Efficient Loop Conditions for Bounded Model Checking Hyperproperties*

Tzu-Han Hsu[1], César Sánchez[2], Sarai Sheinvald[3], and
✉ Borzoo Bonakdarpour[1]

[1] Michigan State University, East Lansing, MI, USA {tzuhan,borzoo}@msu.edu
[2] IMDEA Software Institute, Madrid, Spain cesar.sanchez@imdea.org
[3] Dept. of Software Engineering, Braude College, Israel sarai@braude.ac.il

Abstract. Bounded model checking (BMC) is an effective technique for hunting bugs by incrementally exploring the state space of a system. To reason about infinite traces through a finite structure and to ultimately obtain completeness, BMC incorporates *loop conditions* that revisit previously observed states. This paper focuses on developing loop conditions for BMC of HyperLTL– a temporal logic for hyperproperties that allows expressing important policies for security and consistency in concurrent systems, etc. Loop conditions for HyperLTL are more complicated than for LTL, as different traces may loop inconsistently in unrelated moments. Existing BMC approaches for HyperLTL only considered linear unrollings without any looping capability, which precludes both finding small infinite traces and obtaining a complete technique. We investigate loop conditions for HyperLTL BMC, for HyperLTL formulas that contain up to one quantifier alternation. We first present a general complete automata-based technique which is based on bounds of maximum unrollings. Then, we introduce alternative simulation-based algorithms that allow exploiting short loops effectively, generating SAT queries whose satisfiability guarantees the outcome of the original model checking problem. We also report empirical evaluation of the prototype implementation of our BMC techniques using Z3py.

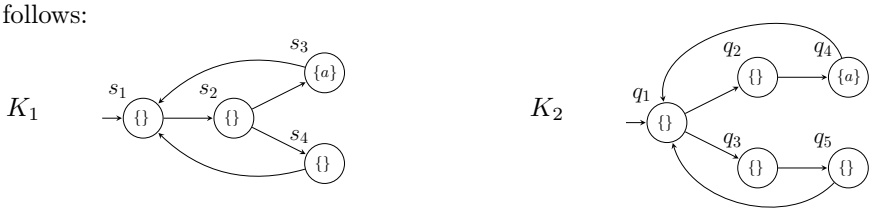
1 Introduction

Hyperproperties [14] have been getting increasing attention due to their power to reason about important specifications such as information-flow security policies that require reasoning about the interrelation among different execution traces. HyperLTL [13] is an extension of the linear-time temporal logic LTL [33] that allows quantification over traces; hence, capable of describing hyperproperties. For example, the security policy *observational determinism* can be specified as

* This research has been partially supported by the United States NSF SaTC Award 2100989, by the Madrid Regional Gov. Project BLOQUES-CM (S2018/TCS-4339), by Project PRODIGY (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the EU NextGenerationEU/PRTR, and by a research grant from Nomadic Labs and the Tezos Foundation.

HyperLTL formula: $\forall\pi.\forall\pi'.(o_\pi \leftrightarrow o_{\pi'})\,\mathcal{W}\,\neg(i_\pi \leftrightarrow i_{\pi'})$, which specifies that for every pair of traces π and π', if they agree on the secret input i, then their public output o must also be observed the same (here '$\mathcal{W}$' denotes the weak until operator).

Several works [15, 24] have studied model checking techniques for HyperLTL specifications, which typically reduce this problem to LTL model checking queries of modified systems. More recently, [29] proposed a QBF-based algorithm for the direct application of bounded model checking (BMC) [12] to HyperLTL, and successfully provided a push-button solution to verify or falsify HyperLTL formulas with an arbitrary number of quantifier alternations. However, unlike the classic BMC for LTL, which included the so-called *loop conditions*, the algorithm in [29] is limited to (non-looping) linear exploration of paths. The reason is that extending path exploration to include loops when dealing with multiple paths simultaneously is not straightforward. For example, consider the HyperLTL formula $\varphi_1 = \forall\pi.\exists\pi'.\ \square(a_\pi \rightarrow b_{\pi'})$ and a pair of Kripke structures K_1 and K_2 as follows:



Assume trace π ranges over K_1 and trace π' ranges over K_2. Proving $\langle K_1, K_2\rangle \not\models \varphi_1$ can be achieved by finding a finite counterexample (i.e., path $s_1s_2s_3$ from K_1). Now, consider $\varphi_2 = \forall\pi.\exists\pi'.\ \square(a_\pi \leftrightarrow a_{\pi'})$. It is easy to see that $\langle K_1, K_2\rangle \models \varphi_2$. However, to prove $\langle K_1, K_2\rangle \models \varphi_2$, one has to show the absence of counterexamples in infinite paths, which is impossible with model unrolling in finite steps as proposed in [29].

In this paper, we propose efficient loop conditions for BMC of hyperproperties. First, using an automata-based method, we show that lasso-shaped traces are sufficient to prove infinite behaviors of traces within finite exploration. However, this technique requires an unrolling bound that renders it impractical. Instead, our efficient algorithms are based on the notion of *simulation* [34] between two systems. Simulation is an important tool in verification, as it is used for abstraction, and preserves ACTL* properties [6, 26]. As opposed to more complex properties such as language containment, simulation is a more local property and is easier to check. The main contribution of this paper is the introduction of practical algorithms that achieve the exploration of infinite paths following a simulation-based approach that is capable of relating the states of multiple models with correct successor relations.

We present two different variants of simulation, $\mathsf{SIM}_{\mathsf{EA}}$ and $\mathsf{SIM}_{\mathsf{AE}}$, allowing to check the satisfaction of $\exists\forall$ and $\forall\exists$ hyperproperties, respectively. These notions circumvent the need to boundlessly unroll traces in both structures and synchronize them. For $\mathsf{SIM}_{\mathsf{AE}}$, in order to resolve non-determinism in the first model, we also present a third variant, where we enhance $\mathsf{SIM}_{\mathsf{AE}}$ by using *prophecy variables* [1, 7]. Prophecy variables allow us to handle cases in which $\forall\exists$ hyperprop-

erties hold despite the lack of a direct simulation. With our simulation-based approach, one can capture infinite behaviors of traces with finite exploration in a simple and concise way. Furthermore, our BMC approach not only model-checks the systems for hyperproperties, but also does so in a way that finds *minimal* witnesses to the simulation (i.e., by partially exploring the existentially quantified model), which we will further demonstrate in our empirical evaluation.

We also design algorithms that generate SAT formulas for each variant (i.e., $\mathsf{SIM}_{\mathsf{EA}}$, $\mathsf{SIM}_{\mathsf{AE}}$, and $\mathsf{SIM}_{\mathsf{AE}}$ with prophecies), where the satisfiability of formulas implies the model checking outcome. We also investigate the practical cases of models with different sizes leading to the eight categories in Table 1. For example, the first row indicates the category of verifying two models of different sizes with the fragment that only allows $\forall\exists$ quantifiers and $\Box$ (i.e., *globally* temporal operator); $\forall_{\texttt{small}}\exists_{\texttt{big}}$ means that the first model is relatively smaller than the second model, and the positive outcome ($\models \forall\exists\Box\varphi$) can be proved by our simulation-based technique $\mathsf{SIM}_{\mathsf{AE}}$, while the negative outcome ($\not\models \forall\exists\Box\varphi$) can be easily checked using non-looping unrolling (i.e., [29]). We will show that in certain cases, one can verify a $\Box$ formula without exploring the entire state space of the `big` model to achieve efficiency.

