Acta Informatica (2025) 62:40
https://doi.org/10.1007/s00236-025-00504-z

RESEARCH

Symbolic execution for refuting ∀∃ hyperproperties

Arthur Correnson[1] · Tobias Nießen[2] · Bernd Finkbeiner[1] · Georg Weissenbacher[2]

Received: 29 November 2024 / Accepted: 11 August 2025 / Published online: 27 October 2025
© The Author(s) 2025

Abstract

Many important hyperliveness properties, such as refinement and generalized non-interference, fall into the class of ∀∃ hyperproperties, and require, for each execution trace of a system, the existence of another execution trace relating to the first one in a certain way. The alternation of quantifiers in the specification renders these hyperproperties extremely difficult to verify, or even just to test. Indeed, contrary to trace properties, where it suffices to find a single counterexample trace, refuting a ∀∃ hyperproperty requires not only to find a trace, but also a proof that no second trace exists that satisfies the specified relation with the first trace. As a consequence, automated testing of ∀∃ hyperproperties falls out of the scope of existing automated testing tools. In this paper, we present a fully automated approach to detect violations of ∀∃ hyperproperties in synchronous and asynchronous infinite-state systems. Our approach extends bug-finding techniques based on symbolic execution with support for trace quantification. We provide a prototype implementation of our approach, and demonstrate its effectiveness on a set of challenging examples.

Arthur Correnson and Tobias Nießen contributed equally to this work.

✉ Arthur Correnson
arthur.correnson@cispa.de

✉ Tobias Nießen
tobias.niessen@tuwien.ac.at

Bernd Finkbeiner
finkbeiner@cispa.de

Georg Weissenbacher
georg.weissenbacher@tuwien.ac.at

1 CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

2 TU Wien, Vienna, Austria

1 Introduction

Hyperproperties relate multiple executions of a system. While most initial interest in hyperproperties came from the area of information-flow security [25], where hyperproperties capture important policies such as noninference and observational determinism [26, 54], hyperproperties have also found numerous applications in other areas, ranging from embedded controllers [6] to concurrency [18] and sorting algorithms [23].

A key benefit of common logical formalizations and specifications of hyperproperties is that they can – concisely and precisely – capture general concepts, such as *symmetry*, that are expected to hold in wide variety of situations. Consider, as an example, two versions of a voting protocol shown in Fig. 1.

Both programs aggregate votes for two candidates A or B. The version on the left correctly tallies the votes of both candidates. The version on the right contains a bug because $countB$ is set to $countA + 1$ instead of the correct $countB + 1$ when a vote for candidate B is received. The fact that the version on the right cannot be correct can easily be seen, without even specifying the functionality of the protocol, by noticing that the faulty variant treats candidate A differently from candidate B.

More formally, the problem with the version of the protocol on the right is revealed by checking the hyperproperty specified by the following trace quantified temporal formula:

$$\forall \pi_1.\ \exists \pi_2.\ \Box(countA_{\pi_1} = countB_{\pi_2} \land countA_{\pi_2} = countB_{\pi_1})$$

The formula specifies that for every possible execution π_1, there must exist a possible execution π_2 such that the counts for A and B are exactly flipped compared to π_1. In other words, throughout the voting process, there should be an opportunity for A to receive exactly the same votes as B (and vice versa). Clearly, the voting protocol on the right violates this property, since any vote for B instantaneously puts B ahead of A by one vote, regardless of the previous values of $countA$ and $countB$.

Much of the research on the verification of hyperproperties has focused on *proving* that a hyperproperty is satisfied. Techniques for locating *violations* of hyperproperties, on the other hand, are either restricted to the analysis of finite-state systems [47], limited to the testing of a fixed property [31, 32], or require human guidance to find errors [35]. This is unfortunate, since the detection of bugs and design errors is of great help during software development [45]. In examples like the voting protocol, we are not so much interested in proving that the version on the left satisfies symmetry – which, by itself, still by no means guarantees that the protocol is functionally correct – than in finding the violation of symmetry in the version on the right, which immediately establishes that the protocol *cannot* be correct.

Fig. 1 Two versions of a simple voting protocol. The version on the right side contains a bug, which is underlined

$countA \leftarrow 0$
$countB \leftarrow 0$
loop
  input $vote \in \{A, B\}$
  if $vote = A$ **then**
    $countA \leftarrow countA + 1$
  else
    $countB \leftarrow countB + 1$
  output $countA, countB$
	$countA \leftarrow 0$
	$countB \leftarrow 0$
	loop
	  input $vote \in \{A, B\}$
	  if $vote = A$ **then**
	    $countA \leftarrow countA + 1$
	  else
	    $countB \leftarrow \underline{countA} + 1$
	  output $countA, countB$

In this paper, we develop new foundations for fully automated "hyperbug" finding, i.e., the detection of hyperproperty violations, in software systems. Contrary to existing approaches, ours does not require human intervention. Further, it is capable of providing concrete counterexamples that demonstrate why a hyperproperty does not hold.

1.1 Challenges

In formalisms for specifying hyperproperties based on trace quantification, hyperproperties are generally classified based on the type of trace quantification that they require [12, 19, 26]. For example, k-safety properties are properties that universally quantify on k traces of a system and express a safety relation between them [25]. Verifying (or testing) k-safety properties can be reduced to analyzing single trace properties on bigger systems obtained by self-composition [4]. It is then possible to exploit symmetries in the resulting composed system (or in the property itself) to drastically speed up the verification and bug-finding algorithms [32, 39, 40]. Other properties can only be expressed with an alternation of universal and existential quantifiers. This includes many interesting *hyperliveness* properties as per the classification by Clarkson and Schneider [25]. $\forall\exists$-quantified hyperproperties specify, for every possible execution trace, the existence of another execution trace that relates to the first one in a certain way. Important examples of such $\forall\exists$ hyperproperties include refinement, generalized non-interference [53], and delimited information release [58].

Verifying $\forall\exists$ hyperproperties cannot be reduced to the verification of simpler trace properties by self-composition. For every universally quantified trace, a corresponding existential witness has to be searched for. Detecting violations of $\forall\exists$ properties is also complex: it requires to find both a trace and a proof that no second trace is compatible. Again, this generally requires to enumerate all possible combinations of traces. In the context of software systems, the number of traces is generally infinite. Popular bug-finding methods such as symbolic execution [21, 22, 46, 59] and fuzzing [52] overcome this challenge by exploring only a subset of all possible traces of a system. Such under-approximating methods cannot directly be applied to $\forall\exists$ hyperproperties as considering only a strict subset of all possible traces is not sufficient to disprove the existence of a witness trace for the inner-most existential quantifier.

1.2 Contributions

This paper is an extended version of our prior work [30], in which we present the first symbolic execution method that automatically tests whether one or more programs satisfy a given $\forall\exists$ hyperproperty. The key idea of our approach is to combine two symbolic execution engines: one to find a universal trace, and one to encode the fact that no matching trace exists.

We consider properties expressed in a fragment of $\mathrm{OHyperLTL}$[12], a temporal logic well-suited for specifying hyperproperties of reactive systems. Importantly, the fragment of $\mathrm{OHyperLTL}$ we choose, which we refer to as $\mathrm{OHyperLTL}_{\mathrm{safe}}$, can express a wide range of $\forall\exists$ properties.

A key methodological point is to equip $\mathrm{OHyperLTL}_{\mathrm{safe}}$ with a bounded semantics, allowing to determine whether a formula is violated by only looking at finite execution prefixes. This semantics also extends the original semantics of $\mathrm{OHyperLTL}$ by making it

more suitable for reasoning about systems whose executions may or may not terminate. Importantly, we prove that the bounded semantics *agrees* with the unbounded one, and we demonstrate that it can express properties of a wider range of systems.

Based on the newly introduced bounded semantics of $\text{OHyperLTL}_{\text{safe}}$, we develop an algorithm to automatically locate hyperproperty violations. This new algorithm extends existing bug-finding approaches based on symbolic execution to support arbitrary trace quantification, and we further optimize this algorithm for specifications that require quantifier alternation as it occurs in $\forall\exists$ hyperproperties. We also provide a prototype implementation of the proposed algorithm. We evaluate this prototype on a selection of benchmarks drawn from the relevant literature, as well as new benchmarks specifically designed to test its limitations. Experimental results demonstrate the effectiveness of our algorithm in locating $\forall\exists$ hyperproperty violations, without any human intervention, and within seconds.

This extended version contains explanations, examples, corrections, proofs, and additional experimental evaluations that were omitted from a prior publication [30].

2 Preliminaries

2.1 First-order logic and theories

Throughout this paper, we assume some fixed, arbitrary underlying first-order theory $\mathcal{T}$ with domain $\text{Val}_{\mathcal{T}}$, and we use the following notations and conventions:

- $\text{Term}_{\mathcal{T}}(X)$ is the set of first-order terms over the set of variables X.
- $\text{Form}_{\mathcal{T}}(X)$ is the set of first-order formulas over the set of variables X.
- For any term $t \in \text{Term}_{\mathcal{T}}(X)$ and any variable assignment $\rho \in (\text{Val}_{\mathcal{T}})^X$, we denote by $[\![t]\!]^{\rho}_{\mathcal{T}} \in \text{Val}_{\mathcal{T}}$ the value of t.
- For any formula $\varphi \in \text{Form}_{\mathcal{T}}(X)$ and any variable assignment $\rho \in (\text{Val}_{\mathcal{T}})^X$, we write $\rho \models_{\mathcal{T}} \varphi$ if and only if ρ is a model of φ in $\mathcal{T}$.
- For (not necessarily disjoint) sets of variables X and X', $\sigma \in \text{Term}(X')^X$ is a substitution that maps variables from X to terms over the set of variables X'.
- For $t \in \text{Term}_{\mathcal{T}}(X)$ and $\sigma \in \text{Term}(X')^X$, $\varphi[/\sigma] \in \text{Term}_{\mathcal{T}}(X)$ is the term obtained from t by substituting every free variables x with the term $\sigma(x)$.
- For $\varphi \in \text{Form}_{\mathcal{T}}(X)$ and $\sigma \in \text{Term}(X')^X$, $\varphi[/\sigma] \in \text{Form}_{\mathcal{T}}(X)$ is the formula obtained from φ by substituting every free variables x with the term $\sigma(x)$.

For a detailed introduction to first-order logic and theories, we refer readers to Jon Barwise's "An Introduction to First-Order Logic" [8]. All results naturally translate to many-sorted first-order logic [44].

2.2 Program graphs

Remark Even though the syntax and semantics of the logic that we use to specify and reason about hyperproperties throughout this paper are largely based on [12], we choose not to adopt the representation of programs as *symbolic transition systems* as defined there because it is not well suited for path-sensitive program analysis. Instead, we use a graph-

based representation of programs, which we call *program graphs*. It is common to formally represent programs as directed graphs in one way or another, going back to Floyd's 1967 usage of flowcharts [43] and Allen's definition of control flow graphs consisting of basic blocks in 1970 [1]. Our definition of program graphs necessarily bears some resemblence of Baier's and Katoen's [2] definition, which some readers may be familiar with. Nevertheless, there are significant differences, such as the possibility of non-deterministic effects in our formalism as well as the explicit requirement that all effects must be expressible in the aforementioned theory $\mathcal{T}$.
Throughout this paper, we model programs as program graphs to facilitate reasoning and simplify formal definitions. Program graphs operate on a finite set X of program variables and have a finite set of vertices called program locations. Edges of a program graph are labeled with *guarded* instructions, which are either *variable assignments* representing (conditional) updates of the program variables or skip, which indicates no effect. Assignments are either of the form $x := e$ for $x \in X$ and $e \in \mathrm{Term}_{\mathcal{T}}(X)$, which represents an assignment of an expression to a variable, or of the form $x := *$ for $x \in X$, which represents the assignment of a nondeterministically chosen value to a variable. The latter can be used, for example, to represent external inputs from the environment. We note Instr the set of possible instructions, i.e., $\mathrm{Instr} := \{x := e \mid x \in X, e \in \mathrm{Term}_{\mathcal{T}}(X)\} \cup \{x := * \mid x \in X\} \cup \{\,\mathrm{skip}\,\}$. Formally, program graphs are defined as follows:

Definition 1 (Program graph) A *program graph* (or *control flow graph*) is a tuple

$$G = (\langle \mathcal{L}, \mathcal{E} \rangle, \ell_0, \mathrm{effect}, \mathrm{guard}),$$

where

- $\mathcal{L}$ is a non-empty, finite set of program locations,
- $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{L}$ is a set of directed edges connecting locations,
- $\ell_0 \in \mathcal{L}$ is the initial program location,
- $\mathrm{effect} : \mathcal{E} \to \mathrm{Instr}$ maps each edge to an instruction, and
- $\mathrm{guard} : \mathcal{E} \to \mathrm{Form}_{\mathcal{T}}(X)$ maps edges to (quantifier-free) formulas over program variables X.

For readability, we sometimes represent program graphs using diagrams. Fig. 2 shows how a simple program can be represented as a diagram. In such representations, we omit guard conditions on edges $e \in \mathcal{E}$ if $\mathrm{guard}(e) = \top$ (i.e., the truth constant *true*) since edges with such a guard can be taken unconditionally.

Programs operate on memories $m \in \mathcal{M}$, where $\mathcal{M} := (\mathrm{Val}_{\mathcal{T}})^X$ is the set of all possible assignments of values to program variables. For simplicity, throughout this paper, we assume that there is a particular initial memory $m_0 \in \mathcal{M}$. For example, if $\mathrm{Val}_{\mathcal{T}} = \mathbb{N}$ (or $\mathbb{Z}$), one may initialize the values of all variables to zero. Given a program graph with locations $\mathcal{L}$, the set of its execution states is $\mathcal{S} = \mathcal{L} \times \mathcal{M}$. For a state $s = \langle \ell, m \rangle \in \mathcal{S}$, we define $\mathrm{mem}(s) := m$. For an instruction $i \in \mathrm{Instr}$ and memories $m, m' \in \mathcal{M}$, we note $\langle m, i \rangle \Downarrow m'$ if executing the instruction i on memory m can result in memory m' according to the following semantics.

loop
  input x_{in}
  if $x_{\text{in}} > 0$ **then**
    output 1
  else
    output 0

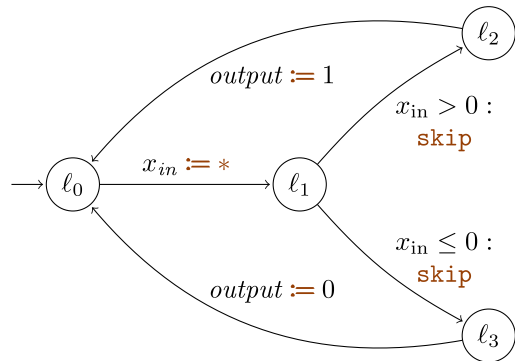


Fig. 2 A simple program and one possible representation as a program graph. Inputs are modeled using nondetermistic assignments. Output is modeled by assigning to a designated program variable

Definition 2 (Semantics of instructions)

$$\text{ASSIGN}\quad \frac{}{\langle m, x := e\rangle \Downarrow m[x \leftarrow [\![e]\!]^m_{\mathcal{T}}]} \qquad \text{HAVOC}\quad \frac{v \in \mathrm{Val}_{\mathcal{T}}}{\langle m, x := *\rangle \Downarrow m[x \leftarrow v]} \qquad \text{SKIP}\quad \frac{}{\langle m, \texttt{skip}\rangle \Downarrow m}$$

Example 1 Assume that $\mathcal{T}$ is the theory of integers, i.e., $\mathrm{Val}_{\mathcal{T}} = \mathbb{Z}$. Let $m \in \mathcal{M}$ be a memory such that $m(x) = 123$ and $m(y) = 456$. We have $\langle m, x := x + y\rangle \Downarrow m[x \leftarrow 579]$ and, independently, $\langle m, y := *\rangle \Downarrow m[y \leftarrow z]$ for any $z \in \mathbb{Z}$.

For a program graph G and two states $s_1, s_2 \in \mathcal{S}$, we note $G \vdash s_1 \hookrightarrow s_2$ if and only if there is a possible (immediate) transition from state s_1 to s_2 in G.

Definition 3 (Transition semantics) Let $G = (\langle \mathcal{L}, \mathcal{E}\rangle, \ell_0, \mathrm{effect}, \mathrm{guard})$ be a program graph. The transition relation $\hookrightarrow$ is defined as follows:

$$\text{STEP}\quad \frac{\langle \ell_1, \ell_2\rangle \in \mathcal{E} \qquad m_1 \models_{\mathcal{T}} \mathit{guard}(\ell_1, \ell_2) \qquad \langle m_1, \mathit{effect}(\ell_1, \ell_2)\rangle \Downarrow m_2}{G \vdash \langle \ell_1, m_1\rangle \hookrightarrow \langle \ell_2, m_2\rangle}$$

The execution of a program graph G progresses through a sequence of transitions of its state according to Definition 3, starting with the state $\langle \ell_0, m_0\rangle$, where ℓ_0 is the initial location of G and m_0 is the fixed initial memory. Execution terminates when a state $s_1 \in S$ is reached such that there exists no state $s_2 \in S$ with $G \vdash s_1 \hookrightarrow s_2$. Otherwise, execution continues by nondeterministically transitioning into any program state $s_2 \in S$ that satisfies $G \vdash s_1 \hookrightarrow s_2$. If only a single such program state s_2 exists, then the transition is deterministic. Nondeterminism can arise both from nondeterministic assignments of values to program variables (i.e., $x := *$) and from program locations with multiple outgoing edges whose guard conditions are not mutually exclusive.

Naturally, not every program state is reachable. The set of reachable program states of a program graph G is the smallest subset of program states such that

1. the initial state $\langle \ell_0, m_0 \rangle$ is reachable and
2. if s_1 is reachable and $G \vdash s_1 \hookrightarrow s_2$, then s_2 is reachable.

Execution traces of a program G are defined as sequences of valid computation steps starting from the state $\langle \ell_0, m_0 \rangle$. We use $\mathcal{S}^*$, $\mathcal{S}^\omega$, and $\mathcal{S}^\infty = \mathcal{S}^* \cup \mathcal{S}^\omega$ to denote the sets of finite, infinite, and mixed traces, respectively.

Definition 4 (Trace semantics) Let $s_0 = \langle \ell_0, m_0 \rangle$ be the initial state and, for any trace $\tau \in \mathcal{S}^\infty$, let τ_i be the *i*-th pair of location and memory within τ. We define:

$$\begin{aligned}
\mathrm{Traces}^*(G) &:= \{\tau \in \mathcal{S}^* \mid \tau_0 = s_0 \land G \vdash \tau_i \hookrightarrow \tau_{i+1} \text{ for } 0 \leq i < |\tau|\} \\
\mathrm{Traces}^k(G) &:= \{\tau \in \mathrm{Traces}^*(G) \mid |\tau| = k\} \\
\mathrm{Traces}^\omega(G) &:= \{\tau \in \mathcal{S}^\omega \mid \tau_0 = s_0 \land G \vdash \tau_i \hookrightarrow \tau_{i+1} \text{ for } i \in \mathbb{N}\}
\end{aligned}$$

Note that a finite trace may end in a non-terminating state and that it may, in fact, be a prefix of a longer (finite or infinite) trace. Every reachable state appears in at least one element of $\mathrm{Traces}^*(G)$, but not necessarily in any element of $\mathrm{Traces}^\omega(G)$. Further, as the following example demonstrates, there may be an infinite number of finite traces of fixed length for program graphs whose behavior is nondeterministic.

