The Complexity of Monitoring Hyperproperties

Borzoo Bonakdarpour
Department of Computer Science
Iowa State University, USA
Email: borzoo@iastate.edu

Bernd Finkbeiner
Reactive Systems Group
Saarland University, Germany
Email: finkbeiner@cs.uni-saarland.de

Abstract—We study the runtime verification of hyperproperties, expressed in the temporal logic HyperLTL, as a means to inspect a system with respect to security polices. Runtime monitors for hyperproperties analyze trace logs that are organized by common prefixes in the form of a tree-shaped Kripke structure, or are organized both by common prefixes and by common suffixes in the form of an acyclic Kripke structure. Unlike runtime verification techniques for trace properties, where the monitor tracks the state of the specification but usually does not need to store traces, a monitor for hyperproperties repeatedly model checks the growing Kripke structure. This calls for a rigorous complexity analysis of the model checking problem over tree-shaped and acyclic Kripke structures.

We show that for *trees*, the complexity in the size of the Kripke structure is **L-complete** independently of the number of quantifier alternations in the HyperLTL formula. For *acyclic* Kripke structures, the complexity is **PSPACE-complete** (in the level of the polynomial hierarchy that corresponds to the number of quantifier alternations). The combined complexity in the size of the Kripke structure and the length of the HyperLTL formula is **PSPACE-complete** for both trees and acyclic Kripke structures, and is as low as **NC** for the relevant case of trees and alternation-free HyperLTL formulas. Thus, the size and shape of both the Kripke structure and the formula have significant impact on the complexity of the model checking problem.

I. INTRODUCTION

Most security properties related to confidentiality and information flow cannot be formulated as trace properties because they relate multiple computations. For example, *observational determinism* [1] is satisfied if on every pair of computation traces where the observable inputs are the same, also the observable outputs are the same. This class of secure information flow policies has been characterized in a set-theoretic framework called *hyperproperties* [2]. Hyperproperties can be expressed in the temporal logic HyperLTL [3], which extends the linear-time temporal logic (LTL) [4] with trace quantifiers and trace variables. Suppose, for example, that the observable input to a system is the atomic proposition i and the output is the atomic proposition o. Observational determinism can then be expressed as the HyperLTL formula

$$\varphi_{\mathsf{obs}} = \forall \pi. \, \forall \pi'. \, \Box(i_\pi \Leftrightarrow i_{\pi'}) \Rightarrow \Box(o_\pi \Leftrightarrow o_{\pi'}),$$

where $\Box$ is the usual "globally" operator of temporal logic: if two traces π and π' agree globally on i, then they must also globally agree on o.

Runtime verification is a technique that inspects the health of a system by evaluating execution traces collected at run time. Existing runtime verification techniques (e.g., [5]–[8])

(a) Set of linear traces.

(b) Equivalent tree-shaped Kripke structure.
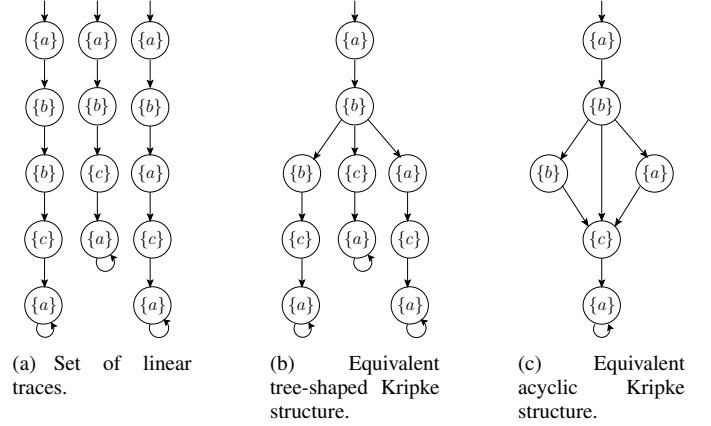
(c) Equivalent acyclic Kripke structure.

Fig. 1: A trace log example and its assembly into space-efficient tree-shaped and acyclic Kripke structures.

evaluate a linear finite trace t against a formula φ expressed in a trace-based language such as LTL or regular expressions. Monitors for trace-based languages typically do not need to record traces that are already evaluated. By contrast, a monitor for hyperproperties must store a set T of traces seen so far and repeatedly check this growing set against the specification (cf. [9]–[11]). For example, to monitor observational determinism φ_{obs}, the monitor has to examine every existing *pair* of traces at all times and, hence, has to keep the pairs that are already evaluated in a trace log. These trace logs may be in the form of a simple linear collection of the traces seen so far or, for space efficiency, 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 (see Fig. 1). Moreover, as a runtime monitor for hyperproperties observes the execution traces while new traces (say T') are produced by the running system over time, the monitor has to evaluate φ with respect to $T \cup T'$ due to inter-trace assertions in φ. Over time, the size of the Kripke structure that represents $T \cup T'$ may grow and its shape may change. Thus, a fundamental research question is to study the complexity of the model checking problem for HyperLTL as the trace log grows over time. For LTL, the complexity of the model checking problem for restricted Kripke structures is known [12]. In particular, the model checking problem is PSPACE-hard (in the size of the formula) only if there exists a strongly connected component with two distinct cycles. For acyclic Kripke structures, the model checking problem is in coNP. If, additionally, the verification problem can be

decomposed into a polynomial number of finite path checking problems, for example, if the Kripke structure is a tree or a directed graph with constant depth, then the complexity reduces further to NC. Prior to our work, the complexity of the model checking problem for hyperproperties over restricted Kripke structures was an open question.

A. Contributions

With this motivation, we study, in this paper, the impact of structural constraints on the complexity of the model checking problem for HyperLTL. As mentioned earlier, we are interested in Kripke structures that are tree-shaped or acyclic as two appropriate shapes to store execution trace logs. With respect to the HyperLTL formula, we are interested in the impact of the quantifier structure. Tables I and II summarize our new complexity results, contrasted with the known results for general Kripke structures [3], [13], [14], related to the equivalent model checking problem. Table I shows the complexity of the model checking problem in terms of the size of the Kripke structure alone. This *system complexity* is often the most relevant complexity in practice, because the system tends to be much larger than the specification. This is in particular true in runtime verification, where the Kripke structure that records the traces seen so far grows over time, while the temporal formula remains the same. Table II shows the *combined complexity* in the full input, consisting of both the Kripke structure and the HyperLTL formula. Our results show that the shape of the Kripke structure plays a crucial role in the complexity of the model checking problem:

- **Trees.** For trees, the complexity in the size of the Kripke structure is L-complete independently of the number of quantifier alternations. The combined complexity in the size of the Kripke structure and the length of the HyperLTL formula is PSPACE-complete (in the level of the polynomial hierarchy that corresponds to the number of quantifier alternations) and is as low as NC for alternation-free fragment as well formulas of the form $\exists\forall$ and $\forall\exists$.

