HyperLasso: Bounded Model Checking of $\forall^+\exists^+$-Liveness Hyperproperties

Anonymous Author(s)

Anonymous Institution

Abstract. This paper presents the first symbolic bounded model checking technique capable of verifying $\forall^+\exists^+$-liveness hyperproperties (expressed in HyperLTL) over arbitrary (non-terminating) reactive systems. Previous bounded procedures for HyperLTL handled only safety hyperproperties or arbitrary properties over terminating systems. We implement our technique as HyperLasso. Our evaluation results show that it consistently outperforms the explicit-state complete model checker AutoHyper (the only existing tool capable of automatically verifying this class of problems) at several complex bug-finding and synthesis problems.

Keywords: Hyperproperties · Bounded Model Checking · Liveness.

1 Introduction

Hyperproperties [15] are specifications that relate multiple traces of execution. They can be used to specify properties that are not possible to express with standard temporal logics that only consider one execution trace at a time, namely security properties such as non-interference or observational determinism. HyperLTL [14] extends the standard *Linear Temporal Logic* (LTL) with explicit quantifiers over traces, thus allowing the specification of general hyperproperties. There has been considerable research addressing the automatic verification of restricted classes of hyperproperties expressed in HyperLTL. However, as far as we are aware, the complete model checker AutoHyper [6], is the only tool capable of verifying arbitrary HyperLTL properties over arbitrary reactive systems. Unfortunately, due to its explicit-state nature, AutoHyper does not cope well with the state explosion problem that characterizes most realistic system models. To overcome this problem, symbolic model checking techniques can be used, namely *Bounded Model Checking* (BMC) [10] procedures that rely on SAT or SMT solving to find bugs in a system, by analyzing all traces up to a given length bound. HyperQB [25,24] is such a symbolic BMC tool for HyperLTL. Currently, it can be used to find bugs of arbitrary hyperproperties on terminating systems, or of safety properties (with any quantifier alternations) on any reactive (possibly non-terminating) system. Other existing symbolic model checking techniques for hyperproperties suffer from similar restrictions.

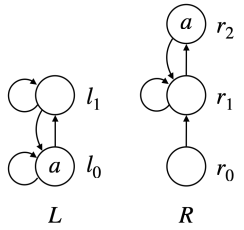

Fig. 1. Example models

This paper presents the first symbolic BMC technique that can be used to find bugs of $\forall^+\exists^+$-liveness properties[1] on reactive systems. In this class of hyperproperties there is a sequence of universal trace quantifiers, followed by a sequence of existential trace quantifiers, followed by an LTL liveness property. Likewise for safety hyperproperties, a single quantifier alternation suffices to express most properties of interest in practice. An example of such a liveness hyperproperty is the following, stating that: for any trace π_1, there exists another trace π_2 where a will eventually be true in both traces at the same time.

$$\forall\pi_1.\exists\pi_2.\Diamond(a_{\pi_1} \wedge a_{\pi_2})$$

Our BMC technique allows quantified trace variables to range over different models. In this example, π_1 and π_2 range, respectively, over traces of models L and R depicted in Fig. 1. Both these models are reactive systems that do not terminate. This hyperproperty is false because, for example, the trace corresponding to path $l_0(l_1)^\omega$ of model L does not have a counterpart trace of model R, since a only holds in the first state of that trace, and any trace of R needs 3 steps to reach a. The challenge of refuting such a hyperproperty is twofold: the $\Diamond$ in the LTL body requires reasoning about infinite paths, while the $\exists$ hyper quantification requires the exploration of all traces even in a BMC setting.

An LTL liveness requirement states that something "good" must happen, so a counter-example must be a full (infinite) path where that "good" thing never happens. With BMC the verification is performed up to a given k, a bound limiting the size of the analyzed paths. Bounded infinite paths are *lasso* shaped, meaning we have a prefix of size l followed by a lasso of size $k-l+1$ that repeats forever, which is denoted a (k,l)-loop or just a k-loop if the concrete value of l is irrelevant. The counter-example to our property, path $l_0(l_1)^\omega$, is a $(1,1)$-loop of model L. Not all infinite paths are lasso shaped, but a classic result from temporal logic is that any property that is false in a model can be falsified by a lasso-shaped path. This result is also valid for HyperLTL [26]. However, since different paths can have lassos of different sizes, the BMC procedure must align the traces resulting from each quantification, which might require exploring the composed models up to depth k^n for n quantified traces.

[1] This is not to be confused with the notion of *hyperliveness* [15], which refers to hyperproperties with existentially quantified traces.

Given a formula φ, $\models \varphi$ means it is valid considering all possible paths, while the bounded version $\models^k \varphi$ means φ is valid considering only paths of size k. BMC for standard LTL is particularly useful for bug finding because if $\not\models^k \varphi$ then $\not\models \varphi$, i.e., any found counter-example is a bug of the full system (in contrast, the absence of counter-examples for bound k does not guarantee φ is valid for higher k values). However, such infinite inference is not valid in a direct application of classical BMC to HyperLTL when liveness properties are involved: such a procedure might report $\not\models^k \varphi$ but it could actually be the case that $\models \varphi$, meaning that a counter-example might be a false positive and not a true bug in the system. Consider again our example property. To refute it with BMC with bound k we must find a witness for its negation $\exists\pi_1.\forall\pi_2.\square(\neg a_{\pi_1} \vee \neg a_{\pi_2})$, i.e., a k-loop π_1 of L such that for all k-loops π_2 of R we have $\square(\neg a_{\pi_1} \vee \neg a_{\pi_2})$. For $k = 0$ the only 0-loop of model L is $(l_0)^\omega$ and there are no 0-loops of model R, so instantiating π_1 as $(l_0)^\omega$ trivially satisfies the negated property for $k = 0$. Unfortunately, this path cannot be used to show that $\not\models \forall\pi_1.\exists\pi_2.\Diamond(a_{\pi_1} \wedge a_{\pi_2})$ with unbounded semantics, because the $(2, 1)$-loop $r_0(r_1r_2)^\omega$ is a valid counter-part in R. For $k = 1$ we have additional 1-loops that satisfy the negated property but are spurious counter-examples, namely the $(1, 0)$-loop $(l_0l_1)^\omega$. However, we also have the 1-loop $l_0(l_1)^\omega$ that is indeed a valid bug. Thus, vanilla BMC for HyperLTL is not usable when we have liveness properties, because one would have to constantly question the validity of counter-examples, completely defeating the premise of BMC. Existing BMC techniques for hyperproperties circumvent these issues by supporting only terminating systems (that stop in k or less steps) or safety hyperproperties.

Our technique solves this issue by pairing a HyperLTL BMC procedure, that will be used as a candidate *synthesizer*, with a *checker* that verifies if candidates are real bugs of the full system. The key insight is that, if we only have a single quantifier alternation, a standard complete model checker for LTL can be used to implement the *checker* using the self-composition technique [2]. When the *checker* flags a candidate as spurious, the *synthesizer* is asked to generate a new one, and the process is iterated until a valid counter-example is found or all possible candidates of size k are exhausted. We implement the technique as the HyperLasso model checker, which we compare against the only existing tool that can automatically verify this class of properties, AutoHyper. To this end, we created a new benchmark consisting of example systems of parameterized complexity, together with liveness hyperproperties, obtained by analyzing prior work. Our evaluation shows that our BMC procedure implemented with a reduction to *Quantified Boolean Formulas* (QBF) formulas is already efficient enough to handle the complexity for multiple traces with different lassos. Moreover, it shows that for large systems, as expected, the symbolic technique implemented in HyperLasso clearly outperforms the explicit-state technique implemented in AutoHyper at bug-finding or synthesis problems.

