOI` [1] 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 we represent automata using symbolic alphabets (similar to `spot`). We store transition formulas as DNFs as this allows for very efficient SAT checks, which are needed during the product construction.

All experiments in this paper were conducted on a Mac Mini with an Intel Core i3 (i3-8100B) and 16GB of memory. We used `spot` version 2.11.1; `RABIT` version 2.4.5; `BAIT` commit 369e1a4; and `FORKLIFT` commit 5d519e3.

Input Formats. `AutoHyper` supports both explicit-state systems (given in a `HANOI`-like [1] input format) and symbolic systems that are internally converted

[2] BMC for trace properties can be made complete by using bounds on the unrolling depth (also called completeness thresholds) [14] and including loop conditions in the encoding [11]. As remarked by Hsu et al. [31], the same is much more challenging for hyperproperties, and no solutions have been proposed. Instead, Hsu et al. [31] propose an alternative unrolling semantics (which they call halting semantics) that can mitigate this incompleteness issue for programs that terminate after a *fixed* number of steps. This is a strong (and often unrealistic) assumption for general reactive systems.

to an explicit-state representation. The support for symbolic systems includes Aiger circuits, symbolic models written in a fragment of the NuSMV input language [13], and a simple boolean programming language [6].

Random Benchmarks. For our evaluation, we use both existing instances from various sources in the literature and randomly generated problems.[3] We generate random transition systems based on the Erdős–Rényi–Gilbert model [28]. 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 $s \to s'$ with probability p. To generate a graph with n edges and, in expectation, constant outdegree of k, we can choose $p = \frac{k}{n}$. We further ensure that the system is connected and all states have at least one outgoing edge. We generate random HyperLTL formulas (with a given quantifier prefix) by sampling the LTL matrix using `spot`'s `randltl`.

6 HyperLTL Model Checking Complexity in Practice

Before we turn our attention to benchmarks found in the literature, we compare the different backend inclusion checkers supported by `AutoHyper` by evaluating them on a large set of synthetic (random) benchmarks (in Section 6.1). Moreover, the random generation of benchmarks allows us to peek at formulas with more than one quantifier alternation. The theoretical hardness of model checking properties with multiple alternations has been studied extensively [15,36], and we analyze, for the first time, how these results transfer to practice (in Section 6.2).

6.1 Performance of Inclusion Checkers

As the first set of benchmarks, we compare the different backend inclusion checkers supported by `AutoHyper`. In Figure 1, we depict how many instances can be solved using the inclusion checks of `spot`, `BAIT`, `RABIT`, and `FORKLIFT` within a timeout of 10s and give the median running time used on the instances that could be solved within the timeout. We observe that `spot` clearly outperforms `RABIT`, `BAIT`, and `FORKLIFT` in terms of the percentage of instances that can be checked within 10s.[4] While, in general, `spot` solves the most instances, a manual inspection reveals that there are also instances that can only be solved by `RABIT`

[3] The advantage of randomly generated instances is twofold. First, it allows for the easy generation of a large set of benchmarks. Second, the random generation is parameterized by multiple parameters (such as system size, transition density, formula size, etc.), enabling a comprehensive analysis of the exact impact of different parameters on the model checking complexity in practice.

[4] We remark that `spot` operates on automata with a symbolic alphabet (i.e., transitions are defined as boolean formulas over AP). In contrast, `RABIT`, `BAIT`, and `FORKLIFT` only support explicit alphabets (i.e., automata with one symbol for each element in 2^{AP}).

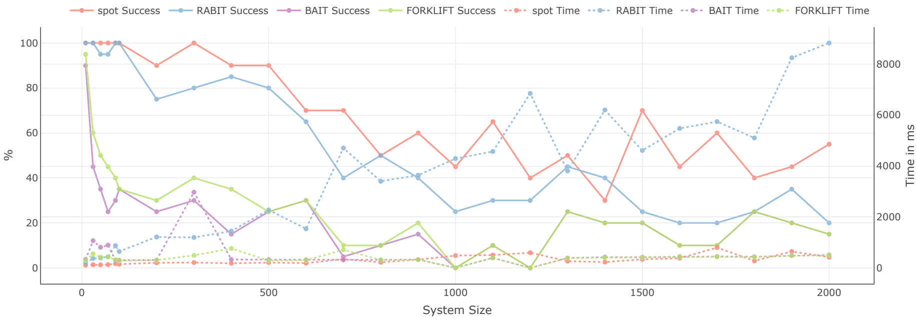

Fig. 1: We evaluate different backend solvers on instances of varying system size with an (on average) constant outdegree of 10 and a fixed property size of 20. We generate 20 samples per system size. We display both the success rate of each solver within a timeout of 10s (on the left axis) and the median running time on the solved instances (on the right axis).