- **Acyclic graphs.** For acyclic Kripke structures, the complexity is NL-complete for the alternation-free fragment and is PSPACE-complete for alternating formulas (in the level of the polynomial hierarchy that corresponds to the number of quantifier alternations). The combined complexity in the size of the Kripke structure and the length of the HyperLTL formula is also PSPACE-complete in the level of the polynomial hierarchy that corresponds to the number of quantifier alternations.

B. Significance of Contributions

The significance of our results is multifold:

- Our results are in sharp contrast to the undecidability result of [15] and the non-elementary complexity of [3], which has commonly been interpreted as suggesting that only the alternation-free fragment is worth considering in practical settings. Our results show that there is a lot that can be done for hyperproperties with alternations without exceeding PSPACE.

- An important observation from Tables I and II is the impact of the shape of the Kripke structure and type of formula on the complexity. For example, the HyperLTL formula for Goguen and Meseguer's non-interference policy [16] is alternation-free for deterministic systems, while the same policy in a non-deterministic setting is of the form $\forall\forall\exists.\psi$, hence, one alternation. This changes the complexity from NL-complete to coNP-complete in acyclic graphs, while it remains L-complete for trees. This shows that there are trade offs, both in the choice of the shape of the trace logs and in the formula that represents the policy, with signficant practical implications. We will present a more detailed motivating example on these trade offs in Section III.

- As discussed in [10], [11], monitoring hyperproperties may depend on the entire set of traces seen so far. This implies that a dependency on the total length and number of the traces is unavoidable. Having said that, our L-completeness result for monitoring trees shows that the dependency in the total length is actually only logarithmic. Also, if the complexity is measured in the length of the traces and the formula, our PSPACE-completeness result shows that monitoring can be accomplished with a linear number of instances of the incremental traces.

- Our results are also of interest in the context of classic model checking. In the restricted Kripke structures, leaves in trees and acyclic graphs are defined to have self-loops, which encode infinite traces. Our results thus have two applications: (1) classic model checking of restricted Kripke structures with infinite traces, and (2) runtime verification of a collected or evolving set of finite traces. Tree-shape and acyclic Kripke structures often occur as the natural representation of the state space of some protocols. 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. The advantage of model checking restricted structures is particularly strong for HyperLTL formulas with many quantifier alternations: while the model checking problem over general Kripke structures cannot be solved by any elementary recursive function [3], [13], [14], the model checking problem for trees and acyclic graphs is in PSPACE. The complexity in the size of a tree-shaped Kripke structure is even just L-complete.

In a nutshell, we believe that the results in this paper provide the fundamental understanding of the runtime verification problem for secure information flow and pave the way for further research on efficient and scalable monitoring techniques.

Organization: The remainder of this paper is organized as follows. In Section II, we review Kripke structures and HyperLTL. We present a detailed motivating example in Section III. Section IV presents our results on the complexity of HyperLTL model checking in the size of the Kripke structure. Section V presents the results on the complexity in the combined input consisting of both the Kripke structure and

	This paper		General
	Tree	**Acyclic**	**General**
$\forall^+/\exists^+$	L-complete *(Theorem 1)*	NL-complete *(Theorem 2)*	NL-complete [13]
$\exists^+\forall^+/\forall^+\exists^+$		NP/coNP-complete *(Theorem 3)*	PSPACE-complete [3]
$(\forall^*\exists^*)^*$		Π_k^p-complete *(Theorem 3)*	$(k-1)$-EXPSPACE-complete [14]
		Σ_k^p-complete *(Theorem 3)*	
		PSPACE-complete *(Corollary 1)*	NONELEMENTARY [3]

TABLE I: Complexity of the HyperLTL model checking problem in the size of the Kripke structure, where k is the number of quantifier alternations in $(\forall^*\exists^*)^*$.

	This paper		General
	Tree	**Acyclic**	**General**
$\exists^k/\forall^k$	NC *(Theorem 4)*	NP/coNP-complete *(Theorem 7)*	PSPACE-complete [13]
$\exists^+/\forall^+$	NP/coNP-complete *(Theorem 6)*		
$\exists\forall/\forall\exists$	NC *(Theorem 5)*	Σ_2^p/Π_2^p-complete *(Theorem 7)*	EXPSPACE-complete [3]
$(\forall^*\exists^*)^*$	Π_{k+1}^p-complete *(Theorems 6,7)*		k-EXPSPACE-complete [3]
	Σ_{k+1}^p-complete *(Theorems 6,7)*		
	PSPACE-complete *(Corollaries 2,3)*		NONELEMENTARY [3]

TABLE II: Complexity of the HyperLTL model checking problem in the combined input, consisting of the Kripke structure and the HyperLTL formula, where k is the number of quantifier alternations in $(\forall^*\exists^*)^*$.

the HyperLTL formula. We discuss related work in Section VI. Finally, we make concluding remarks in Section VII.

II. PRELIMINARIES

We begin with a quick review of Kripke structures and HyperLTL.

A. Kripke Structures

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

$$t = t(0)t(1)t(2)\cdots .$$

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 $(s, s') \in \delta$.

For example, in Fig. 2, we have that $L(s_{init}) = \{a\}, L(s_3) = \{b\}$, 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 terminal states, i.e., on states that have no other outgoing transition. See Fig. 2 for an example. Since Definition 1 does not allow terminal states,
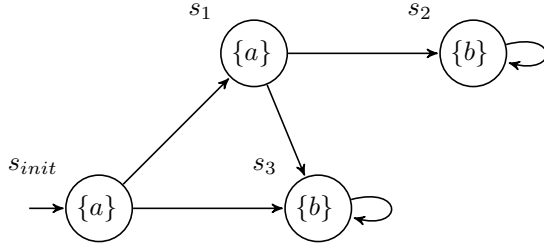
Fig. 2: Example of an acyclic Kripke structure (with self-loops at otherwise terminal states).

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.

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 $i \geq 0$.

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 $i \geq 0$. 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 the context of monitoring, we assume that traces of a system under inspection are given as a tree-shaped or acyclic Kripke structure. These type of Kripke frames are obviously more space efficient than a set of linear traces, because trees allow us to organize the traces according to common prefixes and acyclic graphs according to both common prefixes and common suffixes.

B. HyperLTL

HyperLTL [3] is a temporal logic for expressing hyperproperties. A hyperproperty [2] is a set of sets of execution traces. HyperLTL generalizes LTL by allowing explicit quantification over multiple execution traces simultaneously. The set of HyperLTL formulas is defined inductively by the following grammar:

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

where $a \in \mathsf{AP}$ is an atomic proposition and π 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* $\Diamond \varphi \equiv \texttt{true} \: \mathcal{U} \: \varphi$, *globally* $\Box \varphi \equiv \neg \Diamond \neg \varphi$, and *weak until*: $\varphi \: \mathcal{W} \: \psi \equiv (\varphi \: \mathcal{U} \: \psi) \vee \Box \varphi$. The quantified formulas $\exists \pi$ and $\forall \pi$

are read as 'along some trace π' and 'along all traces π', respectively. A *sentence* is a closed formula, i.e., the formula that has no free trace variables. A formula with only universal or only existential quantifiers is called *alternation-free*. Such formulas have *alternation depth* 0. The alternation depth of formulas with both existential and universal quantifiers is the number of alternations from existential to universal quantifiers and from universal to existential quantifiers.