Example 2 Let G be the program graph from Fig. 2, and assume again that $\mathcal{T}$ is the theory of integers, i.e., $\mathrm{Val}_\mathcal{T} = \mathbb{Z}$. The first transition during any execution of G is necessarily $G \vdash \langle \ell_0, m_0 \rangle \hookrightarrow \langle \ell_1, m_1 \rangle$, where $m_1 := m_0[x_{\text{in}} \leftarrow z]$ for any $z \in \mathbb{Z}$. Therefore, $\mathrm{Traces}^2(G) = \{(\langle \ell_0, m_0 \rangle \langle \ell_1, m_0[x_{\text{in}} \leftarrow z] \rangle) \mid z \in \mathbb{Z}\}$ contains infinitely many traces of length two. The target location of the second transition depends on the value of the program variable x_{in} in m_1 since ℓ_1 has two outgoing edges and only one of their guard conditions is satisfied by any m_1. Hence,

$$\begin{aligned}
\mathrm{Traces}^3(G) = &\big\{(\langle \ell_0, m_0 \rangle \langle \ell_1, m_1 \rangle \langle \ell_2, m_1 \rangle) \bigm| z > 0 \land m_1 = m_0[x_{\text{in}} \leftarrow z]\big\} \cup \\
&\big\{(\langle \ell_0, m_0 \rangle \langle \ell_1, m_1 \rangle \langle \ell_3, m_1 \rangle) \bigm| z \leq 0 \land m_1 = m_0[x_{\text{in}} \leftarrow z]\big\}, \text{ and} \\
\mathrm{Traces}^4(G) = &\big\{(\langle \ell_0, m_0 \rangle \langle \ell_1, m_1 \rangle \langle \ell_2, m_1 \rangle \langle \ell_0, m_2 \rangle) \bigm| \\
&\qquad z > 0 \land m_1 = m_0[x_{\text{in}} \leftarrow z] \land m_2 = m_1[\text{output} \leftarrow 1]\big\} \cup \\
&\big\{(\langle \ell_0, m_0 \rangle \langle \ell_1, m_1 \rangle \langle \ell_3, m_1 \rangle \langle \ell_0, m_2 \rangle) \bigm| \\
&\qquad z \leq 0 \land m_1 = m_0[x_{\text{in}} \leftarrow z] \land m_2 = m_1[\text{output} \leftarrow 0]\big\}.
\end{aligned}$$

3 Specifying hyperproperties of programs in $\mathrm{OHyperLTL}_{\text{safe}}$

This paper is concerned with automated testing of hyperproperties. We consider hyperproperties expressed in *Observation-based HyperLTL* (OHyperLTL) [12], a temporal logic for specifying hyperproperties of software systems. OHyperLTL, like HyperLTL [26], allows

universal and existential quantification over traces of a system, and uses temporal operators to express relations between the quantified traces. Additionally, OHyperLTL facilitates reasoning about asynchronous, infinite-state systems by incorporating arbitrary background theories instead of limiting specifications to atomic propositions, as well as by introducing a concept of *observations* that allows specifying asynchronous hyperproperties. While, recently, other formalizations of asynchronous hyperproperties have been proposed [7, 9, 19], this combination makes OHyperLTL particularly well-suited for the specification of hyperproperties of software systems.

In this paper, we consider a fragment of OHyperLTL that restricts specifications to (relational) invariants. We call this fragment $\mathrm{OHyperLTL}_{\mathrm{safe}}$. This section introduces the syntax and the semantics of $\mathrm{OHyperLTL}_{\mathrm{safe}}$.

3.1 Syntax

An $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula begins with a sequence of universal and existential trace quantifiers $Q^{G_i}\pi_i : O_i$, where $Q \in \{\forall, \exists\}$, each quantifying over the execution traces π_i of a program G_i. Additionally, each quantifier specifies a set of locations O_i at which traces should be observed. Such observation points serve the purpose of synchronizing the different traces and deciding when they should be compared. The remainder of the formula specifies an invariant $\Box\varphi$ that should always hold across all quantified traces.

Definition 5 (Syntax of $\mathrm{OHyperLTL}_{\mathrm{safe}}$) The syntax of $\mathrm{OHyperLTL}_{\mathrm{safe}}$ is defined as follows:

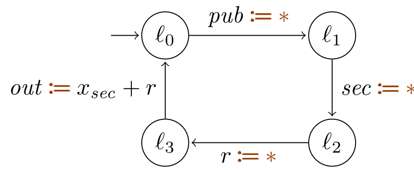
$$\psi \quad ::= \forall^G \pi : O.\ \psi \mid \exists^G \pi : O.\ \psi \mid \Box\varphi,$$

where G is a program graph with program locations $\mathcal{L}$, π is a trace variable drawn from a set $\mathcal{V}$, $O \subseteq \mathcal{L}$ is a set of observed locations, and $\varphi \in \mathrm{Form}_{\mathcal{T}}(X_{\mathcal{V}})$ is a quantifier-free first-order formula over $\mathcal{T}$ with free variables in $X_{\mathcal{V}} = \bigcup_{\pi \in \mathcal{V}} X_\pi$, i.e., for any program variable $x \in X$ and any trace variable $\pi \in \mathcal{V}$, $x_\pi \in X_{\mathcal{V}}$.

Remark Annotating trace quantifiers with program graphs allows specifying relations across executions of multiple distinct programs. In the original presentation of the logic OHyperLTL[12], trace quantifiers are annotated with programs only in examples, and these annotations are omitted from the formal definition of syntax and semantics to simplify notation. However, observation formulas, which OHyperLTL uses instead of observed locations, tend to be highly specific to the program that traces are drawn from, e.g., $(\mathrm{pc} = 2)$, where pc is a program variable representing the program counter (i.e., location). Due to this typically tight coupling between the program and the observation formula, and because of the usefulness of such annotations when reasoning about multiple programs, we choose to annotate quantifiers with program graphs in $\mathrm{OHyperLTL}_{\mathrm{safe}}$. Where a stricter separation of syntax and semantics is desired, quantifiers can instead be annotated with variables that are later resolved to sets of traces as part of the semantics.

Figure 3 shows how generalized non-interference, an important security property, can be precisely expressed in $\mathrm{OHyperLTL}_{\mathrm{safe}}$ for a simple program. Generalized non-interference

Program G:
loop
  input $pub, sec \in \mathbb{Z}$
  havoc $r \in \mathbb{Z}$
  output $sec + r$



Specification of GNI

$$\forall^G \pi_1 : \{\ell_0\}. \forall^G \pi_2 : \{\ell_0\}. \exists^G \pi_3 : \{\ell_0\}. \; \square(pub_{\pi_1} = pub_{\pi_3} \wedge out_{\pi_1} = out_{\pi_3} \wedge sec_{\pi_2} = sec_{\pi_3})$$

Fig. 3 A simple program that satisfies GNI, and a specification of GNI in $\text{OHyperLTL}_{\text{safe}}$

Program G':
loop
  input $pub, sec \in \mathbb{Z}$
  havoc $\underline{r \in \mathbb{N}}$
  output $sec + r$

Program G'':
loop
  input $pub, sec \in \mathbb{Z}$
  havoc $r \in \mathbb{Z}$
  output $sec + \underline{pub}$

Fig. 4 Two variants of the program G from Fig. 3, neither of which satisfies GNI. The relevant differences from G are underlined

(GNI) requires that any sequence of low-security (public) input/output pairs that occurs for *some* sequence of high-security (secret) inputs can occur for *any* (sufficiently long) sequence of secret inputs [53]. Thus, an adversary observing a sequence of public input/output pairs cannot definitively determine or rule out specific values of the secret input.

The program graph G in Fig. 3 does indeed satisfy GNI, and the $\text{OHyperLTL}_{\text{safe}}$ specification holds. However, both of the two variants of G in Fig. 4 violate GNI for different reasons.

The first program graph G' from Fig. 4, unlike G from Fig. 3, violates GNI because observing a negative output allows an adversary to infer that the value of sec must be negative. The second program graph G'' from Fig. 4, on the other hand, violates GNI because an adversary can compute the exact value of sec from the observed output and the known value of pub.

We note that the ability to choose observation points is an important feature to synchronize multiple traces evolving at different paces [12]. For example, the following two programs both compute double their input, but the program on the right does so using more computation steps to achieve the goal. Observation points allow to synchronize traces of both programs when they reach the output instruction, and to ignore intermediate computation steps.

```
loop                    loop
   input x ∈ ℤ             input x ∈ ℤ
  y ← 2x                  y ← x
   output y               y ← y + x
                           output y
```

Another important feature of OHyperLTL is that trace quantification is relative to user-specified programs. This allows to draw execution traces from different programs to express relational hyperproperties such as refinement. For example, Fig. 5 shows how to specify that

a program MIN calculating the minimum of two integers x and y is a *refinement* of a program FLIP that nondeterministically selects either x or y.

As will be apparent once we establish semantics in the following section, the specification in Fig. 5 holds: for every execution of MIN, there exists an execution of FLIP such that their inputs and outputs match. However, if the two programs were to be swapped, i.e., if the specification began with the quantifiers $\forall^{\text{FLIP}}\pi_1 : \{\ell_0\}.\ \exists^{\text{MIN}}\pi_2 : \{\ell_0\}$ instead, the property would clearly not hold. In other words, MIN is a refinement of FLIP, but FLIP is not a refinement of MIN.

3.2 Semantics

$\text{OHyperLTL}_{\text{safe}}$ formulas are evaluated on projections of traces that effectively hide computation steps occurring between two observation points. We define these projections in the form of *observed traces* as follows.

Definition 6 (Observational trace semantics) Let G be a program graph with program locations $\mathcal{L}$ and let $O \subseteq \mathcal{L}$ be a set of *observed locations*. We define

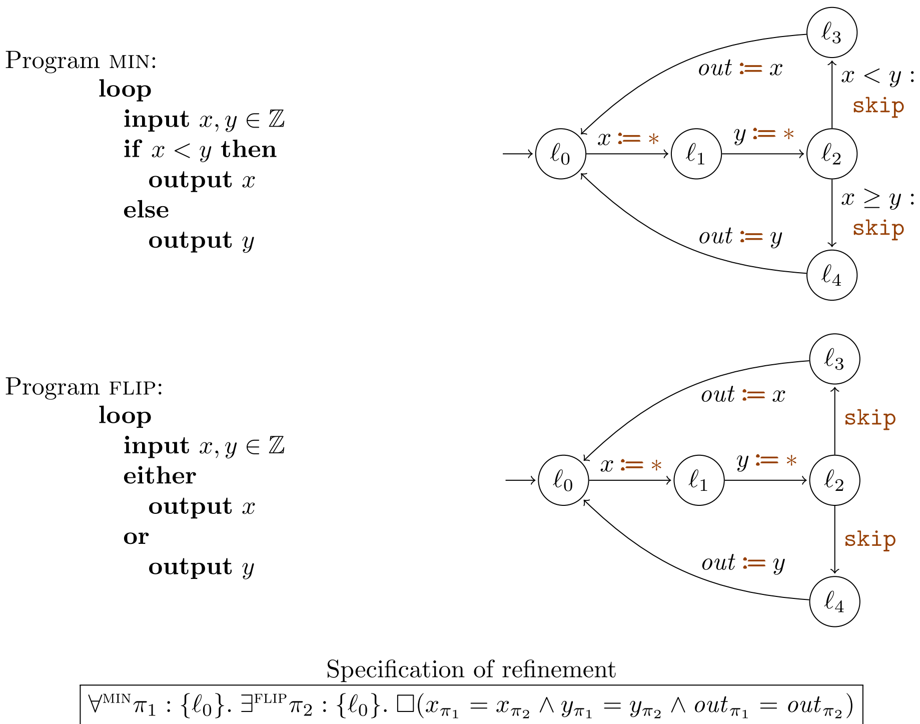


Fig. 5 Two programs with the equivalent program graphs, and a specification of refinement in $\text{OHyperLTL}_{\text{safe}}$

$$\begin{aligned}
\mathrm{Traces}^{\omega}_{O}(G) &:= \{[\tau]_O \mid \tau \in \mathrm{Traces}^{\omega}(G) \wedge |\tau|_O = \infty\} \\
\mathrm{Traces}^{*}_{O}(G) &:= \{[\tau]_O \mid \tau \in \mathrm{Traces}^{*}(G)\} \\
\mathrm{Traces}^{k}_{O}(G) &:= \{\tau \in \mathrm{Traces}^{*}_{O}(G) \mid |\tau| = k\},
\end{aligned}$$

where $[\tau]_O$ is the sequence of states obtained from a trace τ by removing all states whose locations are not in O, and $|\tau|_O = |\,[\tau]_O\,|$ is the number of states in τ with a location in O.

Given a formula ψ and a partial map $\Pi : \mathcal{V} \to \mathcal{S}^{\omega}$ mapping the free trace variables of ψ to concrete (projections of) traces, we write $\Pi \models^{\omega} \psi$ if and only if the traces in Π satisfy ψ as per the following definition of the semantics.

Definition 7 (Infinite-trace semantics of $\mathrm{OHyperLTL}_{\mathrm{safe}}$)

$$\begin{aligned}
\Pi \models^{\omega} \forall^{G}\pi : O.\psi &\iff \forall \tau \in \mathrm{Traces}^{\omega}_{O}(G),\ \Pi[\pi \leftarrow \tau] \models^{\omega} \psi \\
\Pi \models^{\omega} \exists^{G}\pi : O.\psi &\iff \exists \tau \in \mathrm{Traces}^{\omega}_{O}(G),\ \Pi[\pi \leftarrow \tau] \models^{\omega} \psi \\
\Pi \models^{\omega} \Box\varphi &\iff \forall i \in \mathbb{N},\ \Pi_i \models_{\mathcal{T}} \varphi, \\
&\qquad \text{where } \Pi_i(x_\pi) := \mathrm{mem}(\Pi(\pi)(i))(x) \text{ and } \mathrm{mem}(\langle \ell, m\rangle) := m \\
\models^{\omega} \psi &\iff \emptyset \models^{\omega} \psi
\end{aligned}$$

It is important to note that this semantics, which closely follows the original definition of OHyperLTL[12], completely disregards finite traces, as well as infinite traces with only finitely many observation points. This is not generally desirable for software systems and can become a major obstacle when specifying properties of programs that exhibit both terminating and non-terminating behaviors. In particular, any OHyperLTL formula starting with a universal quantifier $\forall^{G} \ldots$ is trivially satisfied if G is a terminating program or a program whose traces have only finitely many observation points. Dually, any OHyperLTL formula starting with an existential quantifier $\exists^{G} \ldots$ is trivially violated for such a program G. In other words, under this semantics, universal trace quantification is too weak, and existential trace quantification is too strong. For example, the following echo server trivially satisfies non-interference under the infinite-trace semantics, even though it clearly leaks the secret input:

```
havoc n ∈ ℕ
repeat n times
  input  pub , sec  ∈ ℤ
  output sec
```

Regardless of the (random) value of the program variable $n \in \mathbb{N}$, the loop terminates after a finite number of iterations. Hence, the program produces only finite traces.

In the following, we progressively refine the semantics of $\mathrm{OHyperLTL}_{\mathrm{safe}}$ to address this issue. The goal is to obtain a semantics that coincides with Definition 7 for programs with only traces with infinitely many observations, while providing a more intuitive treatment of traces with finitely many observations.

3.3 Bounded semantics

As a first step, we begin by introducing a bounded variant of the semantics that only considers traces with exactly k observations.

Definition 8 (Bounded semantics of $\text{OHyperLTL}_{\text{safe}}$)

$$
\begin{aligned}
\Pi \models^k \forall^G \pi : O.\ \psi &\iff \forall \tau \in \text{Traces}_O^k(G),\ \Pi[\pi \leftarrow \tau] \models^k \psi \\
\Pi \models^k \exists^G \pi : O.\ \psi &\iff \exists \tau \in \text{Traces}_O^k(G),\ \Pi[\pi \leftarrow \tau] \models^k \psi \\
\Pi \models^k \Box \varphi &\iff \forall i \in [0, k-1],\ \Pi_i \models_{\mathcal{T}} \varphi, \\
&\qquad \text{where } \Pi_i(x_\pi) := \text{mem}(\Pi(\pi)(i))(x) \\
\models^k \psi &\iff \emptyset \models^k \psi
\end{aligned}
$$

Intuitively, $\models^k \psi$ means that ψ holds (i.e., is not violated) if we consider only prefixes of observational length exactly k. However, it does not provide any information on traces with strictly less or strictly more observations. Before we explore this further, we show that the bounded semantics agrees with the unbounded semantics for non-terminating programs that only have traces with infinitely many observations. We call such programs *infinitely observable*.

Definition 9 (Infinitely observable programs) Let G be a program graph and let O be a set of observed locations. We say that G is *infinitely observable with respect to* O if, for every $k \in \mathbb{N}$, every trace $\tau \in \text{Traces}_O^k(G)$ is a prefix of some trace $\bar{\tau} \in \text{Traces}_O^\omega(G)$.

In other words, infinitely observable programs are non-terminating programs such that all their traces have infinitely many observations. This definition is naturally extended to $\text{OHyperLTL}_{\text{safe}}$ specifications as follows:

Definition 10 (Infinitely observable specifications) Let ψ be an $\text{OHyperLTL}_{\text{safe}}$ specification. We say that ψ is *infinitely observable* if, for each quantified program G annotated with observed locations O in ψ, G is infinitely observable with respect to O.

Theorem 1 *Let ψ be an infinitely observable $OHyperLTL_{\text{safe}}$ specification and let $k \in \mathbb{N}$. Then, $\models^\omega \psi \implies \models^k \psi$.*

Proof We prove this theorem by induction on the structure of the formula ψ for arbitrary mappings $\Pi \in (\mathcal{S}^\omega)^\mathcal{V}$ and $\Pi' \in (\mathcal{S}^k)^\mathcal{V}$ such that $\Pi'(\pi)$ is a prefix of $\Pi(\pi)$ for every $\pi \in \mathcal{V}$ (we note $\Pi' \sqsubset \Pi$). In the following, we assume that $\Pi \models^\omega \psi$ holds. In each case, we need to show $\Pi' \models^k \psi$.

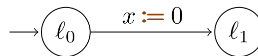
- Suppose $\psi = \Box\varphi$ for some φ. Clearly, $\Pi' \models^k \Box\varphi$ holds since $\Pi' \sqsubset \Pi$.
- Suppose $\psi = \forall^G \pi : O.\ \psi'$ for some ψ'. Let $\tau \in \text{Traces}_O^k(G)$ be arbitrary. It suffices to show $\Pi'[\pi \leftarrow \tau] \models^k \psi'$. Since G is infinitely observable, τ can be extended to $\bar{\tau} \in \text{Traces}_O^\omega(G)$, and since $\Pi \models^\omega \psi$, we have $\Pi[\pi \leftarrow \bar{\tau}] \models^\omega \psi'$. Further, since $\Pi' \sqsubset \Pi$, it is easy to see that $\Pi'[\pi \leftarrow \tau] \sqsubset \Pi[\pi \leftarrow \bar{\tau}]$. By induction hypothesis, it follows that

$\Pi'[\pi \leftarrow \tau] \models^k \psi'$.