or BAIT/FORKLIFT. This justifies why AutoHyper supports multiple backed inclusion checkers that implement different algorithms and thus excel on different problems (we will confirm this in Section 7). Moreover, our experiments provide evidence that HyperLTL MC is a natural source for challenging language inclusion benchmarks (see the full version [9]).

We remark that we set the timeout of 10s deliberately low to compute (and reproduce) the plots in a reasonable time (computing Figure 1 took about 3.5h). If a user wants to verify a given instance and does not require a result within a few seconds, running the solver for even longer will likely increase the success rate further (see also the evaluation in Section 7).

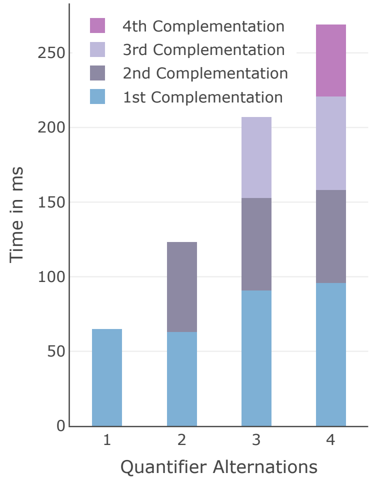

Fig. 2: For properties with a varying number of quantifier alterations, we display the average time spent on the automata complementation during model checking.

6.2 Model Checking Beyond $\forall^*\exists^*$

Using randomly generated benchmarks, we can also peek at the practical complexity of model checking in the presence of multiple quantifier alternations. In *theory*, the model checking complexity of HyperLTL increases by one exponent with each quantifier alternation [15,36]. Using AutoHyper, we can, for the first time, investigate the model checking complexity in *practice*.

We model check randomly generated formulas with 1 to 4 quantifier alternations and visualize the total running time based on the cost of each complementation (using spot) in Figure 2 (recall that checking a formula with k alternations

Table 1: We depict the running time of `AutoHyper` when verifying GNI on the boolean programs taken from [6] and [10]. We give the program, the bitwidth (bw), the size of the intermediate explicit-state representation (Size), and the time taken by each solver. The timeout is set to 60s and indicated by a "-". The property holds in all cases. Times are given in seconds.

Program	bw	Size	$t_{\texttt{spot}}$	$t_{\texttt{RABIT}}$	$t_{\texttt{BAIT}}$	$t_{\texttt{FORKLIFT}}$
[6].1	1-bit	17	**0.52**	0.59	0.80	0.61
	3-bit	65	**0.56**	0.86	-	22.73
	4-bit	129	**0.99**	5.51	-	-
[6].2	1-bit	55	**0.53**	0.69	-	5.49
[6].3	1-bit	20	**0.52**	0.61	3.05	0.98
	3-bit	80	**0.61**	1.31	-	-
[6].4	1-bit	29	**0.52**	0.56	0.58	0.57
	3-bit	113	**0.67**	1.74	-	-

Program	bw	Size	$t_{\texttt{spot}}$	$t_{\texttt{RABIT}}$	$t_{\texttt{BAIT}}$	$t_{\texttt{FORKLIFT}}$
[10].1	1-bit	5	0.52	**0.56**	0.58	0.57
[10].2	1-bit	11	**0.51**	0.57	0.72	0.61
	2-bit	27	**0.52**	0.65	35.7	5.43
	4-bit	291	**1.46**	-	-	-
[10].3	1-bit	21	**0.52**	0.60	3.15	1.00
	3-bit	225	-	**45.2**	-	-
[10].4	1-bit	25	**0.52**	0.71	12.8	1.63
	3-bit	193	**0.98**	-	-	-

using ABV requires k automaton complementations). Although the number of quantifier alternations has an undeniable impact on the total running time (the cumulative height of each bar), the increase in runtime is not proportional to the (non-elementary) increase suggested by the theoretical analysis. Different from the theoretical analysis (where the $(k+1)$th complementation is exponentially more expensive than the kth), the cost of each complementation barely increases (or even decreases). This suggests that the $\mathcal{T}$-equivalent automata constructed in each iteration are, in practice, much smaller than indicated by the worst-case theoretical analysis. Verification of properties beyond one alternation is thus less infeasible than the theory suggests (at least on randomly generated test cases).