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 Π, a trace variable π, and a trace t, we denote by $\Pi[\pi \to t]$ the assignment that coincides with Π everywhere but at π, which is mapped to t. Furthermore, $\Pi[j, \infty]$ denotes the assignment mapping each trace π in Π's domain to

$$\Pi(\pi)(j)\Pi(\pi)(j+1)\Pi(\pi)(j+2)\cdots.$$

The satisfaction of a HyperLTL formula φ over a trace assignment Π and a set T of traces, denoted by $T, \Pi \models \varphi$, is defined as follows:

$$T, \Pi \models a_\pi \quad \text{iff} \quad a \in \Pi(\pi)(0),$$

$$T, \Pi \models \neg \phi \quad \text{iff} \quad T, \Pi \not\models \phi,$$

$$T, \Pi \models \phi_1 \vee \phi_2 \quad \text{iff} \quad T, \Pi \models \phi_1 \text{ or } T, \Pi \models \phi_2,$$

$$T, \Pi \models \bigcirc \phi \quad \text{iff} \quad T, \Pi[1, \infty] \models \phi,$$

$$T, \Pi \models \phi_1 \: \mathcal{U} \: \phi_2 \quad \text{iff} \quad \exists i \geq 0 : T, \Pi[i, \infty] \models \phi_2 \: \wedge$$
$$\forall j \in [0, i) : T, \Pi[j, \infty] \models \phi_1,$$

$$T, \Pi \models \exists \pi. \: \varphi \quad \text{iff} \quad \exists t \in T : T, \Pi[\pi \to t] \models \varphi,$$

$$T, \Pi \models \forall \pi. \: \varphi \quad \text{iff} \quad \forall t \in T : T, \Pi[\pi \to t] \models \varphi.$$

We say that a set T of traces satisfies a sentence φ, denoted by $T \models \varphi$, if $T, \Pi_\emptyset \models \varphi$. A Kripke structure $\mathcal{K} = \langle S, s_{init}, \delta, L \rangle$ satisfies a HyperLTL formula φ, denoted by $\mathcal{K} \models \varphi$, iff $\mathsf{Traces}(\mathcal{K}, s_{init}) \models \varphi$.

Example. Consider the HyperLTL formula

$$\varphi = \forall \pi_1. \forall \pi_2. \: a_{\pi_1} \: \mathcal{U} \: b_{\pi_2}$$

and the Kripke structure in Fig. 2. The Kripke structure does not satisfy φ. For example, the trace assignment Π that assigns to π_1 the trace $\{a\}\{b\}^\omega$ and to π_2 the trace $\{a\}\{a\}\{b\}^\omega$ does not satisfy $a_{\pi_1} \: \mathcal{U} \: b_{\pi_2}$.

Standard *linear-time temporal logic* (LTL) is the fragment of HyperLTL with a single quantifier. Typically, the quantifier is universal and is left implicit, i.e., the LTL formula $\varphi = \forall \pi. \psi$ is written as ψ with the index π omitted from all atomic propositions. We say that a trace t satisfies an LTL formula φ, denoted by $t \models \varphi$, if $\{t\} \models \varphi$.

We note that although our focus in this paper is on runtime verification (hence, a finite number of finite traces), for simplicity and without loss of generality, we use the infinite semantics of HyperLTL. To this end, we assume that the leaves of Kripke frames have self-loops that corresponds to the "stuttering" semantics of finite-trace temporal logics.

III. MOTIVATING EXAMPLE

A. EDAS Conference Manager Bug

We demonstrate the importance of the problem under investigation in this paper with a real-life information leak encountered by the first author while using the EDAS Conference Management System[1]. Fig. 3 shows an anonymized screenshot of the EDAS web interface [9]. The color-coded table displays the status of submitted papers by the user: accepted (green), rejected (orange), withdrawn (grey), and pending (yellow). Now, consider the well-known Goguen and Meseguer's *non-interference* (GMNI) security policy [16] for *deterministic* systems, where a low-privileged user (in this case, the author) should not be able to acquire any information about the activities (if any) of the high-privileged user (in this case, the conference PC chair). The HyperLTL formula for this policy in the context of our example is the following:

$$\varphi_{\text{GMNI}} = \forall \pi. \forall \pi'. \Big(\Box (dec = \lambda)_{\pi'} \wedge \Box (dec_\pi \neq dec_{\pi'}) \Big) \Rightarrow$$
$$\Box \Big(ses_\pi \Leftrightarrow ses_{\pi'} \Big)$$

where high input variable dec for a submission, ranging over $\{\mathsf{acc}, \mathsf{rej}, \mathsf{undec}\}$, contains the internal decision of the conference chair for the submission and low output proposition ses represents whether or not the submission is assigned to a session for presentation. By abuse of notation, we denote the value of variable dec in the associated state of trace π by dec_π. Finally, λ denotes an arbitrary dummy value for the dec.

The web interface exhibits the following blunt violation of GMNI, i.e., the author can learn the internal decision of the chair, while the status of the paper is pending. The first two rows show the status of two papers submitted to a conference after their notification (i.e., values sent on the low-observable channel): the first paper is accepted while the second is rejected. The last two rows show two other papers submitted to a different conference whose status are pending at the time the screenshot is taken. Although the authors should not be able to infer the internal decision making activities (i.e., high inputs) of the conference chair before the notification, this table leaks these activities as follows. When the chair sets $dec = \mathsf{acc}$, the paper is supposed to be assigned to a session in the technical program, while a rejected paper (i.e., $dec = \mathsf{rej}$) does not need to be assigned to a session. Now, by comparing the rows, one can observe that their 'Session' column have the same value (i.e., 'not yet assigned'). Likewise, the second and

the last rows have an empty 'Session' column. This simply means that the table reveals the internal status of the fourth and last papers as accepted and rejected, respectively, although their external status are pending. More specifically, in formula φ_{GMNI}, if π' and π are instantiated by the last two yellow rows, respectively, then purging dec by λ in π' will result in different ses observations, which clearly is a violation of non-interference through the four independent executions to generate the HTML table rows[2].

B. The Need for Runtime Monitoring

The above example illustrates how a security policy can easily be violated due to a careless implementation, where the value of high variable dec flows in the publicly-observable variable ses, although the chair did not take any inappropriate action that directly violates the security policy. This example demonstrates the need for designing techniques for monitoring the functional as well as security aspects of systems such as an online conference manager to inspect their health at run time or through periodic offline trace log analysis.

A key step in deploying any type of verification is identifying the specification of the system in terms of a formula. For our conference manager system, we now identify the specification in a sequence of steps starting from a simple formula, which is evolved into more complex ones:

- If the specification is only concerned with monitoring non-interference in deterministic executions, then formula φ_{GMNI} suffices. In this case, according to Table I, the complexity of monitoring a tree-shaped (respectively, acyclic) trace log is L-complete (respectively, NL-complete) in the size of the log.