Case	φ with $\Box$	$\neg\varphi$ with $\Diamond$
$\forall_{\texttt{small}}\ \exists_{\texttt{big}}$	$\mathsf{SIM}_{\mathsf{AE}} \rightarrow \models \forall\exists\Box\varphi$	$\mathsf{BMC} \rightarrow \not\models \forall\exists\Box\varphi$
$\forall_{\texttt{big}}\ \exists_{\texttt{small}}$	$\mathsf{SIM}_{\mathsf{AE}} \rightarrow \models \forall\exists\Box\varphi$	$\mathsf{BMC} \rightarrow \not\models \forall\exists\Box\varphi$
$\exists_{\texttt{small}}\ \forall_{\texttt{big}}$	$\mathsf{SIM}_{\mathsf{EA}} \rightarrow \models \exists\forall\Box\varphi$	$\mathsf{BMC} \rightarrow \not\models \exists\forall\Box\varphi$
$\exists_{\texttt{big}}\ \forall_{\texttt{small}}$	$\mathsf{SIM}_{\mathsf{EA}} \rightarrow \models \exists\forall\Box\varphi$	$\mathsf{BMC} \rightarrow \not\models \exists\forall\Box\varphi$

Table 1: Eight categories of HyperLTL formulas with different forms of quantifiers, sizes of models, and different temporal operators.

We have implemented our algorithms[1] using Z3py, the Z3 [17] API in python. We demonstrate the efficiency of our algorithm exploring a subset of the state space for the larger (i.e., `big`) model. We evaluate the applicability and efficiency with cases including conformance checking for distributed protocol synthesis, model translation, and path planning problems. In summary, we make the following contributions: (1) a bounded model checking algorithm for hyperproperties with loop conditions, (2) three different practical algorithms: $\mathsf{SIM}_{\mathsf{EA}}$, $\mathsf{SIM}_{\mathsf{AE}}$, and $\mathsf{SIM}_{\mathsf{AE}}$ with prophecies, and (3) a demonstration of the efficiency and applicability by case studies that cover through all eight different categories of HyperLTL formulas (see Table 1).

Related Work. Hyperproperties were first introduced by Clarkson and Schneider [14]. HyperLTL was introduced as a temporal logic for hyperproperties in [13]. The first algorithms for model checking HyperLTL were introduced in [24] using alternating automata. Automated reasoning about HyperLTL specifications has received attention in many aspects, including static verification [16, 22–24] and monitoring [2, 9, 11, 20, 21, 28, 35]. This includes tools support, such as MCHyper [16, 24] for model checking, EAHyper [19] and MGHyper [18] for satisfiability

[1] Available at: https://github.com/TART-MSU/loop_condition_tacas23

checking, and RVHyper [20] for runtime monitoring. However, the aforementioned tools are either limited to HyperLTL formulas without quantifier alternations, or requiring additional inputs from the user (e.g., manually added strategies [16]).

Recently, this difficulty of alternating formulas was tackled by the bounded model checker HyperQB [29] using QBF solving. However, HyperQB lacks loop conditions to capture early infinite traces in finite exploration. In this paper, we develop simulation-based algorithms to overcome this limitation. There are alternative approaches to reason about infinite traces, like reasoning about strategies to deal with $\forall\exists$ formulas [15], whose completeness can be obtained by generating a set of prophecy variables [8]. In this work, we capture infinite traces in BMC approach using simulation. We also build an applicable prototype for model-check HyperLTL formulas with models that contain loops.

2 Preliminaries

Kripke structures. A *Kripke structure* K is a tuple $\langle S, S^0, \delta, \mathsf{AP}, L\rangle$, where S is a set of *states*, $S^0 \subseteq S$ is a set of *initial states*, $\delta \subseteq S \times S$ is a total *transition relation*, and $L : S \to 2^{\mathsf{AP}}$ is a *labeling function*, which labels states $s \in S$ with a subset of atomic propositions in AP that hold in s. A *path* of K is an infinite sequence of states $s(0)s(1)\cdots \in S^\omega$, such that $s(0) \in S^0$, and $(s(i), s(i+1)) \in \delta$, for all $i \geq 0$. A *loop* in K is a finite path $s(n)s(n+1)\cdots s(\ell)$, for some $0 \leq n \leq \ell$, such that $(s(i), s(i+1)) \in \delta$, for all $n \leq i < \ell$, and $(s(\ell), s(n)) \in \delta$. Note that $n = \ell$ indicates a *self-loop* on a state. A *trace* of K 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 $i \geq 0$. We denote by $\mathit{Traces}(K, s)$ the set of all traces of K with paths that start in state $s \in S$. We use $\mathit{Traces}(K)$ as a shorthand for $\bigcup_{s\in S^0} \mathit{Traces}(K, s)$, and $\mathcal{L}(K)$ as the shorthand for $\mathit{Traces}(K)$.

Simulation relations. Let $K_A = \langle S_A, S^0_A, \delta_A, \mathsf{AP}_A, L_A\rangle$ and $K_B = \langle S_B, S^0_B, \delta_B, \mathsf{AP}_B, L_B\rangle$ be two Kripke structures. A *simulation relation* R from K_A to K_B is a relation $R \subseteq S_A \times S_B$ that meets the following conditions:

1. For every $s_A \in S^0_A$ there exists $s_B \in S^0_A$ such that $(s_A, s_B) \in R$.
2. For every $(s_A, s_B) \in R$, it holds that $L_A(s_A) = L_B(s_B)$.
3. For every $(s_A, s_B) \in R$, for every $(s_A, s'_A) \in \delta_A$, there exists $(s_B, s'_B) \in \delta_B$ such that $(s'_A, s'_B) \in R$.

The Temporal Logic HyperLTL. HyperLTL [13] 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* $\Box\varphi \equiv \neg\Diamond\neg\varphi$. A formula is *closed* (i.e., a *sentence*) if all trace variables used in the formula are quantified. We assume, without loss of generality, that no trace variable is quantified twice. We use $\mathit{Vars}(\varphi)$ for the set of trace variables used in formula φ.

Semantics. An interpretation $\mathcal{T} = \langle T_\pi \rangle_{\pi \in \mathit{Vars}(\varphi)}$ of a formula φ consists of a tuple of sets of traces, with one set T_π per trace variable π in $\mathit{Vars}(\varphi)$, denoting the set of traces that π ranges over. Note that we allow quantifiers to range over different models, which is called the *multi-model semantics* [25, 29][2]. That is, each set of traces comes from a Kripke structure and we use $\mathcal{K} = \langle K_\pi \rangle_{\pi \in \mathit{Vars}(\varphi)}$ to denote a *family* of Kripke structures, so $T_\pi = \mathit{Traces}(K_\pi)$ is the traces that π can range over, which comes from $K_\pi \in \mathcal{K}$. Abusing notation, we write $\mathcal{T} = \mathit{Traces}(\mathcal{K})$.

The semantics of HyperLTL is defined with respect to a trace assignment, which is a partial map $\Pi \colon \mathit{Vars}(\varphi) \rightharpoonup \Sigma^\omega$. The assignment with the 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 $\Pi[\pi \rightarrow t]$ 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 evaluating position. The semantics is defined as follows:

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

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 $\langle \mathit{Traces}(K_\pi) \rangle_{\pi \in \mathit{Vars}(\varphi)} \models \varphi$. When the same Kripke structure K is used for all path variables we write $K \models \varphi$.

