Bounded Model Checking for Hyperproperties

Tzu-Han Hsu[1], César Sánchez[2], and Borzoo Bonakdarpour[1]

[1] Department of Computer Science and Engineering
Michigan State University, USA
{tzuhan,borzoo}@msu.edu
[2] IMDEA Software Institute, Spain
cesar.sanchez@imdea.org

Abstract. Hyperproperties are properties of systems that relate multiple computation traces, including security properties and properties in concurrency. This paper introduces a bounded model checking (BMC) algorithm for hyperproperties expressed in HyperLTL, which—to the best of our knowledge—is the first such algorithm. Just as the classic BMC technique for LTL primarily aims at finding bugs, our approach also targets identifying counterexamples. LTL describes a property via inspecting individual traces, so BMC for LTL is reduced to SAT solving. HyperLTL allows explicit and simultaneous quantification over traces and describes properties that involves multiple traces and, hence, our BMC approach naturally reduces to QBF solving. We report on successful and efficient model checking, implemented in a tool called HyperQube, of a rich set of experiments on a variety of case studies, including security/privacy, concurrent data structures, and path planning in robotics applications.

1 Introduction

Hyperproperties [10] have been shown to be a powerful framework for specifying and reasoning about important classes of requirements that were not possible with trace-based languages such as the classic temporal logics. Examples include information-flow security, consistency models in concurrent computing [5], and robustness models in cyber-physical systems [6,33]. The temporal logic HyperLTL [9] extends LTL by allowing explicit and simultaneous quantification over execution traces, describing the property of multiple traces. For example, the security policy *observational determinism* can be specified by the following HyperLTL formula:

$$\forall \pi_A.\forall \pi_B.(o_{\pi_A} \leftrightarrow o_{\pi_B}) \; \mathcal{W} \; \neg(i_{\pi_A} \leftrightarrow i_{\pi_B})$$

which stipulates that every pair of traces π_A and π_B have to agree on the value of the (public) output o as long as they agree on the value of the (secret) input i, where '$\mathcal{W}$' denotes the weak until operator.

There has been a recent surge of model checking techniques for HyperLTL specifications [9,12,22,24]. These approaches employ various techniques (e.g.,

alternating automata, model counting, strategy synthesis, etc) to verify hyper-properties. However, they generally fall short in proposing an effective method to deal with identifying bugs with respect to alternating HyperLTL formulas. Indeed, quantifier alternation has been shown to generally elevate the complexity class of model checking HyperLTL specifications in different shapes of Kripke structures (KS) [2, 9]. For example, consider the simple Kripke structure K in Fig. 1 and HyperLTL formulas $\varphi_1 = \forall \pi_A. \forall \pi_B. \Box(p_{\pi_A} \leftrightarrow p_{\pi_B})$ and $\varphi_2 = \forall \pi_A. \exists \pi_B. \Box(p_{\pi_A} \not\leftrightarrow p_{\pi_B})$. Proving that $K \not\models \varphi_1$ (where traces for π_A and π_B are taken from K) can be reduced to building the self-composition of K and applying standard LTL model checking, resulting in worst-case complexity $|K|^2$ in the size of the system. On the contrary, proving that $K \models \varphi_2$ is not as straightforward. In the worst case, this requires a subset generation to encode the existential quantifier within the Kripke structure, resulting in $|K| \cdot 2^{|K|}$ blow up. In addition, the quantification is over traces rather than states, adding to the complexity of reasoning.
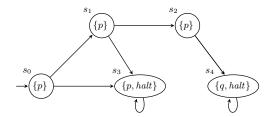

Fig. 1: A Kripke structure.

Following the great success of bounded model checking (BMC) for LTL specifications [8], in this paper, we propose the first BMC algorithm for HyperLTL. To the best of our knowledge this is the first such algorithm. Just as BMC for LTL is reduced to SAT solving to search for a counterexample trace whose length is bounded by some integer k, we reduce BMC for HyperLTL to QBF solving to be able to deal with quantified counterexample traces in the input model. More formally, given a HyperLTL formula (for example, of the form) $\varphi = \forall \pi_A. \exists \pi_B. \psi$ and a family of Kripke structures $\mathcal{K} = (K_A, K_B)$ (one per trace variable), the reduction involves three main components. First, the transition relation of K_π (for every π) is represented by a Boolean encoding $[\![K_\pi]\!]$. Secondly, the inner LTL subformula ψ is translated to a Boolean fixpoint representation $[\![\psi]\!]$ in a similar fashion to the standard BMC technique for LTL. This way, the QBF encoding for a bound $k \geq 0$ roughly appears as:

$$[\![\mathcal{K}, \neg\varphi]\!]_k = \exists \overline{x_A}. \forall \overline{x_B}. [\![K_A]\!]_k \wedge \left([\![K_B]\!]_k \rightarrow [\![\neg\psi]\!]_k\right) \qquad (1)$$

where the vector of Boolean variables $\overline{x_A}$ (respectively, $\overline{x_B}$) are used to represent the state and propositions of the kripke structures K_A (resp. K_B) for steps from 0 to k. Formulas $[\![K_A]\!]_k$ and $[\![K_B]\!]_k$ are the unrollings K_A (using $\overline{x_A}$) and

K_B (using $\overline{x_B}$), and $\llbracket \neg\psi \rrbracket$ (that uses both $\overline{x_A}$ and $\overline{x_B}$) is the fixpoint Boolean encoding of $\neg\psi$. We note that the proposed technique in this paper does not incorporate a loop condition, as implementing such a condition for multiple traces is not straightforward at all. This, of course, comes at the cost of lack of a completeness result.

While our QBF encoding is a natural generalization of BMC for HyperLTL, the first contribution of this paper is a more refined view of how to interpret the behavior of the formula beyond the unrolling depth k. Consider LTL formula $\forall\pi.\,\square p_\pi$. BMC for LTL attempts to find a counterexample by unrolling the model and check for satifiability of $\exists\pi.\,\Diamond \neg p_\pi$. In this case satifiability means existence of a counterexample within the first k steps. Now consider LTL formula $\forall\pi.\,\Diamond p_\pi$ whose negation is of the form $\exists\pi.\,\square \neg p_\pi$. In the classic BMC, due to its *pessimistic* handling of $\square$ the unstatisfiability of the formula can not be established in the finite unrolling (handling these formulas requires to appeal to either looping conditions or to reach the diameter of the the system). This is because $\square\neg p_\pi$ is not *sometimes finitely satisfiable* (SFS), in the terminology introduced by Havelund and Peled [27], meaning that not all satisfying traces of $\square p_\pi$ have a finite prefix that witness the satisfiability.

We propose a method that allows to interpret a wide range of outcomes of the QBF solver and relate these to the original model checking decision problem. To this end, we propose the following semantics for BMC for HyperLTL:

- *Pessimistic* semantics (which is the common for LTL BMC) under which pending eventualities are considered to be unfulfilled. This semantics work for sometime finitely satisfiable temporal formulas and paves the way for bug hunting.
- *Optimistic* semantics considers the dual case, where pending eventualities are assumed to be fulfilled at the end of the trace. This semantics work for *sometimes finitely refutable* formulas, and allows us to interpret unsatisfiability of QBF as proof of verification even with bounded traces.
- *Halting* variants of the optimistic and pessimistic semantics, which allows sound and complete decision on a verdict for terminating models.

We have fully implemented our technique in the tool HyperQube. Our experimental evaluation includes a rich set of case studies, such as information-flow security/privacy, concurrent data structures (in particular, linearizability), and path planning in robotic applications. Our evaluation shows that our technique is effective and efficient in identifying bugs in several prominent examples. We also show that our QBF-based approach is certainly more efficient than an brute-force SAT-based approach, where universal and existential quantifiers are eliminated by combinatorial expansion to conjunctions and disjunctions. We also show that in some cases our approach can also be used as as tool for synthesis. Indeed, a witness to an existential quantifier in a HyperLTL formula is an execution path that satisfies the formula. For example, our experiments on path planning for robots showcases this feature of HyperQube.

In summary, the contributions of this paper are as follows:

- We propose a QBF-based BMC approach for verification and falsification of HyperLTL specifications.
- We introduce complementary semantics that allow proving and disproving formulas, given a finite set of finite traces.
- We rigorously analyze the performance of our technique by case studies from different areas of computing.

The rest of the paper is structured as follows. Section 2 contains the preliminaries. Section 3 introduces the different bounded semantics for HyperLTL. Section 4 presents the encoding into QBF of the different formulas and bounded semantics of choice, and what can be inferred in each case about the HyperLTL model checking problem in each case. Sections 5 and 6 present an empirical evaluation of our tool HyperQube. Section 7 presents the related work and Section 8 concludes.

2 Preliminaries

2.1 Kripke Structures

Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the *alphabet*. A *letter* is an element of Σ. A *trace* $t \in \Sigma^\omega$ over alphabet Σ is an infinite sequence of letters: $t = t(0)t(1)t(2)\cdots$

Definition 1. *A* Kripke structure *is a tuple* $K = \langle S, S_{init}, \delta, L \rangle$, *where*

- S *is a finite set of* states;
- $S_{init} \subseteq S$ *is the set of* initial states;
- $\delta \subseteq S \times S$ *is a* transition relation, *and*
- $L : S \to \Sigma$ *is a* labeling function *on the states of* K.

We require that for each $s \in S$, *there exists* $s' \in S$, *such that* $(s, s') \in \delta$.

Fig. 1 shows a Kripke structure, where $S_{init} = \{s_0\}$, $L(s_0) = \{p\}$, $L(s_4) = \{q, halt\}$, etc.

The *size* of the Kripke structure is the number of its states.

A *loop* in K is a finite sequence $s(0)s(1)\cdots s(n)$, such that $(s(i), s(i+1)) \in \delta$, for all $0 \le i < n$, and $(s(n), s(0)) \in \delta$. We call a Kripke frame *acyclic*, if the only loops are self-loops on otherwise terminal states, i.e., on states that have no other outgoing transition. Since Definition 1 does not allow terminal states, we only consider acyclic Kripke structures with such added self-loops. We also label such states by atomic proposition *halt*.

A *path* of a Kripke structure is an infinite sequence of states $s(0)s(1)\cdots \in S^\omega$, such that:

- $s(0) \in S_{init}$, and
- $(s(i), s(i+1)) \in \delta$, for all $i \ge 0$.

A trace of a Kripke structure is a trace $t(0)t(1)t(2)\cdots \in \Sigma^\omega$, such that there exists a path $s(0)s(1)\cdots \in S^\omega$ with $t(i) = L(s(i))$ for all $i \ge 0$. We denote by *Traces*(K, s) the set of all traces of K with paths that start in state $s \in S$, and use *Traces*(K) as a short for $\bigcup_{s \in S_{init}}$ *Traces*(K, s).