- Next, let us imagine, the generation of the HTML report is accomplished by a set of concurrent threads. In this case, we need to refine φ_{GMNI} to obtain a stronger notion of confidentiality known as the Generalized Non-interference (GNI) [17], which permits nondeterminism in the behavior, but stipulates that low-security outputs may not be altered by the injection of high-security inputs:

$$\varphi_{\text{GNI}} = \forall \pi. \forall \pi'. \exists \pi''. \Box (dec_\pi = dec_{\pi''}) \wedge$$
$$\Box (ses_{\pi'} \Leftrightarrow ses_{\pi''})$$

The trace π'' is an interleaving of the high inputs trace π and the low outputs of the trace π'. In this case, according to Table I, the complexity of monitoring a tree-shaped (respectively, acyclic) trace log is L-complete (respectively, coNP-complete) in the size of the log. As can be seen, in case of asyclic trace logs, there is a significant jump in the complexity hierarchy of monitoring.

- As mentioned earlier, the EDAS information leak was due to an implementation bug, rather than by a mistake by the chair. For the designer of the conference management system, this is an important distinction:
  - Information leaks caused by an incorrect implementation should be fixed by eliminating the bug in the implementation, and

---

EDAS Conference and Journal Management System

Click on the menu items above to submit and review papers.

Please indicate whether you want to receive call-for-papers by updating your areas of interest.

Your conflicts-of-interest have not been updated in the last three months. (Persons with conflicts-of-interest are those who should not review p
the same institution.)

My pending, active and accepted papers

Only papers for upcoming conferences are shown.

Conference	Paper title (details)	Abstract or manuscript deadline	Edit	Add and delete authors	Upload paper	Files	Withdraw	Session
IEEE IPDPS 2015	██████	February 2, 2015 Anywhere on Earth	🔵	+	final deadline		❌	(not yet assigned)
IEEE IPDPS 2015	██████	October 18, 2014 Anywhere on Earth			paper status	📄		
IEEE IPDPS 2015	██████	October 18, 2014 Anywhere on Earth			withdrawn			
ICDCS 2015	██████	December 23, 2014 Anywhere on Earth	🔵	+	paper deadline	📄	❌	(not yet assigned)
ICDCS 2015	██████	December 23, 2014 Anywhere on Earth	🔵	+	paper deadline	📄	❌	

Fig. 3: EDAS conference management website's information leak.

○ Information leaks caused by the user could be fixed by educating the user or by improving the user interface, for example by issuing an explicit warning, or might not even need fixing if the information leak was intentional.

In the next step, we will further refine the specification to *only* refer to information leaks that are due to errors in the implementation, ignoring information leaks that are caused by the conference chair. For this purpose, we specify that a trace π_1 is OK (from the system implementation's point of view) even if π_1 results in a leak, as long as there exists a trace π_2, representing a different interaction of the chair with the system, that avoids the leak. In order to prevent trivial alternatives, such as "do nothing", we only consider alternative user behaviors that would accomplish the same objectives. Now, suppose that for two traces π_1 and π_2, the predicate $obj(\pi_1, \pi_2)$ indicates that π_1 and π_2 accomplish the same functional objectives, e.g.,

$$obj(\pi_1, \pi_2) =$$
$$\left(\Diamond(dec = \mathsf{acc})_{\pi_1} \Rightarrow \Diamond(dec = \mathsf{acc})_{\pi_2} \right) \wedge$$
$$\left(\Diamond(dec = \mathsf{rej})_{\pi_1} \Rightarrow \Diamond(dec = \mathsf{rej})_{\pi_2} \right)$$

i.e., if π_1 prescribes that a paper is accepted (respectively, rejected), then the same decision is made for the paper in π_2 as well. Then, our refined property is

expressed by the following HyperLTL formula:

$$\varphi_{\mathsf{ref}} = \forall \pi_1. \exists \pi_2. \forall \pi_3. \exists \pi_4.\ obj(\pi_1, \pi_2) \wedge$$
$$\Box(dec_{\pi_4} = dec_{\pi_2}) \wedge$$
$$\Box(ses_{\pi_4} \Leftrightarrow ses_{\pi_3})$$

The formula (with three quantifier alternations) expresses noninterference with the modification that the universally quantified trace π_1 is replaced by a existentially quantified trace π_2 that satisfies the same objectives. According to Table I, the complexity of monitoring the refined property in a tree-shaped (respectively, acyclic) trace log is L-complete (respectively, Π_4^p-complete) in the size of the Kripke structure. If we did not allow non-determinism, which translates to removing the innermost existential quantifier and, hence, one less alternation in the formula quantifiers, the complexities would be L-complete and Π_3^p-complete, respectively.

As can be seen in this example, the choice of the shape of the Kripke structure and the HyperLTL formula play a crucial role in the complexity of the model checking problem. This observation motivates rigorously investigating the complexity of RV for tree-shaped and acyclic Kripke structures. Our findings, summarized in Tables I and II, are presented in detail in Sections IV and V, respectively.

IV. SYSTEM COMPLEXITY

In this section, we analyze the complexity of the model checking problem in the size of the Kripke structure. We use

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

$$\text{MC[Fragment, Frame Type],}$$

where

- MC is the model checking problem, i.e., the problem to determine whether or not $\mathcal{K} \models \varphi$, where $\mathcal{K}$ is a Kripke structure and φ is a closed HyperLTL formula;
- Fragment is one of the following for φ:
  - AF-HyperLTL refers to the alternation-free fragment of HyperLTL (i.e., $\exists^+\psi$ or $\forall^+\psi$);
  - (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.

A. Tree-shaped Graphs

Our first result is that the model checking problem for tree-shaped Kripke structures is L-complete in the size of the Kripke structures. This result is particularly interesting, as system trace logs are very often stored as a set of traces grouped by common prefixes.

Theorem 1: MC[HyperLTL, tree] is L-complete in the size of the Kripke structure.

Proof: For the upper bound, 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 model checking algorithm maintains for each trace variable a counter on the number of traces, i.e., a logarithmic number of bits in size of the Kripke structure. To evaluate the inner LTL subformula, determine, in a backwards fashion, whether a subformula holds for a particular trace position. We need two counters on the length of the trace (corresponding to the variables i and j in the semantics of Until) for each Until subformula. Since the length of the trace is again bounded by the number of states, again a logarithmic number of bits will suffice. (Note that, since we are only interested in the complexity in the size of the Kripke structure, we consider the number of subformulas to be constant.)

The lower bound follows from the L-hardness of ORD [18]. ORD is the graph-reachability problem for directed line graphs. Graph reachability from s to t can be expressed with the formula $\exists\pi.\ \Diamond(s_\pi \land \Diamond t_\pi)$. ∎

B. Acyclic Graphs

We now turn to acyclic graphs. Acyclic Kripke structures are interesting in two contexts: (1) efficient storage of system trace logs in runtime verification, grouping the traces according to common prefixes *and* common suffixes, and (2) analyzing

certain security protocols, in particular authentication algorithms, which often consist of sequences of phases with no repetitions or loops. Such applications result in acyclic Kripke structures. We develop results for three different fragments of HyperLTL: (1) the alternation-free fragment (Theorem 2), the bounded-alternation fragment (Theorem 3), and (3) full HyperLTL (Corollary 1).

