# Program Repair for Hyperproperties

Borzoo Bonakdarpour[1] and Bernd Finkbeiner[2]

[1] Iowa State University, USA, borzoo@iastate.edu
[2] Saarland University, Germany, finkbeiner@cs.uni-saarland.de

**Abstract.** We study the repair problem for hyperproperties specified in the temporal logic HyperLTL. Hyperproperties are system properties that relate multiple computation traces. This class of properties includes information flow policies like noninterference and observational determinism. The repair problem is to find, for a given Kripke structure, a substructure that satisfies a given specification. We show that the repair problem is decidable for HyperLTL specifications and finite-state Kripke structures. We provide a detailed complexity analysis for different fragments of HyperLTL and different system types: tree-shaped, acyclic, and general Kripke structures.

## 1 Introduction

*Information-flow security* is concerned with the detection of unwanted flows of information from a set of variables deemed as secrets to another set of variables that are publicly observable. Information-flow security is foundational for some of the pillars of cybersecurity such as confidentiality, secrecy, and privacy. Information-flow properties belong to the class of hyperproperties [12], which generalize trace properties to sets of sets of traces. Trace properties are usually insufficient, because information-flow properties relate multiple executions. This also means that classic trace-based specification languages such as linear-time temporal logic (LTL) cannot be used directly to specify information-flow properties. HyperLTL [11] is an extension of LTL with trace variables and quantifiers. HyperLTL can express information-flow properties by simultaneously referring to multiple traces. For example, *noninterference* [28] between a secret input $h$ and a public output $o$ can be specified in HyperLTL by stating that, for all pairs of traces $\pi$ and $\pi'$, if the input is the same for all input variables $I$ except $h$, then the output $o$ must be the same at all times:

$$\forall \pi . \forall \pi' .\ \Box\Big( \bigwedge_{i \in I \setminus \{h\}} i_\pi = i_{\pi'} \Big) \ \Rightarrow\ \Box(o_\pi = o_{\pi'})$$

Another prominent example is *generalized noninterference* (GNI) [36], which can be expressed as the following HyperLTL formula:

$$\forall \pi . \forall \pi' . \exists \pi'' .\ \Box(h_\pi = h_{\pi''}) \ \wedge\ \Box(o_{\pi'} = o_{\pi''})$$

The existential quantifier is needed to allow for nondeterminism. Generalized noninterference permits nondeterminism in the low-observable behavior, but stipulates that low-security outputs may not be altered by the injection of high-security inputs.

There has been a lot of recent progress in automatically *verifying* [14, 23–25] and *monitoring* [2, 8, 9, 21, 22, 29, 39] HyperLTL specifications. The automatic *construction* of systems that satisfy a given set of information-flow properties is still, however, in its infancy. So far, the only known approach is bounded synthesis [14, 19], which searches for an implementation up to a given bound on the number of states. While there has been some success in applying bounded synthesis to systems like the dining cryptographers [10], this approach does not yet scale to larger systems. The general synthesis problem (without the bound on the number of states) becomes undecidable as soon as the HyperLTL formula contains two universal quantifiers [19]. A less complex type of synthesis is *program repair*, where, given a model $\mathcal{K}$ and a property $\varphi$, the goal is to construct a model $\mathcal{K}'$, such that (1) any execution of $\mathcal{K}'$ is also an execution of $\mathcal{K}$, and (2) $\mathcal{K}'$ satisfies $\varphi$. A useful application of program repair is *program sketching*, where the developer provides a program with "holes" that are filled in by the synthesis algorithm [38]. Filling a hole in a program sketch is a repair step that eliminates nondeterminism. While such a repair is guaranteed to preserve trace properties, it is well known that this is not the case in the context of information-flow security policies [30]. In fact, this problem has not yet been studied in the context of hyperproperties.

In this paper, we study the problem of automated program repair of finite-state systems with respect to HyperLTL specifications. We provide a detailed analysis of the complexity of the repair problem for different shapes of the structure: we are interested in *general*, *acyclic*, and *tree-shaped* Kripke structures. The need for investigating the repair problem for tree-shaped and acyclic graphs stems from two reasons. First, many trace logs that can be used as a basis for example-based synthesis [4] and repair are in the form of a simple linear collection of the traces seen so far. Or, for space efficiency, the traces are organized by common prefixes and assembled into a tree-shaped Kripke structure, or by common prefixes as well as suffixes assembled into an acyclic Kripke structure. The second reason is that tree-shaped and acyclic Kripke structures often occur as the natural representation of the state space of a protocol. For example, certain security protocols, such as authentication and session-based protocols (e.g., TLS, SSL, SIP) go through a finite sequence of *phases*, resulting in an acyclic Kripke structure.

Table 1 summarizes the contributions of this paper. It shows our results on the complexity of automated program repair with respect to different fragments of HyperLTL. The complexities are in the size of the Kripke structure. This *system complexity* is the most relevant complexity in practice, because the system tends to be much larger than the specification. Our results show that the shape of the Kripke structure plays a crucial role in the complexity of the repair problem:

– **Trees.** For trees, the complexity in the size of the Kripke structure does not go beyond $\mathsf{NP}$. The problem for the alternation-free fragment and the fragment with one quantifier alternation where the leading quantifier is ex-

<table>
<tr><th>HyperLTL fragment</th><th>Tree</th><th colspan="2">Acyclic</th><th colspan="2">General</th></tr>
<tr><td>$E^*$</td><td rowspan="4">L-complete<br>*(Theorem 1)*</td><td colspan="2" rowspan="3">NL-complete<br>*(Theorems 5 and 6)*</td><td colspan="2">NL-complete<br>*(Theorem 8)*</td></tr>
<tr><td>$A^*$</td><td colspan="2">NP-complete<br>*(Theorem 9)*</td></tr>
<tr><td>$EA^*$</td><td rowspan="2">PSPACE</td><td rowspan="6">*(Theorem 10)*</td></tr>
<tr><td>$E^*A^*$</td><td>$\Sigma_2^p$</td><td rowspan="5">*(Theorem 7)*</td></tr>
<tr><td>$AE^*$</td><td>P-complete<br>*(Theorem 2)*</td><td rowspan="2">$\Sigma_2^p$-complete</td><td rowspan="2">PSPACE-complete</td></tr>
<tr><td>$A^*E^*$</td><td rowspan="4">NP-complete<br>*(Corollary 1)*</td></tr>
<tr><td>$(E^*A^*)^k$</td><td>$\Sigma_k^p$-complete</td><td rowspan="2">$(k-1)$-EXPSPACE-complete</td></tr>
<tr><td>$(A^*E^*)^k$</td><td>$\Sigma_{k+1}^p$-complete</td></tr>
<tr><td>$(A^*E^*)^*$</td><td colspan="2">PSPACE<br>*(Corollary 2)*</td><td colspan="2">NONELEMENTARY<br>*(Corollary 3)*</td></tr>
</table>

Table 1: Complexity of the HyperLTL repair problem in the size of the Kripke structure, where $k$ is the number of quantifier alternations in the formula.

istential is L-complete. The problem for the fragment with one quantifier alternation where the leading quantifier is universal is P-complete and is NP-complete for full HyperLTL.

- **Acyclic graphs.** For acyclic Kripke structures, the complexity is NL-complete for the alternation-free fragment and the fragment with one quantifier alternation where the leading quantifier is existential. The complexity is in the level of the polynomial hierarchy that corresponds to the number of quantifier alternations.
- **General graphs.** For general Kripke structures, the complexity is NL-complete for the existential fragment and NP-complete for the universal fragment. The complexity is PSPACE-complete for the fragment with one quantifier alternation and $(k-1)$-EXPSPACE-complete in the number $k$ of quantifier alternations.

We believe that the results of this paper provide the fundamental understanding of the repair problem for secure information flow and pave the way for further research on developing efficient and scalable techniques.

*Organization* The remainder of this paper is organized as follows. In Section 2, we review Kripke structures and HyperLTL. We present a detailed motivating example in Section 3. The formal statement of our repair problem is in Section 4. Section 5 presents our results on the complexity of repair for HyperLTL in the size of tree-shaped Kripke structures. Sections 6 and 7 present the results on the complexity of repair in acyclic and general graphs, respectively. We discuss related work in Section 8. We conclude with a discussion of future work in Section 9. Detailed proofs appear in the appendix.

## 2 Preliminaries

### 2.1 Kripke Structures

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

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

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

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

Figure 1 shows an example Kripke structure where $L(s_{init}) = \{a\}, L(s_3) = \{b\}$, etc. The *size* of the Kripke structure is the number of its states. The directed graph $\mathcal{F} = \langle S, \delta\rangle$ is called the *Kripke frame* of the Kripke structure $\mathcal{K}$. A *loop* in $\mathcal{F}$ is a finite sequence $s_0 s_1 \cdots s_n$, such that $(s_i, s_{i+1}) \in \delta$, for all $0 \le i < n$, and $(s_n, s_0) \in \delta$. We call a Kripke frame *acyclic*, if the only loops are self-loops on otherwise terminal states, i.e., on states that have no other outgoing transition. See Fig. 1 for an example. Since Definition 1 does not allow terminal states, we only consider acyclic Kripke structures with such added self-loops.

We call a Kripke frame *tree-shaped*, or, in short, a *tree*, if every state $s$ has a unique state $s'$ with $(s', s) \in \delta$, except for the root node, which has no predecessor, and the leaf nodes, which, again because of Definition 1, additionally have a self-loop but no other outgoing transitions.

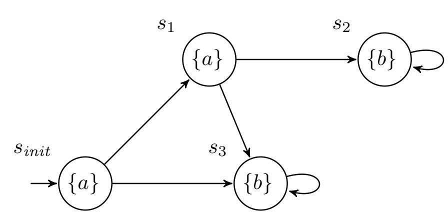

Fig. 1: An acyclic Kripke structure.

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

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

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

In some cases, the system at hand is given as a tree-shaped or acyclic Kripke structure. Examples include session-based security protocols and space-efficient execution logs, because trees allow us to organize the traces according to common prefixes and acyclic graphs according to both common prefixes and common suffixes.

### 2.2 The Temporal Logic HyperLTL

HyperLTL [11] is an extension of linear-time temporal logic (LTL) for hyperproperties. The syntax of HyperLTL formulas is defined inductively by the following grammar:

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

where $a \in \mathsf{AP}$ is an atomic proposition and $\pi$ is a trace variable from an infinite supply of variables $\mathcal{V}$. The Boolean connectives $\neg$ and $\vee$ have the usual meaning, $\mathcal{U}$ is the temporal *until* operator and $\bigcirc$ is the temporal *next* operator. We also consider the usual derived Boolean connectives, such as $\wedge$, $\Rightarrow$, and $\Leftrightarrow$, and the derived temporal operators *eventually* $\Diamond \varphi \equiv \mathsf{true}\, \mathcal{U}\, \varphi$ and *globally* $\Box \varphi \equiv \neg \Diamond \neg \varphi$. The quantified formulas $\exists \pi$ and $\forall \pi$ are read as 'along some trace $\pi$' and 'along all traces $\pi$', respectively.