- Suppose $\psi = \exists^G \pi : O.\, \psi'$. By definition of the bounded semantics, it suffices to find some $\tau \in \mathrm{Traces}_O^k(G)$ such that $\Pi'[\pi \leftarrow \tau] \models^k \psi'$. Since $\Pi \models^\omega \psi'$, there exists $\bar{\tau} = \bar{\tau}_0\bar{\tau}_1\bar{\tau}_2 \ldots \in \mathrm{Traces}_O^\omega(G)$ such that $\Pi[\pi \leftarrow \bar{\tau}] \models^\omega \psi'$. We pick $\tau = \bar{\tau}_0 \ldots \bar{\tau}_{k-1} \in \mathrm{Traces}_O^k(G)$. Clearly, $\Pi'[\pi \leftarrow \tau] \sqsubset \Pi[\pi \leftarrow \bar{\tau}]$ and, by induction hypothesis, $\Pi'[\pi \leftarrow \tau] \models^k \psi'$ follows. □

In the context of automated bug finding, Theorem 1 is crucial as it guarantees that violations detected with respect to the bounded semantics immediately translate to violations with respect to the unbounded semantics.

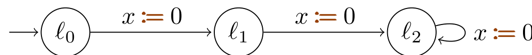
For specifications that are not infinitely observable, the infinite semantics and the bounded semantics disagree by design. Indeed, the bounded semantics addresses the problem presented above: it does not ignore finite traces, nor infinite traces with only finitely many observations. However, it still has some limitations. Importantly, $\models^{k+1} \psi$ does not imply $\models^k \psi$: even if no violations are detected for trace prefixes with *k* observations, there could still be violations for trace prefixes with $k' < k$ observations. This is somewhat counterintuitive, in particular when compared to usual bounded semantics for single trace logics such as LTL [15]. For example, consider the following program graph *G*



and the specification $\psi = \forall^G \pi : \{\ell_1\}.\, \Box(x_\pi \neq 0)$. Clearly, the program should be considered to violate the specification as it assigns $x := 0$, thus the invariant $x \neq 0$ does not globally hold. Nevertheless, under the unbounded semantics of $\mathrm{OHyperLTL}_{\mathrm{safe}}$ (see Definition 7) as well as under the original semantics of OHyperLTL[12], ψ holds.

Returning to the new bounded semantics (see Definition 8), for $k = 1$, we indeed have $\not\models^k \psi$ as expected. However, $\models^k \psi$ trivially holds for any $k > 1$ because, for such values of *k*, the set of traces $\mathrm{Traces}_{\{\ell_1\}}^k(G)$ is empty.

Note that this problem still occurs even if we restrict ourselves to non-terminating programs. For example, for the following non-terminating variant of *G*, the same specification ψ is still satisfied for $k > 1$ because it steps through ℓ_1 only once, which is the only observed location:



Fortunately, in order to achieve a more intuitive *k*-bounded semantics, it suffices to consider bad interactions between traces with at most *k* observations, instead of traces with exactly *k* observations.

Definition 11 (Upper-bounded semantics) Let ψ be a specification and let $k \in \mathbb{N}$. We define $\models^{\leq k} \psi$ such that $\models^{\leq k} \psi \iff \forall k' \leq k, \models^{k'} \psi$.

By definition, and contrary to $\models^k$, the semantics of $\models^{\leq k}$ enjoys the intuitive property of being monotonic.

Theorem 2 *(Monotonicity of $\models^{\leq k}$) Let ψ be a specification and let $k, k' \in \mathbb{N}$ such that $k' < k$. Then $\models^{\leq k} \psi \implies \models^{\leq k'} \psi$.*

Proof Assume that $\models^{\leq k} \psi$ holds. As per Definition 11, this implies that $\models^{k''} \psi$ holds for all $k'' \leq k$. Because $k' < k$, this also implies that $\models^{k''} \psi$ holds for all $k'' \leq k'$. By definition, this is equivalent to $\models^{\leq k'} \psi$. □

Further, it is easy to see that $\models^{\leq k}$ still agrees with $\models^{\omega}$ for infinitely observable specifications (see Definition 10).

Theorem 3 *Let ψ be an infinitely observable specification and let $k \in \mathbb{N}$. Then, $\models^{\omega} \psi \implies \models^{\leq k} \psi$.*

Proof Assume that $\models^{\omega} \psi$ holds. As per Theorem 1, because ψ is infinitely observable, for all $k' \in \mathbb{N}$, we have $\models^{k'} \psi$. Thus, for all $k' \leq k$, we have $\models^{k'} \psi$, and hence, $\models^{\leq k} \psi$ holds.□

We conclude this section with further examples that demonstrate the expressivity of $\text{OHyperLTL}_{\text{safe}}$.

Example 3 The tremendous difficulty of verifying or refuting hyperproperties expressed in logics such as $\text{OHyperLTL}_{\text{safe}}$ is apparent in the fact that many open mathematical problems can easily and intuitively be formulated as such hyperproperties. For example, given a program graph $\textsc{Even}$ that simply outputs random, even natural numbers and a program graph $\textsc{Prime}$ that outputs random prime numbers, Goldbach's conjecture can be formulated as the $\text{OHyperLTL}_{\text{safe}}$ specification

$$\forall^{\textsc{Even}} \pi_1 : \{\ell_{out}\}.\ \exists^{\textsc{Prime}} \pi_2 : \{\ell_{out}\}.\ \exists^{\textsc{Prime}} \pi_3 : \{\ell_{out}\}.\\ \Box(out_{\pi_1} > 2 \implies out_{\pi_1} = out_{\pi_2} + out_{\pi_3}),$$

which, of course, is neither known to hold nor not to hold.

Asynchronous hyperproperties can also be useful for specifying properties of concurrent systems [18]. In terms of program graphs (see Definition 1), concurrent (interleaved) execution of two threads can be represented, for example, by storing an explicit program counter for each thread in a separate program variable.

Example 4 The following program repeatedly and nondeterministically increases the value of an integer variable by either one or two and outputs the new value in each iteration.

```
repeat
   havoc  inc ∈ { 1, 2 }
   variable ← variable + inc
   output  variable
```

Let $\textsc{Sequential}$ be the above program, and let w_{s} be the location of the **output** statement in this program. Let $\textsc{Parallel}$ be a program that executes two copies of the above program concurrently such that they share the same program variable variable, and let w_{t0} and w_{t1} be the locations of the **output** statements in the two threads, respectively.

The following $\text{OHyperLTL}_{\text{safe}}$ specification is a variant of linearizability [18] and establishes that every output sequence that can be produced by PARALLEL must also be possible to produce by executing SEQUENTIAL.

$$\forall^{\text{PARALLEL}}\pi_1 : \{ w_{\mathtt{t0}}, w_{\mathtt{t1}} \}.\ \exists^{\text{SEQUENTIAL}}\pi_2 : \{ w_{\mathtt{s}} \}.\ \square(\mathtt{variable}_{\pi_1} = \mathtt{variable}_{\pi_2}).$$

If the read-write operation that increases the value of variable is atomic, the property holds. Otherwise, if either of the two concurrent threads of PARALLEL might be interrupted after reading the value of variable and before updating the value of variable, the property does not hold. In this case, the shortest counterexample requires exploration up to the second synchronized observation: Assume that variable is initialized to zero in both PARALLEL and in SEQUENTIAL, and assume that the first thread of PARALLEL is scheduled for execution first. It randomly sets its local copy of $\text{inc} \in \{1, 2\}$ and reads the value of variable (storing it in some temporary register or on some stack), but is then interrupted by the second thread, which also randomly sets its own local copy of $\text{inc} \in \{1, 2\}$ and then also reads the value of the shared variable. Next, the first thread is resumed, which adds to the previously obtained value of variable the value of inc that the same first thread determined before. This new value is produced as an output and an observed location is reached, namely, w_{t0}. At this observation, the specification holds since any possible value of variable_{π_1} is either one or two and thus can also be obtained from an execution of SEQUENTIAL up to the first occurrence of its observed location w_{s}. Now, assume that the second thread is resumed and assigns to the shared variable variable the sum of the value that it previously read from variable (i.e., zero) and its local variable inc (i.e., either one or two). The second thread subsequently reaches the observed location w_{t1}, which is synchronized with the second observation point of the execution of SEQUENTIAL. If variable_{π_1} is equal to two at this second observation point, then the property again holds for this particular assignment of π_1 because an execution of SEQUENTIAL exists that increased the value of variable_{π_2} by one twice, thus yielding the value two as well. However, if the second thread of PARALLEL selected $\text{inc} = 1$, then we have $\text{variable}_{\pi_1} = 1$ at this second observation point, and no execution of SEQUENTIAL can satisfy $\text{variable}_{\pi_1} = \text{variable}_{\pi_2}$ at this point. Hence, the specification does not hold.

This example requires quantifier alternation in the specification due to the repeated nondeterministic behavior of the program. Further, such specifications tend to be difficult to verify or to test due to the vast complexity of considering all possible interleavings of multiple concurrent threads. Nevertheless, for this example, the approach that we describe in the next section rapidly locates a specification violation.

4 Finding counterexamples by symbolic execution

In the previous section, we have defined $\text{OHyperLTL}_{\text{safe}}$, a logic well-suited to specify the presence of good and the absence of bad interactions between multiple executions of a program. The semantics that we defined for $\text{OHyperLTL}_{\text{safe}}$ in Section 3.3 deviates from other existing relational logics such as OHyperLTL[12] as it allows to consistently reason about both terminating and non-terminating executions of programs.

Even in the case of the bounded semantics of $\text{OHyperLTL}_{\text{safe}}$, finding a counterexample to an $\text{OHyperLTL}_{\text{safe}}$ formula might require inspecting an infinite number of traces. In

particular, to find counterexamples to specifications of the form $\forall\pi_1\exists\pi_2\ldots$, we need to identify a trace π_1 that cannot be matched with any corresponding trace π_2. In turn, this requires exhaustively exploring the set of all candidate traces π_2. As we have seen in Example 2, the set of all traces of a program is usually infinite, even if their length is restricted. Nevertheless, it is possible to efficiently compute a (finite) symbolic representation of all traces of length *k* using symbolic execution [51]. In this section, we take advantage of this observation to develop a symbolic encoding of the bounded semantics of $\mathrm{OHyperLTL}_{\mathrm{safe}}$. We will then use this symbolic encoding to devise an algorithm that is capable of finding counterexamples.

4.1 Symbolic encoding of the bounded semantics

We start by defining symbolic semantics and encodings for program graphs. In the following, we let V_* be an infinite set of unique *fresh variables* over $\mathcal{T}$ such that $V_* \cap X = \emptyset$, where X is the set of program variables (see Section 2). Further, we assume a procedure $\textsc{fresh}()$ that generates a new variable in V_*, different from all the other variables in context.

Symbolic execution aims to generate a symbolic encoding of sets of execution paths. Following [16] and [29], we present symbolic execution as another semantics of programs in which concrete memories, states, and traces are replaced with symbolic memories, states, and traces, respectively.

In the context of symbolic execution, programs operate on a *symbolic memory* $\hat{m} \in \hat{\mathcal{M}}$, where $\hat{\mathcal{M}} = (\mathrm{Term}_{\mathcal{T}}(V_*))^X$, which maps program variables to symbolic expressions, i.e., to terms over $\mathcal{T}$ whose free variables are fresh variables from V_*. Recall that $m_0 \in \mathcal{M}$ is some fixed initial memory as described in Section 2. We use $\hat{m}_0$ to denote some fixed symbolic memory that corresponds to the initial memory m_0, i.e., $\hat{m}_0 \in \hat{\mathcal{M}}$ satisfies $[\![\hat{m}_0(x)]\!]^{\emptyset}_{\mathcal{T}} = m_0(x)$ for all $x \in X$. While the choice of $\hat{m}_0$ is not necessarily unique, typically, $\hat{m}_0(x)$ is chosen to be a constant in the language of the theory $\mathcal{T}$, i.e., $m_0(x) = [\![c_x]\!]_{\mathcal{T}}$ for some constant c_x in the language of $\mathcal{T}$.

A *symbolic state* is a triple $\hat{s} = \langle \ell, \varphi, \hat{m}\rangle$, where ℓ is a program location, $\varphi \in \mathrm{Term}_{\mathcal{T}}(V_*)$ is a quantifier-free *path formula*, and $\hat{m} \in \hat{\mathcal{M}}$ is a symbolic memory. A *symbolic trace* is a finite sequence of symbolic states. For a given symbolic trace $\hat{\tau} = \langle \ell_0, \varphi_0, \hat{m}_0\rangle \ldots \langle \ell_n, \varphi_n, \hat{m}_n\rangle$, we use $\mathrm{path}(\hat{\tau}) := \varphi_n$ to denote the accumulated path formula φ_n and $|\hat{\tau}| := n + 1$ to denote the length of $\hat{\tau}$. Within this model of symbolic execution, we can equip program graphs with a *symbolic semantics* that defines how to compute symbolic encodings of program traces. The symbolic semantics mimics the concrete semantics presented in Definition 3, but replaces concrete memory assignments with symbolic ones.

Definition 12 (Symbolic operational semantics)

$$\frac{}{\langle \hat{m}, x \coloneqq e \rangle \Downarrow_{sym} \hat{m}[x \leftarrow e[/\hat{m}]]}\ \text{SYM-ASSIGN} \qquad \frac{}{\langle \hat{m}, x \coloneqq * \rangle \Downarrow_{sym} \hat{m}[x \leftarrow \text{FRESH}()]}\ \text{SYM-HAVOC} \qquad \frac{}{\langle \hat{m}, \texttt{skip} \rangle \Downarrow_{sym} \hat{m}}\ \text{SYM-SKIP}$$

$$\text{SYM-STEP}\quad \frac{\langle \ell_1, \ell_2 \rangle \in \mathcal{E} \qquad \varphi_2 = \varphi_1 \wedge \mathit{guard}(\ell_1, \ell_2)[/\hat{m}_1] \qquad \exists \rho, \rho \models_{\mathcal{T}} \varphi_2 \qquad \langle \hat{m}_1, \mathit{effect}(\ell_1, \ell_2) \rangle \Downarrow_{sym} \hat{m}_2}{G \vdash \langle \ell_1, \varphi_1, \hat{m}_1 \rangle \hookrightarrow_{sym} \langle \ell_2, \varphi_2, \hat{m}_2 \rangle}$$

Analogously to finite concrete traces (see Definition 4) and their projections to sequences of observations (see Definition 6), we define symbolic traces and observations over symbolic traces as follows.

Definition 13 (Symbolic traces of program graphs) Let G be a program graph with program locations $\mathcal{L}$ and let $O \subseteq \mathcal{L}$ be a set of observed locations. For any sequence of symbolic states $\hat{\tau} = \langle \ell_0, \varphi_0, \hat{m}_0 \rangle \dots \langle \ell_n, \varphi_n, \hat{m}_n \rangle$, let τ_i be $\langle \ell_i, \varphi_i, \hat{m}_i \rangle$. We define:

$$\begin{aligned}
\text{SymTraces}^*(G) &:= \{\hat{\tau} \mid \hat{\tau}_0 = \langle \ell_0, \mathsf{true}, \hat{m}_0 \rangle \ \wedge\ G \vdash \hat{\tau}_i \hookrightarrow_{\text{sym}} \hat{\tau}_{i+1} \text{ for all } i < |\hat{\tau}| - 1\} \\
\text{SymTraces}^*_O(G) &:= \{[\hat{\tau}]_O \mid \hat{\tau} \in \text{SymTraces}^*(G)\} \\
\text{SymTraces}^k_O(G) &:= \{\hat{\tau} \in \text{SymTraces}^*_O(G) \mid |\hat{\tau}| = k\},
\end{aligned}$$

where $[\hat{\tau}]_O$ is the sequence of states obtained from a trace τ by removing all symbolic states whose locations are not in O.

Given an assignment $\rho : V_* \to \text{Val}_{\mathcal{T}}$ for the fresh variables generated by the procedure FRESH during symbolic execution, a symbolic trace $\hat{\tau}$ can be concretized into a unique (concrete) trace $\tau = \gamma_\rho(\hat{\tau})$ as follows.

Definition 14 (Concretization) In the following, we use $x \mapsto [\![\hat{m}(x)]\!]^\rho_{\mathcal{T}}$ to denote the function with domain X that is the concretization of a symbolic memory $\hat{m} \in \hat{\mathcal{M}}$ with respect to the assignment ρ. In other words, $(x \mapsto [\![\hat{m}(x)]\!]^\rho_{\mathcal{T}}) \in \mathcal{M}$ is a memory as defined in Section 2.

$$\begin{aligned}
\gamma_\rho(\langle \ell_0, \varphi_0, \hat{m}_0 \rangle \dots \langle \ell_n, \varphi_n, \hat{m}_n \rangle) &:= \langle \ell_0, x \mapsto [\![\hat{m}_0(x)]\!]^\rho_{\mathcal{T}} \rangle \dots \langle \ell_n, x \mapsto [\![\hat{m}_n(x)]\!]^\rho_{\mathcal{T}} \rangle \\
\gamma(\hat{\tau}) &:= \{\gamma_\rho(\hat{\tau}) \mid \rho : V_* \to \text{Val}_{\mathcal{T}} \text{ s.t. } \rho \models_{\mathcal{T}} \text{path}(\hat{\tau})\} \\
\gamma(T) &:= \bigcup_{\hat{\tau} \in T} \gamma(\hat{\tau}), \quad \text{where } T \text{ is a set of symbolic traces.}
\end{aligned}$$

Importantly, the concretization of the set of all symbolic traces is equal to the set of all (finite) concrete traces.

Theorem 4 *(Equivalence of symbolic and concrete trace semantics) For any program graph* G*,* $\gamma(\mathrm{SymTraces}^*(G)) = \mathrm{Traces}^*(G)$ *holds.*

We omit the proof of Theorem 4 due to its lengthy nature. Equivalent results for similar formalizations have been established in various works [16, 30] and naturally translate to our use of symbolic semantics. Further, we obtain the following corollary that applies to the bounded trace semantics with observations as follows.