1) Alternation-free Formulas:

Theorem 2: MC[AF-HyperLTL, acyclic] is NL-complete in the size of the Kripke structure.

Proof: For the upper bound, we consider the case that the HyperLTL formula is existential, i.e., it is of the form

$$\exists\pi_1\ldots\exists\pi_k.\varphi,$$

where φ does not contain any trace quantifiers. For the case that the formula is universal, i.e., it is of the form

$$\forall\pi_1\ldots\forall\pi_k.\varphi,$$

we check the formula $\exists\pi_1\ldots\exists\pi_k.\neg\varphi$ and report the complemented result.

We consider the self-composition of the Kripke structure. Let $\mathcal{K} = \langle S, s_{init}, \delta, L\rangle$ be a Kripke structure, and let $\exists\pi_1\ldots\exists\pi_k.\varphi$ be an existential HyperLTL formula. The *self-composition* of $\mathcal{K}$ is the Kripke structure $\mathcal{K}' = \langle S^k, s_{init}^k, \delta', L'\rangle$, where

$$S^k = \overbrace{S \times S \times \cdots \times S}^{k \text{ times}}$$

$$s_{init}^k = \overbrace{(s_{init}, s_{init}, \ldots, s_{init})}^{k \text{ times}}$$

$$\delta' = \Big\{ \big((s_1,\ldots,s_k),(s_1',\ldots,s_k')\big) \mid \forall i \in [1,k] : (s_i, s_i') \in \delta \Big\}$$

$$L'(s_1,\ldots,s_k) = \Big\{ a_i \mid \exists i \in [1,k] : a \in L(s_i) \Big\}.$$

It is easy to see that the self-composition of an acyclic Kripke structure is again acyclic.

For the HyperLTL formula $\exists\pi_1\ldots\exists\pi_k.\varphi$, let φ' be the same as inner LTL formula φ, where every indexed proposition a_{π_i}, for some $i \in [1,k]$, is replaced by the atomic proposition a_i. Now, the Kripke structure $\mathcal{K}$ satisfies $\exists\pi_1\ldots\exists\pi_k.\varphi$, iff there is a path in the self-composition $\mathcal{K}'$, such that the corresponding trace satisfies φ'. Since the Kripke structure is acyclic, the length of the traces is bounded by the number of states of the Kripke structure. We can, therefore, nondeterministically guess the trace that satisfies φ', using a counter with a logarithmic number of bits in the number of states of $\mathcal{K}$.

The lower bound follows from the NL-hardness of the graph-reachability problem for ordered graphs [19]. Ordered graphs are acyclic graphs with a vertex numbering that is a topological sorting of the vertices. As in the proof of Theorem 1, we express graph reachability from s to t with the formula $\exists\pi.\ \Diamond(s_\pi \land \Diamond t_\pi)$. ∎

2) Formulas with Bounded Alternation Depth:

Next, we consider formulas where the number of quantifier alternations is bounded by a constant k. We show that changing

the frame structure from a tree to an acyclic graph results in significant increase in complexity (see Table I).

Theorem 3: MC[(EA)k-HyperLTL, acyclic] is Σ_k^p-complete in the size of the Kripke structure. MC[(AE)k-HyperLTL, acyclic] is Π_k^p-complete in the size of the Kripke structure.

Proof: We show membership in Σ_k^p and Π_k^p, respectively, by induction over k. According to Theorem 2, the model checking problem for $k = 0$, where the formula is alternation-free, is solvable in polynomial time. For $k + 1$ quantifier alternations, 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 existentially quantified traces in polynomial time and then verify the correctness of the guess, by the induction hypothesis, in Π_k^p. Hence, the model checking problem for $k+1$ is in Σ_{k+1}^p. Likewise, if the first quantifier is universal, we universally guess the universally quantified traces in polynomial time and verify the correctness of the guess, by the induction hypothesis, in Σ_k^p. Hence, the problem of determining $\mathcal{K} \models \varphi$ for $k + 1$ alternations in φ is in Π_{k+1}^p.

For the lower bound, we show that the model checking problem for HyperLTL formula with k alternations is Σ_k^p-hard and Π_k^p-hard, respectively, via a reduction from the *quantified Boolean formula* (QBF) satisfiability problem [20]:

> *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 y is restricted to at most k alternations of quantifiers, then QBF satisfiability is complete for Σ_{k+1}^p if $\mathbb{Q}_1 = \exists$, and for Π_k^p if $\mathbb{Q}_1 = \forall$. 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 a mapping from an arbitrary instance of QBF with k alternations and where $\mathbb{Q}_1 = \exists$ to the model checking problem of an acyclic Kripke structure and a HyperLTL formula with k quantifier alternations. Then, we show that the Kripke structure satisfies the HyperLTL formula

if and only if the answer to the QBF problem is affirmative. Figures 4 and 5 show an example.

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

- *(Atomic propositions* 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. Thus,

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

- *(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:

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

  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,

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

- *(Transition relation δ)* We now identify the members of δ:
  - 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,

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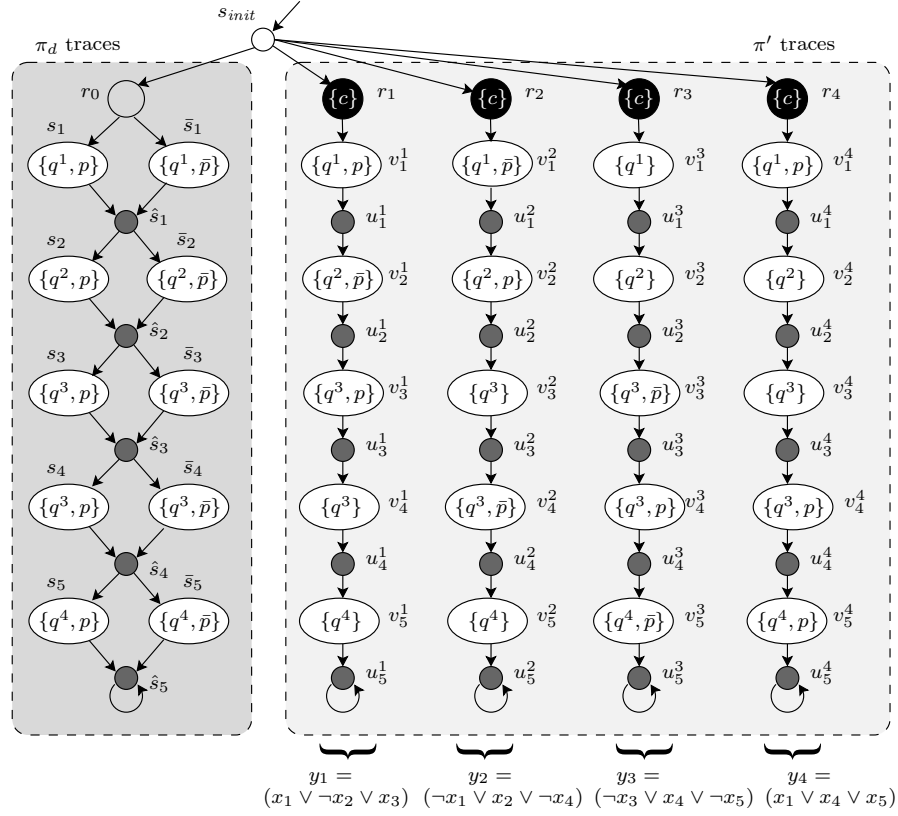$$\{(\hat{s}_i, s_{i+1}), (\hat{s}_i, \bar{s}_{i+1}) \mid i \in [1, n)\}.$$