The semantics of HyperLTL is defined with respect to a trace assignment, a partial mapping $\Pi \colon \mathcal{V} \to \Sigma^\omega$. The assignment with empty domain is denoted by $\Pi_\emptyset$. Given a trace assignment $\Pi$, a trace variable $\pi$, and a concrete trace $t \in \Sigma^\omega$, we denote by $\Pi[\pi \to t]$ the assignment that coincides with $\Pi$ everywhere but at $\pi$, which is mapped to trace $t$. Furthermore, $\Pi[j, \infty]$ denotes the assignment mapping each trace $\pi$ in $\Pi$'s domain to $\Pi(\pi)(j)\Pi(\pi)(j+1)\Pi(\pi)(j+2)\cdots$. The satisfaction of a HyperLTL formula $\varphi$ over a trace assignment $\Pi$ and a set of traces $T \subseteq \Sigma^\omega$, denoted by $T, \Pi \models \varphi$, is defined as follows:

$$\begin{array}{lll}
T, \Pi \models a_\pi & \text{iff} & a \in \Pi(\pi)(0), \\
T, \Pi \models \neg \psi & \text{iff} & T, \Pi \not\models \psi, \\
T, \Pi \models \psi_1 \vee \psi_2 & \text{iff} & T, \Pi \models \psi_1 \text{ or } T, \Pi \models \psi_2, \\
T, \Pi \models \bigcirc \psi & \text{iff} & T, \Pi[1, \infty] \models \psi, \\
T, \Pi \models \psi_1 \mathcal{U} \psi_2 & \text{iff} & \exists i \geq 0 : T, \Pi[i, \infty] \models \psi_2 \ \wedge \ \forall j \in [0, i) : T, \Pi[j, \infty] \models \psi_1, \\
T, \Pi \models \exists \pi .\ \psi & \text{iff} & \exists t \in T : T, \Pi[\pi \to t] \models \psi, \\
T, \Pi \models \forall \pi .\ \psi & \text{iff} & \forall t \in T : T, \Pi[\pi \to t] \models \psi.
\end{array}$$

We say that a set $T$ of traces satisfies a sentence $\varphi$, denoted by $T \models \phi$, if $T, \Pi_\emptyset \models \varphi$. If the set $T$ is generated by a Kripke structure $\mathcal{K}$, we write $\mathcal{K} \models \varphi$.

## 3 Motivating Example

A real-life example that demonstrates the importance of the problem under investigation in this paper is the information leak in the EDAS Conference Management System[3], first reported in [2]. The system manages the review process

[3] http://www.edas.info

```
void Output(){
  bool ntf = GetNotificationStatus();
  bool dec = GetDecision();
  bool ses = getSession();

  string status =
    if (ntf)
    then if (dec)
         then "Accept"
         else "Reject"
    else "Pending";

  string session =
    if (?)
    then "Yes"
    else "No"

  Print(status, session);
}
```

Fig. 2: Program sketch for a conference management system.

for papers submitted to conferences. Throughout this process, authors can check on the status of their papers, but should not learn whether or not the paper has been accepted until official notifications are sent out. The system is correctly programmed to show status “Pending” before notification time and “Accept” or “Reject” afterwards. The leak (which has since then been fixed) occurred through another status display, which indicates whether or not the paper has been scheduled for presentation in a session of the conference. Since only accepted papers get scheduled to sessions, this allowed the authors to infer the status of their paper.

The problem is shown in Table 2. The first two rows show the output in the web interface for the authors regarding two papers submitted to a conference after their notification, where the first paper is accepted while the second is rejected. The last two rows show two other papers where the status is pending. The internal decisions on notification (`ntf`), acceptance (`dec`), and session (`ses`), shown in the table with a gray background, are not part of the observable output and are added for the reader’s convenience. However, by comparing the rows for the two pending papers, the authors can observe that the Session column values are not the same. Thus, they can still deduce that the first paper is rejected and the second paper is accepted.

| | Internal Decisions | | | **Output** | |
|---|---|---|---|---|---|
| **Paper** | `ntf` | `dec` | `ses` | **Status** | **Session** |
| *foo1* | true | true | true | Accept | Yes |
| *bar1* | true | false | false | Reject | No |
| *foo2* | false | false | false | Pending | No |
| *bar2* | false | true | true | Pending | Yes |

Table 2: Output with leak.

| | Internal Decisions | | | **Output** | |
|---|---|---|---|---|---|
| **Paper** | `ntf` | `dec` | `ses` | **Status** | **Session** |
| *foo1* | true | true | true | Accept | Yes |
| *bar1* | true | false | false | Reject | No |
| *foo2* | false | false | false | Pending | No |
| *bar2* | false | true | true | Pending | No |

Table 3: Output without leak.

The information leak in the EDAS system has previously been addressed by adding a monitor that detects such leaks [2, 6]. Here, we instead eliminate the leak constructively. We use *program sketching* [38] to automatically generate the code of our conference manager system. A program *sketch* expresses the high-level structure of an implementation, but leaves "holes" in place of the low-level details. In our approach, the holes in a sketch are interpreted as nondeterministic choices. The repair eliminates nondeterministic choices in such a way that the specification becomes satisfied.

Figure 2 shows a simple sketch for the EDAS example. The hole in the sketch (line 14) is indicated by the question mark in the `if` statement. The replacement for the hole determines how the the value of the `session` output in the the web interface for the authors is computed. We wish to repair the sketch so that whenever two computations both result in `status` = `"Pending"`, the value of `session` is also the same. This requirement is expressed by the following HyperLTL formula:

$$\varphi = \forall\pi.\forall\pi'.\Box\Big(\big((\texttt{status} = \texttt{"Pending"})_\pi \;\wedge\; (\texttt{status} = \texttt{"Pending"})_{\pi'}\big) \Rightarrow\; (\texttt{session}_\pi \Leftrightarrow \texttt{session}_{\pi'})\Big)$$

In this example, an *incorrect* repair would be to replace the hole in line 14 with `ses`, which would result in the output of Table 2. A *correct* repair would be to replace the hole with the Boolean condition $\texttt{ntf} \wedge \texttt{ses}$, which would result in the output of Table 3.

In the rest of the paper, we formally define the repair problem and study its complexity for different fragments of HyperLTL.

## 4 Problem Statement

The *repair problem* is the following decision problem. Let $\mathcal{K} = \langle S, s_{init}, \delta, L\rangle$ be a Kripke structure and $\varphi$ be a closed HyperLTL formula. Does there exist a Kripke structure $\mathcal{K}' = \langle S', s'_{init}, \delta', L'\rangle$ such that:

- $S' = S$,
- $s'_{init} = s_{init}$,
- $\delta' \subseteq \delta$,
- $L' = L$, and
- $\mathcal{K}' \models \varphi$?

In other words, the goal of the repair problem is to identify a Kripke structure $\mathcal{K}'$, whose set of traces is a subset of the traces of $\mathcal{K}$ that satisfies $\varphi$. Note that since the witness to the decision problem is a Kripke structure, following Definition 1, it is implicitly implied that in $\mathcal{K}'$, for every state $s \in S'$, there exists a state $s'$ such that $(s, s') \in \delta'$. In other words, the repair does not create a *deadlock* state.

We use the following notation to distinguish the different variations of the problem:

$$\mathsf{PR[Fragment, Frame\ Type]},$$

where

– PR is the *program repair* decision problem as described above;
– Fragment is one of the following for $\varphi$:
  - We use regular expressions to denote the order and pattern of repetition of quantifiers. For example, $\mathsf{E^*A^*}$-HyperLTL denotes the fragment, where an arbitrary (possibly zero) number of existential quantifiers is followed by an arbitrary (possibly zero) number of universal quantifiers. Also, $\mathsf{AE^+}$-HyperLTL means a lead universal quantifier followed by one or more existential quantifiers. $\mathsf{E^{\leq 1}A^*}$-HyperLTL denotes the fragment, where zero or one existential quantifier is followed by an arbitrary number of universal quantifiers.
  - (EA)$k$-HyperLTL, for $k \geq 0$, denotes the fragment with $k$ alternations and a lead existential quantifier, where $k = 0$ means an alternation-free formula with only existential quantifiers;
  - (AE)$k$-HyperLTL, for $k \geq 0$, denotes the fragment with $k$ alternations and a lead universal quantifier, where $k = 0$ means an alternation-free formula with only universal quantifiers,
  - HyperLTL is the full logic HyperLTL, and
– Frame Type is either tree, acyclic, or general.

## 5 Complexity of Repair for Tree-shaped Graphs

In this section, we analyze the complexity of the program repair problem for trees. This section is organized based on the rows in Table 1. We consider the following three HyperLTL fragments: (1) $\mathsf{E^*A^*}$, (2) $\mathsf{AE^*}$, and (3) the full logic.

### 5.1 The $\mathsf{E^*A^*}$ Fragment

Our first result is that the repair problem for tree-shaped Kripke structures can be solved in logarithmic time in the size of the Kripke structure for the fragment with only one quantifier alternation where the leading quantifier is existential. This fragment is the least expensive to deal with in tree-shaped Kripke structures and, interestingly, the complexity is the same as for the model checking problem [6].

**Theorem 1.** PR[$\mathsf{E^*A^*}$-HyperLTL, tree] *is L-complete in the size of the Kripke structure.*

*Proof.* We note that the number of traces in a tree is bounded by the number of states, i.e., the size of the Kripke structure. The repair algorithm enumerates all possible assignments for the existential trace quantifiers, using, for each existential trace variable, a counter up to the number of traces, which requires only a logarithmic number of bits in size of the Kripke structure. For each such assignment to the existential quantifiers, the algorithm steps through the assignments to the universal quantifiers, which again requires only a logarithmic number of bits in size of the Kripke structure. We consider only assignments with traces that have also been assigned to a existential quantifier. For each assignment of the trace variables, we verify the formula, which can be done in logarithmic space [6]. If the verification is affirmative for all assignments to the universal variables, then the repair consisting of the the traces assigned to the existential variables satisfies the formula.

In order to show completeness, we prove that the repair problem for the existential fragment is L-hard. The L-hardness for PR[E*-HyperLTL, tree] and PR[A*-HyperLTL, tree] follows from the L-hardness of ORD [16]. ORD is the graph-reachability problem for directed line graphs. Graph reachability from $s$ to $t$ can be checked with with the repair problems for $\exists \pi.\ \Diamond(s_\pi \wedge \Diamond t_\pi)$ or $\forall \pi.\ \Diamond(s_\pi \wedge \Diamond t_\pi)$. □

### 5.2 The AE* Fragment

We now consider formulas with one quantifier alternation where the leading quantifier is universal. The type of leading quantifier has a significant impact on the complexity of the repair problem: the complexity jumps from L-completeness to P-completeness, although the model checking complexity for this fragment remains L-complete [6].

**Theorem 2.** PR[AE*-HyperLTL, tree] *is P-complete in the size of the Kripke structure.*[4]

*Proof sketch.* Membership to P can be shown by the following algorithm. For $\varphi = \forall \pi_1.\exists \pi_2.\ \psi$, we begin by marking all the leaves. Then, in several rounds, we go through all marked leaves $v_1$ and instantiate $\pi_1$ with the trace leading to $v_1$. We then again go through all marked leaves $v_2$ and instantiate $\pi_2$ with the trace leading to $v_2$, and check $\psi$ on the pair of traces. If the check is successful for some instantiation of $\pi_2$, we leave $v_1$ marked, otherwise we remove the mark. When no more marks can be removed, we eliminate all branches of the tree that are not marked. For additional existential quantifiers, the number of rounds will increase linearly.