7 Evaluation on Symbolic Systems

In this section, we challenge `AutoHyper` with complex model checking problems found in the literature. Our benchmarks stem from a range of sources, including non-interference in boolean programs [6], symmetry in mutual exclusion algorithms [19], non-interference in multi-threaded programs [37], fairness in non-repudiation protocols [32], mutation testing [27], and path planning [39].

7.1 Model Checking GNI on Boolean Programs

We use `AutoHyper` to verify GNI on a range of boolean programs that process high-security and low-security inputs (taken from [6,10]). Table 1 depicts the runtime results using different backend solvers. We test each program with varying bitwidth and depict the largest bitwidth that can be solved by at least one solver (within a timeout of 60s). We, again, note that `spot` performs better than

Table 2: We evaluate `HyperQube` and `AutoHyper` on the benchmarks from [31]. We list the system and the property (as given in [31, Table 2]), the quantifier structure ($\boldsymbol{Q}^*$), the verification result (Res) (✓ indicates that the property holds and ✗ that it is violated), and the total running time of either tool (t). For `HyperQube`, we additionally list the unrolling bound (k) and the unrolling semantics (Sem). For `AutoHyper`, we additionally list the size of the intermediate explicit state space (Size). Times are given in seconds.

				HyperQube [31]			**AutoHyper**	
System	**Spec**	$\boldsymbol{Q}^*$	**Res**	$\boldsymbol{k}$	**Sem**	$\boldsymbol{t}$	**Size**	$\boldsymbol{t}$
Bakery_3	φ_{S1}	∃∃	✗	7	pes	**1.9**	167	2.3
Bakery_3	φ_{S2}	∀∃	✗	12	pes	**2.0**	167	4.2
Bakery_3	φ_{S3}	∃∀	✗!	20	pes	**2.8**	167	34.6
Bakery_3	φ_{sym1}	∀∃	✗	10	pes	**1.7**	167	16.2
Bakery_3	φ_{sym2}	∀∃	✗	10	pes	**1.6**	167	2.9
Bakery_5	φ_{sym1}	∀∃	✗	10	pes	**17.3**	996	282.1
Bakery_5	φ_{sym2}	∀∃	✗	10	pes	18.2	996	**18.0**
SNARK-bug1	φ_{lin}	∀∃	✗	26	hpes	618.0	4941	**96.1**
$\text{3-Thread}_{correct}$	φ_{NI}	∀∃	✓	10	hopt	1.6	64	**1.3**
$\text{3-Thread}_{incorrect}$	φ_{NI}	∀∃	✗	57	hpes	12.8	368	**7.7**
$NRP : T_{correct}$	φ_{fair}	∃∀	✓	15	hopt	1.3	55	**0.5**
$NRP : T_{incorrect}$	φ_{fair}	∃∀	✓!	15	hopt	1.4	54	**0.8**
Mutant	φ_{mut}	∃∀	✓	8	hopt	1.1	32	**0.8**

other inclusion checkers and, in particular, scales better when the size of the system increases. Note that the number of atomic propositions is 3 in all instances, so `spot`'s support for symbolic alphabets has a negligible impact on the running time. We emphasize that not all instances in Table 1 can be verified using SBV [19,7] without a user-provided fixed lookahead. Likewise, BMC [31] can *never* verify GNI. This provides further evidence why complete model checking tools (of which `AutoHyper` is the first) are necessary.

7.2 Explicit Model Checking of Symbolic Systems

In this section, we evaluate `AutoHyper` on challenging symbolic models (NuSMV models [13]) that were used by Hsu et al. [31] to evaluate `HyperQube`.

The properties we verify cover a wide range of properties. For example, we verify that Lamport's bakery algorithm [33] does not satisfy various symmetry properties (as the algorithm prioritizes processes with a lower ticket ID); We check linearizability[5] [30] on the SNARK datastructure [23] and identify a previously known bug; And, we generate model-based mutation test cases using the approach proposed by Fellner et al. [27]. Further details on the benchmarks are provided in [31].