Definition 1. A *nondeterministic Büchi automaton* (NBW) is a tuple $A = \langle \Sigma, Q, Q_0, \delta, F \rangle$, where Σ is an *alphabet*, Q is a nonempty finite set of

[2] In terms of the model checking problem, multi-model and (the conventional) single-model semantics where all paths are assigned traces from the same Kripke structure [13] are equivalent (see [25, 29]).

states, $Q_0 \subseteq Q$ is a set of *initial states*, $F \subseteq Q$ is a set of *accepting states*, and $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*.

Given an infinite word $w = \sigma_1\sigma_2\cdots$ over Σ, a *run of* A *on* w is an infinite sequence of states $r = (q_0, q_1, \ldots)$, such that $q_0 \in Q_0$, and $(q_{i-1}, \sigma_i, q_i) \in \delta$ for every $i > 0$. The run is *accepting* if r visits some state in F infinitely often. We say that A *accepts* w if there exists an accepting run of A on w. The *language* of A, denoted $\mathcal{L}(A)$, is the set of all infinite words accepted by A. An NBW A is called a *safety* NBW if all of its states are accepting. Every safety LTL formula ψ can be translated into a safety NBW over 2^{AP} such that $\mathcal{L}(A)$ is the set of all traces over AP that satisfy ψ [30].

3 Adaptation of BMC to HyperLTL on Infinite Traces

There are two main obstacles in extending the BMC approach of [29] to handle infinite traces. First, a trace may have an irregular behavior. Second, even traces whose behavior is regular, that is, lasso shaped, are hard to synchronize, since the length of their respective prefixes and lassos need not to be equal. For the latter issue, synchronizing two traces whose prefixes and lassos are of lengths p_1, p_2 and l_1, l_2, respectively, is equivalent to coordinating the same two traces, when defining both their prefixes to be of length $\max\{p_1, p_2\}$, and their lassos to be of length $\mathrm{lcm}\{l_1, l_2\}$, where 'lcm' stands for 'least common multiple'. As for the former challenge, we show that restricting the exploration of traces in the models to only consider lasso traces is sound. That is, considering only lasso-shaped traces is equivalent to considering the entire trace set of the models.

Let $K = \langle S, S^0, \delta, \mathsf{AP}, L\rangle$ be a Kripke structure. A *lasso path* of K is a path $s(0)s(1)\ldots s(\ell)$ such that $(s(\ell), s(n)) \in \delta$ for some $0 \leq n < \ell$. This path induces a *lasso trace* (or simply, a *lasso*) $L(s_0)\ldots L(s_{n-1})\,(L(s_n)\ldots L(s_\ell))^\omega$. Let $\langle K_1, \ldots, K_k\rangle$ be a multi-model. We denote the set of lasso traces of K_i by C_i for all $1 \leq i \leq k$, and we use $\mathcal{L}(C_i)$ as the shorthand for the set of lasso traces of K_i.

Theorem 1. *Let* $\mathcal{K} = \langle K_1, \ldots, K_k\rangle$ *be a multi-model, and let* $\varphi = \mathbb{Q}_1\pi_1.\cdots\mathbb{Q}_k\pi_k.\psi$ *be a* HyperLTL *formula, both over* AP*, then* $\mathcal{K} \models \varphi$ *iff* $\langle C_1, \ldots, C_k\rangle \models \varphi$.

Proof. (sketch) For an LTL formula ψ over $\mathsf{AP} \times \{\pi_i\}_{i=1}^k$, we denote the translation of ψ to an NBW over $2^{\mathsf{AP}\times\{\pi_i\}_{i=1}^k}$ by A_ψ [36]. Given $\alpha = \mathbb{Q}_1\pi_1\cdots\mathbb{Q}_k\pi_k$, where $\mathbb{Q}_i \in \{\exists, \forall\}$, we define the satisfaction of A_ψ by $\mathcal{K}$ w.r.t. α, denoted $\mathcal{K} \models (\alpha,\ A_\psi)$, in the natural way: $\exists\pi_i$ corresponds to the existence of a path assigned to π_i in K_i, and dually for $\forall\pi_i$. Then, $\mathcal{K} \models (\alpha, A_\psi)$ iff the various k-assignments of traces of $\mathcal{K}$ to $\{\pi_i\}_{i=1}^k$ according to α are accepted by A_ψ, which holds iff $\mathcal{K} \models \varphi$.

For a model K, we denote by $K \cap_k A_\psi$ the intersection of K and A_ψ w.r.t. $\mathsf{AP} \times \{\pi_k\}$, taking the projection over $\mathsf{AP} \times \{\pi_i\}_{i=1}^{k-1}$. Thus, $\mathcal{L}(K \cap_k A_\psi)$ is the set of all $(k-1)$-words that *an extension* (i.e., $\exists$) by a word in $\mathcal{L}(K)$ to a k-word in $\mathcal{L}(A_\psi)$. Oppositely, $\mathcal{L}(\overline{K \cap_k \overline{A_\psi}})$ is the set of all $(k-1)$-words that *every extension* (i.e., $\forall$) by a k-word in $\mathcal{L}(K)$ is in $\mathcal{L}(A_\psi)$.

We first construct NBWs $A_2, \ldots, A_{k-1}, A_k$, such that for every $1 < i < k$, we have $\langle K_1, \ldots, K_i \rangle \models (\alpha_i, A_{i+1})$ iff $\mathcal{K} \models (\alpha, A_\psi)$, where $\alpha_i = \mathbb{Q}_1\pi_1 \ldots \mathbb{Q}_i\pi_i$.

For $i = k$, if $\mathbb{Q}_k = \exists$, then $A_k = K_k \cap_k A_\psi$; otherwise if $\mathbb{Q}_k = \forall$, $A_k = \overline{K_k \cap_k \overline{A_\psi}}$. For $1 < i < k$, if $\mathbb{Q}_i = \exists$ then $A_i = K_i \cap_i A_{i+1}$; otherwise if $\mathbb{Q}_i = \forall$, $A_i = \overline{K_i \cap_i \overline{A_{i+1}}}$. Then, for every $1 < i < k$, we have $\langle K_1, \ldots, K_i \rangle \models (\alpha_i, A_{i+1})$ iff $\langle K_1, \ldots, K_k \rangle \models \varphi$.

We now prove by induction on k that $\mathcal{K} \models \varphi$ iff $\langle C_1, \ldots C_k \rangle \models \varphi$. For $k = 1$, it holds that $\mathcal{K} \models \varphi$ iff $K_1 \models (\mathbb{Q}_1\pi_1, A_2)$. If $\mathbb{Q}_1 = \forall$, then $K_1 \models (\mathbb{Q}_1\pi_1, A_2)$ iff $K_1 \cap \overline{A_2} = \emptyset$. If $\mathbb{Q}_1 = \exists$, then $K_1 \models (\mathbb{Q}_1\pi_1, A_2)$ iff $K_1 \cap A_2 \neq \emptyset$. In both cases, a lasso witness to the non-emptiness exists. For $1 < i < k$, we prove that $\langle C_1, \ldots, C_i, K_{i+1} \rangle \models (\alpha_{i+1}, A_{i+2})$ iff $\langle C_1, \ldots, C_i, C_{i+1} \rangle \models (\alpha_{i+1}, A_{i+2})$. If $\mathbb{Q}_i = \forall$, then the first direction simply holds because $\mathcal{L}(C_{i+1}) \subseteq \mathcal{L}(K_{i+1})$. For the second direction, every extension of $c_1, c_2, \ldots c_i$ (i.e., lassos in $C_1, C_2, \ldots C_i$) by a path τ in K_{i+1} is in $\mathcal{L}(A_{i+2})$. Indeed, otherwise we can extract a lasso c_{i+1} such that $c_1, c_2, \ldots c_{i+1}$ is in $\overline{\mathcal{L}(A_{i+2})}$, a contradiction. If $\mathbb{Q}_i = \exists$, then $\mathcal{L}(C_{i+1}) \subseteq \mathcal{L}(K_{i+1})$ implies the second direction. For the first direction, we can extract a lasso $c_{i+1} \in \mathcal{L}(C_{i+1})$ such that $\langle c_1, c_2, \ldots c_i, c_{i+1} \rangle \in \mathcal{L}(A_{i+2})$. □