2.2 The Temporal Logic HyperLTL

HyperLTL [9] is an extension of the linear-time temporal logic (LTL) for hyperproperties. The syntax of HyperLTL formulas is defined inductively by the following grammar:

$$\varphi ::= \exists \pi.\varphi \mid \forall \pi.\varphi \mid \phi$$
$$\phi ::= \mathsf{true} \mid a_\pi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi\,\mathcal{U}\,\phi \mid \phi\,\mathcal{R}\,\phi \mid \bigcirc\phi$$

where $a \in \mathsf{AP}$ is an atomic proposition and π is a *trace variable* from an infinite supply of variables $\mathcal{V}$. The Boolean connectives $\neg$, $\vee$ and $\wedge$ have the usual meaning, $\mathcal{U}$ is the temporal *until* operator, $\mathcal{R}$ is the temporal *release* operator, and $\bigcirc$ is the temporal *next* operator. We also consider other derived Boolean connectives, such as $\rightarrow$, and $\leftrightarrow$, and the derived temporal operators *eventually* $\Diamond\varphi \equiv \mathsf{true}\,\mathcal{U}\,\varphi$ and *globally* $\Box\varphi \equiv \neg\Diamond\neg\varphi$. Even though the set of operators presented is not minimal, we have introduced this set to uniform the treatment with the variants in Section 3. The quantified formulas $\exists\pi$ and $\forall\pi$ are read as "along some trace π" and "along all traces π", respectively. A formula is *closed* (i.e., a *sentence*) if all trace variables used in the formula are quantified. We assumed, without lost of generality that no variable is quantified twice. We use $Vars(\varphi)$ for the set of path variables used in formula φ.

Semantics. An interpretation $\mathcal{T} = \langle T_\pi \rangle_{\pi \in Vars(\varphi)}$ of a formula φ consists of a set of traces, one set T_π per trace variable π in $Vars(\varphi)$. We use T_π for the set of traces assigned to π. The idea here is to allow quantifiers to range over different models. We will use this feature in the verification of hyperproperties such as linearizabiliity, where different quantifiers are associated with different sets of executions (in this case one for the concurrent implementation and one for the sequential implementation). That is, each set of traces comes from a Kripke structure and we use $\mathcal{K} = \langle K_\pi \rangle_{\pi \in Vars(\varphi)}$ to denote a *family* of Kripke structure, so $T_\pi = Traces(K_\pi)$ is the traces that π can range over, which comes from K_π. Abusing notation, we write $\mathcal{T} = Traces(\mathcal{K})$.

Note that all trace sets being the same set of traces for a single Kripke structure K (i.e. $K_\pi = K$ for all π) is a particular case, which leads to the original semantics of HyperLTL [9]. The semantics of HyperLTL are defined with respect to a trace assignment, which is a partial map $\Pi\colon Vars(\varphi) \rightharpoonup \Sigma^\omega$. The assignment with empty domain is denoted by $\Pi_\emptyset$. Given a trace assignment Π, a trace variable π, and a concrete trace $t \in \Sigma^\omega$, we denote by $\Pi[\pi \to t]$ the assignment that coincides with Π everywhere but at π, which is mapped to trace t.

The satisfaction of a HyperLTL formula φ is a binary relation $\models$ that associates a formula to the models $(\mathcal{T}, \Pi, i)$ where $i \in \mathbb{Z}_{\geq 0}$ is a pointer that indicates

the current position of all traces in $\mathcal{T}$. The semantics is defined as follows:

$$
\begin{aligned}
(\mathcal{T}, \Pi, 0) &\models \exists \pi.\ \psi & \text{iff} \quad & \text{there is a } t \in T_\pi \text{ such that } (\mathcal{T}, \Pi[\pi \to t], 0) \models \psi, \\
(\mathcal{T}, \Pi, 0) &\models \forall \pi.\ \psi & \text{iff} \quad & \text{for all } t \in T_\pi \text{ such that } (\mathcal{T}, \Pi[\pi \to t], 0) \models \psi, \\
(\mathcal{T}, \Pi, i) &\models \mathsf{true} & & \\
(\mathcal{T}, \Pi, i) &\models a_\pi & \text{iff} \quad & a \in \Pi(\pi)(i), \\
(\mathcal{T}, \Pi, i) &\models \neg \psi & \text{iff} \quad & (\mathcal{T}, \Pi, i) \not\models \psi, \\
(\mathcal{T}, \Pi, i) &\models \psi_1 \vee \psi_2 & \text{iff} \quad & (\mathcal{T}, \Pi, i) \models \psi_1 \text{ or } (\mathcal{T}, \Pi, i) \models \psi_2, \\
(\mathcal{T}, \Pi, i) &\models \psi_1 \wedge \psi_2 & \text{iff} \quad & (\mathcal{T}, \Pi, i) \models \psi_1 \text{ and } (\mathcal{T}, \Pi, i) \models \psi_2, \\
(\mathcal{T}, \Pi, i) &\models \bigcirc \psi & \text{iff} \quad & (\mathcal{T}, \Pi, i+1) \models \psi, \\
(\mathcal{T}, \Pi, i) &\models \psi_1\, \mathcal{U}\, \psi_2 & \text{iff} \quad & \text{there is a } j \geq i \text{ for which } (\mathcal{T}, \Pi, j) \models \psi_2 \text{ and} \\
& & & \quad \text{for all } k \in [i, j), (\mathcal{T}, \Pi, k) \models \psi_1, \\
(\mathcal{T}, \Pi, i) &\models \psi_1\, \mathcal{R}\, \psi_2 & \text{iff} \quad & \text{either for all } j \geq i,\ (\mathcal{T}, \Pi, j) \models \psi_2, \text{ or,} \\
& & & \quad \text{for some } j \geq i, (\mathcal{T}, \Pi, j) \models \psi_1 \text{ and} \\
& & & \quad \text{for all } k \in [i, j] : (\mathcal{T}, \Pi, k) \models \psi_2.
\end{aligned}
$$

We say that an interpretation $\mathcal{T}$ satisfies a sentence φ, denoted by $\mathcal{T} \models \varphi$, if $(\mathcal{T}, \Pi_\emptyset, 0) \models \varphi$. We say that a family of Kripke structures $\mathcal{K}$ satisfies a sentence φ, denoted by $\mathcal{K} \models \varphi$, if $\langle Traces(K_\pi)\rangle_{\pi \in Vars(\varphi)} \models \varphi$. When the same kripke structure K is used for all path variables we write $K \models \varphi$.

For example, the Kripke structure in Fig. 1 satisfies HyperLTL formula $\varphi = \forall \pi_A.\exists \pi_B.\Diamond(p_{\pi_A} \leftrightarrow q_{\pi_B})$.

These semantics are slightly different from the definition in [9], but equivalent. First, we use the pointer i instead of chopping the trace with the head elements when traversing the tuple of traces forward, but this is clearly equivalent and more convenient later in the paper when we define finite unrollings. Second, we use a multi-model semantics allowing different trace variables to choose traces from different trace sets. In terms of the model checking problem, multi-model and (the conventional) single-model semantics [9] are equivalent. One can instantiate all models with the same Kripke structure (so multi-model can simulate single-model). For the other direction, one can merge all Kripke structures into a single Kripke structure and add fresh predicates to distinguish each Kripke structure, and then require each path to belong to the desired original Kripke structure.

2.3 Quantified Boolean Formula Satisfiability

The *quantified Boolean formula* (QBF) satisfiability problem [25] is the following:

Given is a set of Boolean variables, $\{x_1, x_2, \ldots, x_n\}$, and a quantified Boolean formula $F = \mathbb{Q}_1 x_1.\mathbb{Q}_2 x_2 \ldots \mathbb{Q}_{n-1} x_{n-1}.\mathbb{Q}_n x_n.\psi$, where each $\mathbb{Q}_i \in \{\forall, \exists\}$ ($i \in [1, n]$) and ψ is an arbitrary Boolean formula over variables $\{x_1, \ldots, x_n\}$. Is F true?

Solving the satisfiability problem for QBF is known to be PSPACE-complete. Figure 2 shows a satisfying model for the following formula:

$$F = \exists x_1. \forall x_2. \exists x_3. \exists x_4. \forall x_5. (x_1 \vee \neg x_2 \vee x_3) \ \wedge \ (\neg x_1 \vee x_2 \vee \neg x_4) \ \wedge$$
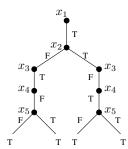$$(\neg x_3 \vee x_4 \vee \neg x_5) \ \wedge \ (x_1 \vee x_4 \vee x_5).$$

Fig. 2: Model for the QBF formula.

3 Bounded Semantics for HyperLTL

In this section, we introduce the bounded semantics of HyperLTL, which will be later used in Section 4 to generate queries to a QBF solver to aid solving the model checking problem.

3.1 Bounded Semantics

We assume the formula is closed and of the form:

$$\mathbb{Q}_A \pi_A. \mathbb{Q}_B \pi_B \ldots \mathbb{Q}_Z \pi_Z. \psi$$

where $\mathbb{Q} \in \{\forall, \exists\}$ and it has been converted into negation-normal form (NNF) so that the negation symbol only appears in front of atomic propositions, e.g., $\neg a_{\pi_A}$. Without loss of generality and for the sake of clarity from other numerical indices, we use roman alphabet as indices of trace variables. Thus, we assume that $Vars(\varphi) \subseteq \{\pi_A, \pi_B, \ldots, \pi_Z\}$. The main idea of bounded model checking is to perform incremental exploration of the state space of the systems by unrolling the systems and the formula up-to a bound. Let $k \geq 0$ be the unrolling *bound* and let $\mathcal{T} = \langle T_A \ldots T_Z \rangle$ be a tuple of finite sets of finite traces, one per trace variable. We start by defining a satisfaction relation between HyperLTL formulas for a bounded exploration k and models $(\mathcal{T}, \Pi, i)$, where $\mathcal{T}$ is the tuple of set of traces, Π is a trace assignment mapping (as defined in Section 2), and $i \in \mathbb{Z}_{\geq 0}$ that points to the position of traces. We will define different finite satisfaction relations for general models (for $* = pes, opt, hpes, hopt$):

- $\models_k^*$, the common satisfaction relation among all semantics,
- $\models_k^{pes}$, called *pessimistic* semantics,
- $\models_k^{opt}$, called *optimistic* semantics, and
- $\models_k^{hpes}$ and $\models_k^{hopt}$, variants of $\models_k^{pes}$ and $\models_k^{opt}$, respectively, for Kripke structures that encode termination of traces (modeled as self-loops to provide infinite traces).

All these semantics coincide in the interpretation of quantifiers, Boolean connectives, and in the interpretation of the temporal operators up-to instant $k-1$, but differ in their assumptions about unseen future events after the bound of observation k.