For the lower bound, we reduce the *Horn satisfiability* problem, which is P-hard, to the repair problem for AE* formulas. We first transform the given Horn formula to one that every clause consists of two negative and one positive literals. We map this Horn formula to a tree-shaped Kripke structure and a constant-size HyperLTL formula. For example, formula $(\neg x_1 \vee \neg x_2 \vee f) \wedge (\neg x_3 \vee \neg f \vee x_4) \wedge (\neg x_2 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_1 \vee \bot)$ is mapped to the Kripke structure in Fig. 3.

The Kripke structure includes one branch for each clause of the given Horn formula, where the length of each branch is in logarithmic order of the number of variables in the Horn formula. We use atomic propositions $neg_1$ and $neg_2$

[4] Detailed proofs appear in the appendix.

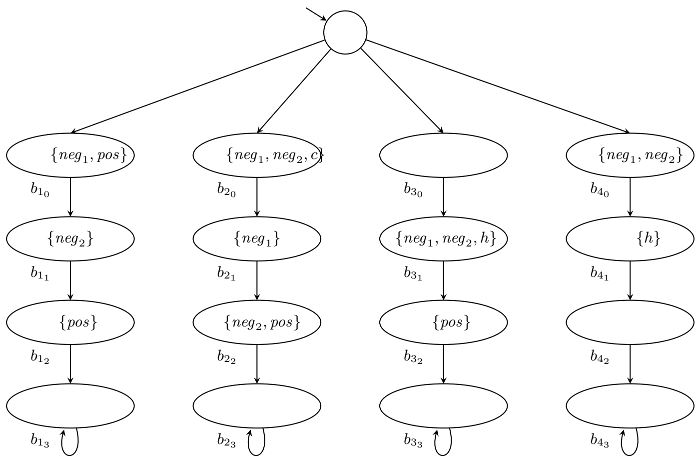


Fig. 3: The Kripke structure of the Horn formula.

to indicate negative literals and *pos* for the positive literal. We also include propositions $c$ and $h$ to mark each clause with a bitsequence. That is, for each clause $\{\neg x_{n_1} \vee \neg x_{n_2} \vee x_p\}$, we label states of its branch by atomic proposition $neg_1$ according to the bitsequence of $x_{n_1}$, atomic proposition $neg_2$ according to the bitsequence of $x_{n_2}$, and atomic proposition *pos* according to the bitsequence of $x_p$. We reserve values 0 and $|X| - 1$ for $\perp$ and $\perp$, respectively, where $X$ is the set of variables of the Horn formula. Finally, we use the atomic proposition $c$ to assign to each clause a number (represented as the bitsequence of valuations of $c$, starting with the lowest-valued bit; the position after the highest-level bit is marked by the occurrence of atomic proposition $h$, which does not appear anywhere else).

The HyperLTL formula enforces that (1) $\top$ is assigned to true, (2) $\perp$ is assigned to false, (3) all clauses are satisfied, and (4) if a positive literal $l$ appears on some clause in the repaired Kripke structure, then all clauses with $l$ must be preserved by the repair. □

### 5.3 The Full Logic

We now turn to full HyperLTL. We first show that the repair problem is in NP.

**Theorem 3.** PR[HyperLTL, tree] *is in NP in the size of the Kripke structure.*

*Proof.* We nondeterministically guess a solution $\mathcal{K}'$ to the repair problem. Since determining whether or not $\mathcal{K}' \models \varphi$ can be solved in logarithmic space [6], the repair problem is in NP. □

For the lower bound, the intuition is that an additional leading universal quantifier allows us to encode full Boolean satisfiability, instead of just Horn satisfiability as in the previous section. Interestingly, the model checking problem remains L-complete for this fragment [6].

**Theorem 4.** PR[AAE-HyperLTL, tree] *is NP-hard in the size of the Kripke structure.*

*Proof sketch.* We map an instance of the 3SAT problem to a Kripke structure and a HyperLTL formula. Figure 4 shows an example, where each clause in 3SAT is mapped to a distinct branch and each literal in the clause is mapped to a distinct sub-branch. We label positive and negative literals by *pos* and *neg*, respectively. Also, propositions $c$ and $h$ are used to mark the clauses with bitsequences in the same fashion as in the construction of proof of Theorem 2. The HyperLTL formula $\varphi_{\mathsf{map}}$ ensures that (1) at least one literal in each clause is true, (2) a literal is not assigned to two values, and (3) all clauses are preserved during repair:

$$\varphi_{\mathsf{map}} = \forall \pi_1.\forall \pi_2.\exists \pi_3.\left[\Box\left(\neg pos_{\pi_1} \ \vee \ \neg neg_{\pi_2}\right)\right] \wedge$$
$$\bigcirc\left[\left(\left(c_{\pi_2} \wedge \neg c_{\pi_3}\right) \,\mathcal{U}\, \left(\neg c_{\pi_2} \wedge c_{\pi_3} \wedge \bigcirc((c_{\pi_2} \leftrightarrow c_{\pi_3}) \,\mathcal{U}\, h_{\pi_2})\right)\right) \vee \left(c_{\pi_2} \wedge \neg c_{\pi_3}\right) \,\mathcal{U}\, h_{\pi_2}\right]$$

The answer to the 3SAT problem is affirmative if and only if a repair exists for the mapped Kripke structure with respect to formula $\varphi_{\mathsf{map}}$. ☐

**Corollary 1.** *The following are NP-complete in the size of the Kripke structure:* PR[$\mathsf{A}^*\mathsf{E}^*$HyperLTL, tree]*,* PR[$(\mathsf{EA})^k$-HyperLTL, tree]*,* PR[$(\mathsf{AE})^k$-HyperLTL, tree]*, and* PR[HyperLTL, tree]*.*

## 6 Complexity of Repair for Acyclic Graphs

We now turn to acyclic graphs. Acyclic Kripke structures are of practical interest, because certain security protocols, in particular authentication algorithms, often consist of sequences of phases with no repetitions or loops. We develop our results first for the alternation-free fragment, then for formulas with quantifier alternation.

### 6.1 The Alternation-free Fragment

We start with the existential fragment. The complexity of the repair problem for this fragment is interestingly the same as the model checking problem.

**Theorem 5.** PR[$\mathsf{E}^*$-HyperLTL, acyclic] *is NL-complete in the size of the Kripke structure.*

*Proof.* For existential formulas, the repair problem is equivalent to the model checking problem. A given Kripke structure satisfies the formula iff it has a repair. If the formula is satisfied, the repair is simply the original Kripke structure. Since the model checking problem for existential formulas over acyclic graphs is NL-complete [6, Theorem 2], the same holds for the repair problem. ☐

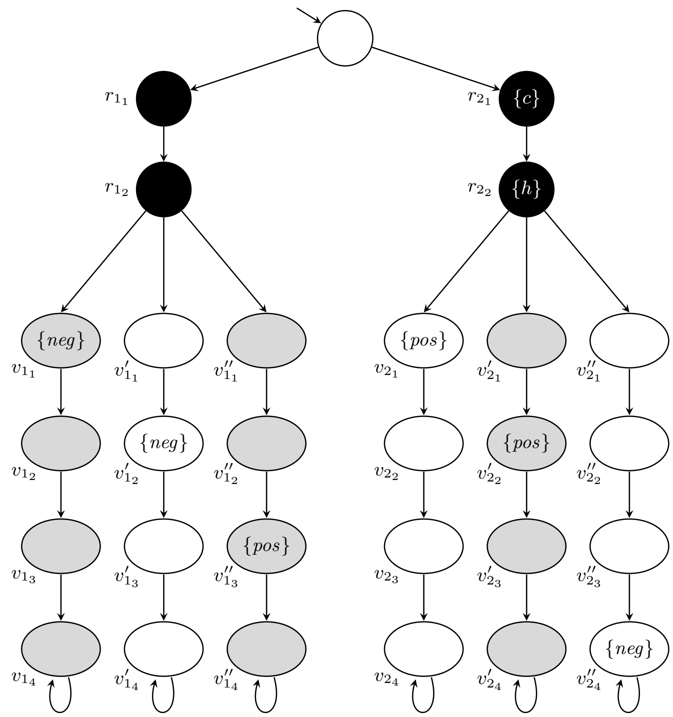


Fig. 4: The Kripke structure for the 3SAT formula $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$. The truth assignment $x_1 = \mathsf{true}$, $x_2 = \mathsf{false}$, $x_3 = \mathsf{false}$, $x_4 = \mathsf{false}$ renders the tree with white branches, i.e., the grey branches are removed during repair.

We now switch to the universal fragment.

**Theorem 6.** PR[A$^*$-HyperLTL, acyclic] *and* PR[EA$^*$-HyperLTL, acyclic] *is* *NL-complete in the size of the Kripke structure.*

*Proof.* To solve the repair problem of a HyperLTL formula $\varphi = \exists\pi.\ \forall\pi_1.\forall\pi_2 \ldots \forall\pi_m.\ \psi(\pi, \pi_1, \pi_2, \ldots, \pi_m)$ with at most one existential quantifier, which appears as the first quantifier, it suffices to find a single trace $\pi$ that satisfies $\psi(\pi, \pi, \ldots, \pi)$: suppose there exists a repair that satisfies $\varphi$ and that has more than one path, then any repair that only preserves one of these paths also satisfies the universal formula $\varphi$. For the upper bound, we nondeterministically guess a path for the trace $\pi$ and remove all other paths. Since the length of the path is bounded by the size of the acyclic Kripke structure, we can guess the path using logarithmically many bits for a counter measuring the length of the path.

NL-hardness of PR[A$^*$-HyperLTL, acyclic] follows from the NL-hardness of the graph-reachability problem for ordered graphs [34]. Ordered graphs are acyclic graphs with a vertex numbering that is a topological sorting of the vertices. We express graph reachability from vertex $s$ to vertex $t$ as the repair problem of the universal formula $\forall\pi.\ \Diamond(s_\pi \wedge \Diamond t_\pi)$. □

### 6.2 Formulas with Quantifier Alternation

Next, we consider formulas where the number of quantifier alternations is bounded by a constant $k$. We show that changing the frame structure from trees to acyclic graphs results in a significant increase in complexity (see Table 1). The complexity of the repair problem is similar to the model checking problem, with the repair problem being one level higher in the polynomial hierarchy (cf. [6]).

**Theorem 7.** *For $k \geq 2$, $\mathsf{PR}[(\mathsf{EA})k\text{-}\mathsf{HyperLTL}, \mathsf{acyclic}]$ is $\Sigma^{\mathsf{p}}_{\mathsf{k}}$-complete in the size of the Kripke structure. For $k \geq 1$, $\mathsf{PR}[(\mathsf{AE})k\text{-}\mathsf{HyperLTL}, \mathsf{acyclic}]$ is $\Sigma^{\mathsf{p}}_{\mathsf{k+1}}$-complete in the size of the Kripke structure.*

*Proof sketch.* For the upper bound, suppose that the first quantifier is existential. Since the Kripke structure is acyclic, the length of the traces is bounded by the number of states. We can thus nondeterministically guess the repair and the existentially quantified traces in polynomial time, and then verify the correctness of the guess by model checking the remaining formula, which has $k-1$ quantifier alternations and begins with a universal quantifier. The verification can be done in $\Pi^{\mathsf{p}}_{\mathsf{k-1}}$ [6, Theorem 3]. Hence, the repair problem is in $\Sigma^{\mathsf{p}}_{\mathsf{k}}$. Analogously, if the first quantifier is universal, the model checking problem in $\Pi^{\mathsf{p}}_{\mathsf{k}}$ and the repair problem in $\Sigma^{\mathsf{p}}_{\mathsf{k+1}}$.