Corollary 5 *Let* G *be a program graph with program locations* $\mathcal{L}$*, let* $O \subseteq \mathcal{L}$ *be a set of observed locations, and let* $k \in \mathbb{N}$*. Then,* $\gamma(\mathrm{SymTraces}^k_O(G)) = \mathrm{Traces}^k_O(G)$ *holds.*

Proof The following equivalence holds.

$$
\begin{aligned}
& \tau \in \mathrm{Traces}^k_O(G) \\
\iff & \exists \tau' \in \mathrm{Traces}^*(G) \text{ s.t. } |\tau'|_O = k \wedge \tau = [\tau']_O && \text{(by Definition 6)} \\
\iff & \exists \tau' \in \gamma(\mathrm{SymTraces}^*(G)) \text{ s.t. } |\tau'|_O = k \wedge \tau = [\tau']_O && \text{(by Theorem 4)} \\
\iff & \exists \hat{\tau} \in \mathrm{SymTraces}^*(G) \text{ s.t. } |\hat{\tau}|_O = k \wedge \tau \in \gamma([\hat{\tau}]_O) && \text{(by Definition 14)} \\
\iff & \exists \hat{\tau} \in \mathrm{SymTraces}^*_O(G) \text{ s.t. } |\hat{\tau}| = k \wedge \tau \in \gamma(\hat{\tau}) && \text{(by Definition 13)} \\
\iff & \exists \hat{\tau} \in \mathrm{SymTraces}^k_O(G) \text{ s.t. } \tau \in \gamma(\hat{\tau}) && \text{(by Definition 13)} \\
\iff & \tau \in \gamma(\mathrm{SymTraces}^k_O(G)). && \text{(by Definition 14)}
\end{aligned}
$$

□

Given a specification ψ and a bound n, our goal is to determine whether or not $\models^{\leq n} \psi$ holds. More precisely, we are interested in finding, as quickly as possible, evidence that $\not\models^{\leq n} \psi$. To do so, we give a symbolic encoding of the bounded semantics of $\mathrm{OHyperLTL}_{\mathrm{safe}}$. The core idea we are building towards is to evaluate $\mathrm{OHyperLTL}_{\mathrm{safe}}$ specifications on the symbolic traces of a system instead of on all the concrete traces. This is possible because symbolic traces faithfully represent concrete ones in virtue of Theorem 4. When encoding an $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula symbolically, universal and existential quantification of concrete traces are naturally replaced by conjunction and disjunction over sets of symbolic traces, respectively.

Definition 15 (Symbolic encoding of $\mathrm{OHyperLTL}_{\mathrm{safe}}$) Given a bound $k \in \mathbb{N}$ and a partial map $\hat{\Pi}$ assigning symbolic traces to trace variables, we define:

$$
\begin{aligned}
[\![\forall^G \pi : O.\ \psi]\!]^k_{\hat{\Pi}} &:= \bigwedge_{\hat{\tau} \in \mathrm{SymTraces}^k_O(G)} \forall(\mathrm{FV}(\hat{\tau})), \Big(\mathrm{path}(\hat{\tau}) \rightarrow [\![\psi]\!]^k_{\hat{\Pi}[\pi \leftarrow \hat{\tau}]}\Big) \\
[\![\exists^G \pi : O.\ \psi]\!]^k_{\hat{\Pi}} &:= \bigvee_{\hat{\tau} \in \mathrm{SymTraces}^k_O(G)} \exists(\mathrm{FV}(\hat{\tau})), \Big(\mathrm{path}(\hat{\tau}) \wedge [\![\psi]\!]^k_{\hat{\Pi}[\pi \leftarrow \hat{\tau}]}\Big) \\
[\![\Box \varphi]\!]^k_{\hat{\Pi}} &:= \bigwedge_{0 \leq i < k} \varphi[/\hat{\Pi}_i], \quad \text{where } \hat{\Pi}_i(x_\pi) := \mathrm{mem}(\hat{\Pi}(\pi)(i))(x),
\end{aligned}
$$

and where $\mathrm{FV}(\hat{\tau}) \subset V_*$ is the set of all free variables that occur in a symbolic trace $\hat{\tau}$, i.e., the set of free variables in $\mathrm{path}(\hat{\tau})$ and in the image of the symbolic memories in $\hat{\tau}$.

Importantly, for any specification ψ and bound k, the encoding $[\![\psi]\!]^k_\emptyset$ as given in Definition 15 is a closed first-order formula over $\mathcal{T}$ that is valid if and only if $\models^k \psi$.

Theorem 6 *(Symbolic encoding of $\models^k$) For any (closed) OHyperLTL$_{\text{safe}}$ formula ψ and any $k \in \mathbb{N}$, we have $\models^k \psi$ if and only if $\models_{\mathcal{T}} [\![\psi]\!]^k_\emptyset$.*

We delay the exposition of a detailed proof of this theorem to the next subsection.

Of course, because of this equivalence, $[\![\psi]\!]^k_\emptyset$ exhibits the same non-monotonicity as $\models^k \psi$, which we discussed in Section 3.3. Nevertheless, we can use the same symbolic encoding to achieve the desired monotonic semantics $\models^{\leq k} \psi$ (see Definition 11).

Corollary 7 *(Symbolic encoding of $\models^{\leq k}$) For any OHyperLTL$_{\text{safe}}$ formula ψ and any $k \in \mathbb{N}$, we have $\models^{\leq k} \psi$ if and only if $\models_{\mathcal{T}} [\![\psi]\!]^{k'}_\emptyset$ holds for all $k' \leq k$.*

Proof By Definition 11, $\models^{\leq k} \psi$ holds if and only if, for all $k' \leq k$, we have $\models^{k'} \psi$. By Theorem 6, this is the case if and only if we have $\models_{\mathcal{T}} [\![\psi]\!]^k_\emptyset$ for all $k' \leq k$. □

4.2 Correctness of the symbolic encoding

Theorem 6 from the previous section claims that, given an OHyperLTL$_{\text{safe}}$formula ψ and a bound $k \in \mathbb{N}$, the validity of the symbolic encoding $[\![\psi]\!]^k_\emptyset$ for some $k \in \mathbb{N}$ entails the validity of ψ for this bound k (i.e., $\models^k \psi$), and vice versa. This subsection is dedicated to a formal proof of this fact.

We start by introducing a *concretization relation* $\Pi \simeq_\rho \hat{\Pi}$ between mappings Π from trace variables to concrete traces, symbolic mappings $\hat{\Pi}$ from trace variables to symbolic traces, and concrete variable assignments ρ from logical variables to concrete values in $\text{Val}_{\mathcal{T}}$. This relation will be used to maintain a connection between concrete and symbolic mappings in proofs by induction on the syntactic structure of OHyperLTL$_{\text{safe}}$formulas. As in Definition 15, for a symbolic trace $\hat{\tau}$, we use $\text{FV}(\hat{\tau})$ to denote the set of all free variables that occur in $\hat{\tau}$, i.e., the set of free variables in $\mathtt{path}(\hat{\tau})$ and in the image of the symbolic memories in $\hat{\tau}$. We further lift $\text{FV}(\hat{\Pi})$ from symbolic traces to mappings of symbolic traces $\hat{\Pi}$ as

$$\text{FV}(\hat{\Pi}) \triangleq \biguplus_{\pi \in \text{dom}(\hat{\Pi})} \text{FV}(\hat{\Pi}(\pi)).$$

Definition 16 (Concretization relation $\simeq_\rho$) Let Π be a mapping from trace variables to concrete traces and let $\hat{\Pi}$ be a mapping from trace variables to symbolic traces. Further, let $\rho : \text{FV}(\hat{\Pi}) \to \text{Val}_{\mathcal{T}}$ be an assignment of the free variables occurring in the symbolic traces of $\hat{\Pi}$. We define

$$\Pi \simeq_\rho \hat{\Pi} \quad \iff \quad \text{dom}(\Pi) = \text{dom}(\hat{\Pi}) \ \wedge \ \forall \pi \in \text{dom}(\Pi).\ \Pi(\pi) = \gamma_\rho(\hat{\Pi}(\pi)).$$

In other words, $\Pi \simeq_\rho \hat{\Pi}$ holds if and only if concretizing all symbolic traces in $\hat{\Pi}$ via γ_ρ (see Definition 14) yields the concrete mapping Π. Importantly, we observe that $\simeq_\rho$ is preserved by extensions of trace mappings.

Lemma 8 *Let Π be a mapping from trace variables to concrete traces, $\hat{\Pi}$ be a mapping from trace variables to symbolic traces, $\hat{\tau}$ be a symbolic trace such that $\mathrm{FV}(\hat{\Pi}) \cap \mathrm{FV}(\hat{\tau}) = \emptyset$, and $\pi \in \mathcal{V}$ be a trace variable such that $\pi \notin \mathrm{dom}(\Pi) \cup \mathrm{dom}(\hat{\Pi})$. Further, let $\rho : \mathrm{FV}(\hat{\Pi}) \to \mathrm{Val}_\mathcal{T}$ and $\rho' : \mathrm{FV}(\hat{\tau}) \to \mathrm{Val}_\mathcal{T}$ be variable assignments for the free variables in traces of $\hat{\Pi}$ and for the free variables in $\hat{\tau}$, respectively. Then,*

$$\Pi \simeq_\rho \hat{\Pi} \quad \iff \quad \Pi[\pi \leftarrow \gamma_{\rho'}(\hat{\tau})] \simeq_{\rho \uplus \rho'} \hat{\Pi}[\pi \leftarrow \hat{\tau}].$$

Proof First, since $\pi \notin \mathrm{dom}(\Pi) \cup \mathrm{dom}(\hat{\Pi})$, we have

$$\mathrm{dom}(\Pi[\pi \leftarrow \gamma_{\rho'}(\hat{\tau})]) = \mathrm{dom}(\hat{\Pi}[\pi \leftarrow \hat{\tau}]) \iff \mathrm{dom}(\Pi) = \mathrm{dom}(\hat{\Pi}). \tag{1}$$

Further, since $\mathrm{FV}(\hat{\Pi}) \cap \mathrm{FV}(\hat{\tau}) = \emptyset$, we have

$$\gamma_{\rho \uplus \rho'}(\hat{\Pi}[\pi \leftarrow \hat{\tau}](\pi)) = \gamma_{\rho \uplus \rho'}(\hat{\tau}) = \gamma_{\rho'}(\hat{\tau}). \tag{2}$$

From Equation 4.2 and Equation 2 and by unfolding Equation 16 for $\simeq_{\rho \uplus \rho'}$, it immediately follows that $\Pi \simeq_\rho \hat{\Pi} \iff \Pi[\pi \leftarrow \gamma_{\rho'}(\hat{\tau})] \simeq_{\rho \uplus \rho'} \hat{\Pi}[\pi \leftarrow \hat{\tau}]$ □

As in Definition 8, for a concrete trace assignment Π, we use Π_i to denote $\Pi_i(x_\pi) := \mathrm{mem}(\Pi(\pi)(i))(x)$, i.e., the function that assigns to each program variable *x* labeled with a trace variable π the value of that program variable in the trace $\Pi(\pi)$ at the *i*-th observation. Similarly, as in Definition 15, for a symbolic trace assignment $\hat{\Pi}$, we use $\hat{\Pi}_i$ to denote $\hat{\Pi}_i(x_\pi) := \mathrm{mem}(\hat{\Pi}(\pi)(i))(x)$, i.e., a substitution function that substitutes each program variable *x* labeled with a trace variable π with a term over V_* that symbolically represents the value of that program variable in the symbolic trace $\hat{\Pi}(\pi)$ at the *i*-th observation.

We show that, if $\Pi \simeq_\rho \hat{\Pi}$, then the interpretation of a first-order formula under a concrete assignment Π_i relates to its interpretation under ρ as follows.

Lemma 9 *Let Π be a mapping from trace variables to concrete traces, $\hat{\Pi}$ be a mapping from trace variables to symbolic traces, and let $\rho : \mathrm{FV}(\hat{\Pi}) \to \mathrm{Val}_\mathcal{T}$ be a variable assignment such that $\Pi \simeq_\rho \hat{\Pi}$. Further, let $\varphi \in \mathrm{Form}_\mathcal{T}(X_\mathcal{V})$ be a quantifier-free first-order formula with free variables in $X_\mathcal{V}$. Then, for all $i \leq k$,*

$$\Pi_i \models_\mathcal{T} \varphi \iff \rho \models_\mathcal{T} \varphi[/\hat{\Pi}_i].$$

Proof Because $\Pi \simeq_\rho \hat{\Pi}$, we have $\Pi(\pi) = \gamma_\rho(\hat{\Pi}(\pi))$ for any trace variable $\pi \in \mathcal{V}$. Therefore, $\mathrm{mem}(\Pi(\pi)(i))(x) = \mathrm{mem}(\gamma_\rho(\hat{\Pi}(\pi))(i))(x) = [\![\mathrm{mem}(\hat{\Pi}(\pi)(i))(x)]\!]^\rho_\mathcal{T}$ for any trace variable π and any program variable *x*. Thus, we have $\Pi_i(x_\pi) = [\![\hat{\Pi}_i(x_\pi)]\!]^\rho_\mathcal{T}$ for any $x_\pi \in X_\mathcal{V}$.

For any free variable x_π that occurs in φ, we therefore have $[\![x_\pi]\!]^{\Pi_i}_\mathcal{T} = [\![x_\pi[/\hat{\Pi}_i]]\!]^\rho_\mathcal{T}$. Because φ is quantifier-free, for any term *t* that occurs in φ, we have $[\![t]\!]^{\Pi_i}_\mathcal{T} = [\![t[/\hat{\Pi}_i]]\!]^\rho_\mathcal{T}$.

Therefore, for any subformula φ' of φ, including φ itself, we have $\Pi_i \models_{\mathcal{T}} \varphi'$ if and only if $\rho \models_{\mathcal{T}} \varphi[/\hat{\Pi}]$. $\square$

With Lemma 8 and Theorem 9, we are now equipped to prove the correctness of our symbolic encoding for $\mathrm{OHyperLTL}_{\mathrm{safe}}$. We start by proving the following stronger theorem, that connects the interpretation of an $\mathrm{OHyperLTL}_{\mathrm{safe}}$formula ψ under a concrete mapping Π with the interpretation of its symbolic encoding $[\![\psi]\!]^k_{\hat{\Pi}}$ for some symbolic mapping $\hat{\Pi}$.

Theorem 10 *Let ψ be an $OHyperLTL_{\mathrm{safe}}$formula, let $k \in \mathbb{N}$, let Π be a partial map assigning trace variables to concrete traces, let $\hat{\Pi}$ be a partial map assigning trace variables to symbolic traces, and let ρ be an assignment of values from $\mathrm{Val}_{\mathcal{T}}$ to variables. If $\Pi \simeq_\rho \hat{\Pi}$ holds, then we have $\Pi \models^k \psi$ if and only if $\rho \models_{\mathcal{T}} [\![\psi]\!]^k_{\hat{\Pi}}$. Formally,*

$$\Pi \simeq_\rho \hat{\Pi} \implies (\Pi \models^k \psi \iff \rho \models_{\mathcal{T}} [\![\psi]\!]^k_{\hat{\Pi}}).$$

Proof We prove this theorem by induction over the structure of the formula ψ. The *induction hypothesis* is that the theorem holds for any $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula that is a strict suffix of ψ for all Π, $\hat{\Pi}$, and ρ for which $\Pi \simeq_\rho \hat{\Pi}$ holds.

Base case: The smallest syntactically correct $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula is $\psi = \square\varphi$ for any $\varphi \in \mathrm{Form}_{\mathcal{T}}(X_{\mathcal{V}})$. In this case, the following equivalence holds:

$$\begin{aligned}
\Pi \models^k \psi &\iff \forall i \in [0, k-1],\ \Pi_i \models_{\mathcal{T}} \varphi, && \text{(by Definition 11)}\\
&\iff \forall i \in [0, k-1],\ \rho \models_{\mathcal{T}} \varphi[/\hat{\Pi}_i] && \text{(by Theorem 2, since } \Pi \simeq_\rho \hat{\Pi})\\
&\iff \rho \models_{\mathcal{T}} \textstyle\bigwedge_{0 \le i < k} \varphi[/\hat{\Pi}_i] && \text{(semantics of } \wedge \text{ in } \mathcal{T})\\
&\iff \rho \models_{\mathcal{T}} [\![\psi]\!]^k_{\hat{\Pi}} && \text{(by Definition 15)}
\end{aligned}$$

Induction step: Otherwise, ψ must be of one of the following forms.

If $\psi = \forall^G \pi : O.\ \psi'$ for some program graph G, a trace variable π, a set of observed locations O, and an $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula ψ', then the following equivalence holds.

$$\begin{aligned}
\Pi \models^k \psi &\iff \forall \tau \in \mathrm{Traces}^k_O(G),\ \Pi[\pi \leftarrow \tau] \models^k \psi' && \text{(by Definition 8)}\\
&\iff \forall \tau \in \gamma(\mathrm{SymTraces}^k_O(G)),\ \Pi[\pi \leftarrow \tau] \models^k \psi' && \text{(by Corollary 5)}\\
&\iff \forall \hat{\tau} \in \mathrm{SymTraces}^k_O(G), \forall \rho_{\hat{\tau}}, && \text{(by Definition 14)}\\
&\qquad \rho_{\hat{\tau}} \models_{\mathcal{T}} \mathrm{path}(\hat{\tau}) \implies \Pi[\pi \leftarrow \gamma_{\rho_{\hat{\tau}}}(\hat{\tau})] \models^k \psi' &&\\
&\iff \forall \hat{\tau} \in \mathrm{SymTraces}^k_O(G), \forall \rho_{\hat{\tau}}, && \text{(by induction hypothesis, using Lemma 8}\\
&\qquad \rho_{\hat{\tau}} \models_{\mathcal{T}} \mathrm{path}(\hat{\tau}) \implies \rho \uplus \rho_{\hat{\tau}} \models_{\mathcal{T}} [\![\psi']\!]^k_{\hat{\Pi}[\pi \leftarrow \hat{\tau}]} && \text{to prove } \Pi[\pi \leftarrow \gamma_{\rho_{\hat{\tau}}}(\hat{\tau})] \simeq_{\rho \uplus \rho_{\hat{\tau}}} \hat{\Pi}[\pi \leftarrow \hat{\tau}])\\
&\iff \forall \hat{\tau} \in \mathrm{SymTraces}^k_O(G), && \text{(semantics in } \mathcal{T})\\
&\qquad \rho \models_{\mathcal{T}} \forall(\mathrm{FV}(\hat{\tau})), \left(\mathrm{path}(\hat{\tau}) \rightarrow [\![\psi']\!]^k_{\hat{\Pi}[\pi \leftarrow \hat{\tau}]}\right) &&\\
&\iff \rho \models_{\mathcal{T}} \textstyle\bigwedge_{\hat{\tau} \in \mathrm{SymTraces}^k_O(G)} && \text{(semantics in } \mathcal{T})\\
&\qquad \forall(\mathrm{FV}(\hat{\tau})), \left(\mathrm{path}(\hat{\tau}) \rightarrow [\![\psi']\!]^k_{\hat{\Pi}[\pi \leftarrow \hat{\tau}]}\right) &&\\
&\iff \rho \models_{\mathcal{T}} [\![\psi]\!]^k_{\hat{\Pi}} && \text{(by Definition 15)}
\end{aligned}$$

Otherwise, $\psi = \exists^G \pi : O.\ \psi'$ for some program graph G, a trace variable π, a set of observed locations O, and an $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula ψ', and the following equivalence holds.