In summary, we make the following contributions to the state-of-art in order to achieve the first usable symbolic BMC for $\forall^+\exists^+$-liveness hyperproperties:

- We propose the first BMC procedure for HyperLTL that handles lasso-shaped paths, by encoding properties to QBF.

- We propose to iterate this BMC procedure and pairing it with a checking step to ensure that the produced counter-examples are true bugs of the system.
- We create, and make publicly available, a benchmark of liveness hyperproperties over systems of parametrized complexity.
- We implement the procedure as HyperLasso, which consumes models and properties in standard formats, and evaluate it against the state-of-the-art.

The remainder of this paper is structured as follows: the next section briefly presents HyperLTL and Section 3 reviews the related work; Section 4 presents our symbolic BMC procedure to implement a counter-example *synthesizer* and how an off-the-shelf model checker can be used to implement the *checker*; Section 5 presents the research questions, benchmark, and results of our evaluation; and, finally, Section 6 concludes the paper, including some ideas for future work.

2 HyperLTL in a nutshell

The syntax of HyperLTL is defined by the following grammar, where $a \in \mathsf{AP}$ is an atomic proposition and $\pi \in \mathcal{V}$ is a path variable.

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

As usual, we also have the derived logical (false, $\vee$, $\rightarrow$) and temporal operators (such as $\Diamond\psi := \mathsf{true}\,\mathcal{U}\,\psi$ and $\Box\psi := \neg(\Diamond\neg\psi)$).

The semantics of HyperLTL is defined over Kripke structures. A Kripke structure is a tuple $K = \langle S, S_0, \delta, L\rangle$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $\delta \subseteq S \times S$ is a transition relation, and $L : S \to \Sigma$ is a labelling function, where $\Sigma ::= 2^{\mathsf{AP}}$. A path in a Kripke structure is an infinite sequence of states $s_0 s_1 s_2 ... \in S^\omega$ such that $s_0 \in S_0$ and $(s_i, s_{i+1}) \in \delta$ for all i. Each path corresponds to a trace $L(s_0)L(S_1)L(S_2) ... \in \Sigma^\omega$, and $\mathit{Traces}(K) \subseteq \Sigma^\omega$ denotes the set of all traces of K.

The semantics of a closed HyperLTL formula is given with respect to a trace assignment $\Pi : \mathcal{V} \hookrightarrow \Sigma^\omega$ and a mapping $\mathbb{T} : \mathcal{V} \hookrightarrow 2^{\Sigma^\omega}$ associating each quantified trace variable with the set of traces over which it ranges:

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

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

We say a formula is valid, denoted $\models \varphi$ iff $\mathbb{T}, \emptyset \models \varphi$ with $\mathbb{T}$ being the mapping that associates each quantified trace variable π with the set $\mathit{Traces}(K_\pi)$ of the

respective Kripke structure K_π. If we are only interested in lasso-shaped paths, the bounded semantics $\models^k \varphi$ is defined in the same way as the unbounded one, except that $\mathbb{T}$ will only consider the subset of traces of each Kripke structure corresponding to k-loops. If non-lasso-shaped paths of size k are also to be considered (paths of which we know a prefix of size k) a "pessimistic" bounded semantics must be considered where the truth value of formulas beyond the last state is assumed to be false [10]. That is why counter-examples to liveness properties cannot be found with non-lasso paths of size k, because we cannot be sure the "good" thing never happens in the future. Since in this paper we focus on the verification of liveness properties we only consider infinite (lasso-shaped) paths of size k, and we omit the pessimistic semantics for the finite ones. Of course, it is trivial to extend our technique to finite paths and have full BMC for HyperLTL.

3 Related work

In recent years considerable work has addressed the problem of verifying hyperproperties encoded in general hyperlogics, namely HyperLTL. In this section, we focus on model checking approaches, although considerable work has also been dedicated to verification of hyperproperties over programs [19,4,27,17].

HyperQB [25] is the state-of-the-art symbolic bounded model checker for HyperLTL. It addresses the verification of arbitrary HyperLTL properties for acyclic Kripke structures through an encoding into QBF. The absence of cycles in the Kripke structure (apart from self-looping states) severely limits its application to reactive systems. HyperQB also implements finite path semantics, which could in principle be used to verify safety hyperproperties for arbitrary Krikpe structures, but not liveness hyperproperties for the reasons presented in Section 1. In subsequent work [26], members of the HyperQB team addressed the model checking of loop-shaped paths for $\forall\exists\Box\psi$ and $\exists\forall\Box\psi$ properties. Unfortunately, the technique requires the enumeration of all states of the Kripke structure, and does not scale for complex systems. To support the verification of asynchronous systems (i.e., that allow stuttering), the authors also proposed a BMC technique [23] for A-HLTL [3], a hyperlogic that allows the quantification of different *trajectories* of a trace. However, A-HLTL model checking is undecidable, and the BMC approach is restricted to acyclic, terminating systems. All these advancements have recently been integrated into a new version of HyperQB [24], as well as an alternative encoding into SMT. None of these BMC approaches support liveness hyperproperties over non-terminanting systems.

Extensive work has focused on explicit-state complete model checking approaches for HyperLTL, since the seminal work by Clarkson et al. showed HyperLTL model checking is decidable and developed a prototype model checker [14]. MCHyper [21] is the first explicit-state model checker for HyperLTL to support models of some complexity. It is an automata-based model checker, but requires user input to solve existential quantifications. AutoHyper [6] is the state-of-the-art automata-based model checker, which fully supports HyperLTL through automata complementations. Subsequent work [7] has extended Auto-

Input: Formula $\exists\pi_1 \dots \exists\pi_n.\forall\pi_{n+1} \dots \forall\pi_m.\psi$, models $K_{\pi_1}, \dots, K_{\pi_m}$, length k.
Output: Traces $t_1, \dots, t_n : \mathcal{V} \hookrightarrow \Sigma^\omega$ or UNSAT
for $i \in [1, \dots, m]$ **do** $k_i \leftarrow k$;
$result \leftarrow \bot$;
while $result = \bot$ **do**
  $[t_1, \dots, t_n] \leftarrow synthesizer(k_1, \dots, k_m, K_{\pi_1}, \dots, K_{\pi_m}, \exists\pi_1 \dots \forall\pi_m.\psi)$;
  if $[t_1, \dots, t_n] =$ UNSAT **then**
    $result \leftarrow$ UNSAT ;
  else
    $K'_{\pi_m}, \dots, K'_{\pi_n} \leftarrow K_{\pi_1} \oplus t_1, \dots, K_{\pi_n} \oplus t_n$;
    $[c_1, \dots, c_m] \leftarrow checker(K'_{\pi_1}, \dots, K'_{\pi_n}, K_{\pi_{n+1}}, \dots, K_{\pi_m}, \forall\pi_1 \dots \forall\pi_m.\psi)$;
    if $[c_1, \dots, c_m] =$ UNSAT **then**
      $result \leftarrow [t_1, \dots, t_n]$;
    else
      for $i \in [n+1, \dots, m]$ **do** $k_i \leftarrow |c_i|$;
return *result*;

Algorithm 1: Overview of the *synthesis*/*checker* verification loop

HYPER to support a version of HyperLTL with propositional quantifications. Although explicit-state model checking approaches allow for the complete verification of arbitrary hyperproperties over non-terminating systems, they suffer from state-explosion which severely hinders scalability. Game-based approaches have also been proposed to automatically derive strategies to instantiate existential quantifiers, both for HyperLTL [16,5] and asynchronous extensions such as OHyperLTL [8] and A-HLTL [9]. The evaluation of [6] suggests that AUTOHYPER can outperform such strategies for HyperLTL.