We establish the lower bound via a reduction from the *quantified Boolean formula* (QBF) satisfiability problem [27]. The Kripke structure (see Fig. 5) contains a path for each clause, and a separate structure that consists of a sequence of diamond-shaped graphs, one for each variable. A path through the diamonds selects a truth value for each variable, by going right or left, respectively, at the branching point.

In our reduction, the quantifiers in the QBF instance are translated to trace quantifiers (one per alternation depth), resulting in a HyperLTL formula with $k$ quantifier alternations and a leading existential quantifier. Note that, the outermost existential quantifiers are not translated to a quantifier, but instead resolved by the repair. For this reason, it suffices to build a HyperLTL formula with one less quantifier alternation than the original QBF instance. Also, in our mapping, we must make sure that the clauses and the diamonds for all variables except the outermost existential variables are not removed during the repair. Similar to the proof of Theorem 4, we add a counter to the clauses and add a constraint to the HyperLTL formula that ensures that all counter values are still present in the repair; for the diamonds of the variables, the valuations themselves form such a counter, and we add a constraint that ensures that all valuations for the variables (except for the outermost existential variables) are still present in the repair. □

Finally, Theorem 7 implies that the repair problem for acyclic Kripke structures and HyperLTL formulas with an arbitrary number of quantifiers is in $\mathsf{PSPACE}$.

**Corollary 2.** *$\mathsf{PR}[\mathsf{HyperLTL}, \mathsf{acyclic}]$ is in $\mathsf{PSPACE}$ in the size of the Kripke structure.*

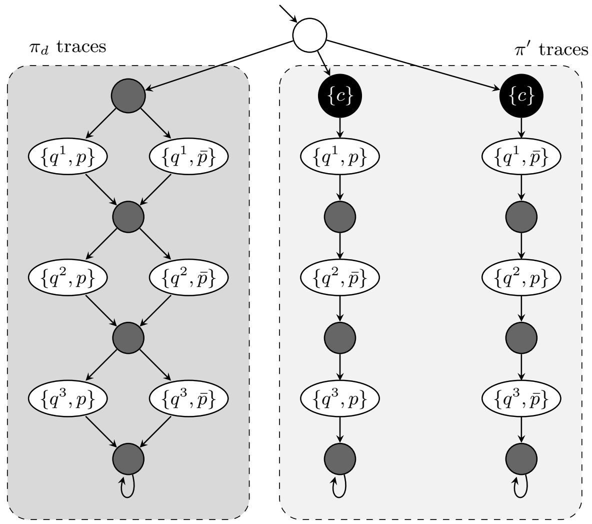


Fig. 5: Kripke structure for the formula $y = \exists x_1.\forall x_2.\exists x_3.(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$.

# 7 Complexity of Repair for General Graphs

In this section, we investigate the complexity of the repair problem for general graphs. We again begin with the alternation-free fragment and then continue with formulas with quantifier alternation.

## 7.1 The Alternation-free Fragment

We start with the existential fragment. Similar to the case of acyclic graphs, the repair problem can be solved with a model checking algorithm.

**Theorem 8.** PR[E$^*$-HyperLTL, general] *is NL-complete in the size of the Kripke structure.*

*Proof.* Analogously to the proof of Theorem 5, we note that, for existential formulas, the repair problem is equivalent to the model checking problem. A given Kripke structure satisfies the formula if and only if it has a repair. If the formula is satisfied, the repair is simply the original Kripke structure. Since the model checking problem for existential formulas for general graphs is NL-complete [25], the same holds for the repair problem. □

Unlike the case of acyclic graphs, the repair problem for the universal fragment is more expensive, although the model checking problem is NL-complete [6].

**Theorem 9.** PR[A$^+$-HyperLTL, general] *is NP-complete in the size of the Kripke structure.*

*Proof.* For membership in NP, we nondeterministically guess a solution to the repair problem, and verify the correctness of the universally quantified HyperLTL formula against the solution in polynomial time in the size of the Kripke structure. NP-hardness follows from the NP-hardness of the repair problem for LTL [5].

### 7.2 Formulas with Quantifier Alternation

Next, we consider formulas where the number of quantifier alternations is bounded by a constant $k$. We show that changing the frame structure from acyclic to general graphs again results in a significant increase in complexity (see Table 1).

**Theorem 10.** PR[$\mathsf{E}^*\mathsf{A}^*$-HyperLTL, general] *is in PSPACE in the size of the Kripke structure.* PR[$\mathsf{A}^*\mathsf{E}^*$-HyperLTL, general] *is PSPACE-complete in the size of the Kripke structure. For $k \geq 2$,* PR[$(\mathsf{EA})^k$-HyperLTL, general] *and* PR[$(\mathsf{AE})^k$-HyperLTL, general] *are $(k-1)$-EXPSPACE-complete in the size of the Kripke structure.*

*Proof idea.* The claimed complexities are those of the model checking problem [37]. We prove that the repair problem has the same complexity as the model checking problem. To show the upper bound of PR[$\mathsf{A}^*\mathsf{E}^*$-HyperLTL, general], we enumerate, in PSPACE, all possible repairs, and then verify against the HyperLTL formula.

For the lower bounds, we modify the Kripke structure and the HyperLTL formula such that the only possible repair is the unchanged Kripke structure. After the modification, the repair problem thus has the same result as the model checking problem. The idea of the modification is to assign numbers to the successors of each state. We add extra states such that the traces that originate from these states correspond to all possible number sequences. Finally, the HyperLTL formula states that for each such number sequence there exists a corresponding trace in the original Kripke structure. ☐

Finally, Theorem 10 implies that the repair problem for general Kripke structures and HyperLTL formulas with an arbitrary number of quantifiers is in NONELEMENTARY.

**Corollary 3.** PR[HyperLTL, general] *is NONELEMENTARY in the size of the Kripke structure.*

## 8 Related Work

There has been a lot of recent progress in automatically *verifying* [14, 23–25] and *monitoring* [2, 8, 9, 21, 22, 29, 39] HyperLTL specifications. HyperLTL is also supported by a growing set of tools, including the model checker MCHyper [14, 25], the satisfiability checkers EAHyper [20] and MGHyper [18], and the runtime monitoring tool RVHyper [21].

Directly related to the repair problem studied in this paper are the satisfiability and synthesis problems. The *satisfiability* problem for HyperLTL was shown to be decidable for the $\exists^*\forall^*$ fragment and for any fragment that includes a $\forall\exists$ quantifier alternation [17]. The hierarchy of hyperlogics beyond HyperLTL has been studied in [13].

The *synthesis* problem was shown to be undecidable in general, and decidable for the $\exists^*$ and $\exists^*\forall$ fragments. While the synthesis problem becomes, in general, undecidable as soon as there are two universal quantifiers, there is a special class of universal specifications, called the linear $\forall^*$-fragment, which is still decidable [19]. The linear $\forall^*$-fragment corresponds to the decidable *distributed synthesis* problems [26]. The *bounded synthesis* problem considers only systems up to a given bound on the number of states. Bounded synthesis from hyperproperties is studied in [14, 19]. Bounded synthesis has been successfully applied to small examples such as the dining cryptographers [10].

The problem of model checking hyperproperties for tree-shaped and acyclic graphs was studied in [6]. Earlier, a similar study of the impact of structural restrictions on the complexity of the model checking problem has also been carried out for LTL [33].

For LTL, the complexity of the repair problem was studied independently in [5,15,32] and subsequently in [7] for distributed programs. The repair problem is also related to *supervisory control*, where, for a given plant, a supervisor is constructed that selects an appropriate subset of the plant's controllable actions to ensure that the resulting behavior is safe [31, 35, 40].

## 9 Conclusion and Future Work

In this paper, we have developed a detailed classification of the complexity of the repair problem for hyperproperties expressed in HyperLTL. We considered general, acyclic, and tree-shaped Kripke structures. We showed that for trees, the complexity of the repair problem in the size of the Kripke structure does not go beyond NP. The problem is complete for L, P, and NP for fragments with only one quantifier alternation, depending upon the outermost quantifiers. For acyclic Kripke structures, the complexity is in PSPACE (in the level of the polynomial hierarchy that corresponds to the number of quantifier alternations). The problem is NL-complete for the alternation-free fragment. For general graphs, the problem is NONELEMENTARY for an arbitrary number of quantifier alternations. For a bounded number $k$ of alternations, the problem is $(k-1)$-EXPSPACE-complete. These results highlight a crucial insight to the repair problem compared to the corresponding model checking problem [6]. With the notable exception of trees, where the complexity of repair is NP-complete, compared to the L-completeness of model checking, the complexities of repair and model checking are largely aligned. This is mainly due to the fact that computing a repair can be done by identifying a candidate substructure, which is comparatively inexpensive, and then verifying its correctness.

The work in this paper opens many new avenues for further research. An immediate question left unanswered in this paper is the lower bound complexity for the $\exists^*\forall^*$ fragment in acyclic and general graphs. It would be interesting to see if the differences we observed for HyperLTL carry over to other hyperlogics (cf. [1, 11, 13, 24]). One could extend the results of this paper to the reactive setting, where the program interacts with the environment. And, finally, the ideas of this paper might help to extend popular synthesis techniques for general (infinite-state) programs, such as program sketching [38] and syntax-guided synthesis [3], to hyperproperties.

## Acknowledgments

We would like to thank Sandeep Kulkrani, Reza Hajisheykhi (Michigan State University), and Shreya Agrawal (Google). The repair problem was originally inspired through our interaction with them. This work is sponsored in part by NSF SaTC Award 1813388. It was also supported by the German Research Foundation (DFG) as part of the Collaborative Research Center "Methods and Tools for Understanding and Controlling Privacy" (CRC 1223) and the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

## References

1. E. Ábrahám and B. Bonakdarpour. HyperPCTL: A temporal logic for probabilistic hyperproperties. In *Proceedings of the 15th International Conference on Quantitative Evaluation of Systems QEST*, pages 20–35, 2018.
2. S. Agrawal and B. Bonakdarpour. Runtime verification of $k$-safety hyperproperties in HyperLTL. In *Proceedings of the IEEE 29th Computer Security Foundations (CSF)*, pages 239–252, 2016.
3. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8, 2013.
4. R. Alur and S. Tripakis. Automatic synthesis of distributed protocols. *SIGACT News*, 48(1):55–90, 2017.
5. B. Bonakdarpour, A. Ebnenasir, and S. S. Kulkarni. Complexity results in revising UNITY programs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1):1–28, January 2009.
6. B. Bonakdarpour and B. Finkbeiner. The complexity of monitoring hyperproperties. In *Proceedings of the IEEE 31th Computer Security Foundations (CSF)*, pages 162–174, 2018.
7. B. Bonakdarpour and S. S. Kulkarni. Masking faults while providing bounded-time phased recovery. In *International Symposium on Formal Methods (FM)*, pages 374–389, 2008.
8. B. Bonakdarpour, C. Sánchez, and G. Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In *Proceedings of the 8th Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 8–27, 2018.