One can use Theorem 1 and the observations above to construct a sound and complete BMC algorithm for both $\forall\exists$ and $\exists\forall$ hyperproperties. Indeed, consider a multi-model $\langle K_1, K_2 \rangle$, and a hyperproperty $\varphi = \forall\pi.\exists\pi'.\ \psi$. Such a BMC algorithm would try and verify $\langle K_1, K_2 \rangle \models \varphi$ directly, or try and prove $\langle K_1, K_2 \rangle \models \neg\varphi$. In both cases, a run may find a short lasso example for the model under $\exists$ (K_2 in the former case and K_1 in the latter), leading to a shorter run. However, in both cases, the model under $\forall$ would have to be explored to the maximal lasso length implicated by Theorem 1, which is doubly-exponential. Therefore, this naive approach would be highly inefficient.

4 Simulation-Based BMC Algorithms for HyperLTL

We now introduce efficient simulation-based BMC algorithms for verifying hyperproperties of the types $\forall\pi.\exists\pi'.\square\mathsf{Pred}$ and $\exists\pi.\forall\pi'.\square\mathsf{Pred}$, where Pred is a *relational predicate* (a predicate over a pair of states). The key observation is that simulation naturally induces the exploration of infinite traces without the need to explicitly unroll the structures, and without needing to synchronize the indices of the symbolic variables in both traces. Moreover, in some cases our algorithms allow to only partially explore the state space of a Kripke structure and give a conclusive answer efficiently.

Let $K_P = \langle S_P, S_P^0, \delta_P, \mathsf{AP}_P, L_P \rangle$ and $K_Q = \langle S_Q, S_Q^0, \delta_Q, \mathsf{AP}_Q, L_Q \rangle$ be two Kripke structures, and consider a hyperproperty of the form $\forall\pi.\exists\pi'.\ \square\mathsf{Pred}$. Suppose that there exists a simulation from K_P to K_Q. Then, every trace in K_P is embodied in K_Q. Indeed, we can show by induction that for every trace $t_p = s_p(1)s_p(2)\ldots$ in K_P, there exists a trace $t_q = s_q(1)s_q(2)\ldots$ in K_Q, such that $s_q(i)$ simulates $s_p(i)$ for every $i \geq 1$; therefore, t_p and t_q are equally labeled. We generalize the labeling constraint in the definition of standard simulation by requiring, given Pred, that if (s_p, s_q) is in the simulation relation, then

$(s_p, s_q) \models \mathsf{Pred}$. We denote this generalized simulation by $\mathsf{SIM_{AE}}$. Following similar considerations, we now have that for every trace t_p in K_P, there exists a trace t_q in K_Q such that $(t_p, t_q) \models \square\mathsf{Pred}$. Therefore, the following result holds:

Lemma 1. *Let K_P and K_Q be Kripke structures, and let $\varphi = \forall\pi.\exists\pi'.\ \square\mathit{Pred}$ be a HyperLTL formula. If there exists $\mathit{SIM_{AE}}$ from K_P to K_Q, then $\langle K_P, K_Q\rangle \models \varphi$.*

We now turn to properties of the type $\exists\pi.\forall\pi'.\ \square\mathsf{Pred}$. In this case, we must find a single trace in K_P that matches every trace in K_Q. Notice that $\mathsf{SIM_{AE}}$ (in the other direction) does not suffice, since it is not guaranteed that the same trace in K_P is used to match all traces in K_Q. However, according to Theorem 1, it is guaranteed that if $\langle K_P, K_Q\rangle \models \exists\pi.\forall\pi'.\ \square\mathsf{Pred}$, then there exists such a single lasso trace t_p in K_P as the witness of the satisfaction. We therefore define a second notion of simulation, denoted $\mathsf{SIM_{EA}}$, as follows. Let $t_p = s_p(1)s_p(2)\dots s_p(n)\dots s_p(\ell)$ be a lasso trace in K_P (where $s_p(\ell)$ closes to $s_p(n)$, that is, $(s_p(\ell), s_p(n)) \in \delta_P$). A relation R from t_p to K_Q is considered as a $\mathsf{SIM_{EA}}$ from t_p to K_Q, if the following holds:

1. $(s_p, s_q) \models \mathsf{Pred}$ for every $(s_p, s_q) \in R$.
2. $(s_p(1), s_q) \in R$ for every $s_q \in S_Q^0$.
3. If $(s_p(i), s_q(i)) \in R$, then for every successor $s_q(i+1)$ of $s_q(i)$, it holds that $(s_p(i+1), s_q(i+1)) \in R$ (where $s_p(\ell+1)$ is defined to be $s_p(n)$).

If there exists a lasso trace t_p, then we say that there exists $\mathsf{SIM_{EA}}$ from K_P to K_Q. Notice that the third requirement in fact unrolls K_Q in a way that guarantees that for every trace t_q in K_Q, it holds that $(t_p, t_q) \models \square\mathsf{Pred}$. Therefore, the following result holds:

Lemma 2. *Let K_P and K_Q be Kripke structures, and let $\varphi = \exists\pi.\forall\pi'.\ \square\mathit{Pred}$. If there exists a $\mathit{SIM_{EA}}$ from K_P to K_Q, then $\langle K_P, K_Q\rangle \models \varphi$.*

Lemmas 1 and 2 enable sound algorithms for model-checking $\forall\pi.\exists\pi'.\ \square\mathsf{Pred}$ and $\exists\pi.\forall\pi'.\ \square\mathsf{Pred}$ hyperproperties with loop conditions. To check the former, check whether there exists $\mathsf{SIM_{AE}}$ from K_P to K_Q; to check the latter, check for a lasso trace t_p in K_P and $\mathsf{SIM_{EA}}$ from t_p to K_Q. Based on these ideas, we introduce now two SAT-based BMC algorithms.

For $\forall\exists$ hyperproperties, we not only check for the existence of $\mathsf{SIM_{AE}}$, but also iteratively seek a small subset of S_Q that suffices to simulate all states of S_P. While finding $\mathsf{SIM_{AE}}$, as for standard simulation, is polynomial, the problem of finding a simulation that uses a bounded number of K_Q states is NP-complete (see Appendix A.1 for details). This allows us to efficiently handle instances in which K_Q is large. Moreover, we introduce in Subsection 4.3 the use of *prophecy variables*, allowing us to overcome cases in which the models satisfy the property but $\mathsf{SIM_{AE}}$ does not exist.

For $\exists\forall$ hyperproperties, we search for $\mathsf{SIM_{EA}}$ by seeking a lasso trace t_p in K_P, whose length increases with every iteration, similarly to standard BMC techniques for LTL. Of course, in our case, t_p must be matched with the states of K_Q in a way that ensures $\mathsf{SIM_{EA}}$. In the worst case, the length of t_p may be

doubly-exponential in the sizes of the systems. However, as our experimental results show, in case of satisfaction the process can terminate much sooner.