Quantifiers. The satisfaction relation for the quantifiers is the following:

$$(\mathcal{T}, \Pi, 0) \models_k^* \exists \pi.\ \psi \quad \text{iff} \quad \text{there is a } t \in T_\pi : (\mathcal{T}, \Pi[\pi \to t], 0) \models_k \psi, \tag{1}$$

$$(\mathcal{T}, \Pi, 0) \models_k^* \forall \pi.\ \psi \quad \text{iff} \quad \text{for all} \quad t \in T_\pi : (\mathcal{T}, \Pi[\pi \to t], 0) \models_k \psi. \tag{2}$$

Boolean operators. For every $i \leq k$, we have:

$$(\mathcal{T}, \Pi, i) \models_k^* \mathsf{true} \tag{3}$$

$$(\mathcal{T}, \Pi, i) \models_k^* a_\pi \qquad \text{iff} \quad a \in \Pi(\pi)(i), \tag{4}$$

$$(\mathcal{T}, \Pi, i) \models_k^* \neg a_\pi \qquad \text{iff} \quad a \notin \Pi(\pi)(i), \tag{5}$$

$$(\mathcal{T}, \Pi, i) \models_k^* \psi_1 \vee \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k \psi_1 \text{ or } (\mathcal{T}, \Pi, i) \models_k \psi_2, \tag{6}$$

$$(\mathcal{T}, \Pi, i) \models_k^* \psi_1 \wedge \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k \psi_1 \text{ and } (\mathcal{T}, \Pi, i) \models_k \psi_2 \tag{7}$$

Temporal connectives. The case where $(i < k)$ is common between the optimistic and pessimistic semantics:

$$(\mathcal{T}, \Pi, i) \models_k^* \bigcirc \psi \qquad \text{iff} \quad (\mathcal{T}, \Pi, i+1) \models_k \psi \tag{8}$$

$$(\mathcal{T}, \Pi, i) \models_k^* \psi_1 \,\mathcal{U}\, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k \psi_2, \text{ or}$$
$$(\mathcal{T}, \Pi, i) \models_k \psi_1 \text{ and } (\mathcal{T}, \Pi, i+1) \models_k \psi_1 \,\mathcal{U}\, \psi_2 \tag{9}$$

$$(\mathcal{T}, \Pi, i) \models_k^* \psi_1 \,\mathcal{R}\, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k \psi_2, \text{ and}$$
$$(\mathcal{T}, \Pi, i) \models_k \psi_1 \text{ or } (\mathcal{T}, \Pi, i+1) \models_k \psi_1 \,\mathcal{R}\, \psi_2 \tag{10}$$

For $(i = k)$, in the pessimistic semantics the eventualities (including $\bigcirc$) are assumed to never be fulfilled in the future, so the current instant k is the last chance:

$$(\mathcal{T}, \Pi, i) \models_k^{pes} \bigcirc \psi \qquad \text{iff} \quad \text{never happens} \tag{P_1}$$

$$(\mathcal{T}, \Pi, i) \models_k^{pes} \psi_1 \,\mathcal{U}\, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^{pes} \psi_2 \tag{P_2}$$

$$(\mathcal{T}, \Pi, i) \models_k^{pes} \psi_1 \,\mathcal{R}\, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^{pes} \psi_1 \wedge \psi_2 \tag{P_3}$$

On the other hand, in the optimistic semantics the eventualities are assumed to be fulfilled in the future:

$$(\mathcal{T}, \Pi, i) \models_k^{opt} \bigcirc \psi \qquad \text{iff} \quad \text{always happens} \qquad (O_1)$$
$$(\mathcal{T}, \Pi, i) \models_k^{opt} \psi_1 \, \mathcal{U} \, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^{opt} \psi_1 \vee \psi_2 \qquad (O_2)$$
$$(\mathcal{T}, \Pi, i) \models_k^{opt} \psi_1 \, \mathcal{R} \, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^{opt} \psi_2 \qquad (O_3)$$

In order to capture the halting semantics, we assume that the Kripke structure is equipped with a predicate *halt* that is true if the state corresponds to a halting state, and define the auxiliary predicate $\textit{halted} \overset{\text{def}}{=} \bigwedge_{\pi\, Vars(\varphi)} halt_\pi$ that holds whenever all traces have halted (and their final state will be repeated ad infinitum), where *halt* is an atomic proposition denoting the termination of a trace. Then, the halted semantics of the temporal case for $i = k$ in the pessimistic case consider the halting case to infer the actual value of the temporal operators on the (now fully known) trace:

$$(\mathcal{T}, \Pi, i) \models_k^{hpes} \bigcirc \psi \qquad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^* \textit{halted} \text{ and } (\mathcal{T}, \Pi, i) \models_k^{hpes} \psi \quad (HP_1)$$
$$(\mathcal{T}, \Pi, i) \models_k^{hpes} \psi_1 \, \mathcal{U} \, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^{hpes} \psi_2 \qquad (HP_2)$$
$$(\mathcal{T}, \Pi, i) \models_k^{hpes} \psi_1 \, \mathcal{R} \, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^{hpes} \psi_1 \wedge \psi_2, \text{ or}$$
$$\qquad\qquad (\mathcal{T}, \Pi, i) \models_k^* \textit{halted} \text{ and } (\mathcal{T}, \Pi, i) \models_k^{hpes} \psi_2 \quad (HP_3)$$

Dually, in the halting optimistic case:

$$(\mathcal{T}, \Pi, i) \models_k^{hopt} \bigcirc \psi \qquad \text{iff} \quad (\mathcal{T}, \Pi, i) \not\models_k^* \textit{halted} \text{ or } (\mathcal{T}, \Pi, i) \models_k^{hopt} \psi \qquad (HO_1)$$
$$(\mathcal{T}, \Pi, i) \models_k^{hopt} \psi_1 \, \mathcal{U} \, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^{hopt} \psi_2, \text{ or}$$
$$\qquad\qquad (\mathcal{T}, \Pi, i) \not\models_k^* \textit{halted} \text{ and } (\mathcal{T}, \Pi, i) \models_k^{hopt} \psi_1 \quad (HO_2)$$
$$(\mathcal{T}, \Pi, i) \models_k^{hopt} \psi_1 \, \mathcal{R} \, \psi_2 \quad \text{iff} \quad (\mathcal{T}, \Pi, i) \models_k^{hpes} \psi_2 \qquad (HO_3)$$

Complete semantics. We are now ready to define the four semantics:
- The pessimistic semantics $\models_k^{pes}$ is comprised of rules (1)-(10) and (P_1)-(P_3).
- The optimistic semantics $\models_k^{opt}$ consists of rules (1)-(10) and (O_1)-(O_3).
- The halting pessimistic semantics $\models_k^{hpes}$ use rules (1)-(10) and (HP_1)-(HP_3).
- The halting optimistic semantics $\models_k^{hopt}$ use rules (1)-(10) and (HO_1)-(HO_3).

3.2 The Logical Relation between Different Semantics

Observe that the pessimistic semantics is the semantics in the traditional BMC for LTL, where pending eventualities are considered to be unfulfilled. In the pessimistic semantics a formula is declared false unless it is witnessed to be true within the bound explored. In other words, formulas can only get "truer" with more information obtained by a longer unrolling. Dually, the optimistic semantics considers a formula true unless there is evidence within the bounded exploration on the contrary. Therefore, formulas only get "falser" with further unrolling. For example, formula $\square p$ always evaluates to false in the pessimistic semantics. In the optimistic semantics, it evaluates to true upto bound k if p holds in all states of the trace upto and including k. However, if the formula evaluates to false at some point before k, then it evaluates to false for all $j \geq k$.

The following lemma formalizes this intuition in HyperLTL.

Lemma 1. *Let $k \leq j$. Then,*

1. *If $(\mathcal{T}, \Pi, 0) \models_k^{pes} \varphi$, then $(\mathcal{T}, \Pi, 0) \models_j^{pes} \varphi$.*
2. *If $(\mathcal{T}, \Pi, 0) \not\models_k^{opt} \varphi$, then $(\mathcal{T}, \Pi, 0) \not\models_j^{opt} \varphi$.*
3. *If $(\mathcal{T}, \Pi, 0) \models_k^{hpes} \varphi$, then $(\mathcal{T}, \Pi, 0) \models_j^{hpes} \varphi$.*
4. *If $(\mathcal{T}, \Pi, 0) \not\models_k^{hopt} \varphi$, then $(\mathcal{T}, \Pi, 0) \not\models_j^{hopt} \varphi$.*

In turn, the verdict obtained from the exploration up-to k can (in some cases) be used to infer the verdict of the model checking problem. As in classical BMC, if the pessimistic semantics find a model, then it is indeed a model. Similarly, if the optimistic semantics fail to find a model, then there is no model. The next lemma formally captures this intuition.

Lemma 2 (Infinite inference). *The following hold for every k,*

1. *If $(\mathcal{T}, \Pi, 0) \models_k^{pes} \varphi$, then $(\mathcal{T}, \Pi, 0) \models \varphi$.*
2. *If $(\mathcal{T}, \Pi, 0) \not\models_k^{opt} \varphi$, then $(\mathcal{T}, \Pi, 0) \not\models \varphi$.*
3. *If $(\mathcal{T}, \Pi, 0) \models_k^{hpes} \varphi$, then $(\mathcal{T}, \Pi, 0) \models \varphi$.*
4. *If $(\mathcal{T}, \Pi, 0) \not\models_k^{hopt} \varphi$, then $(\mathcal{T}, \Pi, 0) \not\models \varphi$.*

3.3 Examples

Consider the Kripke structure in Fig. 1, bound $k = 3$, and formula

$$\varphi_1 = \forall \pi_A . \exists \pi_B . \big((p_{\pi_A} \not\leftrightarrow p_{\pi_B}) \, \mathcal{R} \, \neg q_{\pi_A} \big)$$

It is easy to see that instantiating π_A with trace $s_0 s_1 s_2 s_4$ is a trace π_A of the negation, $\neg \varphi_1$ as follows, in the pessimistic semantics.

$$\neg \varphi_1 = \exists \pi_A . \forall \pi_B . \big((a_{\pi_A} \leftrightarrow p_{\pi_B}) \, \mathcal{U} \, q_{\pi_A} \big)$$

By Lemma 2, this counterexample shows that the kripke structure is a model of $\neg \varphi_1$ in the infinite semantics as well. That is, $K \models_3^{pes} \neg \varphi_1$ and, hence, $K \models \neg \varphi_1$, so $K \not\models \varphi_1$.