9. N. Brett, U. Siddique, and B. Bonakdarpour. Rewriting-based runtime verification for alternation-free HyperLTL. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 77–93, 2017.
10. D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, 1985.
11. M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *Proceedings of the 3rd Conference on Principles of Security and Trust (POST)*, pages 265–284, 2014.
12. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
13. N. Coenen, B. Finkbeiner, C. Hahn, and J. Hofmann. The hierarchy of hyperlogics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2019.
14. N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup. Verifying hyperliveness. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 121–139, Cham, 2019. Springer International Publishing.
15. A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *On Principles of Distributed Systems (OPODIS)*, pages 275–290, 2005.
16. K Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54(3):400–411, 1997.
17. B. Finkbeiner and C. Hahn. Deciding hyperproperties. In *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR)*, pages 13:1–13:14, 2016.
18. B. Finkbeiner, C. Hahn, and T. Hans. MGHyper: Checking satisfiability of HyperLTL formulas beyond the $\exists^*\forall^*$ fragment. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis*, pages 521–527, Cham, 2018. Springer International Publishing.
19. B. Finkbeiner, C. Hahn, P. Lukert, M. Stenger, and L. Tentrup. Synthesizing reactive systems from hyperproperties. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, pages 289–306, 2018.
20. B. Finkbeiner, C. Hahn, and M. Stenger. EAHyper: Satisfiability, implication, and equivalence checking of hyperproperties. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)*, pages 564–570, 2017.
21. B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. RVHyper: A runtime verification tool for temporal hyperproperties. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 194–200, 2018.
22. B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. Monitoring hyperproperties. *Formal Methods in System Design*, Jun 2019.
23. B. Finkbeiner, C. Hahn, and H. Torfah. Model checking quantitative hyperproperties. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 144–163, Cham, 2018. Springer International Publishing.
24. B. Finkbeiner, Ch. Müller, H. Seidl, and E. Zalinescu. Verifying Security Policies in Multi-agent Workflows with Loops. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 633–645, 2017.
25. B. Finkbeiner, M. N. Rabe, and C. Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pages 30–48, 2015.

26. B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proceedings of LICS*, pages 321–330. IEEE Computer Society, 2005.
27. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.
28. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy (S & P)*, pages 11–20, 1982.
29. C. Hahn, M. Stenger, and L. Tentrup. Constraint-based monitoring of hyperproperties. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 115–131, Cham, 2019. Springer International Publishing.
30. J. Jacob. On the derivation of secure components. In *Proceedings of the IEEE Symposium on Security and Privacy (S & P)*, page 242–247, 1989.
31. S. Jiang and R. Kumar. Supervisory control of discrete event systems with ctl* temporal logic specifications. *SIAM Journal on Control and Optimization*, 44(6):2079–2103, 2006.
32. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238, 2005.
33. L. Kuhtz and B. Finkbeiner. Weak Kripke structures and LTL. In *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR)*, pages 419–433, 2011.
34. T. Lengauer and K.W. Wagner. The correlation between the complexities of the nonhierarchical and hierarchical versions of graph problems. *Journal of Computer and System Sciences*, 44(1):63 – 93, 1992.
35. F. Lin. Analysis and synthesis of discrete event systems using temporal logic. In *Proceedings of the 1991 IEEE International Symposium on Intelligent Control*, pages 140–145, Aug 1991.
36. D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–186, April 1988.
37. M. N. Rabe. *A Temporal Logic Approach to Information-flow Control*. PhD thesis, Saarland University, 2016.
38. A. Solar-Lezama. Program sketching. *Springer Journal on Software Tool for Technology Transfer STTT*, 15(5-6):475–495, 2013.
39. S. Stucki, C. Sánchez, G. Schneider, and B. Bonakdarpour. Graybox monitoring of hyperproperties. In *Proceedings of the 23rd International Symposium on Formal Methods (FM)*, 2019. To appear.
40. J. G. Thistle and W. M. Wonham. Control problems in a temporal logic framework. *International Journal of Control*, 44(4):943–976, 1986.

## Appendix

### Proof of Theorem 2

We show membership to P with a simple marking algorithm. Let $\varphi = \forall \pi_1 . \exists \pi_2 .\ \psi$. We begin by marking all leaves. We then proceed in several rounds, such that in each round, at least one mark is removed. We, hence, terminate within linearly many rounds in the size of the tree-shaped Kripke structure.

In each round, we go through all marked leaves $v_1$ and instantiate $\pi_1$ with the trace leading to $v_1$. We then again go through all marked leaves $v_2$ and instantiate $\pi_2$ with the trace leading to $v_2$, and check $\psi$ on the pair of traces. If the check is successful for some instantiations of $\pi_2$, we leave $v_1$ marked, otherwise we remove the mark. If no mark was removed by the end of the round, we terminate. Each round of the marking algorithm takes linear time in the size of the tree, the complete algorithm thus takes quadratic time. Once the marking algorithm has terminated, we remove all branches of the tree that are not marked. As established by the final round of the marking algorithm, the remaining tree satisfies $\varphi$. For additional existential quantifier, the number of rounds will increase linearly.

For the lower bound, we reduce HORN-SAT, which is P-complete, to the repair problem for AE$^*$ formulas. HORN-SAT is the following problem:

> Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of propositional variables. A *Horn clause* is a clause over $X$ with at most one positive literal. Is $y = y_1 \wedge y_2 \wedge \cdots \wedge y_m$, where each $y_j$ is a Horn clause for $j \in [1, m]$, satisfiable? That is, does there exist an assignment of truth values to the variables in $X$, such that all clauses of $y$ evaluate to true?

In the following, we work with a modified version of HORN-SAT, where every clause consists of two negative and one positive literals. In order to transform any arbitrary Horn formula $y$ to another one $y'$ that consists of two negative and one positive literals, we apply the following:

- To ensure that every clause contains a positive literal, we introduce a fresh variable $\bot$ with the intended meaning "false". We add $\bot$ as a positive literal to all clauses that have no positive literal.
- To ensure that every clause contains at least two negative literals, we introduce a fresh variable $\top$ with the intended meaning "true". We add $\top$ as a negative literal to all clauses that have no negative literals. (Clauses with only one negative literal count as clauses with two negative literals with two identical negative literals.)
- To ensure that no clause contains more than two negative literals, we reduce the number of negative literals as follows: Let $l_1$ and $l_2$ be two negative literals in a clause with more than two negative literals. We introduce a fresh variable $f$ and replace $l_1$ and $l_2$ with $\neg f$; we furthermore add $\{l_1, l_2, f\}$ as a new clause.

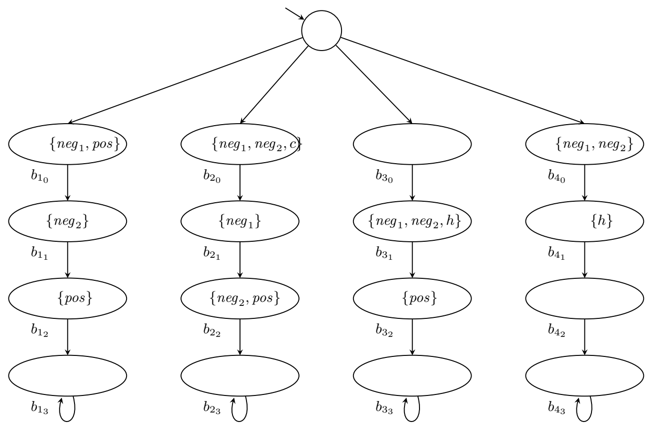


Fig. 6: The Kripke structure for Horn formula $y = (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_1)$.

– In order to account for the intended meaning of $\top$ and $\bot$, we modify the HORN-SAT problem to check if there exists a truth assignment to the variables in $X \cup \{\top, \bot\}$ (union fresh variables $f$ to break clauses as described above), such that all clauses in $y$ evaluate to true *and* $\bot$ evaluates to false and $\top$ evaluates to true.

For example, we transform Horn formula:

$$y = (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_1)$$

to the following:

$$\begin{aligned} y' = &(\neg x_1 \vee \neg x_2 \vee f) \wedge \\ &(\neg x_3 \vee \neg f \vee x_4) \wedge (\neg x_2 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_1 \vee \bot) \end{aligned}$$

Hence, the set of propositional variables for the transformed formula $y'$ is updated to $X = \{x_1, x_2, x_3, x_4, f, \top, \bot\}$. Since the modified problem and the original problem are obviously equivalent, it follows that the modified problem is $\mathsf{P}$-complete as well. We now describe our mapping (see Fig. 6 for an example).

**Kripke structure** We translate the (modified) HORN-SAT problem to a tree-shaped Kripke structure as follows.

– *(Atomic propositions* $\mathsf{AP}$*)* We include atomic propositions $neg_1$ and $neg_2$ to indicate negative literals and $pos$ for the positive literals. We also include $c$ and $h$ to mark the clause bitsequences. Thus,

$$\mathsf{AP} = \{neg_1, neg_2, pos, c, h\}.$$

– *(Set of states S)* We now identify the members of $S$:

- First, we include an initial state $s_{init}$, which is labeled with the empty set of atomic propositions.
- We represent propositional variables $x_i \in X$ (i.e., the updated set including $\top$, $\bot$, and fresh $f$ variables) as a set of states

$$\Big\{b_{j_i} \mid j \in [1,m] \land i \in [0, \log(|X|)]\Big\}$$

  That is, for each clause $y_j$, we include a bitstring that represents which literals participate in $y_j$. More specifically, let $y_j = \{\neg x_{n_1} \lor \neg x_{n_2} \lor x_p\}$ be a Horn clause. We label states $b_{j_0}, b_{j_1}, \ldots, b_{j_{\log(X)-1}}$ by atomic proposition $neg_1$ according to the bitsequence of $x_{n_1}$, atomic proposition $neg_2$ according to the bitsequence of $x_{n_2}$, and atomic proposition $pos$ according to the bitsequence of $x_p$. We reserve values 0 and $X-1$ for $\bot$ and $\bot$, respectively.

– *(Transition relation $\delta$)* We also represent clauses as paths. That is, we include the following transitions:
  - We connect the states that represent bitstrings:

$$\Big\{(b_{j_i}, b_{j_{i+1}}) \mid j \in [1,m] \;\land i \in [0, \log(X)]\Big\}$$

  - We also connect the initial state to the beginning of bitsequence of each clause:
$$\Big\{(s_{init}, b_{j_0}) \mid j \in [1,m]\Big\}$$
  - Finally, we add a self-loop to the end of each branch:

$$\Big\{\;(b_{j_{\log(X)}}, b_{j_{\log(X)}}) \mid j \in [1,m]\Big\}$$

– Finally, we group clauses according to the positive literal. We use the atomic proposition $c$ to assign to each clause a number (represented as the bitsequence of valuations of $c$, starting with the lowest-valued bit; the position after the highest-level bit is marked by the occurrence of atomic proposition $h$, which does not appear anywhere else), such that (1) two different clauses with the same positive literal have different numbers, and (2) all numbers occur for each positive literal. To ensure (2), we add additional copies of the clauses on extra paths, where the same clause appears with a different number. The purpose of this numbering is that we can check if all clauses for a certain positive literal are present in the Kripke structure.

It is easy to see that the Kripke structure that represents the Horn clauses is a tree. It branches into the paths that represent the clauses (see Fig. 6).