Most of the tools described above support models at low levels of abstraction or subsets of the SMV language. Recent work has provided a higher-level abstraction for state-of-the-art symbolic and explicit-state model checkers [32].

4 BMC for liveness hyperproperties

We reduce the problem of bounded model checking $\forall^+\exists^+$ hyperproperties with a bound k, i.e., $\models_k \forall^+\exists^+\psi$, to the problem of checking its negation, i.e., $\models_k \exists^+\forall^+\neg\psi$. If this (negated) property is valid, the BMC procedure will yield a counter-example for the original formula. This counter-example is an instantiation for the existentially quantified traces for which $\forall^+\neg\psi$ holds, and since ψ may be a liveness property, refuting it requires lasso-shaped paths.

Algorithm 1 presents an overview of the proposed procedure. As mentioned in the introduction, it searches for k-loop candidates using a *synthesizer* that implements a BMC procedure by reduction to QBF solving, extending the one implemented in HYPERQB [25] with support for lasso-shaped paths. Candidates are then verified by a *checker*, to see if they are true paths of the full system or spurious candidates. Given that the input formula has a single quantifier alternation, once a candidate is found for the outermost existential quantifiers, its

validation amounts to checking $\forall^+\psi$ with the existential quntifiers instantiated with the candidate. If the formula contains more than one universal quantifier, although still a hyperproperty, it can be analyzed using a standard model checker using self-composition [2]. Thus, this *checker* can be implemented using an off-the-self model checker for LTL. If a candidate is found to be spurious, the *synthesizer* is asked to generate a new one, and the process is iterated until a valid candidate is found or all possible candidate k-loops are exhausted, in which case $\not\models_k \exists^+\forall^+\neg\psi$.

To avoid iterating too many candidates, information from the *checker* when a candidate is identified as spurious is integrated into the next iteration of the *synthesizer*, implementing something similar to a CEGAR verification loop [13]. A candidate (an instantiation of the existentially quantified traces) is flagged as spurious by the *checker* when the LTL model checker finds a counter-part instantiation for the universally quantified traces that breaks the property. Rather than just negating the spurious candidate in the *synthesizer* and asking for another candidate, we instead expand the search space of the universally quantified traces. For the candidate to be spurious the counter-part instantiation must contain at least a k'-loop with $k' > k$, otherwise it would have already been found by the *synthesizer*. The k values for the universal quantifications are incremented to this higher k' for the next iteration of the *synthesizer* (bound k is always used for the existentially quantified traces). This excludes the previously found spurious candidate (since the counter-part will be found within the k' length), but also any other potential candidates that have counter-parts of similar length.[2]

Theorem 1 (Infinite inference). *All k-loops returned by Algorithm 1 are witnesses of both $\models_k \exists^+\forall^+\psi$ and $\models \exists^+\forall^+\psi$.*

Proof. $\models_k \exists^+\forall^+\psi$ is guaranteed because the *synthesizer* directly implements the BMC semantics described in Section 2. The returned k-loops are also witnesses of $\models \exists^+\forall^+\psi$ because, to become invalid for any $k' > k$, it would need to become an invalid instantiation of the existential quantifiers (not possible for looping paths because they can be unrolled for any $k' > k$) or for the universal quantification to find longer counter-parts, but this is already discarded by the complete *checker*.

Lemma 1 (Bounded inference). *If Algorithm 1 does not return a witness, then $\not\models_k \exists^+\forall^+\psi$.*

Proof. If no witness is returned, then the procedure found some $k' \geq k$ such that every k-loop instantiation of the existentially quantified traces is rendered invalid by some k'-loop instantiation for the universally quantified traces. Increasing the k' bound for the universally quantified traces does not remove traces from the search-space, so k-loop candidates cannot become valid.

In the remainder of this section we detail the *synthesizer* and the *checker*.

[2] In a sense, this process tries to iteratively approximate the completeness threshold for the instantiated inner problem, the value of k for which an invalid outcomes applies to any trace. Note that this is now a regular LTL problem, for which classical results exist [28]. Nonetheless, statically determining a tight threshold is very difficult in practice or it would be too large for the analysis to be feasible.

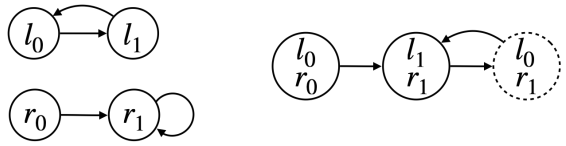

Fig. 2. Lasso synchronization

4.1 The synthesizer

The *synthesizer* implements a rather standard BMC procedure. Likewise the procedure implemented by HyperQB [25], trace quantifiers are handled by reduction to a QBF which can then be checked by an off-the-shelf solver. There are, however, two main differences when compared to HyperQB: we consider lasso-shaped paths in order to enable the verification of liveness properties over arbitrary reactive systems; and we support different bounds for the quantified traces in order to implement the synthesis loop described in Algorithm 1. The second generalization is rather trivial, and to simplify the presentation we will explain the procedure using the same bound k for all traces. The trickier part is the first generalization, because the lassos of different traces can have different lengths which must be aligned.

Going back to our running example hyperproperty, to generate a candidate with BMC we must show that $\models^k \exists\pi_1.\forall\pi_2.\Box(\neg a_{\pi_1} \vee \neg a_{\pi_2})$ holds, that is, find a k-loop π_1 of model L such that for all k-loops π_2 of model R the negation of the original inner LTL formula holds. For $k = 1$ one such candidate is the path $(l_0 l_1)^\omega$ depicted in the left-upper corner of Fig. 2. To show that this is a valid instantiation for π_1 with the bounded semantics we must explore all 1-loops of model R, for example the path $r_0(r_1)^\omega$ depicted in the left-lower corner of Fig. 2. For each such pair of k-loops π_1 and π_2 we must show that $\Box(\neg a_{\pi_1} \vee \neg a_{\pi_2})$ holds. This requires exploring all pairs of states reachable when executing both models synchronously, which in turn requires aligning the lassos of different k-loops to fully explore the combined paths. Given a (k, i)-loop and a (k, j)-loop, to align the respective lassos, we will end-up with a combined lasso path of size $\max(i, j) + \mathrm{lcm}(k+1-i, k+1-j)$, where the last state loops back to state $\max(i, j)$ [26]. For the $(1, 0)$- and $(1, 1)$-loops in the left side of Fig. 2, the combined path is the $(2, 1)$-loop on the right side. The length of the combined path can thus be $(k + 1) \times k$ in the worst case. This exponential growth in the bound necessary to perform BMC of hyperproperties was considered to be too inefficient by the authors of HyperQB [26], and one of the reasons to adopt a bounded semantics considering only finite paths of size k (with the consequent limitation to safety properties or terminating systems). However, as we will show in the evaluation, with proper encodings it is efficient enough to enable BMC of $\forall^+\exists^+$-liveness properties in many complex reactive systems.