Fig. 4: Mapping quantified Boolean formula $y = \exists x_1.\forall x_2.\exists x_3.\exists x_4.\forall x_5.(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_3 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee x_4 \vee x_5)$ to an instance of MC[(EA)k-HyperLTL, acyclic].

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

$$\varphi_{map} = \exists \pi_{k+1}.\forall \pi_k \cdots \exists \pi_2.\forall \pi_1.\forall \pi'.$$

$$\left(\bigwedge\nolimits_{d \in \{1,3,\ldots,k\}} \bigcirc \neg c_{\pi_d} \wedge \bigcirc c_{\pi'} \right) \Rightarrow$$

$$\left(\bigwedge\nolimits_{d \in \{2,4,\ldots,k+1\}} \bigcirc \neg c_{\pi_d} \wedge \right.$$

$$\left. \diamondsuit \left[\bigvee\nolimits_{d \in [1,k+1]} \left((q^d_{\pi_d} \Leftrightarrow q^d_{\pi'}) \wedge \right.\right.\right.$$

$$\left.\left.\left. ((p_{\pi'} \wedge p_{\pi_d}) \vee (\bar{p}_{\pi'} \wedge \bar{p}_{\pi_d})) \right) \right] \right)$$

Note that the formula has k alternations. Intuitively, this formula expresses the following: for all the (clause) traces that are universally quantified (i.e., the left side of the implication), there exist (clause) traces, where either p or $\bar{p}$ eventually matches its counterpart position in any trace π'. The matching positions identify the assignments of Boolean variables in the corresponding clauses that make the QBF instance true.

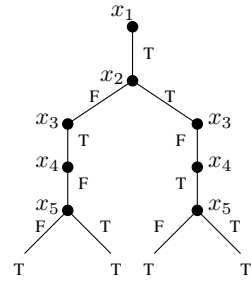We now show that the given quantified Boolean formula

Fig. 5: Model for the QBF instance in Fig. 4.

is *true* if and only if the Kripke structure obtained by our mapping satisfies the HyperLTL formula φ_{map}.

($\Rightarrow$)   Suppose that y is true. Then, there is an instantiation of existentially quantified variables for each value of universally quantified variables, such that each clause y_j, where $j \in [1, m]$ becomes true (see Fig. 5 for an example). We now use these instantiations to instantiate each $\exists \pi_{x_d}$ in HyperLTL formula φ_{map}, where $d \in \{2, 4, \ldots, k+1\}$ as follows. For each existentially quantified variable x_i, where $i \in [1, n]$, in depth $d \in [1, k+1]$, if $x_i = \texttt{true}$, we instantiate π_d with a trace that includes state s_i. Otherwise, the trace will include state $\bar{s}_i$. We now show that this trace instantiation

evaluates formula φ_{map} to true. Observe that the left side of the implication in the formula is basically distinguishing traces (i.e., clause traces π', where $\bigcirc c$ holds and traces corresponding to universal variables, where $\neg\bigcirc c$ is true). Since each y_j is true, for any instantiation of universal quantifiers, there is at least one literal in y_j that is true. If this literal is of the form x_i, then we have $x_i = \mathtt{true}$ and trace π_d will include s_i, which is labeled by p and q^d. Hence, the values of p (respectively, q^d), in both π_d and π' instantiated by trace

$$s_{init} r_j v_1^j \cdots u_n^j$$

are eventually equal. If the literal in y_j is of the form $\neg x_i$, then $x_i = \mathtt{false}$ and, hence, some trace π_d will include $\bar{s}_i$. Again, the values of $\bar{p}$ (respectively, q^d), in both π_d and π' are eventually equal. Finally, since all clauses are true, all traces π' reach a state where the right side of the implication becomes true.

($\Leftarrow$) Suppose our mapped Kripke structure satisfies the HyperLTL formula φ_{map}. This means that for each trace π' of the form

$$s_{init} r_j v_1^j \cdots u_n^j,$$

there exists a state u_i^j, where the values of q^d and either p or $\bar{p}$ are eventually equal to their counterparts in some trace π_d. If this trace is existentially quantified and includes s_i, then we assign $x_i = \mathtt{true}$ for the preceding quantifications. If the trace includes $\bar{s}_i$, then $x_i = \mathtt{false}$. Observe that since in no state p and $\bar{p}$ are simultaneously true and no trace includes both s_i and $\bar{s}_i$, variable x_i will have only one truth value. This way, a model similar to Fig. 5 can be constructed. Similar to the forward direction, it is straightforward to see that this valuation makes every clause y_j of the QBF instance true.

To establish the hardness for HyperLTL formulas where the first quantifier is universal, we analogously map an instance of QBF with k alternations and where $\mathbb{Q}_1 = \forall$ to the model checking problem of an acyclic Kripke structure and a HyperLTL formula that also begins with a universal quantifier. This time, the HyperLTL formula has $k+1$ quantifier alternations, because the inner-most quantifier is universal. We have thus reduced a Π_{k+1}^p-hard problem to the model checking problem for HyperLTL formulas with $k+1$ quantifier alternations where the first quantifier is universal. Hence, the model checking problem for formulas with k quantifier alternations where the first quantifier is universal is Π_k^p-hard. ∎

An important case of Theorem 3 are formulas with a single quantifier alternation, i.e., $k = 1$. This class of formulas contains, for example, generalized noninference [21], which can be expressed as a $\forall\exists$ and generalized non-interference [17], which can be expressed as a $\forall\forall\exists$ HyperLTL formula [3].

According to the polynomial hierarchy, the model checking problem for acyclic graphs is NP-complete for formulas of the form $\exists^+\forall^+\psi$ and coNP-complete for formulas of the form $\forall^+\exists^+\psi$.

It is worth noting that the special case of a single quantifier alternation consisting of a single existential and a single universal quantifier is already NP/coNP-complete for acyclic graphs, but still in L for trees. The intuitive reason is the repeated-diamonds structure in Fig. 4, which is possible in acyclic graphs, but not in trees. This structure allows us to select multiple Boolean values with a single trace quantifier.