**HyperLTL formula** We interpret the repaired Kripke structure as a solution to HORN-SAT assigning false to every variable that appears as a positive literal on some path, and assigning true to all other variables. We define a HyperLTL formula that ensures that this valuation satisfies the clause set. Let $\varphi = \varphi_F \land \varphi_T \land \varphi_P \land \varphi_C$ be a HyperLTL formula with the following conjuncts:

– Formula $\varphi_\top$ enforces that $\top$ is assigned to true. This is expressed by requiring that, on all traces, $\top$ does not appear as a positive literal. That is,

$$\varphi_\top = \forall \pi_1. \Diamond(pos_{\pi_1}).$$

– Formula $\varphi_\bot$ stipulate that $\bot$ is assigned to false. This is expressed by requiring that there exists a trace where $\bot$ appears as a positive literal. That is,

$$\varphi_\bot = \exists \pi_2. \Box(\neg pos_{\pi_2}).$$

– Formula $\varphi_C$ ensures that all clauses are satisfied. This is expressed as a forall-exists formula that requires, for every trace in the repaired Kripke structure, that for one of the variables that appear as negative literals in the clause, there must exist a trace where the same variable appears as the positive literal. That is,

$$\varphi_C = \forall \pi_1. \exists \pi_3. \Box\Big((neg_{1_{\pi_1}} \Leftrightarrow pos_{\pi_3}) \vee (neg_{2_{\pi_1}} \Leftrightarrow pos_{\pi_3})\Big).$$

– Finally, formula $\varphi_P$ requires that if a positive literal $l$ appears on some clause in the repaired Kripke structure, then all clauses with $l$ must be preserved by the repair. This is expressed by a forall-exists formula that states that for all traces where some positive literal $l$ appears with some number $n_c$ (encoded as the bitsequence of atomic proposition $c$), there must exist a trace with positive literal $l$ and number $n_c + 1 \mod n$. Addition plus 1 between the number on trace $\pi$ and the number on trace $\pi'$ is expressed as the formula:

$$\varphi_P = \forall \pi_1. \exists \pi_4. \Big((c_{\pi_1} \wedge \neg c_{\pi_4})\ \mathcal{U}\ (\neg c_{\pi_1} \wedge c_{\pi_4} \wedge \bigcirc((c_{\pi_1} \leftrightarrow c_{\pi_4})\ \mathcal{U}\ h_{\pi_4})\Big) \vee \Big((c_{\pi_1} \wedge \neg c_{\pi_4})\ \mathcal{U}\ h_{\pi_4}\Big).$$

The second disjunct allows for the reset to 0 in the modulo addition.

Overall, $\varphi$ needs only one universal quantifier and three exists quantifiers (one each for $\varphi_\bot$, $\varphi_P$, and $\varphi_C$). $\square$

### Proof of Theorem 4

We reduce the 3-SAT problem to our repair problem. The 3-SAT problem is as follows:

> Let $\{x_1, x_2, \ldots, x_n\}$ be a set of propositional variables. Given is a Boolean formula $y = y_1 \wedge y_2 \wedge \cdots \wedge y_m$, where each $y_j$, for $j \in [1, m]$, is a disjunction of exactly three literals. Is $y$ satisfiable? That is, does there exist an assignment of truth values to $x_1, x_2, \ldots, x_n$ such that $y$ evaluates to true.

We now present a mapping from an arbitrary instance of 3SAT to the repair problem of an tree-shaped Kripke structure and a HyperLTL formula of the form $\forall\forall\exists.\psi$. Then, we show that the Kripke structure satisfies the HyperLTL formula if and only if the answer to the 3SAT problem is affirmative. Figures 7 shows an example.

**Kripke structure $\mathcal{K} = \langle S, s_{init}, \delta, L \rangle$:**

– *(Atomic propositions* $\mathsf{AP}$*)* We include two atomic propositions: $c$ is used to mark the clauses and $p$ is used to force clauses to become true if a Boolean variable appears in a clause. Thus,

$$\mathsf{AP} = \{c, pos, neg, h\}.$$

– *(Set of states $S$)* We now identify the members of $S$:

- First, we include an initial state $s_{init}$. Then, for each clause $y_j$, where $j \in [1, m]$, we include the set of states

$$r_j = \Big\{ r_{j_i} \mid i \in [1, \log(m) + 1] \Big\}.$$

  We label each state $r_{j_i}$ with proposition $c$ based on the bitsequence of $j-1$. Intuitively, these states and their labeling by $c$ ensure that clauses of the input 3SAT formula are not removed during repair. We also label state $r_{m_{\log(m)+1}}$ by $h$. The idea here is the same as the proof of Theorem to implement a modulo addition counter.
- Let $y_j = (l, l', l'')$ be a clause in the 3SAT formula. We include the following set of states:

$$\Big\{ v_{j_i}, v'_{j_i}, v''_{j_i} \mid i \in [1, n] \Big\}.$$

  If $l = x_i$ in $y_j$, then we label state $v_{j_i}$ with proposition $pos$. If $l = \neg x_i$ in $y_j$, then we label state $v_{j_i}$ with proposition $neg$. We analogously label $v'$ and $v''$ associated to $l'$ and $l''$, respectively.

Thus, we have

$$S = \Big\{ r_{j_i} \mid j \in [1, m] \ \wedge \ i \in [1, \log(m) + 1] \Big\} \cup \Big\{ v_{j_i}, v'_{j_i}, v''_{j_i} \mid i \in [1, n] \ \wedge \ j \in [1, m] \Big\}.$$

– *(Transition relation $\delta$)* We now identify the members of $\delta$:

- We include a transition $(s_{init}, r_{1_j})$, for each $j \in [1, m]$.
- We connect all the states corresponding to each clause in a sequence. That is, we include the following transitions for each $j \in [1, m]$:

$$\Big\{ (r_{j_i}, r_{j_{i+1}}) \mid i \in [1, \log(m)) \Big\}.$$

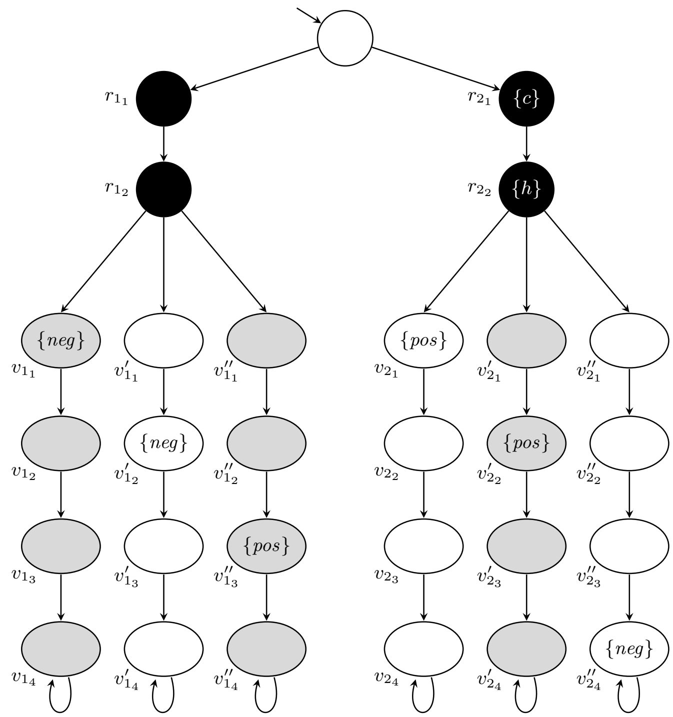


Fig. 7: The Kripke structure for the 3SAT formula $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$. The truth assignment $x_1 = \mathsf{true}$, $x_2 = \mathsf{false}$, $x_3 = \mathsf{false}$, $x_4 = \mathsf{false}$ renders the tree with white branches, i.e., the grey branches are removed during repair.

- We also connect all the states corresponding to the states corresponding to the propositional variables in a sequence. That is, we include the following transitions for each $j \in [1, m]$:

$$\Big\{(r_{j_{\log(m)+1}}, v_{j_1}), (r_{j_{\log(m)+1}}, v'_{j_1}), (r_{j_{\log(m)+1}}, v''_{j_1})\Big\} \cup \Big\{(v_{j_i}, v_{j_{i+1}}), (v'_{j_i}, v'_{j_{i+1}}), (v''_{j_i}, v''_{j_{i+1}}) \mid i \in [1, n)\Big\}$$

- We also add a self-loop at each leaf state :

$$\Big\{(v_{j_n}, v_{j_n}), (v'_{j_n}, v'_{j_n}), (v''_{j_n}, v''_{j_n}) \mid j \in [1, m]\})\Big\}$$

**HyperLTL formula:** The HyperLTL formula in our mapping is the following:

$$\varphi_{\mathsf{map}} = \forall\pi_1.\forall\pi_2.\exists\pi_3.\left[\Box\left(\neg pos_{\pi_1} \ \vee \ \neg neg_{\pi_2}\right)\right] \wedge$$
$$\left[\left(\left(c_{\pi_2} \wedge \neg c_{\pi_3}\right) \ \mathcal{U} \ \left(\neg c_{\pi_2} \wedge c_{\pi_3} \wedge \bigcirc((c_{\pi_2} \leftrightarrow c_{\pi_3})\,\mathcal{U}\,h)\right)\right) \vee \right.$$
$$\left. (c_{\pi_2} \wedge \neg c_{\pi_3}) \ \mathcal{U} \ h \right]$$

We now show that the given 3SAT formula is satisfiable if and only if the Kripke structure obtained by our mapping satisfies the HyperLTL formula $\varphi_{\mathsf{map}}$:

- ($\Rightarrow$) Supposed the 3SAT formula $y$ is satisfiable. This means there exists an assignment to the propositional variables $x_1, x_2, \ldots, x_n$ that makes $y$ true. This implies that each $y_j$ also becomes true, which in turn means that there exists at least one literal in each $y_j$ that evaluates to true. Now, given this assignment, we identify a Kripke structure $\mathcal{K}'$ that satisfies the conditions of our repair problem stated in Section 4. Suppose $y_j = (l \vee l' \vee l'')$, for some $j \in [1, m]$. Also, suppose that $l = x_i$, for some $i \in [1, n]$. If $x_i = \mathsf{true}$ in the answer to the 3SAT problem, then we keep states $v_{j_1}, v_{j_2}, \ldots, v_{j_n}$ and all incoming and outgoing transitions to them. We also remove states $v'_{j_1}, v'_{j_2}, \ldots, v'_{j_n}$ and $v''_{j_1}, v''_{j_2}, \ldots, v''_{j_n}$. Likewise, suppose that $l = \neg x_i$, for some $i \in [1, n]$. If $x_i = \mathsf{false}$ in the answer to the 3SAT problem, then we keep states $v_{j_1}, v_{j_2}, \ldots, v_{j_n}$ and all incoming and outgoing transitions to them. We also remove states $v'_{j_1}, v'_{j_2}, \ldots, v'_{j_n}$ and $v''_{j_1}, v''_{j_2}, \ldots, v''_{j_n}$ and all incoming and outgoing transitions to them. The case for literals $l', l''$ and states $v', v''$ follow trivially.
  It is straightforward to see that the Kripke structure obtained after removing the aforementioned states satisfies $\varphi_{\mathsf{map}}$. Observe that the second conjunct is satisfied since a propositional variable cannot be simultaneously true and false in the answer to the 3SAT problem. Thus, if we keep the states corresponding to a true variable $x_i$, the branches of the where some $v_{j_i}$ is labeled by $neg$ is removed. Same argument holds for $v'$ and $v''$ states and the case where a variable evaluates to false. The first conjunct in $\varphi_{\mathsf{map}}$ also evaluates to true, since we do not remove an $r$ while refining $\mathcal{K}$ to obtain $\mathcal{K}'$.
- ($\Leftarrow$) Suppose the answer to the repair problem is affirmative. That is, there is a repair of our mapped Kripke structure, namely $\mathcal{K}'$ that satisfies the HyperLTL formula $\varphi_{\mathsf{map}}$. This means that (1) all the $r$ states are preserved (otherwise, the second conjunct would have been violated), and (2) there are no pairs of $v$, $v'$, or $v''$ states at the same height of the tree of the repaired Kripke structure, such that both $pos$ and $neg$ are true. We now describe how one can obtain a truth assignment to the propositional variables that satisfies the input 3SAT formula. Suppose that state $v_{j_1}, v_{j_2}, \ldots, v_{j_n}$ appear in $\mathcal{K}'$ for some $i$ and $j$, such that some state $v_{j_i}$ is labeled by $pos$. We assign truth value $\mathsf{true}$ to variable $_i$. This assignment makes clause $y_j$ true, since