We check each instance using both HyperQube and AutoHyper and depict the results in Table 2.[6] When using AutoHyper we always apply spot's inclusion checker.[7] For HyperQube we use the unrolling semantics and unrolling depth listed in [31, Table 2]. We observe that for most instances – despite using explicit state methods and thus being complete (cf. Section 7.4) – AutoHyper performs on par with HyperQube. On instances using Lamport's bakery algorithm, BMC only needs to unroll to very shallow depths, resulting in very efficient solving, whereas AutoHyper's running time is dominated by spot's LTL-to-NBA translation (consuming up to 98% of the total time). Conversely, on the large SNARK example, AutoHyper performs significantly better.

7.3 Hyperproperties for Path Planning

As a last set of benchmarks, we use planning problems for robots encoded into HyperLTL as proposed by Wang et al. [39]. For example, the synthesis of a shortest path can be phrased as a $\exists\forall$ 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. [39] propose a solution to check the resulting HyperLTL property by encoding it in first-order logic, which is then solved by an SMT solver. While not competitive with state-of-the-art planning tools, HyperLTL allows one to express a broad range of problems (shortest path, path robustness, etc.) in a very general way. Hsu et al. [31] observe that the QBF encoding implemented in HyperQube outperforms the SMT-based approach by Wang et al. [39]. In this section, we evaluate AutoHyper on these planning-hyperproperties and compare it with HyperQube[8].

We depict the results in Table 3. It is evident that AutoHyper outperforms HyperQube, 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 in-

[5] Linearizability asserts that any execution of a concurrent data structure corresponds to a sequential execution, which is naturally expressed as a $\forall\exists$ hyperproperty.

[6] For the two verification instances (Bakery$_3$,φ_{S3}) and (NRP : $T_{incorrect}$, φ_{fair}) HyperQube provides the wrong verification result. We mark such instances with a "!" to avoid confusion when comparing Table 2 with [31, Table 2]. In particular, the supposedly unfair version of the NRP protocol is, in fact, fair.

[7] The automata use a symbolic alphabet with up to 18 letters. A conversion to an explicit alphabet – as required for RABIT, BAIT, and FORKLIFT – is thus infeasible (this would require 2^{18} symbols).

[8] AutoHyper is intended as a model checking tool, i.e., it only checks if a property holds or is violated. However, as we show in the full version [9], we could use the counterexamples returned by the inclusion checker to *synthesize* an actual plan.

Table 3: We evaluate HyperQube 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 size of the grid (Grid), and the times taken by HyperQube and AutoHyper (t). For HyperQube, we additionally give the unrolling depth used (k) and the file size of the QBF generated (|QBF|). For AutoHyper, we additionally give the size of the generated explicit state space (Size). Times are given in seconds. The timeout is set to 20 min and indicated by a "-".

		HyperQube [31]			**AutoHyper**	
Spec	**Grid**	***k***	**|QBF|**	***t***	**Size**	***t***
φ_{sp}	10 × 10	20	8 MB	4.6	146	**0.7**
	20 × 20	40	26 MB	168.1	188	**1.5**
	40 × 40	80	-	-	408	**22.7**
	60 × 60	120	-	-	404	**88.8**
φ_{rp}	10 × 10	20	13 MB	4.2	266	**0.6**
	20 × 20	40	84 MB	22.4	572	**0.7**
	40 × 40	80	419 MB	265.0	1212	**1.6**
	60 × 60	120	-	-	1852	**3.7**

termediate QBF indicates that a more optimized encoding (perhaps specific to path planning) could improve the performance of BMC on such examples.

7.4 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 [20]. Similarly, in the context of *alternation-free* hyperproperties, symbolic verification tools such as MCHyper [29] (which internally reduces to the verification of a circuit using ABC [12]) 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 model checking is currently the only method that is *complete*. While 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) and can thus not verify any of the instances in Table 1. The most significant advantage of explicit-state MC (as implemented in AutoHyper) is thus 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, cf. Table 3). We stress that this is despite the fact that for the evaluation of HyperQube we already fix an unrolling depth and unrolling semantics, thus creating favorable conditions for HyperQube.[9] 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.

8 Evaluating Strategy-based Verification

So far, we have 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 [19,7].

8.1 Effectiveness of Strategy-based Verification

SBV is known to be incomplete [19,7]. 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 [7], which synthesizes a strategy for the $\exists$-player by translating the formula to a deterministic parity automaton (DPA) [35] and phrases the synthesizes as a parity game.