Finally, Theorem 3 implies that the model checking problem for acyclic Kripke structures and HyperLTL formulas with an arbitrary number of quantifiers is in PSPACE. Moreover, its proof of lower bound shows that the problem is at least as hard as QBF, making it PSPACE-hard.

Corollary 1: MC[HyperLTL, acyclic] is PSPACE-complete in the size of the Kripke structure.

V.   COMBINED COMPLEXITY

We now analyze the complexity of the model checking problem in the size of the *combined* input, consisting of both the Kripke structure *and* the HyperLTL formula. Again, we separately focus on trees and acyclic graphs.

A. Trees

For tree-shaped Kripke structures, we first show that model checking is *efficiently parallelizable* for two fragments: (1) the alternation-free fragment (Theorem 4), and (2) formulas with one alternation consisting of a single universal and a single existential quantifier (Theorem 5). We denote (2) as (AE/EA)-HyperLTL. As we already noted, this model checking problem is particularly interesting because its complexity is significantly different for trees and acyclic graphs. This is again true for the combined complexity, which is in NC for trees, but Σ_2^p-complete or Π_2^p-complete, depending on whether the leading quantifier is existential or universal, for acyclic graphs.

Theorem 4: MC[(A/E)k-HyperLTL, tree] is in NC.

Proof: A decision problem is in NC, if there exists a parallel algorithm that runs in time $O(\log^c n)$ with $O(n^{c'})$ processors for some constants c and c'.

To verify an alternation-free formula with k quantifiers, we consider all combinations of k traces in the Kripke structure. Since k is a constant and the number of traces is bounded by the number of states of the tree, there is only a polynomial number of combinations. The evaluation of an individual combination corresponds to the evaluation of an LTL formula over a single trace, which can be done in NC [22]. We evaluate all combinations in parallel.

For universal quantifiers, we then compute the conjunction over these results by evaluating a binary tree of conjunctions. The height of the tree is logarithmic in the number of combinations. Using a linear number of processors, the evaluation therefore is done in logarithmic time. Likewise, for existential

quantifiers, we compute the disjunction over the results by evaluating a binary tree of disjunctions. ∎

Theorem 5: MC[(AE/EA)-HyperLTL, tree] is in NC.

Proof: Analogously to the proof of Theorem 4, we consider all pairs of traces in the Kripke structure. Since the number of traces is bounded by the number of states of the tree, the number of pairs is polynomial. The evaluation of an individual pair corresponds to the evaluation of an LTL formula over a single trace, which can be done in NC [22]. We evaluate all pairs in parallel. If the formula is of the form $\forall\exists$, we then need to evaluate the conjunction over all first elements of the pair, and the disjunction over all second elements. This can be done by a binary tree, where the upper part consists of conjunctions and the lower part consists of disjunctions. The height of the tree is logarithmic in the number of pairs. Using a linear number of processors, the evaluation is therefore done in logarithmic time. Likewise, if the formula is of the form $\exists\forall$, we compute the disjunction over the results by evaluating a binary tree, where the upper part consists of disjunctions and the lower part of conjunctions. ∎

Theorem 6: MC[(EA/AE)k-HyperLTL, tree] is Σ^p_{k+1}-complete in the combined size of the Kripke structure and the formula, if the leading quantifier is existential and is Π^p_{k+1}-complete if the leading quantifier is universal.

Proof: Matching upper bounds are provided in the proof of Theorem 7 in the next subsection for the more general case of acyclic graphs. We now show that the model checking problem is Σ^p_k-hard (respectively, Π^p_k-hard) via a reduction from QBF satisfiability, where the leading quantifier is existential (respectively, universal). In contrast to the proof of Theorem 3, we do not assume a specific form of the Boolean formula.

Let the quantified Boolean formula consist of Boolean variables $\{x_1, x_2, \ldots, x_n\}$, and a formula with k alternations

$$y = \mathbb{Q}_1 x_1.\mathbb{Q}_1 x_2 \ldots \mathbb{Q}_{n-1} x_{n-1}.\mathbb{Q}_n x_n. \varphi$$

where each $\mathbb{Q}_i \in \{\forall, \exists\}$ ($i \in [1, n]$) and φ is an arbitrary Boolean formula over variables $\{x_1, \ldots, x_n\}$. Satisfiability for QBF formulas of this type is complete for Σ^p_{k+1} if $\mathbb{Q}_1 = \exists$, and for Π^p_{k+1} if $\mathbb{Q}_1 = \forall$.

We reduce the satisfiability problem for a quantified Boolean formula to the model checking problem for a HyperLTL formula with the same quantifier structure.

- **Kripke structure** $\mathcal{K} = \langle S, s_{init}, \delta, L \rangle$. We use the simple Kripke structure shown in Fig. 6, which contains two traces $\{\}\{x\}^\omega$ and $\{\}\{\}^\omega$.

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

$$\mathbb{Q}_1 \pi_1.\mathbb{Q}_1 \pi_2 \ldots \mathbb{Q}_{n-1} \pi_{n-1}.\mathbb{Q}_n \pi_n. \varphi' \qquad (1)$$

where φ' is constructed from φ by replacing every occurrence of a variable x_i in the Boolean formula with $\bigcirc x_{\pi_i}$ in the HyperLTL formula.

The given formula is *true* if and only if the Kripke structure obtained by our mapping satisfies HyperLTL formula (1). We translate every assignment to the trace quantifiers to a corresponding assignment of the Boolean variables, and vice
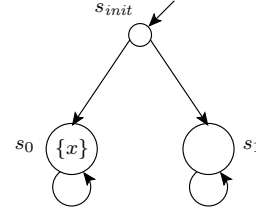

Fig. 6: Kripke structure in the proof of Theorem 6.

versa, as follows: Assigning the trace $\{\}\{x\}^\omega$ to π_i means that x_i is set to true, and assigning the trace $\{\}\{\}^\omega$ to π_i means that x_i is set to false. ∎

Corollary 2: MC[HyperLTL, tree] is PSPACE-complete.

B. Acyclic Kripke Structures

For HyperLTL formulas with bounded quantifier alternation, trees and acyclic graphs have the same model checking complexity (except for the special case of exactly one universal and one existential quantifier). We match the lower bounds for trees from Theorem 6 with upper bounds for acyclic graphs.

Theorem 7: MC[(EA/AE)k-HyperLTL, acyclic] is Σ^p_{k+1}-complete in the combined size of the Kripke structure and the formula, if the leading quantifier is existential and Π^p_{k+1}-complete if the leading quantifier is universal.

Proof: We show membership in Σ^p_{k+1} and Π^p_{k+1}, respectively, by induction over k. For the base case, $k = 0$, where the formula is alternation-free, the model checking problem can be solved in NP and co-NP, respectively, as follows. If the quantifiers are existential, we can nondeterministically guess a combination of the traces and verify the correctness of the guess in polynomial time, as the length of each trace is bounded by the number of states. Likewise, if the quantifiers are universal, we can universally guess a combination of the traces and verify the correctness of the guess in polynomial time.