$x_i$ is a literal in $y_j$. On the contrary, if state $v_{j_i}$ is labeled by $neg$, then we assign truth value $\mathsf{false}$ to variable $_i$. This assignment makes clause $y_j$ true, since $\neg x_i$ is a literal in $y_j$. Same argument holds for $v'$ and $v''$ states. Furthermore, since all $r$ states are preserved all clauses evaluate to true and, hence, $y$ evaluates to true.

And this concludes the proof. □

### 9.1 Proof of Theorem 7

We show membership in $\Sigma_k^p$ and $\Sigma_{k+1}^p$, respectively, as follows. Suppose that the first quantifier is existential. Since the Kripke structure is acyclic, the length of the traces is bounded by the number of states. We can thus nondeterministically guess the repair and the existentially quantified traces in polynomial time, and then verify the correctness of the guess by model checking the remaining formula, which has $k-1$ quantifier alternations and begins with a universal quantifier. The verification can be done in $\Pi_{k-1}^p$ [6, Theorem 3]. Hence, the repair problem is in $\Sigma_k^p$.

If the first quantifier is universal, we apply the same procedure except that we only guess the repair (there are no leading existential quantifiers). In this case, the formula for the model checking problem has $k$ quantifier alternations. Hence, we solve the model checking problem in $\Pi_k^p$ and the repair problem in $\Sigma_{k+1}^p$.

We give a matching lower bound for PR[(AE)$k$-HyperLTL, acyclic]. Since the (AE)k-HyperLTL formulas are contained in the (EA)k+1-HyperLTL formulas (not using the outermost existential quantifiers), this also provides a matching lower bound for PR[(EA)$k$-HyperLTL, acyclic].

We establish the lower bound for for PR[(AE)$k$-HyperLTL, acyclic] via a reduction from the *quantified Boolean formula* (QBF) satisfiability problem [27]:

> *Given is a set of Boolean variables, $\{x_1, x_2, \ldots, x_n\}$, and a quantified Boolean formula*
>
> $$y = \mathbb{Q}_1 x_1.\mathbb{Q}_1 x_2 \ldots \mathbb{Q}_{n-1} x_{n-1}.\mathbb{Q}_n x_n.(y_1 \wedge y_2 \wedge \cdots \wedge y_m)$$
>
> *where each $\mathbb{Q}_i \in \{\forall, \exists\}$ ($i \in [1, n]$) and each clause $y_j$ ($j \in [1, m]$) is a disjunction of three literals (3CNF). Is $y$ true?*

If $\mathbb{Q}_1 = \exists$ and $y$ is restricted to at most $k$ alternations of quantifiers, then QBF satisfiability is complete for $\Sigma_k^p$. We note that in the given instance of the QBF problem:

- The clauses may have more than three literals, but three is sufficient of our purpose;
- The inner Boolean formula has to be in conjunctive normal form in order for our reduction to work;

– Without loss of generality, the variables in the literals of the same clause are different (this can be achieved by a simple pre-processing of the formula), and
– If the formula has $k$ alternations, then it has $k+1$ alternation *depths*. For example, formula

$$\forall x_1.\exists x_2.(x_1 \vee \neg x_2)$$

has one alternation, but two alternation depths: one for $\forall x_1$ and the second for $\exists x_2$. By $d(x_i)$, we mean the alternation depth of Boolean variable $x_i$.

We now present, for $k \geq 1$, a mapping from an arbitrary instance of QBF with $k$ alternations and where $\mathbb{Q}_1 = \exists$ to the repair problem of an acyclic Kripke structure and a HyperLTL formula with $k-1$ quantifier alternations and a leading universal quantifier. Then, we show that the Kripke structure has a repair that satisfies the HyperLTL formula if and only if the answer to the QBF problem is affirmative.

The reduction is similar to the reduction from QBF satisfiability to the HyperLTL model checking problem [6, Theorem 3] except for the treatment of the outermost existential quantifiers. In the reduction to the model checking problem, these quantifiers are translated to trace quantifiers, resulting in an HyperLTL formula with $k$ quantifier alternations and a leading existential quantifier. In the reduction to the repair problem, the outermost existential quantifiers are resolved by the repair. For this reason, it suffices to build a HyperLTL formula with one less quantifier alternation, i.e., with $k-1$ quantifier alternations, and a leading universal quantifier.

In the following, we first describe the Kripke structure from the reduction to the model checking problem [6, Theorem 3] and then describe the necessary additions for the reduction to the repair problem. Figure 5 shows an example.
**Kripke structure $\mathcal{K} = \langle S, s_{init}, \delta, L\rangle$:**

– *(Atomic propositions* $\mathsf{AP}$*)* For each alternation depth $d \in [1, k+1]$, we include an atomic proposition $q^d$. We furthermore include three atomic propositions: $c$ is used to mark the clauses, $p$ is used to force clauses to become true if a Boolean variable appears in a clause, and proposition $\bar{p}$ is used to force clauses to become true if the negation of a Boolean variable appears in a clause in our reduction.
– *(Set of states $S$)* We now identify the members of $S$:
  - First, we include an initial state $s_{init}$ and a state $r_0$. Then, for each clause $y_j$, where $j \in [1, m]$, we include a state $r_j$, labeled by proposition $c$.
  - For each clause $y_j$, where $j \in [1, m]$, we introduce the following $2n$ states:

$$\left\{v_i^j, u_i^j \mid i \in [1, n]\right\}.$$

  Each state $v_i^j$ is labeled with propositions $q^{d(x_i)}$, and with $p$ if $x_i$ is a literal in $y_j$, or with $\bar{p}$ if $\neg x_i$ is a literal in $y_j$.

- For each Boolean variable $x_i$, where $i \in [1, n]$, we include three states $s_i$, $\bar{s}_i$, and $\hat{s}_i$. Each state $s_i$ (respectively, $\bar{s}_i$) is labeled by $p$ and $q^{d(x_i)}$ (respectively, $\bar{p}$ and $q^{d(x_i)}$).

Thus,

$$\begin{aligned} S =& \big\{s_{init}\big\} \cup \big\{r_j \mid j \in [0, m]\big\} \cup \\ & \big\{v_i^j, u_i^j, s_i, \bar{s}_i, \hat{s}_i \mid i \in [1, n] \wedge j \in [1, m]\big\}. \end{aligned}$$

– *(Transition relation $\delta$)* We now identify the members of $\delta$:
  - We include a transition $(s_{init}, r_j)$, for each $j \in [0, m]$.
  - We add transitions $(r_j, v_1^j)$ for each $j \in [1, m]$.
  - For each $i \in [1, n]$ and $j \in [1, m]$, we include transitions $(v_i^j, u_i^j)$. For each $i \in [1, n)$ and $j \in [1, m]$, we include transitions $(u_i^j, v_{i+1}^j)$.
  - For each $i \in [1, n]$, we include transitions $(s_i, \hat{s}_i)$ and $(\bar{s}_i, \hat{s}_i)$. For each $i \in [1, n)$, we include transitions $(\hat{s}_i, s_{i+1})$ and $(\hat{s}_i, \bar{s}_{i+1})$.
  - We include two transitions $(r_0, s_1)$ and $(r_0, \bar{s}_1)$.
  - Finally, we include self-loops $(\hat{s}_n, \hat{s}_n)$ and $(u_n^j, u_n^j)$, for each $j \in [1, m]$.

Thus,

$$\begin{aligned} \delta =& \big\{(s_{init}, r_j), (r_j, v_1^j), (u_n^j, u_n^j) \mid j \in [0, m]\big\} \cup \\ & \big\{(r_0, s_1), (r_0, \bar{s}_1)\big\} \cup \\ & \big\{(v_i^j, u_i^j) \mid i \in [1, n] \ \wedge \ j \in [1, m]\big\} \cup \\ & \big\{(u_i^j, v_{i+1}^j) \mid i \in [1, n) \ \wedge \ j \in [1, m]\big\} \cup \\ & \big\{(s_i, \hat{s}_i), (\bar{s}_i, \hat{s}_i) \mid i \in [1, n]\big\} \cup \\ & \big\{(\hat{s}_i, s_{i+1}), (\hat{s}_i, \bar{s}_{i+1}) \mid i \in [1, n)\big\}. \end{aligned}$$

Since our target is the repair problem, not the model checking problem, we need to make sure that the clauses are not deleted by the repair. We accomplish this by adding a numbering of the clauses. The HyperLTL formula contains a requirement that ensures that all numbers are still present in the repair. We add two atomic propositions $cn, h$, resulting in the following complete set of atomic propositions:

$$\mathsf{AP} = \big\{c, p, \bar{p}, cn, h\big\} \ \cup \ \big\{q^d \mid d \in [1, k+1]\big\}.$$

We label the states $v_i^j$, representing clause $j$, with a bitsequence of $cn$ encoding the number $j$ followed by label $\{h\}$ on the state $u_{\log j}^j$ indicating the last (and highest-valued) bit of the bitsequence. We make sure that the number of clauses is a full power of 2 by adding, if necessary, extra copies of the clauses.

**HyperLTL formula:** The HyperLTL formula in our reduction is a conjunction of two formulas: $\varphi = \varphi_{pres} \wedge \varphi_{map}$.

– $\varphi_{pres}$: The role of $\varphi_{pres}$ is to ensure that the clauses and the diamond structure for all alternation depths except $k + 1$ are preserved by the repair.