The reduction of BMC to QBF is presented in Fig. 3 (the direct encoding of derived operators $\Box$ and $\Diamond$ is also included to ease the understanding). To simplify the presentation we assume that each quantified trace variables π is

$$[\![\exists\pi_1 \dots \exists\pi_n.\forall\pi_{n+1} \dots \forall\pi_m.\psi]\!]_k := \exists\overline{x_{\pi_1}}.\exists l_1 \dots \exists\overline{x_{\pi_n}}.\exists l_n.\forall\overline{x_{\pi_{n+1}}}.\forall l_{n+1} \dots \forall\overline{x_{\pi_m}}.\forall l_m.$$
$$\bigvee 0 \leq i_1, \dots, i_m \leq k$$
$$\Big(l_1 = i_1 \wedge \dots \wedge l_m = i_m \wedge [\![\pi_1]\!]_k^{i_1} \wedge \dots \wedge [\![\pi_n]\!]_k^{i_n} \wedge \Big([\![\pi_{n+1}]\!]_k^{i_{n+1}} \wedge \dots \wedge [\![\pi_m]\!]_k^{i_m} \rightarrow {}_0[\![\psi]\!]_k^{\pi_j \mapsto i_j}\Big)\Big)$$

$$[\![\pi]\!]_k^i := I_\pi(x_\pi^0) \wedge R_\pi(x_\pi^0, x_\pi^1) \wedge \dots \wedge R_\pi(x_\pi^{k-1}, x_\pi^k) \wedge R_\pi(x_\pi^k, x_\pi^i)$$

$$\begin{aligned}
{}_i[\![a_\pi]\!]_k^\Lambda &:= \begin{cases} a_\pi^i \text{ if } i < \Lambda(\pi) \\ a_\pi^{\underline{i}} \text{ otherwise} \end{cases} \\
{}_i[\![\neg\psi]\!]_k^\Lambda &:= \neg {}_i[\![\psi]\!]_k^\Lambda \\
{}_i[\![\psi_1 \wedge \psi_2]\!]_k^\Lambda &:= {}_i[\![\psi_1]\!]_k^\Lambda \wedge {}_i[\![\psi_2]\!]_k^\Lambda \\
{}_i[\![\bigcirc\psi]\!]_k^\Lambda &:= \begin{cases} {}_{i+1}[\![\psi]\!]_k^\Lambda \text{ if } i < \underline{k} \\ {}_{\underline{l}}[\![\psi]\!]_k^\Lambda \quad \text{otherwise} \end{cases} \\
{}_i[\![\Box\psi]\!]_k^\Lambda &:= \bigwedge_{\min(i,\underline{l}) \leq j \leq \underline{k}} {}_j[\![\psi]\!]_k^\Lambda \\
{}_i[\![\Diamond\psi]\!]_k^\Lambda &:= \bigvee_{\min(i,\underline{l}) \leq j \leq \underline{k}} {}_j[\![\psi]\!]_k^\Lambda \\
{}_i[\![\psi_1 \,\mathcal{U}\, \psi_2]\!]_k^\Lambda &:= \bigvee_{i \leq j \leq \underline{k}} \Big({}_j[\![\psi_2]\!]_k^\Lambda \wedge \bigwedge_{i \leq n < j} {}_n[\![\psi_1]\!]_k^\Lambda\Big) \vee \\
&\quad \bigvee_{\underline{l} \leq j < i} \Big({}_j[\![\psi_2]\!]_k^\Lambda \wedge \bigwedge_{i \leq n \leq \underline{k}} {}_n[\![\psi_1]\!]_k^\Lambda \wedge \bigwedge_{\underline{l} \leq n < j} {}_n[\![\psi_1]\!]_k^\Lambda\Big)
\end{aligned}$$

$$\begin{aligned}
\underline{i} &:= \Lambda(\pi) + ((i - \Lambda(\pi)) \bmod (k + 1 - \Lambda(\pi))) \\
\text{where } \underline{l} &:= \max(\{l \mid \pi \mapsto l \in \Lambda\}) \\
\underline{k} &:= \underline{l} + \operatorname{lcm}(\{k + 1 - l \mid \pi \mapsto l \in \Lambda\}) - 1
\end{aligned}$$

Fig. 3. Reducing BMC to QBF solving

implicitly associated with the respective Kripke structures K_π. Moreover, $\overline{x_\pi}$ is a vector of Boolean variables with k copies of the state variables of K_π, x_π^i retrieves the variables from that vector relating to state i, and a_π^i refers to the Boolean variable for the atomic proposition a at state i. The encoding $[\![\exists\pi_1 \dots \forall\pi_m.\psi]\!]_k$ of a HyperLTL formula with m quantifications is parametrized by the bound k. For each quantified trace π_j, the encoding starts with a similar quantification over k copies of the respective state variables, plus one quantified variable l_j that will range over all possible loopback indexes. These quantifications ensure that all possible k-loops are analyzed. The rest of the translation depends on the concrete values of the loops, so an outermost disjunction blasts those for all possible combinations. Then, for each quantified trace variable π, we ensure that the k copies of $\overline{x_\pi}$ and the respective loopback index i behave according to the respective Kripke structure using the formula $[\![\pi]\!]_k^i$, where I_π denotes the initial state predicate of the Kripke structure associated with π, and R_π the respective transition predicate. These (symbolic) predicates can be easily derived from high-level descriptions of Kripke structures. The last conjunct of $[\![\pi]\!]_k^i$ ensures that there is a loop back from the last state k to state i.

Finally, formula ${}_i[\![\psi]\!]_k^\Lambda$ encodes the standard bounded semantics of the inner LTL formula over k-loops [10], with adjustments to consider the combined execution of multiple paths. This encoding is parametrized by the bound k, a mapping Λ from trace variables to the respective loopback indexes being consid-

ered, and the state index i over which the formula should be evaluated (initially 0). To determine the value of an atomic proposition a_π at state i we need to reason about the position of i relative to the respective loopback index $\Lambda(\pi)$: if i is in the prefix before the lasso, the i-th copy of the variable is accessed; otherwise, we need to determine which index $\underline{i}$ in the lasso corresponds to the combined path index i, by performing modular arithmetic over the lasso size, which for variable π is $k + 1 - \Lambda(\pi)$. Note that, for efficiency reasons, only the first k states of the individual paths are represented in the encoding. The states of the combined path with indexes $i > k$ are "virtual", and put together on-the-fly by combining states with possibly different indexes in the individual paths. That is the case of the (dashed) state at index 2 in the combined path on the right side of Fig. 2. To encode the standard semantics of the next operator $\bigcirc\phi$ over the combined lasso path, we need to determine both its last state index $\underline{k}$ and loopback index $\underline{l}$ using the maximum and least common multiple functions as described before: if i happens to be the last state $\underline{k}$ then ϕ is evaluated in the loopback state $\underline{l}$. For the remainder temporal operators, standard bounded semantics over lasso-shaped paths is used [10].