Consider again the same Kripke structure, bound $k = 3$, and formula

$$\varphi_2 = \forall \pi_A . \exists \pi_B . \Diamond (p_{\pi_A} \leftrightarrow q_{\pi_B})$$

To disprove φ_2, we need to find a trace π_A such that for all other π_B, proposition q in π_B always disagrees with p in π_A, as the following formula,

$$\neg \varphi_2 = \exists \pi_A . \forall \pi_B . \Box (p_{\pi_A} \not\leftrightarrow q_{\pi_B})$$

It is straightforward to observe that such a trace π_A does not exist. By Lemma 2, proving the formula is not satisfiable upto bound 3 in the optimistic semantics implies that K is not a model of $\neg \varphi_2$ in the infinite semantics. That is, $K \not\models_3^{opt} \neg \varphi_2$ implies $K \not\models \neg \varphi_2$. Hence, we conclude $K \models \varphi_2$.

Consider again the same Kripke structure which has two terminating states, s_3 and s_4, labeled by atomic proposition *halt* with only a self-loop. Let $k = 3$, and formula,

$$\varphi_3 = \forall \pi_A. \exists \pi_B. (\neg q_{\pi_B} \, \mathcal{U} \, \neg p_{\pi_A})$$

To disprove, we want to find a trace π_A that fulfills the negation,

$$\neg \varphi_3 = \exists \pi_A. \forall \pi_B. (q_{\pi_B} \, \mathcal{R} \, p_{\pi_A})$$

Take the halting state in to consideration, $s_0 s_1 s_3$ is a trace of the form $\{p\}^\omega$. It satisfies the halting optimistic semantic of $\mathcal{R}$ in s_3 because of the halting condition. By Lemma 2, the fulfillment of formula implies that in infinite semantics it will be fulfilled as well. That is, $K \models_3^{hpes} \neg \varphi_3$ implies $K \models \neg \varphi_3$. Hence, $K \not\models \varphi_3$.

Consider again the same Kripke structure with halting states and formula,

$$\varphi_4 = \forall \pi_A. \exists \pi_B. \Diamond \Box (p_{\pi_A} \not\leftrightarrow p_{\pi_B})$$

A counterexample is an instantiation of π_A such that for all π_B, both traces will always eventually agree on p as follows,

$$\neg \varphi_4 = \exists \pi_A. \forall \pi_B. \Box \Diamond (p_{\pi_A} \leftrightarrow p_{\pi_B})$$

Trace $s_0 s_1 s_2 s_4$ is of the form $\{p\}\{p\}\{p\}\{r, halt\}^\omega$ with $k = 3$. This trace never agrees with a trace that ends in state s_3 (which is of the form $\{p\}^\omega$) and vice versa. By Lemma 2, the absence of counterexample upto bound 3 in the halting optimistic semantics implies that K is not a model of $\neg \varphi_4$ in the infinite semantics. That is, $K \not\models_3^{hopt} \neg \varphi_4$ implies $K \not\models \neg \varphi_4$. Hence, we conclude $K \models \varphi_4$.

4 Reducing BMC to QBF Solving

We describe in this section (1) how to generate a QBF query from an instance of the model checking problem, and (2) what can be inferred in each case from the outcome of the QBF solver about the model checking problem.

4.1 QBF-based Solution

Given a family of Kripke structures $\mathcal{K}$, a HyperLTL formula φ, and bound $k \geq 0$, our goal is to construct a quantified Boolean formula $[\![\mathcal{K}, \varphi]\!]_k$ whose satisfiability can be used to infer whether or not $\mathcal{K} \models \varphi$. We first describe how to encode the model and the formula, and then how to combine the two to generate the QBF query.

Encoding the models. The unrolling of the transition relation of a Kripke structure $K_A = \langle S, S_{init}, \delta, L \rangle$ up to bound k is analogous to the BMC encoding for LTL [8]. First, note that the state space S can be encoded with a (logarithmic) number of bits in $|S|$. We introduce additional variables $n_0, n_1, \ldots$ to encode

the state of the Kripke structure and use $\mathsf{AP}^* = \mathsf{AP} \cup \{n_0, n_1, \ldots\}$ for the extended alphabet that includes the encoding S. In this manner, the set of initial states of a Kripke structure is a Boolean formula over AP^*. For example, for the Kripke structure K_A in Fig. 1 the set of initial states (in this case $S_{init} = \{s_0\}$) corresponds to the following Boolean formula:

$$I_A := (\neg n_0 \wedge \neg n_1 \wedge \neg n_2) \wedge p \wedge \neg q \wedge \neg halt$$

assuming that $(\neg n_0 \wedge \neg n_1 \wedge \neg n_2)$ encodes state s_0 (we need three bits to encode five states.) Similarly R_A is a binary relation that encodes the transition relation δ of K_A (encoding the relation between a state and its successor). The encoding into QBF works by introducing fresh Boolean variables (a new copy of AP^* for each Kripke structure K_A and position), and then producing a Boolean formula that encodes the unrolling up-to k. We use x_A^i for the set of fresh copies of the variables AP^* of K_A corresponding to position $i \in [0, k]$. Therefore, there are $k|x_A| = k|\mathsf{AP}_A^*|$ Boolean variables to represent the unrolling of K_A. We use $I_A(x)$ for the Boolean formula (using variables from x) that encodes the initial states, and $R_A(x, x')$ (for two copies of the variables x and x') for the Boolean formula whether x' encodes a successor states of x.

For example, for bound $k = 3$, we unroll the transition relation up-to 3 as follows,

$$[\![K_A]\!]_3 = I_A(x_A^0) \wedge R_A(x_A^0, x_A^1) \wedge R(x_A^1, x_A^2) \wedge R(x_A^2, x_A^3)$$

which is the Boolean formula representing valid traces of length 4, using four copies of the variables AP_A^* that represent the Kripke structure K_A.

Encoding the inner LTL formula. The idea of the construction of the inner LTL formula is analogous to the standard BMC as well, except for the choice of different semantics described in Section 3. In particular, we introduce the following inductive construction and define four different unrollings for a given k: $[\![\cdot]\!]_{i,k}^{pes}$, $[\![\cdot]\!]_{i,k}^{opt}$, $[\![\cdot]\!]_{i,k}^{hpes}$, and $[\![\cdot]\!]_{i,k}^{hopt}$.

– **Inductive Case**: Since the semantics only differ on the temporal operators at the end of the unrolling, the inductive case is common to all unrollings and we use $[\![\cdot]\!]_{i,k}^*$ to mean any of the choices of semantic (for $* = pes, opt, hpes, hopt$). For all $i \leq k$:

$$
\begin{aligned}
[\![p_\pi]\!]_{k,i}^* &:= p_\pi^i \\
[\![\neg p_\pi]\!]_{k,i}^* &:= \neg p_\pi^i \\
[\![\psi_1 \vee \psi_2]\!]_{k,i}^* &:= [\![\psi_1]\!]_{k,i}^* \vee [\![\psi_2]\!]_{k,i}^* \\
[\![\psi_1 \wedge \psi_2]\!]_{k,i}^* &:= [\![\psi_1]\!]_{k,i}^* \wedge [\![\psi_2]\!]_{k,i}^* \\
[\![\psi_1 \, \mathcal{U} \, \psi_2]\!]_{k,i}^* &:= [\![\psi_2]\!]_{k,i}^* \vee \left([\![\psi_1]\!]_{k,i}^* \wedge [\![\psi_1 \, \mathcal{U} \, \psi_2]\!]_{k,i+1}^* \right) \\
[\![\psi_1 \, \mathcal{R} \, \psi_2]\!]_{k,i}^* &:= [\![\psi_2]\!]_{k,i}^* \wedge \left([\![\psi_1]\!]_{k,i}^* \vee [\![\psi_1 \, \mathcal{R} \, \psi_2]\!]_{k,i+1}^* \right) \\
[\![\bigcirc \psi]\!]_{k,i}^* &:= [\![\psi]\!]_{k,i+1}^*
\end{aligned}
$$

Note that, for a given path variable π_A, the atom $p_{\pi_A}^i$ that results from $[\![p_\pi]\!]_{k,i}^*$ is one of the Boolean variables in x_A^i.

– For the **base case**, the formula generate is different depending on the intended semantics:

$$\llbracket\psi\rrbracket^{pes}_{k,k+1} := \mathsf{false} \qquad\qquad \llbracket\psi\rrbracket^{opt}_{k,k+1} := \mathsf{true}$$
$$\llbracket\psi\rrbracket^{hpes}_{k,k+1} := \llbracket halted\rrbracket^{hpes}_{k,k} \wedge \llbracket\psi\rrbracket^{hpes}_{k,k} \qquad \llbracket\psi\rrbracket^{hopt}_{k,k+1} := \llbracket halted\rrbracket^{hopt}_{k,k} \to \llbracket\psi\rrbracket^{hopt}_{k,k}$$

Note that the base case defines the value to be assumed for the formula after the end k of the unrolling, which is spawned in the temporal operators in the inductive case at k. The pessimistic semantics assume the formula to be false, and the optimistic semantics assume the formula to be true. The halting cases consider the case at which the traces have halted (using in this case the evaluation at k) and using the unhalting choice otherwise.

Combining the encodings. Now, let φ be a HyperLTL formula of the form $\varphi = \mathbb{Q}_A\pi_A.\mathbb{Q}_B\pi_B.\dots.\mathbb{Q}_Z\pi_Z.\psi$ and $\mathcal{K} = \langle K_A, K_B, \dots, K_Z\rangle$. Combining all the components, the encoding of the HyperLTL BMC problem in QBF is the following (for $* = pes, opt, hpes, hopt$):

$$\llbracket\mathcal{K}, \varphi\rrbracket^*_k = \mathbb{Q}_A\overline{x_A}.\mathbb{Q}_B\overline{x_B}\cdots.\mathbb{Q}_Z\overline{x_Z}\Big(\llbracket K_A\rrbracket_k \circ_A \llbracket K_B\rrbracket_k \circ_B \cdots \llbracket K_Z\rrbracket_k \circ_Z \llbracket\psi\rrbracket^*_{0,k}\Big)$$

where $\llbracket\psi\rrbracket^*_{0,k}$ is the choice of semantics and, $\circ_j = \wedge$ if $\mathbb{Q}_j = \exists$ and $\circ_j = \to$ if $\mathbb{Q}_j = \forall$, for $j \in Vars(\varphi)$.