- We ensure that clause number 0 is present with the following formula:

$$\exists \pi.\ \bigcirc(c_\pi \wedge (\neg cn_\pi \,\mathcal{U}\, h_\pi))$$

- To ensure that all other clauses are present as well, we use the following adaption of the counter formula from the proof of Theorem 2:

$$\begin{aligned}&\varphi_{pres,clauses} = \forall \pi.\ \exists \pi'.\ (\bigcirc c_\pi) \rightarrow\\ &(\bigcirc((cn_\pi \wedge \neg cn_{\pi'})\,\mathcal{U}\,(\neg cn_\pi \wedge cn_{\pi'}\\ &\qquad\qquad \wedge \bigcirc((cn_\pi \leftrightarrow cn_{\pi'})\,\mathcal{U}\, h)))\\ &\vee(cn_\pi\,\mathcal{U}\, h_\pi)).\end{aligned}$$

- Similarly, we ensure that the path that interprets all variables (except for the outermost existential variables) is still present in the repair:

$$\exists \pi. \bigcirc(\neg c \wedge \Box \bigwedge_{2\leq i\leq k} q^i \rightarrow \neg p)$$

- To ensure that all other valuations are present as well, we use the following adaption of the counter formula from the proof of Theorem 9:

$$\begin{aligned}&\varphi_{pres,clauses} = \forall \pi.\ \exists \pi'.\ (\bigcirc c_\pi) \rightarrow\\ &\neg q^2\,\mathcal{U}\ (q^2 \wedge (\textstyle\bigvee_{2\leq i\leq k} q^i \rightarrow (p_\pi \wedge \neg p_{\pi'}))\\ &\quad \mathcal{U}\,((\textstyle\bigvee_{2\leq i\leq k} q^i) \wedge (\neg p_\pi \wedge p_{\pi'}\\ &\qquad \wedge \bigcirc(((\textstyle\bigvee_{2\leq i\leq k} q^i) \rightarrow (p_\pi \leftrightarrow p_{\pi'})\\ &\qquad\quad \mathcal{U}\ \bigcirc\Box \textstyle\bigwedge_{2\leq i\leq k} \neg q^i)))))\\ &\vee \Box(\textstyle\bigvee_{2\leq i\leq k} q^i) \rightarrow p_\pi.\end{aligned}$$

– $\varphi_{map}$: The role of $\varphi_{map}$ is to ensure that the QBF instance is satisfiable iff the HyperLTL formula is satisfied on the repair.

$$\begin{aligned}\varphi_{map} &= \forall \pi_{k+1}.\forall \pi_k.\exists \pi_{k-1}\cdots\exists \pi_2.\forall \pi_1.\forall \pi'.\\ &\left(\textstyle\bigwedge_{d\in\{1,3,\ldots,k\}} \bigcirc\neg c_{\pi_d} \ \wedge\ \bigcirc c_{\pi'}\right) \Rightarrow\\ &\Bigg(\textstyle\bigwedge_{d\in\{2,4,\ldots,k+1\}} \bigcirc\neg c_{\pi_d} \wedge\\ &\quad \Diamond\Bigg[\textstyle\bigvee_{d\in[1,k+1]}\Big(\big(q^d_{\pi_d} \Leftrightarrow q^d_{\pi'}\big) \wedge\\ &\qquad \big((p_{\pi'} \wedge p_{\pi_d})\ \vee\ (\bar{p}_{\pi'} \wedge \bar{p}_{\pi_d})\big)\Big)\Bigg]\Bigg)\end{aligned}$$

Intuitively, $\varphi_{map}$ expresses the following: for all the clause traces $\pi'$ and all traces that valuate universally quantified variables $(\pi_1, \pi_3, \ldots)$, there exist traces evaluating the existentially quantified variables $(\pi_2, \pi_4, \ldots)$, where either $p$ or $\bar{p}$ eventually matches its counterpart position in the clause trace $\pi'$. The dependencies between the trace quantifiers for the valuation of the variables match the dependencies in the quantified Boolean formula. A special case are the outermost existential variables in the QBF. The corresponding trace quantifier (for $\pi_{k+1}$) is universal, rather than existential. As a result, the formula has only $k-1$ alternations. We allow the repair to reduce the valuations for the outermost existential variables to a single valuation. Hence, universal and existential quantification is the same. $\square$

### Proof of Theorem 10

The claimed complexities are those of the model checking problem [37]. We prove that the repair problem has the same complexity as the model checking problem. For the upper bound, we enumerate, in PSPACE, all possible repairs, and then verify against the HyperLTL formula.

For the lower bound, we modify the Kripke structure and the HyperLTL formula such that the only possible repair is the unchanged Kripke structure. After the modification, the repair problem thus has the same result as the model checking problem. The idea of the modification is to assign numbers to the successors of each state. We add extra states such that the traces that originate from these states correspond to all possible number sequences. Finally, the HyperLTL formula states that for each such number sequence there exists a corresponding trace in the original Kripke structure. A technical difficulty is that the HyperLTL formula needs to be fixed for all Kripke structures, but different Kripke structures may differ in their branching degree. For this reason we first (Step 1) transform the given Kripke structure and HyperLTL formula into an equivalent problem where every state has precisely two successors; afterwards (Step 2) we carry out the modification that ensures that the repair cannot remove any edges.

*Step 1: Fixing the branching degree.* We first translate the given Kripke structure $\mathcal{K} = \langle S, s_{init}, \delta, L\rangle$ into a Kripke structure $\mathcal{K}' = \langle S', s'_{init}, \delta', L'\rangle$ where every state has at most two successors; afterwards we construct another Kripke structure $\mathcal{K}''$ where every state has exactly two successors. Let $d(s) = |\{s' \in S \mid (s, s') \in \delta\}$ be the branching degree of state $s \in S$ and let $d = \max_{s\in S} d(s)$ be the branching degree. We add a marker $m$ to identify the states of the original Kripke structure.

- $\mathsf{AP}' = \mathsf{AP} \cup \{m\}$.
- $S' = S \cup \{(s, i) \mid s \in S, 2 \leq i \leq d(s)\}$
- $s'_{init} = s_{init}$
- $\delta' = \bigcup_{s\in S}\{(s, s'_1), (s, (s, 2)), ((s, 2), s'_2), ((s, 2), s'_2),$
  $((s, 2), (s, 3)), ((s, 3), s'_3), \ldots,$
  $((s, d(s) - 1), (s, s_{d(s)-1})), (s, d(s) - 1), s'_{d(s)})\}$
- $L'(s) = L'(s) \cup \{m\}$ for $s \in S$ and
  $L(s, i) = \emptyset$ for $(s, i) \in S'$

Let $\varphi$ be a given HyperLTL formula over $\mathsf{AP}$. We define a new HyperLTL formula $\varphi'$ over $\mathsf{AP}'$ such that $\mathcal{K}$ satisfies $\varphi$ iff $\mathcal{K}'$ satisfies $\varphi'$. We transform $\varphi$ inductively as follows:

- if $\varphi = \bigcirc \psi$, then $\varphi' = (\neg m) \,\mathcal{U}\, (m \wedge \psi')$,
- if $\varphi = \psi_1 \,\mathcal{U}\, \psi_2$, then $\varphi' = (m \rightarrow \psi') \,\mathcal{U}\, (m \wedge \psi_2)$,
- if $\varphi = \exists \pi.\ \psi$ then $\varphi' = \exists \pi.\ \psi'$
- if $\varphi = \forall \pi.\ \psi$ then $\varphi' = \forall \pi.\ \psi'$
- if $\varphi = \neg \psi$ then $\varphi' = \neg \psi'$
- if $\varphi = \psi_1 \vee \psi_2$ then $\varphi' = \psi'_1 \vee \psi'_2$

where $\psi', \psi'_1, \psi'_2$ are the transformations of $\psi$, $\psi_1$, and $\psi_2$, respectively.

The Kripke structure $\mathcal{K}'$ may still contain states that have only one successor. We construct a Kripke structure $\mathcal{K}'' = \langle S'', s''_{init}, \delta'', L'' \rangle$ that has the same traces over $\mathsf{AP}'$ as $\mathcal{K}'$, and, hence, satisfies the same HyperLTL formulas over $\mathsf{AP}'$, but only has states with exactly two successors. We furthermore distinguish the two successors with a fresh proposition $c$.

- $\mathsf{AP}'' = \mathsf{AP}' \cup \{c\}$.
- $S'' = (S' \times \{0, 1\})$
- $s''_{init} = (s'_{init}, 0)$
- $\delta'' = \bigcup_{(s,i) \in S''} \{((s, i), (s', 0)), ((s, i), (s'', 1)) \mid (s, s'), (s, s'') \in \delta$ and $s' < s''$ if $d(s) = 2$ and $s' = s''$ if $d(s) = 1\}$
- $L''(s, 0) = L'(s)$ and
  $L''(s, 1) = L'(s) \cup \{c\}$ for $s \in S'$

where $<$ is an arbitrary order on the states in $S'$.

*Step 2: Protecting the Kripke structure from repair.* As described above, we construct another Kripke structure $\mathcal{K}'' = \langle S'', s''_{init}\, \delta''\, L''\ \rangle$ by adding fresh states that generate all sequences of successor numbers, i.e., all bitsequences of $c$. We again use a marker proposition $m''$ to mark the states from $S''$.

- $\mathsf{AP}''' = \mathsf{AP}'' \cup \{m''\}$.
- $S''' = S'' \cup \{0, 1\}$
- $s'''_{init} = s''_{init}$
- $\delta''' = \delta'' \cup \{(s'''_{init}, 0), (s'''_{init}, 1), (0, 1), (0, 0), (1, 0), 1, 1)\}$
- $L'''(s) = L''(s) \cup \{m''\}$ for $s \in S''$, $L(0) = \emptyset$, and $L(1) = \{c\}$.

Finally, we modify the HyperLTL formula $\varphi'$ so that $\varphi'$ only refers to traces in $S''$ and that no transitions can be removed by the repair. Let $\varphi' = Q_1 \pi_1 Q_2 \pi_2 \ldots Q_m \pi_m.\ \psi$, where $\psi$ is quantifier-free. Let $E$ be the set of indices $i$ such that $Q_i$ is existential, and $A$ be the set of indices $i$ such that $Q_i$ is universal. We define

$$\varphi''' = Q_1 \pi_1 Q_2 \pi_2 \ldots Q_m \pi_m.\ (\bigwedge_{m \in E} \bigcirc m''_{\pi_i}) \wedge ((\bigwedge_{m \in E} \bigcirc m''_{\pi_i}) \rightarrow \psi)$$

The final HyperLTL formula $\varphi'''' = \varphi''' \wedge \varphi_{01} \wedge \varphi_{\mathcal{K}}$ is the conjunction of $\varphi'''$ and two constraints that protect the Kripke structure from repair.

- $\varphi_{01}$ protects the transitions to states 0 and 1:

$$\begin{aligned} \varphi_{01} = {} & \exists \pi. \exists \pi'. \exists \pi''. \exists \pi'''. \\ & (\bigcirc(\neg m''_{\pi} \wedge \neg c \wedge \bigcirc \neg c)) \wedge \\ & (\bigcirc(\neg m''_{\pi'} \wedge \neg c \wedge \bigcirc c)) \wedge \\ & (\bigcirc(\neg m''_{\pi''} \wedge c \wedge \bigcirc c)) \wedge \\ & (\bigcirc(\neg m''_{\pi'''} \wedge c \wedge \bigcirc \neg c)) \end{aligned}$$

– $\varphi_{\mathcal{K}}$ protects all transitions in $\mathcal{K}''$:

$$\varphi_{\mathcal{K}} = \forall \pi \exists \pi'. \; (\bigcirc \Box \neg m''_{\pi}) \rightarrow \\ \bigcirc \Box (m''_{\pi'} \wedge c_{\pi} \leftrightarrow c_{\pi'})$$

$\square$