We now describe our BMC algorithms and our SAT encodings in detail. First, we fix the unrolling depth of K_P to n and of K_Q to k. To encode states of K_P we allocate a family of Boolean variables $\{x_i\}_{i=1}^{n}$. Similarly, we allocate $\{y_j\}_{j=1}^{k}$ to represent the states of K_Q. Additionally, we encode the simulation relation T by creating $n \times k$ Boolean variables $\{sim_{ij}\}_{i=1,j=1}^{n,k}$ such that sim_{ij} holds if and only if $T(p_i, q_j)$. We now present the three variations of encoding: (1) EA-Simulation ($\mathsf{SIM_{EA}}$), (2) AE-Simulation ($\mathsf{SIM_{AE}}$), and (3) a special variation where we enrich AE-Simulation with prophecies.

4.1 Encodings for EA-Simulation

The goal of this encoding is to find a lasso path t_p in K_P that guarantees that there exists $\mathsf{SIM_{EA}}$ to K_Q. Note that the set of states that t_p uses may be much smaller than the whole of K_P, while the state space of K_Q must be explored exhaustively. We force x_0 be an initial state of K_P and for x_{i+1} to follow x_i for every i we use, but for K_Q we will let the solver fill freely each y_k and add constraints[3] for the full exploration of K_Q.

- **All states are legal states.** The solver must only search legal encodings of states of K_P and K_Q (we use $K_P(x_i)$ to represent the combinations of values that represent a legal state in S_P and similarly $K_Q(y_j)$ for S_Q):

$$\bigwedge_{i=1}^{n} K_P(x_i) \wedge \bigwedge_{j=1}^{k} K_Q(y_j) \tag{1}$$

- **Exhaustive exploration of K_Q.** We require that two different indices y_j and y_r represent two different states in K_Q, so if $k = |K_Q|$, then all states are represented (note that the validity of states is implied by (1)):

$$\bigwedge_{j \neq r} (K_Q(y_j) \wedge K_Q(y_r)) \rightarrow (y_j \neq y_r) \tag{2}$$

  where $y_j \neq y_r$ captures that some bit distinguishes the states encoded by j and r.
- **The initial S_P^0 state simulates all initial S_Q^0 states.** State x_0 is an initial state of K_P and simulates all initial states of K_Q (we use I_P to represent a legal initial state in K_P and $I_Q(y_j)$ for S_Q^0 of K_Q):

$$I_P(x_0) \wedge \big(\bigwedge_{j=1}^{k} I_Q(y_j) \rightarrow T(x_0, y_j) \big) \tag{3}$$

[3] An alternative is to fix an enumeration of the states of K_Q and force the assignment of $y_0 \ldots$ according to this enumeration instead of constraining a symbolic encoding, but the explanation of the symbolic algorithm above is simpler.

- **Successors in K_Q are simulated by successors in K_P.** We first introduce the following formula $succ_T(x, x')$ to capture one-step of the simulation, that is, x' follows x and for all y if $T(x, y)$ then x' simulates all successors of y (we use $\delta_Q(y, y')$ to represent that y and y' states are in δ_Q of K_Q, similarly for $(x, x') \in \delta_P$ of K_P we use $\delta_P(x, x')$) :

$$succ_T(x, x') \stackrel{\text{def}}{=} \bigwedge_{y=y_1}^{y_k} T(x, y) \rightarrow \big(\bigwedge_{y'=y_1}^{y_k} \delta_Q(y, y') \rightarrow T(x', y') \big)$$

  We can then define that x_{i+1} follows x_i:

$$\bigwedge_{i=1}^{n-1} \big[\delta_P(x_i, x_{i+1}) \wedge succ_T(x_i, x_{i+1}) \big] \tag{4}$$

  And, x_n has a jump-back to a previously seen state:

$$\bigvee_{i=1}^{n} \big[\delta_P(x_n, x_i) \wedge succ_T(x_n, x_i) \big] \tag{5}$$

- **Relational state predicates are fulfilled by simulation.** Everything relating in the simulation fits the relational predicate, defined as a function Pred of two sets of labels (we use $L_Q(y)$ to represent the set of labels on the y-encoded state in K_Q, similarly, $L_P(x)$ for the x-encoded state in K_P):

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} T(x_i, y_j) \rightarrow \mathsf{Pred}(L_P(x_i), L_Q(y_j)) \tag{6}$$

We use $\varphi_{\mathsf{EA}}^{n,k}$ for the SAT formula that results of conjoining (1)-(6) for bounds n and k. If $\varphi_{\mathsf{EA}}^{n,k}$ is satisfiable, then there exists $\mathsf{SIM}_{\mathsf{EA}}$ from K_P to K_Q.

4.2 Encodings for AE-Simulation

Our goal now is to find a set of states $S'_Q \subseteq S_Q$ that is able to simulate all states in K_P. Therefore, as in the previous case, the state space K_P corresponding to the $\forall$ quantifier will be explored exhaustively, and so $n = |K_P|$, while k is the number of states in K_Q, which increases in every iteration. As we have explained, this allows finding a small subset of states in K_Q which suffices to simulate all states of K_P.

- **All states in the simulation are legal states.** Again, every state guessed in the simulation is a legal state from K_P or K_Q:

$$\bigwedge_{i=1}^{n} K_P(x_i) \wedge \bigwedge_{j=1}^{k} K_Q(y_j) \tag{1'}$$

- K_P **is exhaustively explored.** Every two different indices in the states of K_P are different states[4]:

$$\bigwedge_{i \neq r} (K_P(x_i) \wedge K_P(x_r)) \rightarrow (x_i \neq x_r) \tag{2'}$$

- **All initial states in K_P must match with some initial state in K_Q.** Note that, contrary to the $\exists\forall$ case, here the initial state in K_Q may be different for each initial state in S_P:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} I_P(x_i) \rightarrow \big(I_Q(y_j) \wedge T(x_i, y_j)\big) \tag{3'}$$

- **For every pair in the simulation, each successor in K_P must match with some successor in K_Q.** For each (x_i, y_j) in the simulation, every successor state of x_i has a matching successor state of y_j:

$$\bigwedge_{i=1}^{n} \bigwedge_{t=1}^{n} \delta_P(x_i, x_t) \rightarrow \bigwedge_{j=1}^{k} \Big[T(x_i, y_j) \rightarrow \bigvee_{r=1}^{k} \Big(\delta_Q(y_j, y_r) \wedge T(x_t, y_r)\Big)\Big] \tag{4'}$$

- **Relational state predicates are fulfilled.** Similarly, all pairs of states in the simulation should respect the relational Pred:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} T(x_i, y_j) \rightarrow \mathsf{Pred}(L_P(x_i), L_Q(y_j)) \tag{5'}$$

We now use $\varphi_{\mathsf{AE}}^{n,k}$ for the SAT formula that results of conjoining (1′)-(5′) for bounds n and k. If $\varphi_{\mathsf{AE}}^{n,k}$ is satisfiable, then there exists $\mathsf{SIM}_{\mathsf{AE}}$ from K_P to K_Q.

4.3 Encodings for AE-Simulation with Prophecies

The AE-simulation encoding introduced in Section 4.2 can handle most properties of the form $\forall\exists\square\mathsf{Pred}$; however, it is unable to cope well with systems (in particular the system K_P for the $\forall$ quantifier) that exhibit non-determinism. The reason, as illustrated in the following example, is that the simulation matches immediately the successor for the $\exists$ path without inspecting the future of the $\forall$ path. This is illustrated in the following example.