Example. Consider formula φ_1 in Section 3.3, whose negation is the following:

$$\neg\varphi_1 := \exists\pi_A.\forall\pi_B.\underbrace{\big((p_{\pi_A} \leftrightarrow p_{\pi_B})\, \mathcal{U}\, q_{\pi_A}\big)}_{\neg\psi}$$

The unrolling of $\neg\psi$ using the pessimistic semantics is

$$\llbracket\neg\psi\rrbracket^{pes}_{0,3} = \llbracket\big((p_{\pi_A} \leftrightarrow p_{\pi_B})\, \mathcal{U}\, q_{\pi_A}\big)\rrbracket^{pes}_{0,3} =$$
$$= q^0_{\pi_A} \vee \Big((p^0_{\pi_A} \leftrightarrow p^0_{\pi_B}) \wedge \Big(q^1_{\pi_A} \vee \Big((p^1_{\pi_A} \leftrightarrow p^1_{\pi_B}) \wedge \Big(q^2_{\pi_A} \vee \Big((p^2_{\pi_A} \leftrightarrow p^2_{\pi_B}) \wedge \Big(q^3_{\pi_A}\Big)\Big)\Big)\Big)\Big)\Big)$$

Note that in the final encoding, for example the collection x^2_A, contains all variables of AP^* of K_A (for example, $p^2_{\pi_A}$) connecting to the corresponding valuation for p_{π_A} in the trace of K_A at step 2 in the unrolling of K_A. In other words, the formula $\llbracket\neg\psi\rrbracket^{pes}_{0,3}$ uses variables from $x^0_A, x^1_A, x^2_A, x^3_A$ and $x^0_B, x^1_B, x^2_B, x^3_B$ (that is, from $\overline{x_A}$ and $\overline{x_B}$). To combine the model description with the encoding of the HyperLTL formula, we use two identical copies of the given Kripke structure to represent different paths π_A and π_B on the model, denoted as K_A and K_B. The resulting formula is:

$$\llbracket\mathcal{K}, \neg\varphi\rrbracket_3 := \exists\overline{x_A}.\forall\overline{x_B}.\big(\llbracket K_A\rrbracket_3 \wedge (\llbracket K_B\rrbracket_3 \to \llbracket\neg\varphi\rrbracket^{pes}_{0,3})\big)$$

The sequence of assignment $\{(\neg n_2, \neg n_1, \neg n_0, p, \neg q)^0,\ (\neg n_2, \neg n_1, n_0, p, \neg q)^1,$ $(\neg n_2, n_1, \neg n_0, p, \neg q)^2,\ (n_2, \neg n_1, \neg n_0, \neg p, q)^3\}$ on K_A, corresponding to the trace $s_0 s_1 s_2 s_4$, satisfies $[\![\neg\varphi]\!]_{0,3}^{pes}$ for all traces on K_B. The satisfaction results shows that $[\![\mathcal{K}, \neg\varphi]\!]_3^{pes}$ is true, indicating that a witness of violation is found. Theorem 1, by a successful detection of a counterexample witness, and the use of the pessimistic semantics, allows to conclude that $\mathcal{K} \not\models \varphi$. $\qquad\square$

4.2 Soundness Results

Lemma 3. *Let φ be a closed HyperLTL formula and $\mathcal{T} = Traces(\mathcal{K})$ be an interpretation. For $* = pes, opt, hpes, hopt$, it holds that*

$$[\![\mathcal{K}, \varphi]\!]_k^* \text{ is satisfiable if and only if } (\mathcal{T}, \Pi_\emptyset, 0) \models_k^* \varphi.$$

Proof (sketch). The proof proceeds in two steps. First, let ψ be the largest quantifier-free sub-formula of φ. Then, every tuple of traces of length k (one for each π) is in one to one correspondence with the collection of variables p_π^i, that satisfies that the tuple is a model of ψ (in the choice semantics) if and only if the corresponding assigment makes $[\![\psi]\!]_0^*$. Then, the second part shows inductively in the stack of quantifiers that each subformula obtained by adding a quantifier is satisfiable if and only the semantics hold. $\qquad\square$

Lemma 3, together with Lemma 2, allows to infer the outcome of the model checking problem from satisfying (or unsatisfying) instances of QBF queries, summarized in the following theorem.

Theorem 1. *Let φ be a HyperLTL formula. Then,*
1. *For $* = pes, hpes$, if $[\![\mathcal{K}, \neg\varphi]\!]_k^*$ is satisfiable then $\quad \mathcal{K} \not\models \varphi$.*
2. *For $* = opt, hopt$, if $[\![\mathcal{K}, \neg\varphi]\!]_k^*$ is unsatisfiable then $\mathcal{K} \models \varphi$.*

Example. Finally, we make the connection between satisfiability of QBF and the infinite semantics of the examples in Section 3.3 using Theorem 1. Table 1 illustrates what the different semantics allows to soundly conclude.

5 Descriptions of Case Studies

In this section, we introduce a rich set of case studies to verify and falsify hyperproperties for different systems. These include proving symmetry of the Bakery Algorithm mutual exclusion protocol, linearizability of the SNARK algorithm, non-interference in multi-threaded programs and fairness in non-repudiation protocols. [12, 14, 30, 31] We also show to strategies can be synthesized for robotic planning and mutation testing using our QBF encoding [15, 34].

Formula	Bound	Semantics		
		pessimistic	*optimistic*	*halting*
φ_1	$k = 2$	UNSAT (inconclusive)	SAT (inconclusive)	UNSAT (inconclusive)
	$k = 3$	SAT (***counterexample***)	SAT (inconclusive)	UNSAT (inconclusive)
φ_2	$k = 2$	UNSAT (inconclusive)	SAT (inconclusive)	UNSAT (inconclusive)
	$k = 3$	UNSAT (inconclusive)	UNSAT (***proved***)	UNSAT (inconclusive)
φ_3	$k = 2$	UNSAT (inconclusive)	UNSAT (inconclusive)	non-halted (inconclusive)
	$k = 3$	UNSAT (inconclusive)	UNSAT (inconclusive)	halted (***counterexample***)
φ_4	$k = 2$	UNSAT (inconclusive)	UNSAT (inconclusive)	non-halted (inconclusive)
	$k = 3$	UNSAT (inconclusive)	UNSAT (inconclusive)	halted (***proved***)

Table 1: Comparison of Properties with Different Semantics

5.1 Case study 1: Symmetry in the Bakery Algorithm

We first investigate the symmetry property in Lamport's Bakery algorithm for enforcing mutual exclusion in a concurrent program. [12] The Bakery algorithm works as follows. When a process p intends to enter the critical section, p draws a "ticket" modeled by a number. When more than one process attempt to enter the critical section, the process with the smallest ticket number enters first, while other processes wait. In a concurrent program, it is also possible that two or more processes hold tickets with same number if they drew tickets simultaneously. To solve this tie, when processes with the same ticket try to access the critical section, the process with smaller process ID enters first while the other processes wait. The Bakery algorithm is shown in Algorithm 1.

Algorithm 1: Bakery

---

1 init(MAX/ $P_0.ticket...P_n.ticket$/ $P_0.status...P_n.status$):= 0/ 0...0/ noncrit...noncrit ;
2 **while** *true* **do**
3   **foreach** *i in 0...n* **do**
4   **if** *select(P_i)* **then**
5 $P_i.ticket = $ MAX $+ 1$;
6 $P_i.status = $ waiting;
7   **else if** $P_i.status = wait$ **then**
8   **if** $P_i.ticket = min(P_0.ticket... P_n.ticket)$ **then**
9 $P_i.status = $ crit ;
10   **else**
11 $P_i.status = $ waiting ;
12   **end**
13   **end**
14 **end**

We are interested in studying the symmetry property, which informally states that no specific process has special privileges in terms of a faster access to the critical section. We use the atomic proposition *select* to represent the process selected to proceed in the next state, and *pause* to indicate if the processes are both not moving. Each process P_n has a program counter denoted by $pc(P_n)$. The symmetry property for the Bakery algorithm is formally express as follows. For all traces π_A, there exists a trace π_B, such that if both traces at every step select the next process to execute symmetrically, then the program counter of each process would be completely symmetric as well. For example, consider two processes P_0 and P_1 and let trace π_A select P_0 iff trace π_B selects P_1, and π_A select P_1 iff π_B selects P_0. Such a dual choice of selection is presented as $sym(select_{\pi_A}, select_{\pi_B})$. We are ready to describe the symmetry property as the following HyperLTL formula:

Symmetry	$\varphi_{sym} = \forall \pi_A.\exists \pi_B.\ \Box \Big(sym(select_{\pi_A}, select_{\pi_B}) \wedge (pause_{\pi_A} = pause_{\pi_B}) \wedge$ $\big(pc(P_0)_{\pi_A} = pc(P_1)_{\pi_B}\big) \wedge \big(pc(P_1)_{\pi_A} = pc(P_0)_{\pi_B}\big)\Big)$

5.2 Case study 2: Linearizability of the SNARK Algorithm

Next, we investigate whether the SNARK algorithm [14] satisfies the linearizability property.

Linearizability is a correctness property of concurrent libraries or datatypes [29]. The *history* of the execution of a concurrent datatype, is the sequence of method *invocations* by the different threads and the *response* observed. A *history* is *linearizable*, if there exists a sequential order of invocations and responses, such that the same responses could be produced with atomic executions of the methods invoked. A concurrent datatype is linearizable if all possible histories are linearizable. In [5], the authors show that linearizability is a hyperproperty of the form $\forall \exists$, where the domain of the universal quantifier ranges over all possible executions of the concurrent data structure and the domain of the existential quantifier ranges over all possible executions of a sequential implementation of the data structure (or over the sequential reference implementation or declarative specification of the datatype). Thus, reasoning about linearizability requires our multi-model semantics introduced in Section 2.

The SNARK algorithm [14] is a concurrent implementation of a double-ended queue data structure (the pseudo-code is shown in Algorithm 2). It uses double-compare-and-swap (DCAS) with doubly linked-list that stores values in nodes while each node is connected to its two neighbors, L and R. When a modification of data happens, e.g., by invoking pushRight() or popLeft(), SNARK performs a DCAS by comparing two memory locations to decide if such modification is appropriate.