4.2 The checker

Given a HyperLTL formula $\exists\pi_1 ... \forall\pi_m.\psi$ the *checker* is fairly simple to implement. A candidate is an instantiation $\{\pi_1 \mapsto t_1, ..., \pi_n \mapsto t_n\}$ to the existentially quantified trace variables. To check that it is a valid candidate we essentially need to verify that $\forall\pi_{n+1} ... \forall\pi_m.\psi[t_1/\pi_1, ..., t_n/\pi_n]$ holds, that is, that there are no possible counter-part traces that could possibly satisfy the original formula, regardless of the k value. In general this is still a HyperLTL formula, but since it contains only universally quantified trace variables it can be verified by a standard LTL model checker using self-composition [2].

Since performing the substitution $\psi[t_1/\pi_1, ..., t_n/\pi_n]$ is not trivial, we instead verify formula $\forall\pi_1 ... \forall\pi_n.\forall\pi_{n+1} ... \forall\pi_m.\psi$ using self-composition, adjusting models K_{π_j} for $1 \leq j \leq n$ to only admit the single trace t_i. This can be done by adding an extra variable s_{π_i} to each model π_i ranging over possible state indexes 0 to k, constraining this variable to encode the correct evolution of the indexes in the k-loop t_i, and adding invariants ensuring the state variables of model π_i have the concrete values at each state in t_i; we denote this procedure by $K_{\pi_j} \oplus t_j$. When checking if the candidate $\pi_1 = (l_0 l_1)^\omega$ of model L satisfies $\forall\pi_2.\square(\neg a_{\pi_1} \vee \neg a_{\pi_2})$, the complete model checker will return a counter-example because the candidate is spurious. As mentioned above, this counter-example is inspected by the *checker* to determine the value $k' > k$ to be used in the next iteration of the *synthesizer* to bound the existentially quantified traces.

4.3 HyperLasso

We implement the technique described above as the HyperLasso bounded model checker for HyperLTL[3]. HyperLasso receives a HyperLTL formula (spec-

[3] The tool is publicly available at anonymous.4open.science/r/HyperLasso-083C/.

ified with the HyperQB syntax), a bound k, and a model for each quantified trace (defined in the SMV syntax, from which the Kripke structures can be easily extracted). For our running example, the input formula to be checked and the respective SMV models are presented in Fig. 4. Each model from Fig. 1 contains a single integer enumeration variable whose value ranges over possible state indexes (for example, in the left model, $l = k$ iff the system is in state l_k).

HyperLasso then checks if there exist k-loop counter-examples for the outermost $\forall$ quantifiers that refute the formula. If no such counter-examples are found, the formula is reported as "maybe valid" as counter-examples might still exist for higher values of k. HyperLasso relies on off-the-shelf tools that follow standard formats, and can easily be swapped given advances in the state of the art. The overall technique and implementation are evaluated in the next section.

Synthesizer To solve QBF we use Z3 [20], using a subset of the SMT-LIB language [1] consisting of quantifiers over bit-vector variables. In comparison to the encoding described in Fig. 3, the most significant remark is that we do not need to blast all loop combinations eagerly at the start. Instead, we lazily blast loop combinations only when temporal operators appear in a formula, and only for the traces that are mentioned in the respective sub-formula. Our preliminary experiments revealed that this strategy yields smaller formulas that are also easier to solve.. Since Z3 is quite sensitive to the number of quantified variables we also implemented a couple more optimizations: we identify "assignment expressions" (such as $x = 5$ or $x = y$) and avoid introducing quantifiers for the "assigned" variable whose value is already known; in the BMC procedure we bit-pack all k-expanded variables for each model variable in a single bit vector.

Checker For the LTL model checker, we rely on nuXmv [12]. We first perform the self-composition into a single SMV model of the models corresponding to the innermost universally quantified traces[4]. It is then simple to extend the self-composed model with dedicated variables and assignments that fix the outermost existential traces. The negation of the LTL body of the original HyperLTL formula is specified as the property to be checked. For example, the SMV model in Fig. 5 represents the checking problem when L is fixed to the candidate $(l_0, l_1)^\omega$ (which, as we have seen, is spurious).

5 Evaluation

Our evaluation aimed at answering the following research questions (RQ):

RQ1 How does the efficiency of HyperLasso compare with the state-of-the-art explicit model checking tools?

RQ2 How prevalent is the issue of spurious counter-example candidates?

Recall from Section 3 that the explicit-state AutoHyper is the only model checker for HyperLTL that supports liveness hyperproperties over non-terminating systems, so it is the focus of our comparison with the state-of-the-art.

[4] Future work could consider experimental nuXmv support for self-composition [11].

```
forall L. exists R. F (a[L] & a[R])
```

```
MODULE main
VAR l : 0..1;
ASSIGN
    init(l) := 0;
    next(l) := case
        l = 0 : {0,1};
        TRUE : {0,1};
    esac;
DEFINE a := (l = 0);
```

```
MODULE main
VAR r : 0..2;
ASSIGN
    init(r) := 0;
    next(r) := case
        r = 0 : 1;
        r = 1 : {1,2};
        TRUE : 1;
    esac;
DEFINE a := (r = 2);
```

Fig. 4. Input hyperproperty and SMV models L and R

5.1 Benchmark

To answer these RQs we defined a new benchmark suite[5], since existing benchmarks consisted mostly of safety properties or trivial toy examples. Although HyperLasso supports SMV models written in a declarative style, the models in our benchmark are all described with explicit (possibly non-deterministic) assigns because this is the only style accepted by AutoHyper.

Our benchmark contains 5 verification and 2 synthesis problems. The difference between them is that in the latter the goal is to verify that a $\exists\forall$-safety property is valid by finding appropriate instantiations for the existentially quantified variables. Obviously, this is just the dual problem of $\forall\exists$-liveness and can be verified with the same technique. All our examples are parameterized by some sort of scope that dictates their size, and several instantiations of those scopes were verified to assess the tools' scalability. Our benchmark contains both valid and invalid properties in verification problems, and both satisfiable and unsatisfiable properties in synthesis problems. BMC is particularly useful for invalid / satisfiable properties, where the found counter-examples / instances constitute a proof that applies to the full system, but we also compared the performance in valid / unsatisfiable properties for reasonable bounds k. Table 1 summarizes the information for each **Property**, showing the sequence of trace quantifiers ($\mathbb{Q}^\star$), the **Models** over which the quantifiers range (if a single model is listed, all quantifers range over it), and a restriction on the **Scope** ($-$ means the result holds regardless of scope) to guarantee the expected outcome (**Res**, ✓ for valid / satisfiable verification / synthesis, ✗ for invalid / unsatisfiable verification / synthesis) according to (non-bounded) HyperLTL semantics. The 5 verification problems (and the paper's running toy example) appear at the top, and the 2 synthesis ones at the bottom. We now describe these problems in more detail.