For $k + 1$ quantifier alternations, 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 existentially quantified traces in polynomial time and verify the correctness of the guess, by the induction hypothesis, in Π^p_{k+1}. Hence, the model checking problem for $k + 1$ is in Σ^p_{k+2}. Likewise, if the first quantifier is universal, we universally guess the universally quantified traces in polynomial time and verify the correctness of the guess, by the induction hypothesis, in Σ^p_{k+1}. Hence, the model checking problem for $k + 1$ is in Π^p_{k+2}.

Together with the lower bounds for trees in Theorem 6, we obtain $\Sigma^p_{k+1}/\Pi^p_{k+1}$-completeness for k quantifier alternations.

∎

Corollary 3: MC[HyperLTL, acyclic] is PSPACE-complete.

VI. RELATED WORK

Model checking algorithms for HyperLTL were introduced in [13]. The satisfiability problem for HyperLTL was shown

to be decidable for the $\exists^*\forall^*$ fragment [15]. Runtime verification algorithms for HyperLTL include both automata-based algorithms [9], [11] and rewriting-based algorithms [10]. HyperLTL is also supported by a growing set of tools, including the model checker MCHyper [13], and the decision procedure EAHyper [23], and the runtime monitoring tool RVHyper [24].

A study of the impact of structural restrictions on the complexity of the model checking problem, similar to this paper, has been carried out for LTL [12]. The LTL model checking problem is PSPACE-hard if there exists a strongly connected component with two distinct cycles in the Kripke structure. If no such component exists, then the model checking problem is in coNP. For the special case of finite paths and trees, the LTL model checking problem is in NC, or, more precisely, in $AC^1(logDCFL)$ [22], [25].

VII. CONCLUSION

We have developed a detailed and fundamental classification of the complexity of the model checking problem for hyperproperties expressed in HyperLTL over trace logs that are stored as tree-shaped or acyclic Kripke structures. The complexity analysis is a crucial step for the development of runtime monitors, because in runtime verification methods for hyperproperties, the traces generated over time by the running system have to be stored into a growing data structure. This is a fundamental difference to monitoring techniques for standard trace properties, where the traces are evaluated individually and the monitors are usually memoryless.

We showed that for trees, the model checking complexity in the size of the Kripke structure is L-complete independently of the number of quantifier alternations. 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 combined complexity in the size of the Kripke structure and the length of the HyperLTL formula is in PSPACE for both trees and acyclic Kripke structures, and is as low as NC for the relevant case of trees and alternation-free HyperLTL formulas.

These results highlight two crucial design choices for monitoring algorithms:

- The substantial differences between the complexities reported in Tables I and II, in particular the contrast to the non-elementary complexity of the model checking problem for general graphs, are intriguing. These results suggest that non-exhaustive techniques such as runtime verification, that work on restricted structures, may have a significant complexity advantage over static verification.

- In the context of runtime verification, our results in Tables I and II clearly show the tradeoffs in deploying runtime verification technology in practice. First, note that for runtime verification, the size of the formula is expected to remain constant and, hence, what matters is the size of the Kripke structure. Tables I shows that the model checking complexity remains the same for trees, while it grows significantly for acyclic structures. This justifies careful space vs. time considerations in practical settings.

Our study raises many open questions for future work. An immediate question left unanswered in this paper is the precise complexity for trees and alternation-free HyperLTL formulas within NC. Next, it would be interesting to determine the complexity of the verification problem for further restricted structures such as flat graphs, i.e., graphs that have no nested cycles. Also, there are many extensions of HyperLTL, such as the branching-time logic HyperCTL* [3] and the first-order extension FOHLTL [26]. It would be very interesting to see if the differences we observed for HyperLTL carry over to these much more expressive logics. And, finally, we are currently working on designing runtime verification techniques that can reuse the result of past verification steps as the size of the Kripke structure grows.

REFERENCES

[1] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*, 2003, p. 29.

[2] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[3] 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)*, 2014, pp. 265–284.

[4] A. Pnueli, "The temporal logic of programs," in *Symposium on Foundations of Computer Science (FOCS)*, 1977, pp. 46–57.

[5] D. Giannakopoulou and K. Havelund, "Automata-Based Verification of Temporal Properties on Running Programs," in *Automated Software Engineering (ASE)*, 2001, pp. 412–416.

[6] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky, "Monitoring, Checking, and Steering of Real-Time Systems," *Electronic. Notes in Theoretical Computer Science*, vol. 70, no. 4, 2002.

[7] B. Finkbeiner and L. Kuhtz, "Monitor circuits for LTL with bounded and unbounded future," in *Runtime Verification*, 2009, pp. 60–75.

[8] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[9] S. Agrawal and B. Bonakdarpour, "Runtime verification of k-safety hyperproperties in HyperLTL," in *Proceedings of the IEEE 29th Computer Security Foundations (CSF)*, 2016, pp. 239–252.

[10] 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)*, 2017, pp. 77–93.

[11] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup, "Monitoring hyperproperties," in *Proceedings of the 17th International Conference on Runtime Verification*, 2017, pp. 190–207.

[12] L. Kuhtz and B. Finkbeiner, "Weak kripke structures and LTL," in *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR)*, 2011, pp. 419–433.

[13] 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)*, 2015, pp. 30–48.

[14] M. N. Rabe, "A temporal logic approach to information-flow control," Ph.D. dissertation, Saarland University, 2016.

[15] B. Finkbeiner and C. Hahn, "Deciding hyperproperties," in *Proceedings of the 27th International Conference on Concurrency Theory (CON-CUR)*, 2016, pp. 13:1–13:14.

[16] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the IEEE Symposium on Security and Privacy (S & P)*, 1982, pp. 11–20.

[17] D. McCullough, "Noninterference and the composability of security properties," in *Proceedings of the 1988 IEEE Symposium on Security and Privacy (S & P)*, 1988, pp. 177–186.

[18] K. Etessami, "Counting quantifiers, successor relations, and logarithmic space," *Journal of Compuer and System Sciences*, vol. 54, no. 3, pp. 400–411, 1997.

[19] T. Lengauer and K. Wagner, "The correlation between the complexities of the nonhierarchical and hierarchical versions of graph problems," *Journal of Computer and System Sciences*, vol. 44, no. 1, pp. 63 – 93, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000092900043

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

[21] J. McLean, "A general theory of composition for trace sets closed under selective interleaving functions," in *Proceedings of the IEEE Symposium on Security and Privacy (S & P)*, Apr. 1994, pp. 79–93.

[22] L. Kuhtz and B. Finkbeiner, "LTL path checking is efficiently parallelizable," in *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP 2009)*, 2009, pp. 235–246.

[23] 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)*, 2017, pp. 564–570.

[24] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup, "RVHyper: A runtime verification tool for temporal hyperproperties," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018, pp. 194–200.

[25] L. Kuhtz, "Model checking finite paths and trees," Ph.D. dissertation, Saarland University, 2010.

[26] B. Finkbeiner, C. Müller, H. Seidl, and E. Zalinescu, "Verifying Security Policies in Multi-agent Workflows with Loops," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 633–645.