We define linearizability as a hyperproperty using two different models. Let π_A denote the trace variable over the traces of the *concurrent program* (in this case SNARK). This program is created by allows multiple to execute each method with interleavings. Let π_B represents the trace variable over traces of

17

Algorithm 2: SNARK

---

1 `popRight()`
2 **while** *true* **do**
3 | $rh = RightHat$;
4 | $lh = LeftHat$;
5 | **if** $rh{\rightarrow}R = rh$ **then**
6 | | return "empty";
7 | **end**
8 | **if** $rh = lh$ **then**
9 | | **if** *DCAS(&RightHat, &LeftHat, rh, lh, Dummy, Dummy)* **then**
10 | | | return $rh \rightarrow V$;
11 | | **end**
12 | **else**
13 | | rhL = rh→L;
14 | | **if** *DCAS(&RightHat, &rh→L, rh, rhL, rhL, rh)* **then**
15 | | | $result = rh{\rightarrow}V$;
16 | | | $rh{\rightarrow}R = Dummy$;
17 | | | return $result$;
18 | | **end**
19 **end**
20 `pushRight()`
21 nd = new Node();
22 **if** $nd = null$ **then**
23 | return "full";
24 **end**
25 $nd{\rightarrow}R = Dummy$;
26 $nd{\rightarrow}V = v$;
27 **while** *true* **do**
28 | $rh = RightHat$s;
29 | $rhR = rh{\rightarrow}R$;
30 | **if** $rhR = rh$ **then**
31 | | $nd \rightarrow L = Dummy$;
32 | | $lh = LeftHat$;
33 | | **if** *DCAS(&RightHat, &LeftHat, rh, lh, nd, Dummy)* **then**
34 | | | return success;
35 | | **end**
36 | **else**
37 | | $nd \rightarrow L = rh$;
38 | | **if** *DCAS(&RightHat, &lh→R, rh, rhR, nd, nd)* **then**
39 | | | return success;
40 | | **end**
41 **end**

---

the sequential implementation of a double-ended queue (i.e., the specification), where only atomic invocations are allowed. The HyperLTL formula that specifies linearizability is:

| Linearizability | $\varphi_{lin} = \forall \pi_A.\exists \pi_B.\ \Box(history_{\pi_A} \leftrightarrow history_{\pi_B})$ |

5.3 Case study 3: Non-interference in Typed Multi-threaded Programs

We also investigate *non-interference* in a multi-threaded program with type system. Non-interference is a security policy that states that low-security variables are independent from the high-security variables, thus, preserving secure information flow. Each variable is labeled as a *high-variable* (high security) or *low-variable* (low security). Non-interference requires that all information about a high-variable cannot be inferred by observing any the values of a low-variable. In this case study, we look at a concurrent system example from [31], which contains three threads α, β, and γ. The variables are assigned with different security level as follows: *PIN*, *trigger0*, and *trigger1* are as high-variables, and *maintrigger*, *mask*, and *result* are low-variables.

Assuming that thread scheduling is fair, the program satisfies non-interference, if for all executions, there exists another execution that starts from a different high-inputs (i.e., the values of *PIN* are not equal) and at termination point, they are in low-equivalent states (i.e., the values of *Result* are equal). Furthermore, in order to search for a witness of non-interference violation in bounded time, we also consider *halting* as introduced in Section 3. In this particular program, the execution terminates when the low-variable *MASK* contains value zero. The corresponding HyperLTL formula is:

| NI | $\begin{aligned} \varphi_{NI} = \forall \pi_A.\exists \pi_B.\big(PIN_{\pi_A} \neq PIN_{\pi_B}\big) \wedge \Big((\neg halt_{\pi_A} \vee \neg halt_{\pi_B}) \\ \mathcal{U}\ \big((halt_{\pi_A} \wedge halt_{\pi_B}) \wedge (Result_{\pi_A} = Result_{\pi_B})\big)\Big) \end{aligned}$ |

where atomic proposition *halt* denotes the halting state (*MASK* contains a zero bit) and by abuse of notation PIN_π (respectively, $Result_\pi$) denotes the value of *PIN* (respectively, *Result*) in trace π.

5.4 Case study 4: Fairness in Non-repudiation Protocols

A non-repudiation protocol consists of three parties: a message sender (P), a message receiver (Q), and a *trusted third party* T. In a message exchange event, the message receiver should obtain a receipt from the sender, named *non-repudiation of origin* (NRO), and the message sender should end up having an evidence named *non-repudiation of receipt* (NRR). The three participants can take the following actions:

$$Act_P = \{P \to Q : m,\ P \to T : m,\ P \to Q : NRO, P \to T : NRO,\ P : skip\}$$
$$Act_Q = \{Q \to P : NRR,\ Q \to T : NRR,\ Q : skip\}$$
$$Act_T = \{T \to P : NRR,\ T \to Q : NRO,\ T : skip\}$$

Algorithm 3: Typed Multi-threaded Program

---

1 Thread α:
2 **while** *mask != 0* **do**
3   **while** *trigger0 = 0* **do**
4 | no-op;
5   **end**
6   *result = result ‖ mask* ; // bitwise 'or'
7   *trigger0 = 0*;
8   *maintrigger = matintrigger + 1* ;
9   **if** *maintrigger = 1* **then**
10 | *trigger1 = 1*;
11   **end**
12 **end**
13 Thread β:
14 **while** *mask != 0* **do**
15   **while** *trigger1 = 0* **do**
16 | no-op;
17   **end**
18   *result = result & !mask* ; // bitwise 'and'
19   *trigger1 = 0*;
20   *maintrigger = matintrigger + 1* ;
21   **if** *maintrigger = 1* **then**
22 | *trigger0 = 1*;
23   **end**
24 **end**
25 Thread γ:
26 **while** *mask != 0* **do**
27   *maintrigger = 0* ;
28   **if** *PIN & mask = 0* **then**
29 | *trigger0 = 1*;
30   **else**
31 | *trigger1 = 1*;
32   **end**
33   **while** *maintrigger != 2* **do**
34 | no-op;
35   **end**
36   *mask = mask/2*;
37 **end**
38 *trigger0 = 1*;
39 *trigger1 = 1*;

In this case study, we evaluate two different models of trusted third party from [30]. First, we pick an incorrect implementation from [30], named $T_{incorrect}$, which Q can choose not to send out *NRR* after receiving *NRO*. We also consider a correct implementation of the protocol. Both versions are show in Alg. 4.

Algorithm 4: Non-repudiation Protocol

---

1 $T_{correct}$:	**8** $T_{incorrect}$:
2 (1) *skip* until $P{\rightarrow}T$: *m* ;	**9** (1) *skip* until $P{\rightarrow}T$: *m* ;
3 (2) *skip* until $P{\rightarrow}T$: *NRO* ;	**10** (2) *skip* until $P{\rightarrow}T$: *NRO* ;
4 (3) $T{\rightarrow}Q$: *m* ;	**11** (3) $T{\rightarrow}Q$:*m* ;
5 (4) *skip* until Q${\rightarrow}T$: *NRR*;	**12** (4) $T{\rightarrow}Q$:*NRO*;
6 (5) $T{\rightarrow}Q$: *NRO*;	**13** (5) *skip* until $Q{\rightarrow}T$: *NRR*s;
7 (6) $T{\rightarrow}P$: *NRR*;	**14** (6) $T{\rightarrow}P$:*NRR*;

A *fair* non-repudiation protocol guarantees that two parties can exchange messages fairly without any party being able to deny sending out evidence while having received an evidence. Furthermore, we say that a trace is *effective* if *message*, *NRR*, and *NRO* are all received. Assuming that each party will take turns and take different actions, the fairness of non-repudiation protocol can be defined as a hyperproperty as follows. There exists an *effective* trace π_A, such that for all other traces π_B, if P in both traces always take the same action while Q behave arbitrarily, or both Q take the same action and P behave arbitrarily, then for π_B, eventually *NRR* gets received by P if and only if *NRO* gets received by Q.

The complete specification for non-repudiation is the following:

Fairness	$\varphi_{fair} = \exists\pi_A.\forall\pi_B.\,(\Diamond m_{\pi_A}) \wedge (\Diamond NRR_{\pi_A}) \wedge (\Diamond NRO_{\pi_A}) \wedge$ $\left((\Box\bigwedge_{act\in Act_P} act_{\pi_A} \leftrightarrow act_{\pi_B}) \rightarrow \left((\Diamond NRR_{\pi_B}) \leftrightarrow (\Diamond NRO_{\pi_B})\right)\right) \wedge$ $\left((\Box\bigwedge_{act\in Act_Q} act_{\pi_A} \leftrightarrow act_{\pi_B}) \rightarrow \left((\Diamond NRR_{\pi_B}) \leftrightarrow (\Diamond NRO_{\pi_B})\right)\right)$

Observe that trace π_A expresses effectiveness (i.e., an honest behavior of all parties), while trace π_B is a trace that behaves similarly to trace π_A as far as the actions of P or Q are concerned while ensuring fair receipt of *NRR* and *NRO*.

5.5   Case study 5: Privacy-Preserving Path Planning for Robots

In addition to model checking problems, inspired by the work in [34], we explore other applications of our QBF encoding that also involve hyperproperties with quantifier alternation. One such application is searching the optimal solution for robotic planning. For example, given a 2-D grid with an initial state and a goal state, a shortest path from initial state to goal state is a trace π_A, such that π_A reaches the goal state and for all other traces π_B, π_B has not reached the goal state before π_A has. In other words, the shortest path is a path on the grid that reaches the goal state before all other paths. We express this specification as the following hyperproperty:

Shortest Path	$\varphi_{sp} = \exists\pi_A.\forall\pi_B.(\neg goal_{\pi_B}\; \mathcal{U}\; goal_{\pi_A})$

where the atomic proposition *goal* denotes that the path has reached the goal state.

To further analyze the result, we also consider that traces halt. An optimal path searching should terminate when the shortest path is found because when a shortest path has been discovered on the map, any further exploration will not affect the outcome .

Besides optimal solution searching, HyperLTL also allows us to specify the *robustness* of paths that are derived by uncertainty in robotic planning. For example, instead of one single initial state, we now consider a map with a set of initial states. We are interested in a strategy that can help all traces to reach the goal state regardless of which initial state the path start from. The robust strategy searching problem can be presented as follows. There exists a robust path π_A, such that for all paths π_B starting from as arbitrary state from the set of initial states, π_B is able to reach the goal state using the same strategy as π_A. We use the proposition *strategy* to represent the sequence of movements the path takes. We write the formula as follows:

Robustness	$\varphi_{rb} = \exists \pi_A. \forall \pi_B.\ (strategy_{\pi_B} \leftrightarrow strategy_{\pi_A})\ \mathcal{U}\ (goal_{\pi_A} \wedge goal_{\pi_B})$

5.6 Case study 6: Generate Mutants in Mutation Testing

Another application of hyperproperty with quantifier alternation is the efficient generation of test suites for mutation testing. We look at the beverage machine model from [15]. The beverage machine has three possible inputs: *request*, *fill*, or *none*. Based on the input, the machine may output *coffee*, *tea*, or *none*. We also use an atomic proposition *mut* to mark mutated traces, and $\neg mut$ for non-mutated traces. In this non-deterministic model, a potentially killable mutant is a trace (mutated) trace π_A such that, for all other (non-mutated) π_B, if they have same inputs as π_A, then the outputs eventually diverge.

Mutant in Non-det Model	$\exists \pi_A \forall \pi_B (mut_{\pi_A} \wedge \neg mut_{\pi_B}) \wedge \big((in_{\pi_A} \leftrightarrow in_{\pi_B})\ \mathcal{U}\ (out_{\pi_A} \not\leftrightarrow out_{\pi_B})\big)$

6 Implementation and Empirical Evaluation

We have implemented the technique described in Section 4 in a tool called HyperQube. In this section, we describe this implementation and the empirical evaluation of the case studies described in Section 5. The tool HyperQube works as follows. Given a transition relation, we automatically unfold it up to a given bound $k \geq 0$ by a procedure *genqbf* using a home-grown tool written in Ocaml.