$$
\begin{aligned}
\Pi \models^k \psi \iff & \exists \tau \in \mathrm{Traces}_O^k(G),\ \Pi[\pi \leftarrow \tau] \models^k \psi' && \text{(by Definition 8)}\\
\iff & \exists \tau \in \gamma(\mathrm{SymTraces}_O^k(G)),\ \Pi[\pi \leftarrow \tau] \models^k \psi' && \text{(by Corollary 5)}\\
\iff & \exists \hat{\tau} \in \mathrm{SymTraces}_O^k(G), \exists \rho_{\hat{\tau}}, && \text{(by Definition 14)}\\
& \quad \rho_{\hat{\tau}} \models_{\mathcal{T}} \mathrm{path}(\hat{\tau}) \wedge \Pi[\pi \leftarrow \gamma_{\rho_{\hat{\tau}}}(\hat{\tau})] \models^k \psi' \\
\iff & \exists \hat{\tau} \in \mathrm{SymTraces}_O^k(G), \exists \rho_{\hat{\tau}}, && \text{(induction hypothesis,}\\
& \quad \rho_{\hat{\tau}} \models_{\mathcal{T}} \mathrm{path}(\hat{\tau}) \wedge \rho \uplus \rho_{\hat{\tau}} \models_{\mathcal{T}} [\![\psi']\!]^k_{\hat{\Pi}[\pi \leftarrow \hat{\tau}]} && \Pi \simeq_\rho \hat{\Pi})\\
\iff & \exists \hat{\tau} \in \mathrm{SymTraces}_O^k(G), && \text{(semantics in } \mathcal{T})\\
& \quad \rho \models_{\mathcal{T}} \exists(\mathrm{FV}(\hat{\tau})), \left(\mathrm{path}(\hat{\tau}) \wedge [\![\psi']\!]^k_{\hat{\Pi}[\pi \leftarrow \hat{\tau}]} \right) \\
\iff & \rho \models_{\mathcal{T}} \bigvee_{\hat{\tau} \in \mathrm{SymTraces}_O^k(G)} && \text{(semantics in } \mathcal{T})\\
& \qquad \exists(\mathrm{FV}(\hat{\tau})), \left(\mathrm{path}(\hat{\tau}) \wedge [\![\psi']\!]^k_{\hat{\Pi}[\pi \leftarrow \hat{\tau}]} \right) \\
\iff & \rho \models_{\mathcal{T}} [\![\psi]\!]^k_{\hat{\Pi}} && \text{(by Definition 15)}
\end{aligned}
$$

These cases exhaustively cover any possible structure of ψ (see Definition 5). Hence, we have $\Pi \models^k \psi$ if and only if $\rho \models_{\mathcal{T}} [\![\psi]\!]^k_{\hat{\Pi}}$. □

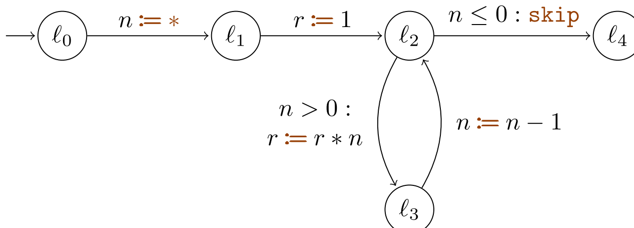
From the previous theorem, we immediately obtain the correctness of our symbolic encoding in the form of the previously presented Theorem 6:

Theorem 6 *(Symbolic encoding of $\models^k$) For any (closed) $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula ψ and any $k \in \mathbb{N}$, we have $\models^k \psi$ if and only if $\models_{\mathcal{T}} [\![\psi]\!]^k_{\emptyset}$.*

Proof This follows from Theorem 10 with $\Pi = \emptyset$, $\hat{\Pi} = \emptyset$, and $\rho = \emptyset$ because $\emptyset \simeq_\emptyset \emptyset$. □

4.3 Computing symbolic encodings

It is important to note that, while the set of symbolic traces of length exactly *k* is always finite, it is not the case for observed symbolic traces $\mathrm{SymTraces}_O^k(G)$. Thus, the symbolic encoding of an $\mathrm{OHyperLTL}_{\mathrm{safe}}$ specification cannot always be computed. For example, consider the following program computing the factorial of some arbitrary integer *n* and suppose we set an observation point at the end of the program (in location ℓ_4):



Clearly, there exist infinitely many symbolic traces with exactly one visit to ℓ_4, each corresponding to the execution of the program with a different input *n*. Formally,

$|\,\mathrm{SymTraces}^1_{\{\ell_2\}}(G)| = \infty$, and consequently, this set cannot be computed by symbolic execution. This issue is inherent to program analysis methods based on loop unwinding: the analysis of unbounded loops never terminates.

Definition 17 (Encodable specifications) Let ψ be an $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula and $k \in \mathrm{N}$. We say that ψ is *k-encodable* if, for any quantified program graph *G* annotated with a set of observed locations *O* in ψ, $\mathrm{SymTraces}^k_O(G)$ is finite.

Theorem 11 *(Finite encodings) Let ψ be a k-encodable specification. Then $[\![\psi]\!]^k_\emptyset$ is a well-defined (finite) first-order formula.*

Proof By Definition 5, ψ is of finite size. By Definition 15, the encoding $[\![\psi]\!]^k_\emptyset$ is well-defined and finite if, for any quantification pattern $Q^G\pi : O$ in ψ, we have that $\mathrm{SymTraces}^k_O(G)$ is finite. By assumption, and by Definition 17, this is the case. □

Requiring specifications to be encodable is a necessary condition to be able to effectively compute the symbolic encoding of a specification. However, it is not a sufficient criterion. Indeed, even if we restrict ourselves to the analysis of programs for which $\mathrm{SymTraces}^k_O(G)$ is finite, it does not necessarily mean that the function can be effectively computed. This can be shown via a reduction from the halting problem.

Theorem 12 *The function $\mathrm{SymTraces}^k_O(G)$ is not computable for arbitrary $k \in \mathbb{N}$, program graphs G, and locations O, even if G is restricted to program graphs for which $|\,\mathrm{SymTraces}^k_O(G)|$ is finite.*

Proof We show that no algorithm can exist that computes the value of the function $\mathrm{SymTraces}^k_O(G)$ as described above by contradiction. For the sake of contradiction, suppose that such an algorithm A exists that computes the set $\mathrm{SymTraces}^k_O(G)$ for arbitrary program graphs *G*, an arbitrary parameter $k \in N$, and an arbitrary set of observed locations *O*, with the restriction that $|\,\mathrm{SymTraces}^k_O(G)|$ shall be finite. We show that, using A, we can construct a procedure that solves the halting problem for any deterministic turing machine, which is impossible.

First, observe that any deterministic turing machine *M* can be encoded as an equivalent program graph *P*. We construct *P* to have a unique location ℓ_{halt} that has no outgoing edges, such that *P* transitions to ℓ_{halt} if and only if the original turing machine *M* halts in a final state. The input of the turing machine *M* is chosen by initializing the memory of *P* before emulating *M*.

Now let $k = 1$ and $O = \{\,\ell_{\mathrm{halt}}\,\}$. By construction of *P*, and because *M* is deterministic, $\mathrm{SymTraces}^k_O(P)$ contains exactly one trace if and only if *P* terminates for its preloaded input, otherwise, *M* diverges and $\mathrm{SymTraces}^k_O(P)$ is empty. Therefore, to decide whether *M* terminates on the fixed input, it suffices to call A to compute $\mathrm{SymTraces}^1_{\{\,\ell_{\mathrm{halt}}\,\}}(P)$ and to check whether its cardinality is zero or one.

Because the existence of a procedure that solves the halting problem for arbitrary (deterministic) turing machines is impossible, this concludes the proof by contradiction. □

In spite of Theorem 12, we demonstrate that a sound (but necessarily incomplete) algorithm exists that computes $\mathrm{SymTraces}^k_O(G)$ by symbolic execution of G. Further, we will show that this algorithm is actually complete for a large class of interesting programs we call finitely observable programs. Algorithm 1 consists of the two procedures $\textsc{Extend}(\hat{\tau})$ and $\textsc{Observe}$. The first procedure takes a symbolic trace $\hat{\tau}$ of length k and computes all feasible extensions of $\hat{\tau}$ of length $k+1$. To check the feasibility of symbolic traces, we assume the existence of a procedure $\mathrm{Sat}(\varphi)$ that (semi-) decides the satisfiability of an SMT formula φ over $\mathcal{T}$. The procedure $\textsc{Observe}$ uses $\textsc{Extend}$ to iteratively compute $\mathrm{SymTraces}^k_O(G)$. The traces are computed by starting from the initial symbolic state $\langle \ell_0, \mathsf{true}, \hat{m}_0 \rangle$, and repeatedly computing all its possible extensions, until the desired number of observation points is reached for all extensions.

Algorithm 1 Symbolic Interpreter

Require: program graph $G = (\langle \mathcal{L}, \mathcal{E} \rangle, \ell_0, \mathit{effect}, \mathit{guard})$
Require: set of observed locations $O \subseteq \mathcal{L}$
Require: number of observations $n \in \mathbb{N}$

```
procedure EXTEND(τ̂)
    T ← ∅
    ⟨ℓ, φ, m̂⟩ ← τ̂_{|τ̂|−1}
    for all (ℓ1, ℓ2) ∈ ℰ such that ℓ1 = ℓ do
        φ' ← φ ∧ guard(ℓ1, ℓ2)[/m̂]
        if SAT(φ') then
            switch effect(ℓ1, ℓ2) do
                case x := e
                    T ← T ∪ {τ̂ · ⟨ℓ2, φ', m̂[x ← e[/m̂]]⟩}
                case x := *
                    T ← T ∪ {τ̂ · ⟨ℓ2, φ', m̂[x ← fresh()]⟩}
                case skip
                    T ← T ∪ {τ̂ · ⟨ℓ2, φ', m̂⟩}
    return T

procedure OBSERVE
    T ← ∅
    todo ← {⟨ℓ0, true, m̂0⟩}
    while todo ≠ ∅ do
        pick τ̂ ∈ todo
        todo ← todo \ {τ̂}
        if |τ̂|_O = n then
            T ← T ∪ {[τ̂]_O}
        else
            todo ← todo ∪ EXTEND(τ̂)
    return T
```

By construction, Algorithm 1 satisfies the following property.

Theorem 13 *(Soundness of the symbolic interpreter) Let G be a program graph, O be a set of observed locations, and n a number of observations. Then, if* $\textsc{Observe}(G, O, n)$ *terminates, it returns* $\mathrm{SymTraces}^n_O(G)$.

Proof Given a symbolic trace $\hat{\tau}$, the procedure $\textsc{Extend}(\hat{\tau})$ computes all symbolic traces of length $|\hat{\tau}| + 1$ that are (immediate) extensions of $\hat{\tau}$. This set is necessarily finite. With that, clearly, any symbolic trace $\hat{\tau} \in \mathrm{SymTraces}^*(G)$ with $|\hat{\tau}|_O \leq n$ will at some point be added to todo by the procedure $\textsc{Observe}$ since any such trace is an extension of the initial symbolic state $\langle \ell_0, \mathsf{true}, \hat{m}_0 \rangle$. Consequently, for any symbolic trace $\hat{\tau} \in \mathrm{SymTraces}^*(G)$ with $|\hat{\tau}|_O = n$, the projection to observations $[\hat{\tau}]_O \in \mathrm{SymTraces}^n_O(G)$ will be added to T at some point. Therefore, when $\textsc{Observe}$ returns, the set T of symbolic traces projected to observations is precisely $\mathrm{SymTraces}^n_O(G)$. □

We can identify a class of *finitely observable* programs for which the symbolic encoding can always be computed. In practice, we found that the vast majority of benchmarks in the relevant literature fall into this class (see Section 5).

Definition 18 (Finite observability of programs) Let G be a program graph with program locations $\mathcal{L}$, let $k \in \mathbb{N}$, and let $O \subseteq \mathcal{L}$ be a set of observed locations. G is said to be (k, O)-observable if and only if there exists a bound $b \in \mathbb{N}$ such that the length of each trace $\tau \in \mathrm{Traces}^*(G)$ that contains strictly fewer than k observations is strictly less than b. Formally, G is (k, O)-observable if and only if

$$\exists b \in \mathbb{N},\ \forall \tau \in \mathrm{Traces}^*(G),\ |\tau|_O < k \implies |\tau| < b.$$

In other words, a program is (k, O)-observable if all of its traces either terminate or reach a k-th observation point within some bounded number of state transitions.

Finite observability is a monotonic property: if a program graph is not (k, O)-observable, it cannot be $(k + 1, O)$-observable. Consequently, given a program graph, if the length of all traces that contain at most k observations is bounded, then the program graph is (k, O)-observable. Note that the other direction generally does not hold, nor is it sufficient to consider only traces that contain precisely k observations.

Theorem 14 *(Relative completeness of the symbolic interpreter) Let G be a program graph, O a set of observed locations,* $n \in \mathbb{N}$ *a number of observations, and suppose that* $\mathcal{T}$ *is a decidable theory. If G is (n, O)-observable, then* $\textsc{Observe}(G, O, n)$ *terminates and returns* $\mathrm{SymTraces}^n_O(G)$.

Proof First, observe that $\textsc{Extend}(\hat{\tau})$ never diverges, assuming that $\mathcal{T}$ is a decidable theory, and always returns a finite set. Suppose that G is (n, O)-observable and let $b \in \mathbb{N}$ be the relevant bound as in Definition 18. By design, Algorithm 1 never extends any symbolic trace $\hat{\tau} \in \mathrm{SymTraces}^*(G)$ with $|\hat{\tau}|_O \geq n$. Combining Definition 18 and Theorem 4, we have that $|\hat{\tau}| < b$ for any symbolic trace $\hat{\tau} \in \mathrm{SymTraces}^*(G)$ with $|\hat{\tau}|_O < n$. Thus, any trace that is added to todo is at most of length b. Following Definition 12, which $\textsc{Extend}(\hat{\tau})$ implements, there is only a finite number of *symbolic* traces of length at most b. Lastly, observe that each such trace $\hat{\tau}$ is extended at most once. It follows that the procedure $\textsc{Observe}$ invokes $\textsc{Extend}(\hat{\tau})$ finitely many times and thus terminates, and the correctness of its return value follows from Theorem 13. □

The notion of finite observability is naturally extended to $\mathrm{OHyperLTL}_{\mathrm{safe}}$ specifications as follows:

Definition 19 (Finitely observable specifications) Let ψ be an $\mathrm{OHyperLTL}_{\mathrm{safe}}$ specification. We say that ψ is *k-observable* if, for each quantified program *G* annotated with observed locations *O* in ψ, *G* is (*k*, *O*)-observable.

Corollary 15 *(Computability of the encoding) Let ψ be an $OHyperLTL_{\mathrm{safe}}$ formula and let $k \in \mathbb{N}$ such that ψ is k-observable. Then, the symbolic encoding $[\![\psi]\!]^k_\emptyset$ can be effectively computed.*

Proof Due to the definition of $[\![\psi]\!]^k_{\hat{\Pi}}$ (see Definition 15), it suffices to show that, for each quantification pattern $Q^G\pi : O$ in ψ, the set $\mathrm{SymTraces}^k_O(G)$ is computable. By assumption, ψ is *k*-observable, thus, *G* is (*k*, *O*)-observable and, therefore, there exists a bound *b* such that each trace in $\mathrm{Traces}^k_O(G)$ can be obtained from a trace of length at most *b*. In turn, the set $\mathrm{SymTraces}^k_O(G)$ can be obtained from the set of symbolic traces of length at most *b*. The set of symbolic traces of length at most *b* is finite, and hence $\mathrm{SymTraces}^k_O(G)$ is computable. □

4.4 Automated bug-finding algorithms

The above definitions and theorems are sufficient for us to formulate a first algorithm that utilizes the symbolic encoding $[\![\psi]\!]^k_\emptyset$ to determine whether or not $\models^{\leq k} \psi$ holds. This approach is depicted in Algorithm 2.

Algorithm 2 Naive bug-finding algorithm

```
Require: OHyperLTL_safe formula ψ
Require: n ∈ ℕ

  procedure NAIVE
      for all k from 1 to n do
          if SAT(¬⟦ψ⟧^k_∅) then
              return "bug found"
```

The correctness of Algorithm 2 is formally stated as follows.

Theorem 16 *(Soundness of Algorithm 2) For any $OHyperLTL_{\mathrm{safe}}$ formula ψ and $n \in \mathbb{N}$, if $Naive(\psi, n)$ reports a specification violation, then $\not\models^{\leq n} \psi$.*

Proof If $\mathrm{Naive}(\psi, n)$ reports a specification violation, then, by design of Algorithm 2, there exists a $k \leq n$ such that $\neg[\![\psi]\!]^k_\emptyset$ is satisfiable over the theory $\mathcal{T}$. Because $[\![\psi]\!]^k_\emptyset$ does not contain any free variables, $\models_\mathcal{T} \neg[\![\psi]\!]^k_\emptyset$ holds in this case. Thus, $\not\models_\mathcal{T} [\![\psi]\!]^k_\emptyset$. Following Theorem 6, this implies $\not\models^k \psi$, and, by Definition 11, $\not\models^{\leq n} \psi$. □

If the specification ψ is not finitely observable, Algorithm 2 might fail to compute the symbolic encoding and diverge. If we require input specifications to be finitely observable, Algorithm 2 is complete with respect to bug finding. Note that this completeness result is relative to the decidability of the underlying first-order theory.

Theorem 17 *(Relative completeness of Algorithm 2) Let $n \in \mathbb{N}$ and let ψ be a n-observable specification, and suppose that $\mathcal{T}$ is a decidable theory. If $\not\models^{\leq n} \psi$, then $Naive(\psi, n)$ reports a violation.*

Proof Suppose $\not\models^{\leq n} \psi$. By Definition 11, there exists a (smallest) $k \leq n$ such that $\not\models^{k} \psi$. Because ψ is n-observable, by Corollary 15, $[\![\psi]\!]^k_\emptyset$ is computable. By Theorem 6, because $[\![\psi]\!]^k_\emptyset$ contains no free variables, $[\![\psi]\!]^k_\emptyset$ is unsatisfiable. Since $\mathcal{T}$ is decidable, the decision procedure determines $\neg[\![\psi]\!]^k_\emptyset$ to be satisfiable and Algorithm 2 reports an error during the k-th iteration of the algorithm's loop. □

Nevertheless, Algorithm 2 is unsatisfactory. Perhaps most importantly, our goal is not only to prove $\not\models^{\leq n} \psi$, but also to produce a witness of this violation. Because $[\![\psi]\!]^k_\emptyset$ is a closed formula, $\mathrm{Sat}(\neg[\![\psi]\!]^k_\emptyset)$ – and thus Algorithm 2 – does not produce any (non-empty) model. Additionally, the size of the symbolic encoding $\neg[\![\psi]\!]^k_\emptyset$ grows rapidly, and the search for relevant traces generally cannot be guided as the entire problem is now encoded as a single SMT query (for each value of k). Lastly, Algorithm 2 is not well-suited for parallelization, which is a desirable property for almost any computationally extensive algorithm due to the parallel architecture of modern processors.

While Algorithm 2 works for arbitrary $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formulas, we are particularly interested in the class of ∀∃ hyperproperties. Therefore, we now assume a specification of the form $\psi = \forall^{G_1}\pi_1 : O_1.\exists^{G_2}\pi_2 : O_2.\Box\varphi$. Given such a specification, we can now improve the search for a counterexample in multiple ways. First, symbolic traces of G_1 can be processed one by one. Instead of generating one big conjunctive query, we try to find a counterexample for every symbolic trace of G_1 independently. More precisely, for every symbolic trace in G_1, we try to prove that there exists no matching trace in G_2. This drastically reduces the size of SMT queries while also allowing more flexibility in the selection of symbolic paths of G_1 that are considered. This approach is depicted in Algorithm 3.