Example 1. Consider Kripke structures K_1 and K_2 from Section 1, and HyperLTL formula $\varphi_2 = \forall\pi.\exists\pi'.\ \square(a_\pi \leftrightarrow a_{\pi'})$. It is easy to see that the two models satisfy φ_2, since mapping the sequence of states $(s_1 s_2 s_3)$ to $(q_1 q_2 q_4)$ and $(s_1 s_2 s_4)$ to $(q_1 q_3 q_5)$ guarantees that the matched paths satisfy $\square(a_\pi \leftrightarrow a_{\pi'})$. However, the technique in Section 4.2 cannot differentiate the occurrences of s_2 in the two different cases. □

[4] As in the previous case, we could fix an enumeration of the states of S_P and fix $x_0 x_1 \ldots$ to be the states according to the enumerations.

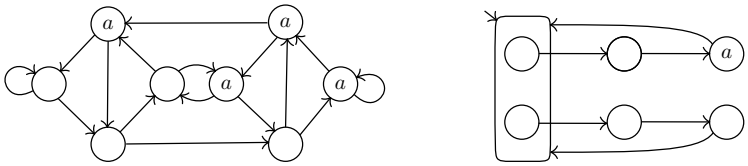


Fig. 1: Prophecy automaton for $\bigcirc\bigcirc a$ (left) and its composition with K_1 (right).

To solve this, we incorporate the notion of *prophecies* to our setting. Prophecies have been proposed as a method to aid in the verification of hyperliveness [15] (see [7] for a systematic method to construct these kind of prophecies). For simplicity, we restrict here to prophecies expressed as safety automata. A safety prophecy over AP is a Kripke structure $U = \langle S, S^0, \delta, \mathsf{AP}, L\rangle$, such that $\mathit{Traces}(U) = \mathsf{AP}^\omega$. The product $K \times U$ of a Kripke structure K with a prophecy U preserves the language of K (since the language of U is universal). Recall that in the construction of the product, states $(s, u) \in (K \times U)$ that have incompatible labels are removed. The direct product can be easily processed by repeatedly removing dead states, resulting in a Kripke structure K' whose language is $\mathit{Traces}(K') = \mathit{Traces}(K)$. Note that there may be multiple states in K' that correspond to different states in K for different prophecies. The prophecy-enriched Kripke structure can be directly passed to the method in Section 4.2, so the solver can search for a $\mathsf{SIM_{AE}}$ that takes the value of the prophecy into account.

Example 2. Consider the prophecy automaton shown in Fig. 1 (left), where all states are initial. Note that for every state, either all its successors are labeled with a (or none are), and all successors of its successors are labeled with a (or none are). In other words, this structure encodes the prophecy $\bigcirc\bigcirc a$. The product K_1' of K_1 with the prophecy automaton U for $\bigcirc\bigcirc a$ is shown in Fig. 1 (right). Our method can now show that $\langle K_1', K_2\rangle \models \varphi_2$, since it can distinguish the two copies of s_1 (one satisfies $\bigcirc\bigcirc a$ and is mapped to $(q_1 q_2 q_4)$, while the other is mapped to $(q_1 q_3 q_5)$). □

5 Implementation and Experiments

We have implemented our algorithms using the SAT solver Z3 through its python API Z3Py [31]. The SAT formulas introduced in Section 4 are encoded into the two scripts `simEA.py` and `simAE.py`, for finding simulation relations for the $\mathsf{SIM_{EA}}$ and $\mathsf{SIM_{AE}}$ cases, respectively. We evaluate our algorithms with a set of experiments, which includes all forms of quantifiers with different sizes of given models, as presented earlier in Table 1. Our simulation algorithms benefit the most in the cases of the form $\forall_{\mathtt{small}}\, \exists_{\mathtt{big}}$. When the second model is substantially larger than the first model, $\mathsf{SIM_{AE}}$ is able to prove that a $\forall\exists$ hyperproperty holds by exploring only a subset of the second model. In this section, besides $\forall_{\mathtt{small}}\, \exists_{\mathtt{big}}$ cases, we also investigate multiple cases on each category in Table 1 to

demonstrate the generality and applicability of our algorithms. All case studies are run on a MacBook Pro with Apple M1 Max chip and 64 GB of memory.

5.1 Case Studies and Empirical Evaluation

Conformance in Scenario-based Programming. In scenario-based programming, scenarios provide a big picture of the desired behaviors of a program, and are often used in the context of program synthesis or code generation. A synthesized program should obey what is specified in the given set of scenarios to be considered *correct*. That is, the program *conforms* with the scenarios. The conformance check between the scenarios and the synthesized program can be specified as a $\forall\exists$-hyperproperty:

$$\varphi_{\mathsf{conf}} = \forall\pi.\exists\pi'.\ \bigwedge_{p\in\mathsf{AP}} \square\ (p_\pi \leftrightarrow p_{\pi'}),$$

where π is over the scenario model and π' is over the synthesized program. That is, for all possible runs in the scenarios, there must exists a run in the program, such that their behaviors always match.

We look into the case of synthesizing an *Alternating Bit Protocol (ABP)* from four given scenarios, inspired by [3]. ABP is a networking protocol that guarantees reliable message transition, when message loss or data duplication are possible. The protocol has two parties: `sender` and `receiver`, which can take three different actions: *send*, *receive*, and *wait*. Each action also specifies which message is currently transmitted: either a *packet* or *acknowledgment* (see [3] for more details). The correctly synthesized protocol should not only have complete functionality but also *include all scenarios*. That is, for every trace that appears in some scenario, there must exist a corresponding trace in the synthesized protocol. By finding $\mathsf{SIM}_{\mathsf{AE}}$ between the scenarios and the synthesized protocols, we can prove the conformance specified with φ_{conf}. Note that the scenarios are often much smaller than the actual synthesized protocol, and so this case falls in the $\forall_{\texttt{small}}\ \exists_{\texttt{big}}$ category in Table 1. We consider two variations: a correct and an incorrect ABP (that cannot handle packet loss). Our algorithm successfully identifies a $\mathsf{SIM}_{\mathsf{AE}}$ that satisfies φ_{conf} for the correct ABP, and returns UNSAT for the incorrect protocol, since the packet loss scenario cannot be simulated.

Verification of Model Translation. It is often the case that in model translation (e.g., compilation), solely reasoning about the source program does not provide guarantees about the desirable behaviors in the target executable code. Since program verification is expensive compared with repeatedly checking the target, alternative approaches such as *certificate translation* [4] are often preferred. Certificate translation takes inputs of a high-level program (source) with a given specification, and computes a set of verification conditions (certificates) for the low-level executable code (target) to prove that a model translation is safe. However, this technique still requires extra efforts to map the certificates to a target language, and the size of generated certificates might explode quickly

(see [4] for retails). We show that our simulation algorithm can directly show the correctness of a model translation more efficiently by investigating the source and target with the same formula φ_{conf} used for ABP. That is, the specifications from the source runs π are always preserved in some target runs π', which infers a correct model translation. Since translating a model into executable code implies adding extra instructions such as writing to registers, it also falls into the $\forall_{\texttt{small}}\,\exists_{\texttt{big}}$ category in Table 1.

We investigate a program from [4] that performs *matrix multiplication (MM)*. When executed, the C program is translated from high-level code (C) to low-level code RTL (Register Transfer Level), which contains extra steps to read from/write to memories. Specifications are triples of $\langle Pre, annot, Post\rangle$, where *Pre*, and *Post* are assertions and *annot* is a partial function from labels to assertions (see [4] for detailed explanations). The goal is to make sure that the translation does not violate the original verified specification. In our framework, instead of translating the certification, we find a simulation that satisfies φ_{conf}, proving that the translated code also satisfies the specification. We also investigate two variations in this case: a correct translation and an incorrect translation, and our algorithm returns SAT (i.e., finds a correct $\mathsf{SIM}_{\mathsf{AE}}$ simulation) in the former case, and returns UNSAT for the latter case.