Given the choice of the semantics (pessimistic, optimistic, h-pessimistic or h-optimistic) the unfolded transition relation is combined with the QBF encoding of the input HyperLTL formula to form a complete QBF instance which is then be fed to the state-of-the-art QBF solver Quabs [28]. All experiments in this section are run on an iMac desktop with Intel i7 CPU @3.4 GHz and 32 GB of RAM.

6.1 Evaluation of Case 1: Symmetry in the Bakery Algorithm

The off-the-self Bakery algorithm described does not satisfy the symmetry property, because when two or more process are intending to enter the critical section with the same tickets number, the algorithm always gives priority to the process with the smaller process ID. We encode the Bakery program as Boolean formulas that encode the initial states and the transition relation. Then, we encoded the negation of the symmetry formula:

$\neg$Symmetry	$\neg\varphi_{sym} = \exists\pi_A.\forall\pi_B.\ \Diamond\Big(\neg sym(select_{\pi_A}, select_{\pi_B}) \vee (pause_{\pi_A} \neq pause_{\pi_B}) \vee$ $(pc(P_0)_{\pi_A} \neq pc(P_1)_{\pi_B}) \vee (pc(P_1)_{\pi_A} \neq pc(P_0)_{\pi_B})\Big)$

HyperQube returns SAT using the *pessimistic* semantics, which indicates that there exists a trace that satisfy $\neg\varphi_{sym}$. The returned trace represents a witness trace of Bakery that violates symmetry and thus falsifies the original formula φ_{sym}.

An observable witness within finite bound is sufficient with the *pessimistic* semantics to infer that all future observations are consistently indicating the given model does not satisfy original property.

6.2 Evaluation of Case 2: Linearizability in SNARK Algorithm

The SNARK algorithm is not linearizable, which means that there is a witness trace that has no sequential equivalent trace. The violation of linearizability can be expressed as xthe negation of the original property, as follows:

$\neg$ Linearizability	$\neg\varphi_{lin} = \exists\pi_A.\forall\pi_B.\ \Diamond(history_{\pi_A} \not\leftrightarrow history_{\pi_B})$

In this case, HyperQube returns SAT using the *pessimistic*semantics, indicating that a witness of linearizability violation has been found. Again, with the use of *pessimistic* semantics, a witness of linearizability violation of length k is enough to infer that the given system does not satisfy the linearizability property. The bug we identified by using HyperQube is the same as the bug trace reported in [14] with an ad-hoc technique.

6.3 Evaluation of Case 3: Non-interference in Typed Multi-threaded Programs

To verify non-interference, we use HyperQube to search for a counterexample exists. We encode the following formula:

$\neg NI$	$\neg\varphi_{NI} = \exists\pi_A.\forall\pi_B.\big(PIN_{\pi_A} \neq PIN_{\pi_B}\big) \rightarrow \Big((terminate_{\pi_A} \wedge terminate_{\pi_B})$ $\mathcal{R}\ \big((\neg terminate_{\pi_A} \vee \neg terminate_{\pi_B}) \vee (Result_{\pi_A} \neq Result_{\pi_B})\big)\Big)$

In this case we use *halting − pessimistic* to further exploit the terminating nature of the system and, HyperQube returns SAT, indicating that there is a trace in which we can detect the difference of high-variable by observing low variable, that is, violating non-interference.

6.4 Evaluation of Case 4: Fairness in Non-repudiation Protocols

In order to handle fairness in non-repudiation protocols we study the negated formula, which is in $\forall\exists$ form against the $T_{incorrect}$ implementation.

$$
\begin{array}{|c|l|}
\hline
\neg \text{ Fairness} &
\begin{aligned}
\neg\varphi_{fair} = &\forall\pi_A.\exists\pi_B.\ \neg\big((\Diamond m_{\pi_A}) \wedge (\Diamond NRR_{\pi_A}) \wedge (\Diamond NRO_{\pi_A})\big)\vee \\
& \Big((\Box\textstyle\bigwedge_{act\in Act_P} act_{\pi_A} \leftrightarrow act_{\pi_B})\ \wedge \neg\big((\Diamond NRR_{\pi_B}) \leftrightarrow (\Diamond NRO_{\pi_B})\big)\Big)\vee \\
& \Big((\Box\textstyle\bigwedge_{act\in Act_Q} act_{\pi_A} \leftrightarrow act_{\pi_B})\ \wedge \neg\big((\Diamond NRR_{\pi_B}) \leftrightarrow (\Diamond NRO_{\pi_B})\big)\Big)
\end{aligned} \\
\hline
\end{array}
$$

We obtain a SAT result from HyperQube, but since the formula passed to the solver is $\forall\exists$ the solver does not return an witness. Alternatively, one could verify the protocol with respect to formula $\exists\pi_A.(\Diamond m_{\pi_A} \wedge \Diamond NRR_{\pi_A} \wedge \Diamond NRO_{\pi_A})$. This step was successful, meaning that an effective trace exists, meaning that the original SAT result implies that the protocol includes an unfair trace.

We then studied the implementation named $T_{correct}$ in [30], where T always guarantees the message exchange event is fair between the two parties. In this case, HyperQube returns UNSAT, which indicates that all traces in the correct system satisfies fairness in non-repudiation. In this case study, both SAT and UNSAT results from HyperQube can be meaningful because of the use of halting semantics (*halting − pessimistic* for falsification of $T_{incorrect}$ and *halting − optimistic* for verification of $T_{correct}$).

6.5 Evaluation of Case 5: Privacy-Preserving Path Planning for Robots

The use of HyperQube for robotic path planning is slightly different from the above-mentioned cases. In this case, we focus on synthesizing a qualified strategy that satisfies the properties described above. Thus, we enforce the original formulas including *shortest path* and *robustness* properties directly with the map model.

– **Shortest path.** By encoding the map grid together with φ_{sp}, HyperQube returns SAT. The returned path as shown in fig. 3 represents a path that can reach the goal from the initial state with the least steps compared to all other paths.
– **Robustness Path.** Encoding the map with , φ_{rb}, HyperQube again returns SAT. This corresponds to a robust strategy, in the sense that all other robots starting from an arbitrary initial state will eventually reach to the goal state by following exactly the same strategy. The result can be visualized in 4
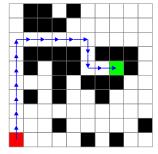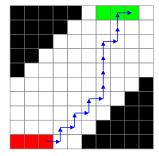
Fig. 3: Shortest Path

Fig. 4: Robust Strategy

We investigate the scalability and performance of our technique for this particular study, in comparison with the technique introduced in [34]. In [34], the paths for robotic planning are synthesized by unfolding the transition relations and properties using python scripts, and solve satisfiability using Z3 SMT solver [13]. The results shown in Table 2 suggest that the QBF-based approach of HyperQube outperforms the solution in [34]—which is based on more mature SMT technology—, on several challenging robotic planning problems. As QBF solvers improve we anticipate that HyperQube will automatically benefit from their improvements.

6.6 Evaluation of Case 6: Generate Mutants in Mutation Testing

We also evaluated HyperQube to synthesize valid mutants for mutation testing as in [15]. We again directly apply the original formula that describes a good mutant together with the model. In this case, HyperQube returns SAT, indicating that we have successfully found a good qualified mutant. Our experiment shows that HyperQube is able to output a mutant with the given formula in a very short amount of time, which provides an efficient solution for test suite generation of mutation testing.

6.7 Summary of Cases Results Evaluations

Table 3 shows the running times of HyperQube in the different case studies.

In Table 4, we separately address how we use Theorem 1 to infer from the output of HyperQube the result of the corresponding model-checking problem.

The results shown in Table 4 (#0.1 to #4.2) illustrate that HyperQube is capable of solving a variety of model checking problems for alternating HyperLTL properties. These instances are very challenging for techniques that attempt reduce to model-checking of LTL because due to the complexity of eliminating the alternation of quantifiers. Additionally, QBF solvers allow to more efficient explore the search space than a brute-force SAT-based approach, where universal and existential quantifiers are eliminated by combinatorial expansion to conjunctions and disjunctions.

		[34]			HyperQube		
	#unroll	gen [s]	Z3 [s]	Total[s]	genqbf [s]	QuAbS [s]	Total [s]
Shortest path ($map\ size : 10^2$)	20	8.31	0.33	**8.64**	1.30	0.57	**1.87**
Shortest path ($map\ size : 20^2$)	40	124.66	6.41	**131.06**	4.53	12.16	**16.69**
Shortest path ($map\ size : 40^2$)	80	1093.12	72.99	**1166.11**	36.04	35.75	**71.79**
Shortest path ($map\ size : 60^2$)	120	4360.75	532.11	**4892.86**	105.82	120.84	**226.66**
Initial state robustness ($map\ size : 10^2$)	20	11.14	0.45	**11.59**	1.40	0.35	**1.75**
Initial state robustness ($map\ size : 20^2$)	40	49.59	2.67	**52.26**	15.92	15.32	**31.14**
Initial state robustness ($map\ size : 40^2$)	80	216.16	19.81	**235.97**	63.16	20.13	**83.29**

Table 2: Case studies results of hyperproperties for robotic planning on larger maps using HyperQube, in comparison with the experimental results in [12]

In cases #5.1 to #6.2, we also demonstrate the ability of HyperQube to solve challenging synthesis problems by leveraging the existential quantifier in a HyperLTL formula as the synthesized result that satisfies the specification.

7 Related Work

There has been a lot of recent progress in automatically verifying [12, 22–24] and monitoring [1, 5, 7, 20, 21, 26, 32] HyperLTL specifications. HyperLTL is also supported by a growing set of tools, including the model checker MCHyper [12, 24], the satisfiability checkers EAHyper [19] and MGHyper [17], and the runtime monitoring tool RVHyper [20].