Algorithm 3 "Lazy" bug-finding algorithm with counterexamples

Require: $\mathrm{OHyperLTL}_{\mathsf{safe}}$ formula $\psi = \forall^{G_1}\pi_1 : O_1.\ \exists^{G_2}\pi_2 : O_2.\ \Box\varphi$
Require: $n \in \mathbb{N}$

procedure LAZY
  for all k from 1 to n **do**
    for all $\hat{\tau}_1 \in \mathrm{SymTraces}^k_{O_1}(G_1)$ **do**
      $C_1 \leftarrow \mathrm{path}(\hat{\tau}_1)$
      $C_2 \leftarrow \neg[\![\exists^{G_2}\pi_2 : O_2.\ \Box\varphi]\!]^k_{[\pi_1 \leftarrow \hat{\tau}_1]}$
      $C \leftarrow C_1 \land C_2$
      if $\mathrm{SAT}(C)$ **then**
        return $\langle \hat{\tau}_1, \mathrm{model}(C), C_2 \rangle$

Additionally, Algorithm 3 now permits recovery of a concrete trace τ_1 of G_1 such that no corresponding concrete trace τ_2 of G_2 can be found. By construction, the free variables in the query C are precisely the free variables in the universally quantified trace $\hat{\tau}_1$. In other words, $\rho := \mathrm{model}(C)$ can be used to construct a concrete trace $\tau_1 = \gamma_\rho(\hat{\tau}_1)$ for π_1 that witnesses the violation of the specification. The query C_2 encodes the impossibility of finding a matching trace in G_2 and acts as an *explanation* as to why τ_1 constitutes a counterexample of ψ.

The correctness of this algorithm follows from the described construction of ρ and τ_1 and Theorem 10.

Theorem 18 *(Soundness of Algorithm 3) For any bound $n \in \mathbb{N}$ and any OHyperLTL$_{\mathrm{safe}}$ formula $\psi = \forall^{G_1}\pi_1 : O_1.\, \exists^{G_2}\pi_2 : O_2.\, \Box\varphi$, if Lazy$(\psi, n)$ reports a specification violation, then $\not\models^{\leq n} \psi$.*

Proof If $\mathrm{Lazy}(\psi, n)$ reports a specification violation, then, by design of Algorithm 3, there exists a $k \leq n$ such that $\hat{\tau}_1 \in \mathrm{SymTraces}^k_{O_1}(G_1)$. Further, because $\mathrm{Sat}(C)$ holds in this case, there exists an assignment $\rho \in (\mathrm{Val}_\mathcal{T})^{\mathrm{FV}(\mathrm{path}(\hat{\tau}_1))}$ such that $\rho \models_\mathcal{T} C$, where $\mathrm{FV}(\mathrm{path}(\hat{\tau}_1))$ is the set of free variables in C. (Both C_1 and C_2 only contain free variables from the set $\mathrm{FV}(\mathrm{path}(\hat{\tau}_1))$.)

Let $\tau_1 = \gamma_\rho(\hat{\tau}_1)$. Because $\hat{\tau}_1 \in \mathrm{SymTraces}^k(G_1)_{O_1}$ and $\rho \models_\mathcal{T} \mathrm{path}(\hat{\tau}_1)$, according to Corollary 5, we have $\tau_1 \in \mathrm{Traces}^k_{O_1}(G_1)$.

Let $\Pi := [\pi_1 \mapsto \tau_1]$ and $\hat{\Pi} := [\pi_1 \mapsto \hat{\tau}_1]$. By construction, we have $\Pi \simeq_\rho \hat{\Pi}$ (see Definition 16), i.e., we have $\mathrm{dom}(\Pi) = \mathrm{dom}(\hat{\Pi}) = \{\pi_1\}$ and $\Pi(\pi_1) = \gamma_\rho(\hat{\Pi}(\pi))$. Because $\rho \models_\mathcal{T} C$ and thus $\rho \models_\mathcal{T} C_2$ holds, $\rho \models_\mathcal{T} \neg[\![\exists^{G_2}\pi_2.\, \Box\varphi]\!]^k_{\hat{\Pi}}$ holds, therefore, we have $\rho \not\models_\mathcal{T} [\![\exists^{G_2}\pi_2 : O_2.\, \Box\varphi]\!]^k_{\hat{\Pi}}$. As per Theorem 10, because $\Pi \simeq_\rho \hat{\Pi}$, this implies $\Pi \not\models^k \exists^{G_2}\pi_2 : O_2.\, \Box\varphi$. Combining these, we have

$$\begin{aligned}
&\exists\tau_1 \in \mathrm{Traces}^k_{O_1}(G_1).\ [\pi_1 \mapsto \tau_1] \not\models^k \exists^{G_2}\pi_2 : O_2.\Box\varphi \\
\Longrightarrow\ &\neg\big(\forall\tau_1 \in \mathrm{Traces}^k_{O_1}(G_1).\ [\pi_1 \mapsto \tau_1] \not\models^k \exists^{G_2}\pi_2 : O_2.\, \Box\varphi\big) \\
\Longrightarrow\ &\not\models^k \forall^{G_1}\pi_1 : O_1.\, \exists^{G_2}\pi_2 : O_2.\, \Box\varphi \\
\Longrightarrow\ &\not\models^k \psi \\
\Longrightarrow\ &\not\models^{\leq n} \psi.
\end{aligned}$$

□

The relative completeness result that we obtained for Algorithm 2 in Theorem 17 can also be attained for Algorithm 3.

Theorem 19 *(Relative completeness of Algorithm 3) Let ψ be an n-observable specification of the form $\psi = \forall^{G_1}\pi_1 : O_1.\, \exists^{G_2}\pi_2 : O_2.\, \Box\varphi$, and suppose that $\mathcal{T}$ is a decidable theory. If $\not\models^{\leq n} \psi$, then Lazy$(\psi, n)$ reports a violation.*

Proof Suppose $\not\models^{\leq n} \psi$. By Definition 11, there exists a (smallest) $k \leq n$ such that $\not\models^{k} \psi$. By Definition 8, there is a $\tau_1 \in \mathrm{Traces}^{k}_{O_1}(G_1)$ such that $[\pi_1 \leftarrow \tau_1] \not\models^{k} \psi$.

Let $\hat{\tau}_1 \in \mathrm{SymTraces}^{k}_{O_1}(G_1)$ be the symbolic trace whose concretization is τ_1, whose existence is guaranteed by Corollary 5. Because ψ is *n*-observable, the inner loop of Algorithm 3 will locate this particular $\hat{\tau}_1$ while iterating over $\mathrm{SymTraces}^{k}_{O_1}(G_1)$.

We know that C_1 is satisfiable, as witnessed by τ_1. We further know that, having fixed such a τ_1, the formula $[\![\exists^{G_2}\pi_2 : O_2.\, \Box\varphi]\!]^{k}_{[\pi_1 \leftarrow \hat{\tau}_1]}$ does not hold for all concretizations of $\hat{\tau}_1$. Thus, $C_2 = \neg[\![\exists^{G_2}\pi_2 : O_2.\, \Box\varphi]\!]^{k}_{[\pi_1 \leftarrow \hat{\tau}_1]}$, which is computable because ψ is *n*-observable, is satisfiable for some concretization of $\hat{\tau}_1$. The fact that the set of models for C_1 and C_2 intersect is witnessed by τ_1, which satisfies $\mathrm{path}(\hat{\tau}_1)$ and, by Theorem 10, it refutes $[\![\exists^{G_2}\pi_2 : O_2.\, \Box\varphi]\!]^{k}_{[\pi_1 \leftarrow \hat{\tau}_1]}$ and thus satisfies C_2. Therefore, $C_1 \wedge C_2$ is satisfiable and Algorithm 3 reports an error during the *k*-th iteration of the outer loop. □

Either algorithm can be used with an upper bound of ∞ on *k* instead of some input or constant *n*. If ψ is *k*-observable for all *k*, and if $\mathcal{T}$ is decidable, the respective algorithm will terminate only if $\not\models^{\leq k} \psi$ for some $k \in \mathbb{N}$, and will generally diverge otherwise. (However, as we will discuss below, Algorithm 3 might still terminate in other cases.) Note that this behavior is not unique to our algorithm or implementation: even for trace properties, various tools for finding specification violations do not terminate in infinite-state settings unless they find a bug, including popular tools for symbolic execution [22, 46, 59] and fuzzing [52].

With this in mind, Algorithm 3 can even be applied to some instances for which $\mathrm{SymTraces}^{k}_{O_1}(G_1)$ is infinite, since traces are processed one by one. Indeed, a counterexample might still be found by looking only at a few of the infinitely many traces in $\mathrm{SymTraces}^{k}_{O_1}(G_1)$. The likelihood of locating any existing specification violation within a finite amount of time can further be affected by adjusting the order in which one iterates over $\mathrm{SymTraces}^{k}_{O_1}(G_1)$. Search heuristics are commonly used by symbolic execution engines that need to explore a large or potentially infinite number of distinct symbolic paths for such scheduling [21, 22].

Many input programs might terminate after some finite number *n* of observations, in which case the set of symbolic traces $\mathrm{SymTraces}^{n+1}_{O_1}(G_1)$ will eventually be empty (and computable). For example, all ORHLE benchmarks [35] only reason about the input-output behavior of programs and each execution trace thus contains a single observation only. If such programs are additionally *n*-observable, Algorithm 3 can fully explore all possible behaviors of the input programs and thus, thanks to Theorem 19, rule out the existence of counterexamples entirely, effectively verifying that the $\mathrm{OHyperLTL}_{\mathrm{safe}}$ specification holds for arbitrary bounds.

Corollary 20 *Let $\psi = \forall^{G_1}\pi_1 : O_1.\, \exists^{G_2}\pi_2 : O_2.\, \Box\varphi$ be an $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula and let $n \in \mathbb{N}$ such that ψ is n-observable and such that Algorithm 3 with parameters ψ and n terminates without returning a counterexample. If $\mathrm{SymTraces}^{n+1}_{O_1}(G_1) = \emptyset$, then $\models^{\leq n'} \psi$ holds for all $n' \in \mathbb{N}$.*

Proof By Theorem 19, $\models^{\leq n} \psi$ holds. If $n' \leq n$, then $\models^{\leq n'} \psi$ holds because $\models^{\leq n} \psi \implies \models^{\leq n'} \psi$ (see Definition 11). If $n' > n$, then $\mathrm{SymTraces}^k_{O_1}(G_1)$ is necessarily empty for $n < k \leq n'$ and, by Corollary 5, so is $\mathrm{Traces}^k_{O_1}(G_1)$. By Definition 8, this implies that $\models^k \psi$ holds for $n < k \leq n'$. Because $\models^{\leq n} \psi$ holds, $\models^k \psi$ also holds for $k < n$. Combined, $\models^k \psi$ holds for $0 \leq k \leq n'$ and thus, by Definition 11, $\models^{\leq n'} \psi$ holds.

□

Lastly, Algorithm 3 – and its inner loop, in particular – is well-suited for parallelization: aside from sharing data as part of further optimizations, the iterations of the inner loop are entirely independent of one another and thus can be executed concurrently. Parallelization has been shown to be a significant improvement for techniques based on symbolic execution [20, 62] by reducing the time required to explore a large number of symbolic paths, at the cost of increasing memory requirements.

4.5 Generalization

Algorithm 3 assumes that the input formula ψ is of the form $\forall^{G_1}\pi_1 : O_1.\ \exists^{G_2}\pi_2 : O_2.\ \Box\varphi$, which permits the specification of important hyperproperties such as refinement. Nevertheless, prominent security properties, such as generalized non-interference [53] and delimited information release [58], fall in the class of $\forall^+\exists^+$ hyperproperties, where sequences of one or more of the same quantifier are allowed. Our approach, and Algorithm 3 specifically, can be generalized to such hyperproperties of this form in a straightforward manner.

Given an $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formula with a quantifier prefix of the form $\forall^{G^\forall_1} \ldots \forall^{G^\forall_m} \exists^{G^\exists_1} \ldots \exists^{G^\exists_n}.$, we could use either self-composition [4] or a product program construction [5] to construct two programs $G^\forall = G^\forall_1 \otimes \ldots \otimes G^\forall_m$ and $G^\exists = G^\exists_1 \otimes \ldots \otimes G^\exists_n$ such that $\mathrm{Traces}^*(G^\forall) \simeq \mathrm{Traces}^*(G^\forall_1) \times \ldots \times \mathrm{Traces}^*(G^\forall_m)$ and $\mathrm{Traces}^*(G^\exists) \simeq \mathrm{Traces}^*(G^\exists_1) \times \ldots \times \mathrm{Traces}^*(G^\exists_n)$. We could then replace the quantifier prefix with $\forall^{G^\forall}\pi_1.\ \exists^{G^\exists}\pi_2$ and rename the variables in the body of the formula accordingly to obtain an equivalent $\mathrm{OHyperLTL}_{\mathrm{safe}}$ specification with only one universal and one existential trace quantifier.

Such product constructions are widely used for the formal verification of *k*-safety hyperproperties [4, 39, 60, 64]. A straightforward way to generate the product of *n* programs is to execute the *n* programs in a lock-step fashion. However, lock-step products often complicate the verification of hyperproperties as they require to find intricate relational invariants [39, 60]. In turn, one of the main challenges in the verification of hyperproperties based on product constructions is to find an appropriate interleaving of the original programs such that the verification is easy, e.g., such that there exists a simple invariance argument [39, 60, 61, 63]. In the context of this work, we are not concerned with formal verification but rather with automated bug-finding. Therefore, finding interleavings leading to easier invariants it not a concern. Instead, simply enumerating all symbolic traces of each universally quantified program execution independently suffices. We present a generalized version of Algorithm 3 for $\mathrm{OHyperLTL}_{\mathrm{safe}}$ formulas with quantifier prefix $\forall^+\exists^+$ in Algorithm 4 below.

Algorithm 4 "Lazy" bug-finding algorithm generalized to $\forall^+\exists^+$ formulas

---

Require: formula $\psi = \forall^{G_1^\forall}\pi_1^\forall : O_1^\forall.\ \dots\forall^{G_p^\forall}\pi_p^\forall : O_p^\forall.\ \exists^{G_1^\exists}\pi_1^\exists : O_1^\exists.\ \dots\exists^{G_q^\exists}\pi_q^\exists.\ \Box\varphi$
Require: $n \in \mathbb{N}$

```
procedure LAZY
    for all k from 1 to n do
        for all ⟨τ̂_1^∀, ..., τ̂_p^∀⟩ ∈ SymTraces^k_{O_1^∀}(G_1^∀) × ... × SymTraces^k_{O_p^∀}(G_p^∀) do
            C_1 ← ⋀_{i=1}^p path(τ̂_i^∀)
            C_2 ← ¬⟦∃^{G_1^∃} π_1^∃ : O_1^∃. ... ∃^{G_q^∃} π_q^∃. □φ⟧^k_{[π_1^∀←τ̂_1^∀, ..., π_p^∀←τ̂_p^∀]}
            C ← C_1 ∧ C_2
            if SAT(C) then
                return ⟨τ̂_1^∀, ..., τ̂_p^∀, model(C), C_2⟩
```

Theorem 21 *(Soundness of Algorithm 4) For any bound $n \in \mathbb{N}$ and any OHyperLTL$_{\text{safe}}$ formula $\psi = \forall^{G_1^\forall}\pi_1^\forall : O_1^\forall.\ \dots\forall^{G_p^\forall}\pi_p^\forall : O_p^\forall.\ \exists^{G_1^\exists}\pi_1^\exists : O_1^\exists.\ \dots\exists^{G_q^\exists}\pi_q^\exists.\ \Box\varphi$, if Lazy$(\psi, n)$ reports a specification violation, then $\not\models^{\leq n} \psi$.*

Proof Let C, C_1, and C_2 be defined as in the inner loop of Algorithm 4. For any model ρ of C, we have $\rho \models C_1$ and hence $\rho \models \bigwedge_{i=1}^{p} \text{path}(\hat{\tau}_i^\forall)$. It follows that $\rho \models \text{path}(\hat{\tau}_i^\forall)$ for all $i \in [1, p]$. The rest of the proof follows the same structure as the proof of Theorem 18. □

5 Implementation and evaluation

To evaluate the effectiveness of our approach, we implemented the lazy ∀∃ bug-finding algorithm (Algorithm 3) in a prototype tool. Our implementation accepts structured programs expressed in a simple imperative programming language with loops, conditionals, and both deterministic and nondeterministic assignments, as well as an |observe| instruction to explicitly annotate programs with observation points. An input file for our tool is composed of a list of program definitions and an OHyperLTL$_{\text{safe}}$ formula with a prefix of the form $\forall^*\exists^*$ quantifying over the traces of the defined programs. Input programs are automatically translated into equivalent program graphs (as established in Definition 1).

If the specification contains more than one universal or existential quantifier, we use the asynchronous product construction that we described in Section 4.5 and Section?? to obtain an equivalent $\forall^1\exists^1$ specification.

Symbolic traces of program graphs are then computed by a custom symbolic execution engine. To check the feasibility of path conditions (see Definition 12) and the satisfiability of the symbolic encodings (see Definition 15), which are formulated in many-sorted first-order logic [44], we use version 2.6.4 of the Yices SMT solver [38].

We evaluated our prototype against a diverse set of examples, many of which we drew from related work [35]. These examples include several instances of refinement, generalized non-interference [53], and delimited information release [58], as well as various other problems. The main questions that we wish to answer with this experimental evaluation are the following:

(Q1) **Effectiveness:** Can our approach effectively find counterexamples to relevant ∀∃ hyperproperties without any expert assistance?
(Q2) **Efficiency:** Is our approach able to report counterexamples within a reasonable amount of time?
(Q3) **Scalability:** How does our approach perform when the number of paths to check and/or number of observation points grow?

To answer these questions, we evaluated our tool against the benchmarks of ORHLE [35], a state-of-the-art deductive verifier for ∀∃ hyperproperties of sequential programs. We additionally developed a new set of benchmarks specifically designed to stress-test our tool and measure its ability to scale when the number of program paths to explore grows. These new benchmarks consist of reactive programs and require to unwind several observation points in order to detect a hyperproperty violation. This type of analysis falls out of the scope of ORHLE.

While existing tools generally do not support the fully automatic refutation of ∀∃ properties of infinite-state systems, we nevertheless are interested in performance comparisons with such existing tools. Therefore, we compared the performance of our tool against ORHLE [35], HyperQB [47], AutoHyper [13], and HyHorn [49]. HyperQB is a bounded model-checker for HyperLTL. It targets finite-state reactive systems and reduces the refutation of hyperproperties to the satisfiability of quantified boolean formulas (QBF) [47]. AutoHyper is an explicit-state model checker that also targets finite-state systems but its *complete* model checking procedure is based on automaton complementation and language inclusion checking [13]. HyHorn is a very recent automated verifier for ∀∃ hyperproperties [49]. It targets the verification of infinite-state reactive systems against hyperproperties expressed in a logic similar to $\mathrm{OHyperLTL}_{\mathrm{safe}}$. Importantly, like ORHLE [35], HyHorn is not fully automatic and it typically requires a user-supplied predicate abstraction [50] to be able to prove certain instances. However, it also features a fully automatic “concrete” mode that does not require any guidance and can also detect violations in some cases. While HyperQB, AutoHyper, and HyHorn tackle slightly different problems than our approach, we believe that these three tools were the most relevant comparison points at the time of writing.