Generalized Non-Interference in a Conference Management System This verification problem is slightly adapted from one of the examples used in the evaluation of [32]. It describes a conference management system where a papers are

[5] The benchmark is publicly available in HyperLasso's repository.

```
MODULE main
VAR
    L_l : 0..1;
    R_r : 0..2;
    L_s : 0..1;
ASSIGN
    init(L_l) := 0;
    next(L_l) := case L_l = 0 : {0,1}; TRUE : {0,1}; esac;
    init(R_r) := 0;
    next(R_r) := case R_r = 0 : 1; R_r = 1 : {1,2}; TRUE : 1; esac;
    init(L_s) := 0;
    next(L_s) := case L_s < 1 : L_s+1; TRUE : 0; esac;
INVAR
    (L_s = 0 & L_l = 0) | (L_s = 1 & L_l = 1);
DEFINE
    L_a := (L_l = 0);
    R_a := (R_r = 2);
LTLSPEC
    G (!L_a | !R_a)
```

Fig. 5. SMV model for checking a candidate

rated by r reviewers, before an acceptance decision is made for each paper. The goal is to verify that the system satisfies *Generalized Non-Interference* (GNI), namely that individual ratings are not leaked to reviewers to which papers are not assigned. The system is non-terminating because reviewers are allowed to change their ratings while no final decision is made on an assigned paper. Unlike traditional GNI, which is just a safety property, this variant is a liveness property because it requires all papers to eventually have final decisions and is also conditional on the alignment of traces: the system model allows stuttering, and as shown in [30], public events must be aligned to happen at the same time in different traces to ensure that counter-examples make sense. This example contains three variants of the system, two where GNI holds and one where it does not: `cms_max_`r`x`a where the final decision for each paper is its highest rating; `cms_ndet_`r`x`a where the final decision for each paper is any of its ratings; and `cms_any_`r`x`a where the final decision for each paper is any rating of any paper.

Isolation Level Equivalence Checker For efficiency reasons, distributed transactional databases can be run with different *isolation levels*. The strongest one is *Serializability* (SER), where the results of trying to commit a set of transactions are the same as if the database was non-distributed. Among the weaker isolation levels we have *Read Committed* (RC), where one transaction is not allowed to read data modified but not yet committed by another transaction. Several formalisms have been proposed to specify such isolation levels, for example the one by Crooks et al. [18] that we use in this problem. It is not trivial to reason about isolation levels, for example to check their equivalence, because they are basically defined as hyperproperties: given a transactional workload, a database

Table 1. Benchmark summary

Property	$\mathbb{Q}^\star$	Model(s)	Scope	Res
toy	$\forall\exists$	left right	–	✗
gni_any_rxa	$\forall\forall\exists$	cms_any_rxa	$r \geq 2 \wedge a \geq 2$	✗
gni_ndet_rxa	$\forall\forall\exists$	cms_ndet_rxa	–	✓
gni_max_rxa	$\forall\forall\exists$	cms_max_rxa	–	✓
rc_ser_txv	$\forall\exists$	rc_txv ser_txv	$t \geq 3 \wedge v \geq 2$	✗
ser_rc_txv	$\forall\exists$	ser_txv rc_txv	–	✓
cni_any_n	$\forall\exists$	mask_any_n	–	✗
cni_lte_n	$\forall\exists$	mask_lte_n	$n \geq 3$	✓
selfstab_gf_n	$\forall\exists$	herman_n	$n \geq 2$	✗
selfstab_lf_n	$\forall\exists$	herman_n	–	✓
robust_crit_n	$\forall\exists$	bakery_n	–	✗
robust_ncrit_n	$\forall\exists$	bakery_n	$n \geq 2$	✓
spec1_sxt	$\exists\forall$	controller_sxt system_s	$s \geq 4$	✓
spec2_sxt	$\exists\forall$	controller_sxt system_s	$t \geq 2$	✓
plan_nxe	$\exists\forall^e$	robot_n	$n > e$	✓
plan_nxe	$\exists\forall^e$	robot_n	$n = e$	✗

satisfies an isolation level if there exists some execution where all transactions can be committed respecting some rules about what values can be read or written. The verified hyperproperty checks if a database running a given isolation level also satisfies another isolation level, assuming a workload of t transactions and v keys each holding one of v possible values. It is a liveness property because the latter database must eventually commit all transactions. Two different models of databases are encoded: ser_txv encodes a database ensuring SER and rc_txv a database ensuring RC. Obviously, SER implies RC (ser_rc_txv), while RC does not imply SER (rc_ser_txv), with the counter-example being the well-known *dirty read* anomaly of RC.

Information Flow in Multi-threaded Programs This problem considers a $\forall\exists$ variant of non-interference (NI) proposed in [33] to check whether high-security information can flow to low-security variables. The model encodes a multi-threaded imperative program that applies a mask to a secret PIN, parameter n denoting the length of the bitvectors. Although a variant of this problem is present in existing hyperproperty benchmarks [25], it is a synchronous and terminating version. The version in our benchmark is asynchronous, with each thread acting interleaved according to a non-deterministic scheduler, as originally defined. This makes the system non-terminating since the scheduler may choose threads that are blocked. The property to be verified is a liveness one because it states that, if a fair scheduler is considered, for whatever trace there is another with a different secret that eventually reaches the same public output. This property is invalid if the mask is chosen arbitrarily (cni_any_n) but valid if the mask is less than half the PIN (cni_lte_n).

Self-stabilization Algorithm for Ring of Processes This problem encodes an algorithm for self-stabilization of a ring of n identical processes proposed by Her-

man [22], inspired by the benchmark of [31]. The goal is to guarantee that, that any ring with an odd number of processes circulating an arbitrary number of tokens eventually stabilizes in a single circulating token. The system is non-terminating because the scheduler chooses the process to act arbitrarily, and the acting process can choose to act or not depending on a coin flip. Self-stabilization is a liveness hyperproperty because it states that whatever execution, there is an execution with the same initial token configuration and scheduler choices that can stabilize. This property is invalid for a global notion of fairness where some process will always eventually act (`selfstab_gf_`n), but not for a local notion of fairness where all processes will always eventually act (`selfstab_lf_`n). [31] constructed a proof for an arbitrary n with additional linear assumptions due to the inability to model fairness constraints.

Robustness of the Bakery Protocol. This problem verifies a robustness hyperproperty [16] on Lamport's Bakery Protocol [29], a classic mutual exclusion algorithm where n processes take tickets to enter a critical section. In this context, robustness defines the system's fault tolerance: specifically, its ability to maintain liveness despite a process failure. This is a non-terminating model since the processes continuously attempt to access the critical section. We verify that if a process halts in the non-critical section, it does not block others from progressing, or more precisely, for every trace in which a process halts outside the critical section, there exists a trace in which that process behaves the same and the remaining processes still succeed in entering the critical section (`robust_ncrit_`n). In general, if a process halts while holding a ticket or inside the critical section, it can permanently starve other processes (`robust_crit_`n).

Controller Synthesis The goal in this problem is to synthesize a controller for two semaphores at an intersection. A controller is a Mealy machine that for each input (requests for the semaphores) decides what is the output (which semaphores are green). The machine can have some sort of memory by having s different states, and can have t transitions, each mapping a source state and input to a destination state and an output. Each state also has a default transition with no input that applies when no other transition can be executed. A controller `controller_`s`x`t can run on a `system_`s with a state variable ranging over the s possible states. We have two variant specifications. In both we wish to ensure that all possible executions satisfy the safety requirement that the two semaphores are not green simultaneously and the (liveness) requirement that all requests are eventually attended: `spec1_`s`x`t additionally requires that when a semaphore turns green it stays green for at least two states in a row, while `spec2_`s`x`t relaxes the requirements by assuming that two semaphore requests cannot appear consecutively. This is an example with a rather degenerate hyperproperty, where the first quantifier ranges over possible controllers modeled just by frozen variables (variables that once assigned in the initial state cannot change value). However, we recon it is a rather common patter in synthesis problems where one only wishes to find an appropriate system configuration.