Compiler Optimization. Secure compiler optimization aims at preserving input-output behaviors of an original implementation and a target program after applying optimization techniques, including security policies. The conformance between source and target programs guarantees that the optimizing procedure does not introduce vulnerabilities such as information leakage. Furthermore, optimization is often not uniform for the same source, because one might compile the source to multiple different targets with different optimization techniques. As a result, an efficient way to check the behavioral equivalence between the source and target provides a correctness guarantee for the compiler optimization.

```
// Source program S
L1: if (j < arr_size) {
L2:     a := arr[0];
L3:     b := arr[j];
L4: } else {
L5:     a := arr[0];
L6:     b := arr[arr_size - 1];
L7: }
// Target program T
L1: a := arr[0];
L2: if (j < arr_size) {
L3:     b := arr[j];
L4: } else {
L5:
L6:     b := arr[arr_size - 1];
L7: }
```

Fig. 2: The common branch factorization example [32].

Imposing optimization usually results in a smaller program. For instance, *common branch factorization* (CBF) finds common operations in an if-then-else structure, and moves them outside of the conditional so that such operation is only executed once. As a result, for these optimization techniques, checking the conformance of the source and target falls in the $\forall_{\texttt{big}}\,\exists_{\texttt{small}}$ category. That is, given two programs, source (`big`) and target (`small`), we check the following formula:

$$\varphi_{\mathsf{sc}} = \forall\pi.\exists\pi'.\ (\mathsf{in}_{\pi} \leftrightarrow \mathsf{in}_{\pi'}) \rightarrow \square\ (\mathsf{out}_{\pi} \leftrightarrow \mathsf{out}_{\pi'}).$$

In this case study we investigate the strategy CBF using the example in Figure 2 inspired by [32]. We consider two kinds of optimized programs for the strategy, one is the correct optimization, one containing bugs that violates the original behavior due to the optimization. For the correct version, our algorithm successfully discovered a simulation relation between the source and target, and the simulation relation returns a smaller subset of states in the second model (i.e., $|Q'| < |Q|$). For the incorrect version, we received UNSAT.

Robust Path Planning. In robotic planning, *robustness planning (RP)* refers to a path that is able to consistently complete a mission without being interfered by the uncertainty in the environment (e.g., adversaries). For instance, in the 2-D plane in Fig. 3, an agent is trying to go from the starting point (blue grid) to the goal position (green grid). The plane also contains three adversaries on the three corners other than the starting point (red-framed grids), and the adversaries move trying to catch the agent but can only move in one direction (e.g., clockwise). This is a $\exists_{\texttt{small}}\ \forall_{\texttt{big}}$ setting, since the adversaries may have several ways to cooperate and attempt to catch the agent. We formulate this planning problem as follows:

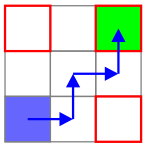


Fig. 3: A robust paths.

$$\varphi_{\mathsf{rp}} = \exists\pi.\forall\pi'.\ \square\ (\mathsf{pos}_{\pi} \not\leftrightarrow \mathsf{pos}_{\pi'}).$$

That is, there exists a robust path for the agent to safely reach the goal regardless of all the ways that the adversaries could move. We consider two scenarios, one in which there exists a way for the agent to form a robust path and one does not. Our algorithm successfully returns SAT for case which the agent can form a robust path, and returns UNSAT for which a robust path is impossible to find.

Plan Synthesis. The goal of *plan synthesis (PS)* is to synthesize a single comprehensive plan that can simultaneously satisfy all given small requirements has wide application in planning problems. We take the well-known toy example, *wolf, goat, and cabbage*[5], as a representative case here. The problem is as follows. A farmer needs to cross a river by boat with a wolf, a goat, and a cabbage. However, the farmer can only bring one item with him onto the boat each time. In addition, the wolf would eat the goat, and the goat would eat the cabbage, if they are left unattended. The goal is to find a plan that allows the farmer to successfully cross the river with all three items safely. A plan requires the farmer to go back and forth with the boat with certain possible ways to carry different items, while all small requirements (i.e., the constraints among each item) always satisfied. In this example, the overall plan is a big model while the requirements form a much smaller automaton. Hence, it is a $\exists_{\texttt{big}}\ \forall_{\texttt{small}}$ problem that can be specified with the following formula:

$$\varphi_{\mathsf{ps}} = \exists\pi.\forall\pi'.\ \square\ (\mathsf{action}_{\pi} \not\leftrightarrow \mathsf{violation}_{\pi'}).$$

[5] https://en.wikipedia.org/wiki/Wolf,_goat_and_cabbage_problem

Type	Quants	Cases	$\lvert S_P\rvert$	$\lvert S_Q\rvert$	Z3	Outcome	solve[s]
$\mathsf{SIM_{AE}}$	$\forall_{\texttt{small}}\ \exists_{\texttt{big}}$	ABP	11	14	sat	$\lvert S'_Q\rvert$=11	9.37
		$\text{ABP}_{\text{w/ bug}}$	11	14	unsat	-	9.46
		MM	27	27	sat	$\lvert S'_Q\rvert$=27	67.74
		$\text{MM}_{\text{w/ bug}}$	27	27	unsat	-	66.85
	$\forall_{\texttt{big}}\ \exists_{\texttt{small}}$	CBF	15	9	sat	$\lvert S'_Q\rvert$=8	3.49
		$\text{CBF}_{\text{w/ bug}}$	15	9	unsat	-	3.51
$\mathsf{SIM_{EA}}$	$\exists_{\texttt{small}}\ \forall_{\texttt{big}}$	RP 3^3	8	9	sat	$\lvert S'_P\rvert$=5	1.09
		RP $3^3_{\text{no sol.}}$	8	9	unsat	-	1.02
	$\exists_{\texttt{big}}\ \forall_{\texttt{small}}$	GCW	16	4	sat	$\lvert S'_P\rvert$=8	3.36
		$\text{GCW}_{\text{no sol.}}$	16	4	unsat	-	2.27

Table 2: Summary of our case studies. The outcomes with simulation discovered show how our algorithms find a smaller subset for either K_P or K_Q.

5.2 Analysis and Discussion

The summary of our empirical evaluation is presented in Table 2. For the $\forall\exists$ cases, our algorithm successfully finds a set $|S'_Q| < |S_Q|$ that satisfies the properties for the cases ABP and CBF. Note that case MM does not find a small subset, since we manually add extra *paddings* on the first model to align the length of both traces. We note that handling this instance without padding requires asynchornicity— a much more difficult problem, which we leave for future work. For the $\exists\forall$ cases, we are able to find a subset of S_P which forms a single lasso path that can simulate all runs in S_Q for all cases RP and GCW. We emphasize here that previous BMC techniques (i.e., HyperQB) cannot handle most of the cases in Table 2 due to the lack of loop conditions.

6 Conclusion and Future Work

We introduced efficient loop conditions for bounded model checking of fragments of HyperLTL. We proved that considering only lasso-shaped traces is equivalent to considering the entire trace set of the models, and proposed two simulation-based algorithms $\mathsf{SIM_{EA}}$ and $\mathsf{SIM_{AE}}$ to realize infinite reasoning with finite exploration for HyperLTL formulas. To handle non-determinism in the latter case, we combine the models with prophecy automata to provide the (local) simulations with enough information to select the right move for the inner $\exists$ path. Our algorithms are implemented using Z3py. We have evaluated the effectiveness and efficiency with successful verification results for a rich set of input cases, which previous bounded model checking approach would fail to prove.