The experimental evaluation was conducted inside Docker containers on Linux 5.10.0, on a system equipped with 1024 GiB of memory and two Intel Xeon Silver 4314 processors, each of which has 32 logical CPU cores. Note that our prototype implementation is entirely single-threaded and thus, unlike other tools, does not benefit from the large number of available CPU cores.

5.1 Comparison with ORHLE (refutation)

As an initial evaluation of our tool, we reused benchmarks from the tool ORHLE [35]. It contains a total of 35 programs and associated hyperproperties including generalized non-interference [53] and refinement, which are expressed as relational pre and post-conditions. Of these, 30 can be expressed in our input programming language (the remaining five use programming constructs that our prototype does not support). Among the 30 remaining programs, 15 contain hyperproperty violations (the others satisfy their specifications). We translated these 15 example programs into our input programming language, and expressed

their associated specifications in $\mathrm{OHyperLTL}_{\mathrm{safe}}$. Importantly, since these programs are sequential, there is only one observation point at the end of each program, which corresponds to the point at which the post-condition needs to be verified.

The results of running our tool on these 15 benchmarks are reported in Table 1. The table recalls the type of hyperproperty being tested, the name of the program, whether the program is finitely observable or not (**FO**), and the number of distinct combinations of symbolic paths that our algorithm had to check in the process (**PC**). The runtime for each example program is expressed in seconds, and corresponds to the median runtime over 100 executions of our tool and ORHLE.

As shown in Table 1, our algorithm produces a counterexample for 14 out of the 15 examples, in much less than a second, and without any expert intervention. In particular, contrary to ORHLE [35], our tool does not require loops to be annotated with invariants.

Our tool was not able to find the violation for one of the 15 benchmarks (`loop-nonrefinement`). This program contains a potentially infinite loop and as such can diverge without ever reaching its observation point, and thus falls out of the class of programs supported by our approach. In particular, it is not finitely observable.

Even though Algorithm 3 would naturally benefit from parallelization, our prototype implementation does not utilize this. ORHLE, on the other hand, uses multiple CPU cores and benefits from the available hardware parallelism. Nevertheless, with the exception of `loop-nonrefinement`, our tool finds specification violations significantly faster than ORHLE in these benchmarks.

5.2 Comparison with ORHLE (verification)

In the previous section, we demonstrated that our prototype tool is able to locate the specification violation in almost all of the ORHLE benchmarks [35] that contain bugs. In this section, we experimentally investigate the outcome of applying our approach to those ORHLE benchmarks that do not contain bugs. As noted in Section 4.4, while our approach is designed specifically to produce counterexamples and may not terminate in the absence of bugs, it can effectively verify $\mathrm{OHyperLTL}_{\mathrm{safe}}$ specifications if it can rule out the existence of counterexamples entirely (see Corollary 20). Even though this might not be widely applicable in general, it is the case for many of the ORHLE benchmarks [35], hence, we compare the runtime of our tool with the runtime of ORHLE. The results are summarized in Table 2.

For fourteen of the fifteen benchmarks, our tool concludes that no counterexample can exist, i.e., that the hyperproperty holds, and it does so faster than ORHLE verifies these instances [35]. It does so by searching for counterexamples along every possible execution path, and for these benchmarks, the number of feasible execution paths is finite. The remaining instance (`loop-refinement`) is again not finitely observable because the input program does not necessarily terminate, and is verified by ORHLE with the help of a user-specified loop invariant.

While our approach is neither meant for nor suitable for verification in general, it is promising that our prototype implementation analyzes these benchmarks faster than a dedicated verification tool and without any expert guidance.

5.3 Custom benchmarks

In the previous set of benchmarks, the number of combination of paths that need to be checked is rather small (at most 64). Further, because of the sequential nature of the programs being analyzed, the hyperproperties are checked only on the final state of each quantified program. Thus, these benchmarks are not exploiting the full potential of $\mathrm{OHyperLTL}_{\mathrm{safe}}$. To further demonstrate the effectiveness of our algorithm, both for an increasing number of observation points, and for large numbers of symbolic paths, we designed a second set of benchmarks. For this set of benchmarks, counterexamples exist but require to check relation at multiple observation points, or to check thousands of combinations of paths. The runtime of our implementation for these benchmarks is depicted in Table 3.

To highlight important aspects of this set of benchmarks, we briefly discuss the benchmark `escalating`, which is one of the fifteen instances. It consists of the two programs Escalating and Limit and a ∀∃ specification that universally quantifies over traces of the first program and existentially quantifies over traces of the second program. The programs are synchronized at the beginning of their respective loops, i.e., their state is observed whenever they enter a loop iteration.

The hyperproperty does not hold because, eventually, the value of y in the left program will exceed the value of max in the right program. To arrive at this conclusion, the algorithm needs to explore an increasing number of iterations and thus has to consider an exponential number of paths in the right program (Limit). The left program assigns a nondeterministically chosen value to s and thus also to x in each iteration, and the choice of the assigned value affects what branch will be taken in the next loop iteration. Therefore, in the left program, the number of feasible paths also grows exponentially in the number of loop iterations. As shown in Table 3, our approach finds the violation after exploring seven iterations of the loop and hundreds of different path combinations within about 0.3 seconds.

Table 1 Runtime of counterexample detection in benchmarks derived from [35]

Class	Type	Program	FO	Bug found	PC	Our tool	ORHLE
∀∃	Refinement	`simple-nonrefinement`	Yes	✓	1	0.004 s	0.228 s
∀∃	Other	`do-nothing`	Yes	✓	1	0.004 s	0.230 s
∀∃	Other	`draw-once`	Yes	✓	1	0.004 s	0.226 s
∀∀∃	GNI	`simple-leak`	Yes	✓	1	0.004 s	0.226 s
∀∀∃	GNI	`nondet-leak2`	Yes	✓	2	0.004 s	0.248 s
∀∀∃	GNI	`smith1`	Yes	✓	2	0.009 s	0.229 s
∀∀∃	Delimited Release	`parity-no-dr`	Yes	✓	2	0.009 s	0.229 s
∀∀∃	GNI	`nondet-leak`	Yes	✓	2	0.010 s	0.237 s
∀∀∃	Delimited Release	`wallet-no-dr`	Yes	✓	2	0.009 s	0.238 s
∀∃	Refinement	`conditional-nonrefinement`	Yes	✓	4	0.015 s	0.237 s
∀∃	Refinement	`add3-shuffled`	Yes	✓	6	0.024 s	0.274 s
∀∀∀∃	Delimited Release	`conditional-no-dr`	Yes	✓	8	0.036 s	0.239 s
∀∀∃	Delimited Release	`median-no-dr`	Yes	✓	4	0.042 s	0.509 s
∀∀∀∃	Delimited Release	`conditional-leak`	Yes	✓	48	0.186 s	0.241 s
∀∃	Refinement	`loop-nonrefinement`	No	✗	N/A	∞	0.263 s

5.4 Case study: escalating

In Section 5.3, we briefly explained the benchmark `escalating`, which we designed to have a number of paths that grows exponentially with the exploration depth, as well as a large state space. We further demonstrate the challenges posed by this kind of input program by parameterizing the benchmark. By varying the initial value of the program variable max in the existentially quantified program Limit, we can effectively adjust the difficulty of the benchmark. Smaller initial values lead to a violation of the hyperproperty within a few loop iterations, whereas larger initial values drastically increase both the number of loop iterations after which the violation occurs as well as the size of the state space (see Table 4).

At the time of writing, there is no particularly common or standardized input format for tools that verify or refute ∀∃ hyperproperties. The vast differences across tools' capabilities and input formats make it difficult to use the same benchmarks across tools. In spite of this, we encoded the same parameterized benchmark `escalating` not only for our own prototype tool, which consumes programs written in an imperative programming language, but also for HyperQB [47] and AutoHyper [13], which consume NuSMV files, and for HyHorn [49], which uses a custom input format to model infinite-state transition systems. This allows us to measure the runtime of all four tools on the various instances of this parameterized benchmark.

For HyperQB, we explicitly use the minimum unwinding depth that is necessary to discover the specification violation. Unlike our approach and HyHorn, HyperQB and AutoHyper only operate on finite-state systems, which is why a general comparison is difficult. However, as shown in Table 4, the program's state space up until the point of the specification violation is finite, and hence, we can encode every instance as a finite-state NuSMV file for HyperQB and AutoHyper.

For our own tool, we let it discover the appropriate depth automatically, which is its default strategy and which is compatible with the semantics of $\models^{\leq k}$ (see Definition 11). As our tool, HyperQB, and AutoHyper are fully automatic tools, we also ran HyHorn in fully automated mode (i.e., without a user-supplied predicate abstraction). We note that even in

Table 2 Runtime of pseudo-verification in benchmarks derived from [35]

Class	Type	Program	FO	Verified	PC	Our tool	ORHLE
∀∀∃	GNI	`simple-nonleak`	Yes	✓	1	0.004 s	0.229 s
∀∀∃	GNI	`nondet-nonleak`	Yes	✓	1	0.007 s	0.237 s
∀∃	Refinement	`conditional-refinement`	Yes	✓	4	0.015 s	0.237 s
∀∃	Refinement	`simple-refinement`	Yes	✓	3	0.015 s	0.227 s
∀∀∃	Delimited Release	`wallet`	Yes	✓	4	0.016 s	0.237 s
∀∀∃	Delimited Release	`parity`	Yes	✓	4	0.017 s	0.236 s
∀∃	Refinement	`perm-inv-refinement`	Yes	✓	4	0.019 s	0.256 s
∃∃	Other	`httpRequest`	Yes	✓	3	0.022 s	0.259 s
∀∀∃	GNI	`nondet-nonleak2`	Yes	✓	16	0.026 s	0.255 s
∀∃	Refinement	`add3-sorted`	Yes	✓	48	0.041 s	0.275 s
∀∃	Other	`draw-until-21`	Yes	✓	19	0.087 s	0.279 s
∃∃	Other	`sleepAndContinue`	Yes	✓	4	0.160 s	0.335 s
∀∀∃	Delimited Release	`median`	Yes	✓	16	0.163 s	0.450 s
∀∀∀∃	Delimited Release	`conditional`	Yes	✓	64	0.245 s	0.246 s
∀∃	Refinement	`loop-refinement`	No	✗	N/A	∞	0.267 s

Table 3 Runtime of counterexample detection for test instances. All input programs are observable, and none of the specifications are valid

Class	Program	Bug found	Depth	Path Combinations	Runtime
∀∃	`even_odd`	✓	1	1	0.002 s
∀∃	`factor2`	✓	2	2	0.004 s
∀∃	`for_loop_simple`	✓	1	2	0.026 s
∀∃	`linear_equation`	✓	22	22	0.060 s
∀∃	`monotonic_increase`	✓	7	7	0.070 s
∀∃	`escalating_inl`	✓	7	40	0.227 s
∀∃	`escalating_2`	✓	7	747	0.257 s
∀∀∃	`secret_pin_leak`	✓	8	11	0.262 s
∀∃	`escalating`	✓	7	1195	0.346 s
∀∃	`escalating_3`	✓	7	1707	0.442 s
∀∀	`obs_determinism`	✓	4	86	1.065 s
∀∃	`no_primes_above_31397`	✓	1	201	2.463 s
∀∀∃	`secret_pin_leak_2`	✓	3	248	5.420 s
∀∃	`exponential_branching_1`	✓	1	1024	5.705 s
∀∃	`exponential_branching_2`	✓	1	2048	10.745 s

Program Escalating:
$x \leftarrow 0$
$y \leftarrow 0$
loop
  if $x \equiv 0 \pmod 2$ **then**
    $y \leftarrow y + 1$
  else
    $y \leftarrow y + x$
  havoc $s \in \{1, 2\}$
  $x \leftarrow x + s$

Program Limit:
$max \leftarrow 15$
loop
  either
    $max \leftarrow max + 1$
  or
    skip

Specification

$$\forall^{\textsc{Escalating}} \pi_1 : \{\ell_{\textbf{loop}}\}.\ \exists^{\textsc{Limit}} \pi_2 : \{\ell_{\textbf{loop}}\}.\ \Box(y_{\pi_1} \leq max_{\pi_2})$$

Fig. 6 The two programs and the specification that are the benchmark `escalating`

this abstraction-less mode, HyHorn reportedly typically outperforms other tools on verification benchmarks [49]. We execute AutoHyper [13] both with its default strategy and with its optional FORQ-based inclusion checker [37].

The runtimes of the four tools on the 56 instances of this parameterized benchmark are depicted in Fig. 7. For all four tools, despite being based on very different automated analysis techniques, we observe that the respective runtimes increase significantly as the initial value of max grows. Nevertheless, our approach still far outperforms HyperQB, AutoHyper, and HyHorn for all tested parameters. For large parameters, all other tools either get close to or exceed the 30 minutes timeout that we have set, whereas our tool finds the specification violation in less than 30 seconds.

We remark that we were unable to run AutoHyper on some instances of the benchmark because its memory requirements exceeded 1024 GiB for large parameters.

5.5 Summary

The outcomes of this evaluation are promising. First, the results obtained on all benchmarks show that our approach is able to find concrete counterexamples to relevant ∀∃ hyperproperties, with convincing execution times: counterexamples are often found within fractions of a second. Thus, we can positively answer questions Q1 and Q2: our approach is an effective and efficient way to detect ∀∃ hyperproperty violations, fully automatically.

Regarding question Q3, our benchmarks contain examples that require to check whether a relational invariant holds across many observation points, as well as examples that require checking thousands of combinations of paths. In Section 5.4, we also observed the rapidly increasing difficulty of a parameterized benchmark. Importantly, for the more challenging benchmark instances, other tools are either significantly slower than our approach, or they simply fail to detect violations without expert guidance. The fact that our prototype resists these stress tests with runtimes not exceeding a few seconds is encouraging, and calls for further development of this new approach.

6 Related work

In this section, we discuss several categories of related work with a focus on the verification and/or refutation of hyperproperties that require quantifier alternation.

6.1 Verification using relational program logics

Dickerson et al. use a relational Hoare-style program logic, RHLE, to verify ∀∃ hyperproperties [35]. They automated their logic by generating verification conditions that are then discharged by an SMT solver. Upon verification failure, their implementation produces a counterexample. However, as is common in all deductive verification approaches, loops in the input program present a major challenge. In fact, the implementation of this program logic, ORHLE [35], requires every loop to be annotated with relevant loop invariants. In

Table 4 Properties of the `escalating` benchmark, parameterized by the initial value of the program variable max

Initial max	0	1	2	...	5	6	...	11	12	...
Analysis depth	4	4	5	...	5	6	...	6	7	...
Max. of x	6	6	8	...	8	10	...	10	12	...
Max. of y	5	5	10	...	10	17	...	17	26	...
Max. of max	3	4	6	...	9	11	...	16	18	...

Initial max	19	20	...	29	30	...	41	42	...	55
Analysis depth	7	8	...	8	9	...	9	10	...	10
Max. of x	12	14	...	14	16	...	16	18	...	18
Max. of y	26	37	...	37	50	...	50	65	...	65
Max. of max	25	27	...	36	38	...	49	51	...	64

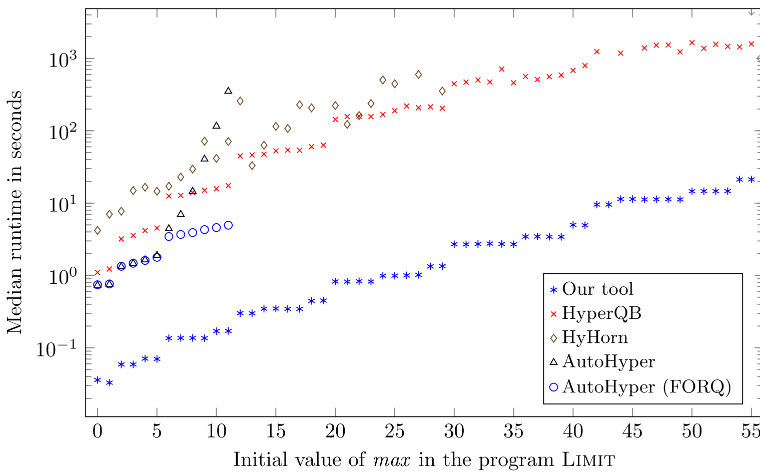


Fig. 7 Median runtimes of our approach, HyperQB [47], HyHorn [49], and AutoHyper [13] on instances of `escalating`. Missing data points indicate timeouts after 30 minutes. The vertical axis is logarithmic

contrast, our approach based on symbolic execution can find counterexamples even without loop invariants.

Beutner [10] also proposed an automated approach to the verification of ∀∃ hyperproperties based on a relational program logic. The associated tool, called ForEx, relies on a notion of *parameterized strongest post-conditions* to automate the verification using symbolic execution techniques. Contrary to ORHLE and to our approach, ForEx cannot detect violations of hyperproperties.

More recently, Dardinier et al. [34] automated the verification of hyperproperties expressed in an extension of Hyper Hoare Logic [33]. Their approach uses deductive program verification based on the Viper verification framework [56]. While this method can effectively and conclusively refute hyperliveness properties by attempting to verify the negation of the original Hyper Hoare Logic specification, it does not produce concrete counterexample traces. Also, in order for the verification (or refutation) to succeed, input programs generally need to be annotated with invariants. Similar to the automated verification of RHLE [35], the handling of loops is limited even with user-defined relational loop invariants.

6.2 Game-based verification of hyperproperties

Beutner and Finkbeiner developed a game-theoretic approach to the verification of ∀∃ -safety properties [12] in infinite-state settings. In their approach, the interaction between universally and existentially quantified traces is interpreted as a game between two agents. If the existential player has a winning strategy, the property holds. This work also introduced *Observation-based HyperLTL* (OHyperLTL) as an asynchronous hyperlogic for infinite-state systems, from which we derived $\text{OHyperLTL}_{\text{safe}}$ in Section 3.

An earlier game-theoretic approach for ∀∃ hyperproperties by Coenen et al. [27], which extends the tool MCHyper [41] to formulas with quantifier alternation, followed a similar idea but was limited to finite-state systems.

Such strategy-based verification methods are generally incomplete, i.e., verification might fail even though the respective property holds, and existing approaches to completeness do not scale beyond very small systems and constrained classes of properties [11, 13]. Additionally, when verification fails, game-based algorithms generally do not produce concrete counterexamples that witness the property violation.

6.3 Automata-based model checking

Beutner and Finkbeiner designed an explicit-state model checker for arbitrary HyperLTL properties [13], which is the first *complete* model checker to support both quantifier alternation and arbitrary temporal operators in the LTL body of HyperLTL formulas. However, like earlier automata-based approaches [41], this method is limited to finite-state systems.