Path Planning In this problem the goal is to synthesize a path for a robot moving in an $n \times n$ board that is guaranteed to avoid e enemies. It is inspired by other robot path planning problems from the hyperproperty literature [25]. The robot starts in position $(0, 0)$ while each enemy $0 \leq i < e$ starts in position $(n-1, i)$ and is only allowed to move in row i. This is a non-terminating system because the robots keep running indefinitely. If $e < n$ it is possible to find such a path – the robot moves down to row $n-1$ and then keeps moving right and left, meaning that it requires a lasso trace to be satisfiable. If seen from a verification perspective, `plan_`n`x`e is a liveness property because no such path exists if any of the enemies eventually catches the robot. Both the robot and the enemies are described with the same generic model `robot_`n, with the start positions and movement restrictions encoded in the property.

5.2 Results

Table 2 contains the results of our evaluation. For each **Property**, we recall the expected outcome (**Res**), and show the time taken by AutoHyper to verify the property (**AH**), and several metrics for HyperLasso, namely: the considered BMC bound k, the bound k' for the inner universally quantified traces by the end of the verification loop, the number of **Cand**idadate**s** tried (for invalid / satisfiable problems, this includes the valid candidate), the time spent in (possibly multiple iterations) of the ***Synth**esizer* and of the ***Check**er*, as well as the **Total** verification time. We run AutoHyper with witness mode enabled and for a fair comparison, we selected the best performing underlying automata solver for each class of examples. All properties were checked in a MacBook Pro M1 machine with 16GB RAM, with a timeout of 300s (marked `TO` in the table). While AutoHyper performs complete model checking, exploring all paths regardless of length, HyperLasso performs bounded model checking, and thus a k bound must be chosen for each problem. For invalid / satisfiable problems, since k-loops returned by HyperLasso are guaranteed to be paths of the full model, we select the minimal value that produces a witness; for valid / unsatisfiable problems, we select the largest k value for which HyperLasso responds within the timeout, i.e., the maximum level of confidence achievable within the timeout.

Regarding **RQ1**, the results show that HyperLasso consistently outperforms AutoHyper for any problem beyond minimal scopes. In some cases this minimal scope required for AutoHyper to provide an answer is below the restrictions stated in Table 1, changing the property's expected outcome (which we mark as red). Obviously AutoHyper is solving a much harder problem than HyperLasso. However, AutoHyper's feedback is all-or-nothing, while HyperLasso guarantees that all k-loop bugs are found, providing a more incremental level of confidence. However, it should be noted that for some valid problems, HyperLasso can provide a false sense of confidence, since small values k also make certain problems vacuous. This is evidenced, for instance, by the sudden decrease of solving time in `selfstab_lf_7`: this problem's fairness constraint forces the token to at least go around the ring once in all traces,

Table 2. Evaluation results

			HyperLasso					
Property	**Res**	**AH**	k	k'	**Cands**	**Synth**	**Check**	**Total**
toy	✗	0.5	1	–	1	0.2	0.1	0.3
gni_any_1x1	✓	12.1	8	–	0	137.9	0.0	137.9
gni_any_2x2	✗	TO	4	–	1	1.3	0.1	1.4
gni_any_3x3	✗	TO	6	–	1	3.7	0.1	3.8
gni_ndet_1x1	✓	19.1	8	–	0	194.4	0.0	194.4
gni_ndet_2x2	✓	TO	5	–	0	165.0	0.0	165.0
gni_ndet_3x3	✓	TO	5	–	0	2.5	0.0	2.5
gni_max_1x1	✓	19.6	8	–	0	209.8	0.0	209.8
gni_max_2x2	✓	TO	5	–	0	96.2	0.0	96.2
gni_max_3x3	✓	TO	5	–	0	2.5	0.0	2.5
rc_ser_2x1	✓	0.6	109	–	0	287.7	0.0	287.7
rc_ser_3x2	✗	TO	3	–	1	0.3	0.1	0.4
rc_ser_4x3	✗	TO	4	–	1	1.5	0.1	1.6
ser_rc_2x1	✓	0.6	99	–	0	295.4	0.0	295.4
ser_rc_3x2	✓	TO	17	–	0	228.1	0.0	228.1
ser_rc_4x3	✓	TO	3	–	0	1.2	0.0	1.2
cni_any_1	✗	1.0	12	–	1	7.9	1.2	9.1
cni_any_2	✗	TO	22	–	1	64.8	0.7	65.5
cni_any_3	✗	TO	22	–	1	156.1	0.2	156.3
cni_any_4	✗	TO	22	–	1	222.2	0.3	222.5
cni_lte_1	✗	1.0	2	–	1	0.3	0.1	0.4
cni_lte_2	✗	TO	23	–	1	65.7	0.2	65.9
cni_lte_3	✓	TO	24	–	0	267.2	0.0	267.2
cni_lte_4	✓	TO	22	–	0	240.6	0.0	240.6
selfstab_gf_3	✗	0.9	1	7	2	0.3	0.3	0.6
selfstab_gf_5	✗	39.6	1	5	2	0.4	0.2	0.6
selfstab_gf_7	✗	TO	1	6	2	0.5	0.2	0.7
selfstab_lf_3	✓	0.8	5	12	2	117.3	0.5	117.8
selfstab_lf_5	✓	TO	4	18	1	86.9	0.1	87.0
selfstab_lf_7	✓	TO	5	–	0	0.8	0.0	0.8
robust_crit_1	✗	0.8	2	–	1	0.2	1.0	1.2
robust_crit_2	✗	TO	2	6	2	0.4	0.3	0.7
robust_crit_3	✗	TO	2	23	2	7.5	1.2	8.7
robust_ncrit_1	✗	0.9	13	–	1	0.2	1.0	1.2
robust_ncrit_2	✓	TO	13	19	1	119.2	0.2	119.4
robust_ncrit_3	✓	TO	11	30	1	264.9	0.2	265.1
spec1_2x0	✗	0.5	27	–	0	220.1	0.0	220.1
spec1_4x0	✓	TO	0	6	3	4.1	0.2	4.3
spec1_4x4	✓	TO	0	3	2	101.3	2.0	103.3
spec2_1x1	✗	TO	24	–	0	192.5	0.0	192.5
spec2_1x2	✓	TO	0	2	2	0.1	0.1	0.2
spec2_4x4	✓	TO	0	3	2	0.5	2.1	2.6
plan_3x1	✓	20.7	2	–	1	0.3	1.0	1.3
plan_3x2	✓	TO	3	–	1	0.4	0.1	0.5
plan_3x3	✗	TO	21	–	0	231.8	0.0	231.8
plan_4x1	✓	TO	2	–	1	0.3	0.2	0.5
plan_4x2	✓	TO	3	–	1	0.6	0.1	0.7
plan_4x3	✓	TO	4	–	1	1.3	0.4	1.7
plan_4x4	✗	TO	18	–	0	246.4	0.0	246.4

which is not possible with $k = 6$. A similar phenomenon occurs, for instance, for `gni_ndet_3x3`, `gni_max_3x3`, or `ser_rc_4x3`.