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 [7]. On the other hand, our results show that random generation produces instances (albeit not many)

[9] In Tables 2 and 3, we perform a single 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 [31]). 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 (if this bound even exists).

on which SBV fails (so far, examples where SBV fails required careful construction by hand). Reverting to SBV as the default verification strategy is thus not possible, further strengthening the case for complete model checking tools (of which `AutoHyper` is the first).

8.2 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, (automata-based) 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 [15,36]. Conversely, SBV is `2-EXPTIME`-complete in the size of the specification but only `PTIME` in the size of the system [19]. Consequently, one would expect that ABV fares better on larger specifications and SBV fares better on larger systems (the more important measure in practice).

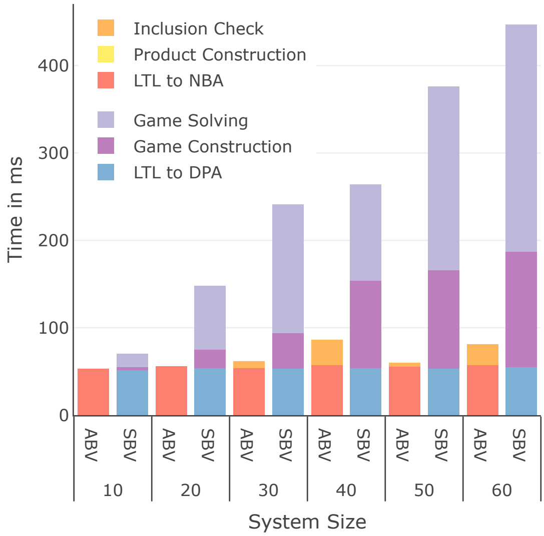

Fig. 3: We compare ABV (`AutoHyper`) and SBV ([7]) on instances of varying system size. We fix the property size to 20. We generate 100 random instances for each size and take the average over the fastest L instances, where L is the minimum number of instances solved within a 5s timeout by both methods.

However, in this section, we show that this does not translate into practice (at least using the current implementation of SBV [7]). We compare the running time of `AutoHyper` (ABV) (using `spot`'s inclusion checker) and SBV. We break the running time into the three main steps for each method. For ABV, 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 Figure 3. We observe that SBV performs worse than ABV and, more importantly, scales poorly in the size of the system. This is contrary to the theoretical analysis of ABV 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 ABV in SBV relative to the property size (we give a plot in the full version [9]). While, 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 not the LTL-to-DPA translation (which, in theory, is exponentially more expensive than the LTL-to-NBA translation used in ABV), but, again the construction and solving of the parity game.

We remark that the SBV engine we used [7] 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 games approaches its worst-case size (see the full version [9]). In contrast, the heavily optimized inclusion checker (in this case `spot`) seems to be able to check inclusion in almost constant time (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 (cf. Section 7) and further strengthens the practical impact of Proposition 1.

9 Conclusion

In this paper, we have presented `AutoHyper`, the first complete model checker for HyperLTL with an arbitrary quantifier prefix. We have demonstrated that `AutoHyper` can check many interesting properties involving quantifier alternations and often outperforms symbolic methods such as BMC, sometimes by orders of magnitude. We believe the biggest advantage of `AutoHyper` to be its push-button functionality combined with its completeness: As a user, one does not need to worry whether `AutoHyper` is applicable to a particular property (different from, e.g., SBV or BMC) and does not need to provide hints (e.g., in the form of explicit strategies in SBV).

Apart from evaluating `AutoHyper`'s performance on a range of benchmarks, we have used `AutoHyper` to **(1)** compare various backend language inclusion checkers, **(2)** explore practical verification beyond one quantifier alternation (which is not as infeasible as suggested by the theory), and **(3)** rigorously evaluate the effectiveness and efficiency of strategy-based verification in practice (which, different than suggested by a theoretical analysis, performs worse than automata-based methods).

Acknowledgments. This work was partially supported by the DFG in project 389792660 (Center for Perspicuous Systems, TRR 248), and by by the ERC Grant HYPER (No. 101055412). R. Beutner carried out this work as a member of the Saarbrücken Graduate School of Computer Science.

Data Availability Statement

`AutoHyper` and all experiments are available at [4].