An extension of an LTL model checking approach by Dietsch et al. [36] to Hyper Temporal Stream Logic (HyperTSL) [28] was recently presented by Finkbeiner et al. alongside an algorithm for finding counterexamples to $\forall^*\exists^*$ fragment of HyperTSL over first-order theories [42]. Like our approach, this algorithm is sound but necessarily incomplete. Contrary to our approach, its incompleteness primarily arises from the use of approximations.

6.4 Verification using constrained horn clauses

More recently, Itzhaky et al. [49] proposed to reduce the verification of hyperproperties, including those that require quantifier alternation, to Constrained Horn Clauses (CHC) satisfiability. They define a fragment of $\mathrm{OHyperLTL}$, which they refer to as $\forall^*\exists^*$-OHyperLTL, and that is very similar to our own fragment $\mathrm{OHyperLTL}_{\mathrm{safe}}$, both syntactically and semantically. They implemented their approach in a tool called HyHorn. For some examples, HyHorn can refute hyperproperties fully automatically. However, as reported in Section 5.4, HyHorn has difficulties finding bugs in some of the more challenging instances.

Earlier work by Unno et al. followed a similar idea as [49] but the resulting formulas fell outside the scope of standard CHC solvers, requiring a custom solver for their particular generalization of Constrained Horn Clauses [63].

6.5 Bounded model checking for hyperproperties

Hsu et al. apply bounded model checking to hyperproperties over finite-state systems by reducing the refutation task to checking the satisfiability of a quantified boolean formula (QBF) [47]. Standard bounded model checking methods encode the behavior of finite-state systems as well as potential specification violations as boolean formulas to be solved by satisfiability (SAT) solvers [14, 15, 24]. Hsu et al. extend this approach with quantifiers, hence yielding QBF formulas to be solved by QBF solvers.

In some sense, our algorithm lifts this approach to the testing of infinite-state systems through the use of an SMT solver in a way that is agnostic of the underlying theory. Further, our approach effectively subdivides the bug finding task by enumerating control flow paths using symbolic execution such that each can be analyzed independently of other paths (see Section 4).

Hsu et al. further developed an extension of their approach to asynchronous hyperproperties [48]. Unlike our method, which follows the semantics of $\mathrm{OHyperLTL}$[12] and thus

relies on explicit observation points for synchronization across executions, their approach extends A-HLTL [9], which is a different asynchronous logic for hyperproperties that uses a concept of *trajectories* to align executions. Lastly, unlike our approach, this asynchronous extension is still limited to finite-state systems.

6.6 SMT solving and quantifier alternation

Naturally, our approach benefits from the significant advancements in the design and implementation of SMT solvers, especially with respect to satisfiability queries that contain universal quantifiers [3, 17, 55, 57]. The Yices SMT solver introduced a dedicated algorithm for solving $\exists\forall$-quantified queries in version 2.2 to aid in the synthesis of system parameters [38]. Their method, which we use for our implementation and experimental evaluation, provides termination guarantees for some underlying theories (including bitvectors and subsets of integer arithmetic).

6.7 Symbolic testing for hypersafety

Daniel et al. propose efficient methods to automatically test 2-safety security properties [31, 32] such as constant-time, secret-erasure, and absence of Spectre attacks at the binary level. Their approach is based on symbolic execution but tailored to these predefined properties, which allows them to develop important optimizations. However, contrary to our approach, their method cannot handle properties that require quantifier alternation in the hyperproperty specification.

7 Conclusion

We have presented a bounded semantics for the logic $\mathrm{OHyperLTL}_{\mathrm{safe}}$, and an algorithm capable of searching for counterexamples that witness the violation of $\forall\exists$ hyperproperties given user-defined input programs for the respective quantified trace variables. We have demonstrated its effectiveness in locating specification violations by evaluating our implementation both against various examples from related work and against our own set of benchmarks, and we have shown experimentally that our approach outperforms existing tools in many cases. We have further characterized precisely under what circumstances our algorithm succeeds (as opposed to diverging).

It is worth noting that the counterexamples produced by the proposed algorithm consist of concrete traces for the universally quantified traces only. While some SAT and SMT solvers can produce proofs of unsatisfiability, these automatically generated proofs are often difficult to relate to the high-level input semantics. Future work includes an extension of the proposed algorithm that not only produces concrete traces for the universally quantified traces but also an explanation as to why these traces form a counterexample, likely in the form of a proof in terms of the high-level input semantics.

Our implementation supports imperative programs and utilizes the SMT theory of integer arithmetic. We intend to extend its capabilities with support for other background theories in the future, which is possible because our method is theory-agnostic.

This novel approach to testing hyperproperties even in the presence of quantifier alternation lays the foundations for future research into fully automated hyperbug detection. It is not only sound but even complete for a large class of programs. Unlike many existing approaches, it is not limited to any particular hyperproperty, and our experimental evaluation has demonstrated its effectiveness on a wide range of specifications. Given the promising results discussed in this paper, it seems likely that future enhancements will further improve the scalability of our approach, in which case it can be expected to become applicable to far more complex specifications and programs.

Acknowledgments This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440 and from the Vienna Science and Technology Fund (WWTF) [10.47379/VRG11005]. This work was also supported by the European Research Council (ERC) Grant HYPER (No. 101055412). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. A. Correnson carried out this work as a member of the Saarbrücken Graduate School of Computer Science.

Author contributions A.C. and T.N. wrote the manuscript text. All authors reviewed the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Allen, F.E.: Control flow analysis. ACM SIGPLAN Notices **5**(7), 1–19 (1970). https://doi.org/10.1145/390013.808479
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press Ser, The MIT Press, Cambridge, Massachusetts and London, England (2008)
3. Barbosa, H., Barrett, C., Brain, M., et al.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman D, Rosu G (eds) Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham, pp 415–442 (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004. IEEE, Pacific Grove, CA, USA, pp 100–114,(2004). https://doi.org/10.1109/CSFW.2004.1310735
5. Barthe, G., Crespo, JM., Kunz, C.: Relational Verification Using Product Programs. In: Butler M, Schulte W (eds) FM 2011: Formal Methods. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, pp 200–214 (2011). https://doi.org/10.1007/978-3-642-21437-0_17

6. Barthe, G., D'Argenio, PR., Finkbeiner, B., et al.: Facets of software doping. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications. Springer International Publishing, Cham, pp 601–608 (2016.) https://doi.org/10.1007/978-3-319-47169-3_46
7. Bartocci, E., Henzinger, TA., Nickovic, D., et al.: Hypernode Automata. LIPIcs, Volume 279, CONCUR 2023 279:21:1–21:16 (2023). https://doi.org/10.4230/LIPICS.CONCUR.2023.21
8. Barwise, J.: An introduction to first-order logic. In: Barwise, J. (ed.) Handook of Mathematical Logic, Studies in Logic and the Foundations of Mathematics, vol. 90, pp. 5–46. Elsevier (1977)
9. Baumeister, J., Coenen, N., Bonakdarpour, B., et al.: A Temporal Logic for Asynchronous Hyperproperties. In: Silva A, Leino KRM (eds) Computer Aided Verification. Springer International Publishing, Cham, pp 694–717 (2021). https://doi.org/10.1007/978-3-030-81685-8_33
10. Beutner, R.: Automated software verification of hyperliveness. In: Finkbeiner B, Kovács L (eds) Tools and Algorithms for the Construction and Analysis of Systems. Springer, Cham, pp 196–216 (2024). https://doi.org/10.1007/978-3-031-57249-4_10
11. Beutner, R., Finkbeine, B.: Prophecy Variables for Hyperproperty Verification. In: 2022 IEEE 35th Computer Security Foundations Symposium (CSF), pp 471–485 (2022a). https://doi.org/10.1109/CSF54842.2022.9919658
12. Beutner, R., Finkbeiner, B.: Software Verification of Hyperproperties Beyond k-Safety. In: Shoham S, Vizel Y (eds) Computer Aided Verification. Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 341–362 (2022b). https://doi.org/10.1007/978-3-031-13185-1_17
13. Beutner, R., Finkbeiner, B.: AutoHyper: Explicit-State Model Checking for HyperLTL. In: Sankaranarayanan S, Sharygina N (eds) Tools and Algorithms for the Construction and Analysis of Systems. Springer, Cham, Lecture Notes in Computer Science, pp 145–163 (2023). https://doi.org/10.1007/978-3-031-30823-9_8
14. Biere, A., Cimatti, A., Clarke, E., et al.: Symbolic Model Checking without BDDs. In: Cleaveland WR (ed) Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, pp 193–207 (1999). https://doi.org/10.1007/3-540-49059-0_14
15. Biere, A., Cimatti, A., Clarke, E.M., et al.: Bounded model checking. Adv Comput **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
16. de Boer, FS., Bonsangue, M.: On the Nature of Symbolic Execution. In: ter Beek MH, McIver A, Oliveira JN (eds) Formal Methods – The Next 30 Years. Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 64–80 (2019). https://doi.org/10.1007/978-3-030-30942-8_6
17. Bonacina, M.P., Graham-Lengrand, S., Vauthier, C.: QSMA: A New Algorithm for Quantified Satisfiability Modulo Theory and Assignment. In: Pientka, B., Tinelli, C. (eds.) Automated Deduction - CADE 29, pp. 78–95. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-38499-8_5
18. Bonakdarpour, B., Sanchez, C., Schneider, G.: Monitoring Hyperproperties by Combining Static Analysis and Runtime Verification. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification and Validation. Verification. Springer International Publishing, Cham, pp 8–27 (2018). https://doi.org/10.1007/978-3-030-03421-4_2
19. Bozzelli, L., Peron, A., Sanchez, C.: Asynchronous Extensions of HyperLTL. In: 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). IEEE, Rome, Italy, pp 1–13 (2021). https://doi.org/10.1109/LICS52264.2021.9470583, https://ieeexplore.ieee.org/document/9470583/
20. Bucur, S., Ureche, V., Zamfir, C., et al.: Parallel symbolic execution for automated real-world software testing. In: Proceedings of the sixth conference on Computer systems. Association for Computing Machinery, New York, NY, USA, EuroSys '11, pp 183–198 (2011). https://doi.org/10.1145/1966445.1966463
21. Cadar, C., Ganesh, V., Pawlowski, PM., et al.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM conference on Computer and communications security. Association for Computing Machinery, New York, NY, USA, CCS '06, pp 322–335 (2006). https://doi.org/10.1145/1180405.1180445
22. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation. USENIX Association, USA, OSDI'08, pp 209–224 (2008)
23. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM **55**(8), 107–115 (2012). https://doi.org/10.1145/2240236.2240262
24. Clarke, E., Biere, A., Raimi, R., et al.: Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design **19**(1), 7–34 (2001). https://doi.org/10.1023/A:1011276507260
25. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
26. Clarkson, MR., Finkbeiner, B., Koleini, M., et al.: Temporal Logics for Hyperproperties. In: Abadi M, Kremer S (eds) Principles of Security and Trust. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, pp 265–284, (2014) https://doi.org/10.1007/978-3-642-54792-8_15

27. Coenen, N., Finkbeiner, B., Sánchez, C., et al.: Verifying Hyperliveness. In: Dillig I, Tasiran S (eds) Computer Aided Verification. Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 121–139 (2019). https://doi.org/10.1007/978-3-030-25540-4_7
28. Coenen, N., Finkbeiner, B., Hofmann, J., et al.: Smart Contract Synthesis Modulo Hyperproperties. In: 2023 IEEE 36th Computer Security Foundations Symposium (CSF). IEEE, Dubrovnik, Croatia, pp 276–291 (2023). https://doi.org/10.1109/CSF57540.2023.00006
29. Correnson, A., Steinhöfel, D.: Engineering a formally verified automated bug finder. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2023, pp 1165–1176 (2023). https://doi.org/10.1145/3611643.3616290
30. Correnson, A., Nießen, T., Finkbeiner, B., et al.: Finding ∀∃ Hyperbugs using Symbolic Execution. Proc ACM Program Lang 8(OOPSLA2):321:1420–321:1445 (2024) https://doi.org/10.1145/3689761
31. Daniel, LA., Bardin, S., Rezk, T.: Hunting the haunter - efficient relational symbolic execution for spectre with Haunted RelSE. Proceedings 2021 Network and Distributed System Security Symposium (2021). https://doi.org/10.14722/ndss.2021.24286
32. Daniel, L.A., Bardin, S., Rezk, T.: Binsec/Rel: Symbolic binary analyzer for security with applications to constant-time and secret-erasure. ACM Trans Priv Secur (2023). https://doi.org/10.1145/3563037
33. Dardinier, T., Müller, P.: Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. Proc ACM Program Lang **8**(PLDI), 207:1485–207:1509 (2024). https://doi.org/10.1145/3656437
34. Dardinier, T., Li, A., Müller, P.: Hypra: A Deductive Program Verifier for Hyper Hoare Logic. Proceedings of the ACM on Programming Languages **8**(OOPSLA2), 1279–1308 (2024). https://doi.org/10.1145/3689756
35. Dickerson, R., Ye, Q., Zhang, MK., et al.: RHLE: Modular Deductive Verification of Relational ∀∃ Properties. In: Sergey I (ed) Programming Languages and Systems. Springer, Cham, Lecture Notes in Computer Science, pp 67–87 (2022). https://doi.org/10.1007/978-3-031-21037-2_4
36. Dietsch, D., Heizmann, M., Langenfeld, V., et al.: Fairness Modulo Theory: A New Approach to LTL Software Model Checking. In: Kroening D, Păsăreanu CS (eds) Computer Aided Verification, vol 9206. Springer International Publishing, Cham, p 49–66 (2015). https://doi.org/10.1007/978-3-319-21690-4_4
37. Doveri, K., Ganty, P., Mazzocchi, N.: FORQ-Based Language Inclusion Formal Testing. In: Shoham S, Vizel Y (eds) Computer Aided Verification. Springer International Publishing, Cham, pp 109–129 (2022). https://doi.org/10.1007/978-3-031-13188-2_6
38. Dutertre, B.: Yices 2.2. In: Biere A, Bloem R (eds) Computer Aided Verification. Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 737–744 (2014). https://doi.org/10.1007/978-3-319-08867-9_49
39. Farzan, A., Vandikas, A.: Automated Hypersafety Verification. In: Dillig I, Tasiran S (eds) Computer Aided Verification. Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 200–218 (2019a). https://doi.org/10.1007/978-3-030-25540-4_11
40. Farzan, A., Vandikas, A.: Reductions for safety proofs. Proc ACM Program Lang (2019b). https://doi.org/10.1145/3371081
41. Finkbeiner, B., Rabe, MN., Sánchez, C.: Algorithms for Model Checking HyperLTL and HyperCTL*. In: Kroening D, Păsăreanu CS (eds) Computer Aided Verification. Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 30–48 (2015). https://doi.org/10.1007/978-3-319-21690-4_3
42. Finkbeiner, B., Frenkel, H., Hofmann, J.: Automata-Based Software Model Checking of Hyperproperties. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods, pp. 361–379. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-33170-1_22
43. Floyd, RW.: Assigning meanings to programs. In: Proceedings of Symposia in Applied Mathematics, vol 19. Amer. Math. Soc., Providence, R.I., pp 19–32 (1967). https://mathscinet.ams.org/mathscinet-getitem?mr=235771
44. Gallier, J.H.: Logic for Computer Science: Foundations of Automatic Theorem Proving, 2nd edn. Dover Books On Computer Science, Dover Publications Inc, Mineola, New York (2015)
45. Godefroid, P.: The soundness of bugs is what matters (position statement). In: BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools) (2005)
46. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. ACM SIGPLAN Notices **40**(6), 213–223 (2005). https://doi.org/10.1145/1064978.1065036
47. Hsu, TH., Sánchez, C., Bonakdarpour, B.: Bounded Model Checking for Hyperproperties. In: Groote JF, Larsen KG (eds) Tools and Algorithms for the Construction and Analysis of Systems, vol 12651. Springer International Publishing, Cham, p 94–112 (2021). https://doi.org/10.1007/978-3-030-72016-2_6
48. Hsu, TH., Bonakdarpour, B., Finkbeiner, B., et al.: Bounded Model Checking for Asynchronous Hyperproperties. In: Sankaranarayanan S, Sharygina N (eds) Tools and Algorithms for the Construction and Analysis of Systems. Springer Nature Switzerland, Cham, pp 29–46 (2023). https://doi.org/10.1007/978-3-031-30823-9_2

49. Itzhaky, S., Shoham, S., Vizel, Y.: Hyperproperty verification as CHC satisfiability. In: Weirich S (ed) Programming Languages and Systems. Springer, Cham, pp 212–241 (2024). https://doi.org/10.1007/978-3-031-57267-8_9
50. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification: Safety and termination. Handbook of Model Checking pp 447–491 (2018). https://doi.org/10.1007/978-3-319-10575-8_15
51. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
52. Manes, V.J., Han, H., Han, C., et al.: The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Trans. Software Eng. **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563
53. McCullough, D.:Noninterference and the composability of security properties. In: Proceedings. 1988 IEEE Symposium on Security and Privacy. IEEE Comput. Soc. Press, Oakland, CA, USA, pp 177–186 (1988). https://doi.org/10.1109/SECPRI.1988.8110
54. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy, pp 79–93 (1994). https://doi.org/10.1109/RISP.1994.296590
55. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
56. Müller, P., Schwerhoff, M., Summers, AJ.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Jobstmann B, Leino KRM (eds) Verification, Model Checking, and Abstract Interpretation. Springer, Berlin, Heidelberg, pp 41–62 (2016). https://doi.org/10.1007/978-3-662-49122-5_2
57. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting Enumerative Instantiation. In: Beyer D, Huisman M (eds) Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham, pp 112–131 (2018). https://doi.org/10.1007/978-3-319-89963-3_7
58. Sabelfeld, A., Myers, AC.: A Model for Delimited Information Release. In: Futatsugi K, Mizoguchi F, Yonezaki N (eds) Software Security - Theories and Systems. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, pp 174–191 (2004). https://doi.org/10.1007/978-3-540-37621-7_9
59. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. ACM SIGSOFT Software Engineering Notes **30**(5), 263–272 (2005). https://doi.org/10.1145/1095430.1081750
60. Shemer, R., Gurfinkel, A., Shoham, S., et al.: Property Directed Self Composition. In: Dillig I, Tasiran S (eds) Computer Aided Verification. Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 161–179 (2019). https://doi.org/10.1007/978-3-030-25540-4_9
61. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, PLDI ’16, pp 57–69 (2016). https://doi.org/10.1145/2908080.2908092
62. Staats, M., Păsăreanu, C.: Parallel symbolic execution for structural test generation. In: Proceedings of the 19th international symposium on Software testing and analysis. Association for Computing Machinery, New York, NY, USA, ISSTA ’10, pp 183–194 (2010). https://doi.org/10.1145/1831708.1831732
63. Unno, H., Terauchi, T., Koskinen, E.: Constraint-Based Relational Verification. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification, pp. 742–766. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_35
64. Yang, W., Vizel, Y., Subramanyan, P., et al.: Lazy Self-composition for Security Verification. In: Chockler H, Weissenbacher G (eds) Computer Aided Verification. Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 136–156 (2018). https://doi.org/10.1007/978-3-319-96142-2_11

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.