Focusing on **RQ2**, results show that, although not prevalent, spurious candidates occurred for both invalid / satisfiable cases, namely `selfstab_gf`, `robust_crit`, `spec1` or `spec2`, where spurious counter-examples are discarded before a valid one is found; and in valid / unsatisfiable problems, namely `selfstab_lf` and `robust_ncrit`, where spurious candidates must be discarded before concluding the formula is valid. This means that the *checker* was effectively needed to guarantee sound results. There are also other problems for which we know spurious candidates exist (such as the satisfiable path planning problems), but the first candidate returned by the *synthesizer* happened to be valid. Another concern was whether the number of iterations would encumber the whole process. For our problems, the number of spurious candidates generated was quite low, never going beyond 2. Results also show that the time spent on the *checker* was negligible compared with that spent on the *synthesizer*, so this extra step does not seem to be a bottleneck. In order to further reason about the impact of the iteration, we look at the final bound $k' > k$ for the inner universally quantified traces. Interestingly, for several problems the process jumped in a single iteration to a k' that ensured termination in the next iteration. That was the case, for example, of `selfstab_lf_5` that jumped from bound 4 to bound 18 in a single iteration, allowing HyperLasso to conclude that the formula was valid for bound k in the next iteration. Notice that this bound k' is not necessarily minimal, since counter-examples generated by nuXmv (in complete mode) are not guaranteed to have minimal length. Nonetheless, this over-approximation did not seem to affect the performance of the procedure.

6 Conclusion

This paper proposed the first symbolic technique for the bounded model checking of arbitrary HyperLTL properties over reactive systems, including the scarcely addressed class of $\forall^+\exists^+$-liveness hyperproperties. The technique consists of a verification loop where candidate witnesses are synthesized by a bounded procedure, and subsequently checked for validity with a complete model checker for plain LTL. We created a benchmark suite specially tailored for this class of properties, and used it to evaluate our implementation of the technique, HyperLasso. Results show that HyperLasso consistently outperforms AutoHyper, the only other state-of-the-art tool that is able to support our benchmarks.

For future work, we intend to generalize this technique for multiple quantifier alternations. This is expected to have a toll on performance, so we may need to implement further optimizations to guarantee scalability, such as more refined strategies to handle the BMC bound or the support for more solving procedures, such as dedicated QBF solvers.

References

1. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)
2. Barthe, G., D'argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical Structures in Computer Science **21**(6), 1207–1252 (2011)
3. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: CAV (1). LNCS, vol. 12759, pp. 694–717. Springer (2021)
4. Beutner, R.: Automated software verification of hyperliveness. In: TACAS (2). LNCS, vol. 14571, pp. 196–216. Springer (2024)
5. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: CSF. pp. 471–485. IEEE (2022)
6. Beutner, R., Finkbeiner, B.: AutoHyper: Explicit-state model checking for HyperLTL. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 145–163. Springer (2023)
7. Beutner, R., Finkbeiner, B.: Model checking omega-regular hyperproperties with AutoHyperQ. In: LPAR. EPiC Series in Computing, vol. 94, pp. 23–35. EasyChair (2023)
8. Beutner, R., Finkbeiner, B.: Predicate abstraction for hyperliveness verification. Formal Methods Syst. Des. **66**(2), 238–277 (2025)
9. Beutner, R., Finkbeiner, B.: Verifying asynchronous hyperproperties in reactive systems. Proc. ACM Program. Lang. **9**(OOPSLA2) (Oct 2025). https://doi.org/10.1145/3763130, https://doi.org/10.1145/3763130
10. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: International conference on tools and algorithms for the construction and analysis of systems. pp. 193–207. Springer (1999)
11. Bombardelli, A., Bozzelli, L., Sanchez, C., Tonetta, S.: (Asynchronous) temporal logics for hyperproperties on finite traces. In: International Symposium on Model Checking Software. pp. 25–43. Springer (2025)
12. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: International Conference on Computer Aided Verification. pp. 334–342. Springer (2014)
13. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) **50**(5), 752–794 (2003)
14. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference on Principles of Security and Trust. pp. 265–284. Springer (2014)
15. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010)
16. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: International Conference on Computer Aided Verification. pp. 121–139. Springer (2019)
17. Correnson, A., Nießen, T., Finkbeiner, B., Weissenbacher, G.: Finding $\forall\exists$ hyperbugs using symbolic execution. Proc. ACM Program. Lang. **8**(OOPSLA2), 1420–1445 (2024)
18. Crooks, N., Pu, Y., Alvisi, L., Clement, A.: Seeing is believing: A client-centric specification of database isolation. In: Proceedings of the ACM Symposium on Principles of Distributed Computing. pp. 73–82 (2017)

19. Dardinier, T., Li, A., Müller, P.: Hypra: A deductive program verifier for hyper Hoare logic. Proc. ACM Program. Lang. **8**(OOPSLA2), 1279–1308 (2024)
20. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
21. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: CAV (1). LNCS, vol. 9206, pp. 30–48. Springer (2015)
22. Herman, T.: Probabilistic self-stabilization. Inf. Process. Lett. **35**(2), 63–67 (1990)
23. Hsu, T., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: Bounded model checking for asynchronous hyperproperties. In: TACAS (1). LNCS, vol. 13993, pp. 29–46. Springer (2023)
24. Hsu, T.H., Rabizadeh, M., Rogale, K., Filippov, F., de Oliveira Batista, M.A., Sánchez, C., Bonakdarpour, B.: Hyperqb 2.0: A bounded model checker for hyperproperties (2025), https://arxiv.org/abs/2109.12989
25. Hsu, T.H., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 94–112. Springer (2021)
26. Hsu, T.H., Sánchez, C., Sheinvald, S., Bonakdarpour, B.: Efficient loop conditions for bounded model checking hyperproperties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 66–84. Springer (2023)
27. Itzhaky, S., Shoham, S., Vizel, Y.: Hyperproperty verification as CHC satisfiability. In: ESOP (2). LNCS, vol. 14577, pp. 212–241. Springer (2024)
28. Kroening, D., Ouaknine, J., Strichman, O., Wahl, T., Worrell, J.: Linear completeness thresholds for bounded model checking. In: CAV. LNCS, vol. 6806, pp. 557–572. Springer (2011)
29. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. In: Concurrency: the works of Leslie Lamport, pp. 171–178 (2019)
30. Lamport, L., Schneider, F.B.: Verifying hyperproperties with TLA. In: 2021 IEEE 34th Computer Security Foundations Symposium (CSF). pp. 1–16. IEEE (2021)
31. Lin, A.W., Rümmer, P.: Liveness of randomised parameterised systems under arbitrary schedulers. In: CAV (2). LNCS, vol. 9780, pp. 112–133. Springer (2016)
32. Macedo, N., Pacheco, H.: Hyper model checking for high-level relational models (2025), https://arxiv.org/abs/2512.12024
33. Smith, G., Volpano, D.M.: Secure information flow in a multi-threaded imperative language. In: POPL. pp. 355–364. ACM (1998)