The complexity of the *model checking* for HyperLTL for tree-shaped, acyclic, and general graphs was rigorously investigated in [2]. The first algorithms for model checking HyperLTL and HyperCTL* using alternating automata were introduced in [24]. These techniques, however, were not able to deal in practice with alternating HyperLTL formulas in a fully automated fashion. We also note that previous approaches that reduce model checking HyperLTL—typically of formulas without quantifier alternations—to model checking LTL can use BMC in the LTL model checking phase. However, this is a completely different approach than the one presented here, as these approaches simply instruct the

#	Model	φ	#unroll	QBF	sems	genqbf [s]	QuAbS [s]	Total [s]
0.1	Bakery.3proc	$\forall\forall\ (sym1)$	7	SAT	*pes*	0.44	0.04	**0.48**
0.2	Bakery.3proc	$\forall\forall\ (sym2)$	12	SAT	*pes*	1.31	0.15	**1.46**
0.3	Bakery.3proc	$\forall\forall\ (sym3)$	20	UNSAT	*opt*	2.86	4.87	**7.73**
1.1	Bakery.3proc	φ_{sym1}	10	SAT	*pes*	0.86	0.11	**0.97**
1.2	Bakery.3proc	φ_{sym2}	10	SAT	*pes*	0.76	0.17	**0.93**
1.3	Bakery.5proc	φ_{sym1}	10	SAT	*pes*	23.57	1.08	**24.65**
1.4	Bakery.5proc	φ_{sym2}	10	SAT	*pes*	29.92	1.43	**31.35**
2.1	SNARK-bug1	φ_{lin}	26	SAT	*pes*	88.42	383.60	**472.02**
2.2	SNARK-bug2	φ_{lin}	40	SAT	*pes*	718.09	779.76	**1497.85**
3.1	3-Thread (*incorrect*)	φ_{NI}	57	SAT	*h-pes*	19.56	46.66	**66.22**
3.2	3-Thread (*correct*)	φ_{NI}	57	UNSAT	*h-opt*	23.91	33.54	**57.45**
4.1	NRP ($T_{incorrect}$)	φ_{fair}	15	SAT	*h-pes*	0.10	0.27	**0.37**
4.2	NRP ($T_{correct}$)	φ_{fair}	15	UNSAT	*h-opt*	0.08	0.12	**0.20**
5.1	Shortest Path	φ_{sp}	20	SAT	*h-pes*	1.30	0.57	**1.87**
5.2	Initial State Robustness	φ_{rb}	20	SAT	*h-pes*	1.40	0.35	**1.75**
6.1	Mutant Synthesis	φ_{mut}	20	SAT	*h-pes*	1.40	0.35	**1.75**

Table 3: Performance of HyperQube in the case studies. Column *case#* identifies the artifact, and the rest of the columns represent the models, properties, number of unrolling in BMC, semantic used for infinite inference, and the running time for generating the query and for solving it.

model checker to use a BMC *after* the problem has beenfully reduced to an LTL model checking problem while we avoid this translation.

These algorithms were then extended to deal with hyperliveness and alternating formulas in [12] by finding a winning strategy in $\forall\exists$ games. In this paper, we take an alternative approach by reducing the model checking problem to QBF solving, which is arguably more effective for finding bugs (in case a finite witness exists).

Semantics	Case#	Property	QBF	Infinite Inference	Conclusion
pessimistic	0.1 0.2 1.1 1.2 1.3 1.4	φ_{sym}	SAT	$\mathcal{K} \models_k^{pes} \neg\varphi$ thus $\mathcal{K} \models \neg\varphi$	$\mathcal{K} \not\models \varphi_{sym}$
	2.1 2.2	φ_{lin}	SAT		$\mathcal{K} \not\models \varphi_{lin}$
optimsitic	0.3	φ_{sym}	UNSAT	$\mathcal{K} \models_k^{opt} \neg\varphi$ thus $\mathcal{K} \not\models \neg\varphi$	$\mathcal{K} \models \varphi_{sym}$
h-pessimistic	3.1	φ_{NI}	SAT	$\mathcal{K} \models_k^{hpes} \neg\varphi$ thus $\mathcal{K} \models \neg\varphi$	$\mathcal{K} \not\models \varphi_{NI}$
	4.1	φ_{fair}	SAT		$\mathcal{K} \not\models \varphi_{fair}$
h-optimsitic	3.2	φ_{NI}	UNSAT	$\mathcal{K} \not\models_k^{hopt} \neg\varphi$ thus $\mathcal{K} \not\models \neg\varphi$	$\mathcal{K} \models \varphi_{NI}$
	4.2	φ_{fair}	UNSAT		$\mathcal{K} \models \varphi_{fair}$
Synthesis (pessimistic)	5.1	φ_{sp}	SAT	$\mathcal{K} \models_k^{pes} \varphi$ thus $\mathcal{K} \models \varphi$	shortest path exists
	5.2	φ_{rb}	SAT		robust path exists
	6.1	φ_{mut}	SAT		mutant synthesized

Table 4: Mappings of cases studies and model checking problem conclusions, with different semantics used for infinite inference from Theorem 1.

The *satisfiability* problem for HyperLTL is shown to be undecidable in general but decidable for the $\exists^*\forall^*$ fragment and for any fragment that includes a $\forall\exists$ quantifier alternation [16]. The hierarchy of hyperlogics beyond HyperLTL were studied in [11]. The synthesis problem for HyperLTL has been studied in problem in [3] in the form of *program repair*, in [4] in the form of *controller synthesis*, and in [18] for the general case.

8 Conclusion and Future Work

In this paper, we introduced the first bounded model checking (BMC) technique for verification of hyperproperties expressed in HyperLTL. To this end, we proposed four different semantics that ensure the soundness of inferring the outcome of the model-checking problem. To handle trace quantification in HyperLTL, we reduced the BMC problem to checking satisfiability of quantified Boolean formulas (QBF). This is analogous to the reduction of BMC for LTL to the simple propositional satisfiability problem. We have introduced different classes of semantics, beyond the pessimistic semantics common in LTL model checking, namely optimistic semantics that allow to infer full verification by observing only

a finite prefix and halting variations of these semantics that additionally exploit the termination of the execution, when available.

Through a rich set of case studies, we demonstrated the effectiveness and efficiency of our approach in verification of information-flow properties, linearizability in concurrent data structures, path planning in robotics, and fairness in non-repudiation protocols.

As for future work, our first step is to solve the loop condition problem. This is necessary to establish completeness conditions for BMC and can help cover even more examples efficiently. The application of QBF-based techniques in the framework of abstraction/refinement is another unexplored area. Success of BMC for hyperproperties inherently depends on effectiveness of QBF solvers. Even though QBF solving is not as mature as SAT/SMT solving techniques, recent breakthroughs on QBF have enabled the construction of HyperQube, and more progress in QBF solving will improve its efficiency.

References

1. S. Agrawal and B. Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *Proceedings of the IEEE 29th Computer Security Foundations (CSF)*, pages 239–252, 2016.
2. B. Bonakdarpour and B. Finkbeiner. The complexity of monitoring hyperproperties. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium CSF*, pages 162–174, 2018.
3. B. Bonakdarpour and B. Finkbeiner. Program repair for hyperproperties. In *Proceedings of the 17th Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 423–441, 2019.
4. B. Bonakdarpour and B. Finkbeiner. Controller synthesis for hyperproperties. In *Proceedings of the 33rd IEEE Computer Security Foundations Symposium (CSF)*, pages 366–379, 2020.
5. B. Bonakdarpour, C. Sánchez, and G. Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In *Proceedings of the 8th Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 8–27, 2018.
6. Borzoo Bonakdarpour, Pavithra Prabhakar, and César Sánchez. Model checking timed hyperproperties in discrete-time systems. In *Proc. of NFM'20*, volume 12229 of *LNCS*, pages 311–328. Springer, 2020.
7. N. Brett, U. Siddique, and B. Bonakdarpour. Rewriting-based runtime verification for alternation-free HyperLTL. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 77–93, 2017.
8. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
9. M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *Proceedings of the 3rd Conference on Principles of Security and Trust POST*, pages 265–284, 2014.
10. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

11. N. Coenen, B. Finkbeiner, C. Hahn, and J. Hofmann. The hierarchy of hyperlogics. In *Proceedings 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2019.

12. N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup. Verifying hyperliveness. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV)*, pages 121–139, 2019.

13. L. de Moura and N. Bjorner. Z3 – a tutorial. Technical report, Microsoft, 2012.

14. S. Doherty, D. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele Jr. DCAS is not a silver bullet for non-blocking algorithm design. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 216–224, 2004.

15. A. Fellner, M. Tabaei Befrouei, and G. Weissenbacher. Mutation testing with hyperproperties. In *Proceedings of the 17th International Conference on Software Engineering and Formal Methods (SEFM)*, pages 203–221. Springer, 2019.

16. B. Finkbeiner and C. Hahn. Deciding hyperproperties. In *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR)*, pages 13:1–13:14, 2016.

17. B. Finkbeiner, C. Hahn, and T. Hans. MGHyper: Checking satisfiability of Hyper-LTL formulas beyond the \exists ^*\forall ^* $\exists_*\forall_*$ fragment. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 521–527, 2018.

18. B. Finkbeiner, C. Hahn, P. Lukert, M. Stenger, and L. Tentrup. Synthesis from hyperproperties. *Acta Informatica*, 57(1-2):137–163, 2020.

19. B. Finkbeiner, C. Hahn, and M. Stenger. Eahyper: Satisfiability, implication, and equivalence checking of hyperproperties. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)*, pages 564–570, 2017.

20. B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. RVHyper: A runtime verification tool for temporal hyperproperties. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 194–200, 2018.

21. B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. Monitoring hyperproperties. *Formal Methods in System Design (FMSD)*, 54(3):336–363, 2019.

22. B. Finkbeiner, C. Hahn, and H. Torfah. Model checking quantitative hyperproperties. In *Proceedings of the 30th International Conference on Computer Aided Verification*, pages 144–163, 2018.

23. B. Finkbeiner, Ch. Müller, H. Seidl, and E. Zalinescu. Verifying Security Policies in Multi-agent Workflows with Loops. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2017.

24. B. Finkbeiner, M. N. Rabe, and C. Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pages 30–48, 2015.

25. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

26. C. Hahn, M. Stenger, and L. Tentrup. Constraint-based monitoring of hyperproperties. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 115–131, 2019.

27. K. Havelund and D. Peled. Runtime verification: From propositional to first-order temporal logic. In *Proceedings of the 18th International Conference on Runtime Verification (RV)*, pages 90–112, 2018.

28. J. Hecking-Harbusch and L. Tentrup. Solving QBF by abstraction. In *Proceedings of the 9th International Symposium on Games, Automata, Logics, and Formal Verification (GandALF)*, volume 277 of *EPTCS*, pages 88–102, 2018.

29. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

30. W. Jamroga, S. Mauw, and M. Melissen. Fairness in non-repudiation protocols. In *Proceedings of the 7th International Workshop on Security and Trust Management (STM)*, volume 7170, pages 122–139. Springer, 2011.

31. G. Smith and D. M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 355–364, 1998.

32. S. Stucki, C. Sánchez, G. Schneider, and B. Bonakdarpour. Graybox monitoring of hyperproperties. In *Proceedings of the 23rd International Symposium on Formal Methods (FM)*, pages 406–424, 2019.

33. Y. Wang, M. Zarei, B. Bonakdarpour, and M. Pajic. Statistical verification of hyperproperties for cyber-physical systems. *ACM Transactions on Embedded Computing systems (TECS)*, 18(5s):92:1–92:23, 2019.

34. S. Nalluri Y. Wang and M. Pajic. Hyperproperties for robotics: Planning via HyperLTL. In *International Conference on Robotics and Automation (ICRA)*, pages 8011–8017, 2019.