References

1. 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. LNCS, vol. 9206. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_31
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
3. 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
4. Beutner, R.: AutoHyper: Explicit-state model checking for HyperLTL (2023). https://doi.org/10.5281/zenodo.7309986
5. Beutner, R., Carral, D., Finkbeiner, B., Hofmann, J., Krötzsch, M.: Deciding hyperproperties combined with functional specifications. In: Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2022. ACM (2022). https://doi.org/10.1145/3531130.3533369
6. Beutner, R., Finkbeiner, B.: A temporal logic for strategic hyperproperties. In: International Conference on Concurrency Theory, CONCUR 2021. LIPIcs, vol. 203. Schloss Dagstuhl (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.24
7. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: IEEE Computer Security Foundations Symposium, CSF 2022. IEEE (2022). https://doi.org/10.1109/CSF54842.2022.00030, https://arxiv.org/abs/2206.01797
8. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: International Conference on Computer Aided Verification, CAV 2022. LNCS, vol. 13371. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_17
9. Beutner, R., Finkbeiner, B.: AutoHyper: Explicit-state model checking for HyperLTL. CoRR **abs/2301.11229** (2023). https://doi.org/10.48550/arXiv.2301.11229
10. Beutner, R., Finkbeiner, B.: HyperATL*: A logic for hyperproperties in multi-agent systems. CoRR **abs/2203.07283** (2023). https://doi.org/10.48550/arXiv.2203.07283
11. 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. LNCS, vol. 1579. Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
12. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: International Conference on Computer Aided Verification, CAV 2010. LNCS, vol. 6174. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
13. 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. LNCS, vol. 2404. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
14. Clarke, E.M., Kroening, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2004. LNCS, vol. 2937. Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_9
15. 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. LNCS, vol. 8414. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15

16. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: IEEE Computer Security Foundations Symposium, CSF 2008. IEEE (2008). https://doi.org/10.1109/CSF.2008.7
17. 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
18. 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. LNCS, vol. 13505. Springer (2022). https://doi.org/10.1007/978-3-031-19992-9_13
19. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: International Conference on Computer Aided Verification, CAV 2019. LNCS, vol. 11561. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_7
20. 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. LNCS, vol. 2102. Springer (2001). https://doi.org/10.1007/3-540-44585-4_43
21. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods Syst. Des. **1**(2/3) (1992). https://doi.org/10.1007/BF00121128
22. 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. LNCS, vol. 10201. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_4
23. Doherty, S., Detlefs, D., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Jr., G.L.S.: DCAS is not a silver bullet for nonblocking algorithm design. In: Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2004. ACM (2004). https://doi.org/10.1145/1007912.1007945
24. Doveri, K., Ganty, P., Mazzocchi, N.: FORQ-based language inclusion formal testing. In: International Conference on Computer Aided Verification, CAV 2022. LNCS, vol. 13372. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_6
25. 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. LIPIcs, vol. 203. Schloss Dagstuhl (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.3
26. 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. LNCS, vol. 13372. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9
27. 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
28. Fienberg, S.E.: A brief history of statistical models for network analysis and open challenges. Journal of Computational and Graphical Statistics **21**(4) (2012)
29. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: International Conference on Computer Aided Verification, CAV 2015. LNCS, vol. 9206. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_3

30. 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
31. 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. LNCS, vol. 12651. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_6
32. Jamroga, W., Mauw, S., Melissen, M.: Fairness in non-repudiation protocols. In: International Workshop on Security and Trust Management, STM 2011. LNCS, vol. 7170. Springer (2011). https://doi.org/10.1007/978-3-642-29963-6_10
33. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM **17**(8) (1974). https://doi.org/10.1145/361082.361093
34. McCullough, D.: Noninterference and the composability of security properties. In: IEEE Symposium on Security and Privacy, SP 1988. IEEE (1988). https://doi.org/10.1109/SECPRI.1988.8110
35. 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
36. Rabe, M.N.: A temporal logic approach to Information-flow control. Ph.D. thesis, Saarland University (2016)
37. Smith, G., Volpano, D.M.: Secure information flow in a multi-threaded imperative language. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998. ACM (1998). https://doi.org/10.1145/268946.268975
38. Tsai, M., Tsay, Y., Hwang, Y.: GOAL for games, omega-automata, and logics. In: International Conference on Computer Aided Verification, CAV 2013. LNCS, vol. 8044. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_62
39. Wang, Y., Nalluri, S., Pajic, M.: Hyperproperties for robotics: Planning via HyperLTL. In: IEEE International Conference on Robotics and Automation, ICRA 2020. IEEE (2020). https://doi.org/10.1109/ICRA40945.2020.9196874