As for future work, we are working on exploiting general prophecy automata (beyond safety) in order to achieve full generality for the $\forall\exists$ case. The second direction is to handle asynchrony between the models in our algorithm. Even though model checking asynchronous variants of HyperLTL is in general undecidable [5, 10, 27], we would like to explore semi-algorithms and fragments with decidability properties. Lastly, exploring how to handle infinite-state systems with our framework by applying *abstraction* techniques is also another promising future direction.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
2. Shreya Agrawal and Borzoo Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *Proc. of the 29th IEEE Computer Security Foundations Symposium (CSF'16)*, pages 239–252. IEEE, 2016.
3. Rajeev Alur, Milo Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing finite-state protocols from scenarios and requirements. In *Haifa Verification Conference*, pages 75–91. Springer, 2014.
4. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, César Kunz, and Anne Pacalet. Implementing a direct method for certificate translation. In *International Conference on Formal Engineering Methods*, pages 541–560. Springer, 2009.
5. Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In *Proc. of the 33rd Int'l Conf. on Computer Aided Verification (CAV'21), Part I*, volume 12759 of *LNCS*, pages 694–717. Springer, 2021.
6. S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proc. 4th Int. Conf. on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 260–273. Springer-Verlag, 1992.
7. Raven Beutner and Bernd Finkbeiner. Prophecy variables for hyperproperty verification. In *Proc. of Computer Security Foundations Symposium (CSF'22)*. IEEE, 2022.
8. Raven Beutner and Bernd Finkbeiner. Prophecy variables for hyperproperty verification. *CoRR*, abs/2206.01797, 2022.
9. Borzoo Bonakdarpour, César Sánchez, and Gerardo Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In *Proc. of the 8th Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18), Part II*, volume 11245 of *LNCS*, pages 8–27. Springer, 2018.
10. Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of HyperLTL. In *Proc. of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'21)*, pages 1–13. IEEE, 2021.
11. Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. Rewriting-based runtime verification for alternation-free HyperLTL. In *Proc. of the 23rd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17), Part II*, volume 10206 of *LNCS*, pages 77–93. Springer, 2017.
12. Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
13. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proc. of the 3rd Int'l Conf. on Principles of Security and Trust (POST'14)*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.
14. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
15. Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In *Proc. of the 31st Int'l Conf. on Computer Aided Verification (CAV'19)*, volume 11561 of *LNCS*, pages 121–139. Springer, 2019.

16. Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In *Proc. of the 31st Int'l Conf. on Computer Aided Verification (CAV'19), Part I*, volume 11561 of *LNCS*, pages 121–139. Springer, 2019.
17. Leonardo M. de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
18. Bernd Finkbeiner, Cristopher Hahn, and Tobias Hans. MGHyper: Checking satisfiability of HyperLTL formulas beyond the $\exists^*\forall^*$ fragment. In *Proc. of the 16th Int'l Symposium on Automated Technology for Verification and Analysis (ATVA'18)*, volume 11138 of *LNCS*, pages 521–527. Springer, 2018.
19. Bernd Finkbeiner, Cristopher Hahn, and Marvin Stenger. Eahyper: Satisfiability, implication, and equivalence checking of hyperproperties. In *Proc. of the 29th Int'l Conf. on Computer Aided Verification (CAV'17), Part II*, volume 10427 of *LNCS*, pages 564–570. Springer, 2017.
20. Bernd Finkbeiner, Cristopher Hahn, Marvin Stenger, and Leander Tentrup. RVHyper: A runtime verification tool for temporal hyperproperties. In *Proc. of the 24th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Part II*, volume 10806 of *LNCS*, pages 194–200. Springer, 2018.
21. Bernd Finkbeiner, Cristopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods in System Design*, 54(3):336–363, 2019.
22. Bernd Finkbeiner, Cristopher Hahn, and Hazem Torfah. Model checking quantitative hyperproperties. In *Proc. of the 30th Int'l Conf. on Computer Aided Verification (CAV'18), Part I*, volume 10981 of *LNCS*, pages 144–163. Springer, 2018.
23. Bernd Finkbeiner, Christian Müller, Helmut Seidl, and Eugene Zalinescu. Verifying security policies in multi-agent workflows with loops. In *Proc. of the 15th ACM Conf. on Computer and Communications Security (CCS'17)*, pages 633–645. ACM, 2017.
24. Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Proc. of the 27th Int'l Conf. on Computer Aided Verification (CAV'15), Part I*, volume 9206 of *LNCS*, pages 30–48. Springer, 2015.
25. Ohad Goudsmid, Orna Grumberg, and Sarai Sheinvald. Compositional model checking for multi-properties. In *Proc. of the 22nd Int'l Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'21)*, volume 12597 of *LNCS*, pages 55–80. Springer, 2021.
26. O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.
27. Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.
28. Cristopher Hahn, Marvin Stenger, and Leander Tentrup. Constraint-based monitoring of hyperproperties. In *Proc. of the 25th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*, volume 11428 of *LNCS*, pages 115–131. Springer, 2019.
29. Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In *Proc. of the 27th Int'l Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'21). Part I*, volume 12651 of *LNCS*, pages 94–112. Springer, 2021.
30. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Computer Aided Verification (CAV)*, pages 172–183, 1999.

31. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
32. Kedar S Namjoshi and Lucas M Tabajara. Witnessing secure compilation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 1–22. Springer, 2020.
33. A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
34. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Proc. Advanced School on Current Trends in Concurrency*, pages 510–584. Volume 224, LNCS, Springer, 1985.
35. Sandro Stucki, César Sánchez, Gerardo Schneider, and Borzoo Bonakdarpour. Graybox monitoring of hyperproperties. In *Proc. of the 23rd Int'l Symposium on Formal Methods (FM'19)*, volume 11800 of *LNCS*, pages 406–424. Springer, 2019.
36. M.Y. Vardi and P. Wolper. Automata theoretic techniques for modal logic of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

A Proofs

A.1 Bounded Simulation

Let K_1 and K_2 be two Kripke structures over AP, and let $k \in \mathcal{N}$. The *bounded simulation problem* for K_1, K_2 and k is to decide whether there exists a simulation relation from K_1 to K_2 that uses at most k states of K_2 (note that in any case, all of the reachable states of K_1 must be used in such a simulation). We prove this problem to be NP-complete.

Theorem 2. *The bounded simulation problem is NP-complete.*

Proof. Let K_1 and K_2 be two Kripke structures with sets of states Q_1 and Q_2, respectively, and let $k \in \mathcal{N}$. A nondeterministic algorithm which guesses a set $Q_2' \subseteq Q_2$ of at most k states, and searches for a simulation from K_1 to K_2 reduced to Q_2'. Finding a simulation can be done in polynomial time, and so the problem is in NP.

We prove NP-hardness by a reduction from the Vertex Cover problem. Given a directed graph $G = V, E,$ where $|E| = m$ and where $V = \{v_1, \ldots v_n\}$, and $k \in \mathcal{N}$, we construct two Kripke structures K_1 and K_2, as follows.

K_1 is composed of m states, where for every $e \in E$ there is a state labeled e, and an additional initial state q labeled q. The transitions are from q to all edge states and vice versa.

K_2 is composed of m states similarly labeled as the edge states of K_1, and additional n states $v_1, \ldots v_n$ all labeled q, all initial. From every v_i there are transitions to all edge states. From every edge state (v_i, v_j) there are transitions to v_i and v_j.

It is easy to see that K_2 can simulate K_1 using at most $m + k$ states iff G has a vertex cover of size at most k.