Canonical Representations of k-Safety Hyperproperties

Bernd Finkbeiner
Reactive Systems Group
Saarland University
Saarbrücken, Germany
finkbeiner@react.uni-saarland.de

Lennart Haas
Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
lennart.haas@stud.uni-saarland.de

Hazem Torfah
Reactive Systems Group
Saarland University
Saarbrücken, Germany
torfah@react.uni-saarland.de

***Abstract*—Hyperproperties elevate the traditional view of trace properties form sets of traces to sets of sets of traces and provide a formalism for expressing information-flow policies. For trace properties, algorithms for verification, monitoring, and synthesis are typically based on a representation of the properties as omega-automata. For hyperproperties, a similar, canonical automata-theoretic representation is, so far, missing. This is a serious obstacle for the development of algorithms, because basic constructions, such as learning algorithms, cannot be applied.**

In this paper, we present a canonical representation for the widely used class of regular k-safety hyperproperties, which includes important polices such as noninterference. We show that a regular k-safety hyperproperty S can be represented by a finite automaton, where each word accepted by the automaton represents a violation of S. The representation provides an automata-theoretic approach to regular k-safety hyperproperties and allows us to compare regular k-safety hyperproperties, simplify them, and learn such hyperproperties. We investigate the problem of constructing automata for regular k-safety hyperproperties in general and from formulas in HyperLTL, and provide complexity bounds for the different translations. We also present a learning algorithm for regular k-safety hyperproperties based on the L* learning algorithm for deterministic finite automata.

***Index Terms*—Hyperproperties, Automata, Learning, Information-flow control.**

I. Introduction

Hyperproperties [1] generalize traces properties to sets of sets of traces. Famous examples of hyperproperties that cannot be expressed as trace properties are information-flow policies, such as noninterference, because they relate multiple runs of a system: a violation of a information-flow policy can therefore only be detected by looking at trace sets with more than one trace.

Many verification and analysis techniques for trace properties are, nowadays, based on the *automata-theoretic approach* [2], whereby the property is translated into an equivalent automaton and then processed by standard operations on automata. For hyperproperties, there is, so far, no automata-theoretic foundation. This means that algorithms for hyperproperties cannot be based directly on automata transformations.

One might argue that the lack of an automata representation is not a big issue, because many verification problems, such as model checking against k-safety hyperproperties, can be reduced, via a self-composition of the system under verification, to standard trace-based model checking against a trace property. However, there are important algorithmic approaches that do not translate this easily. A prime example are learning algorithms like Dana Angluin's L* algorithm [3]. Learning is a fundamental building block for compositional verification [4], synthesis [5], and for mining specifications of malicious behavior [6]–[8]. Generally, the advantage of learning algorithmis like L* compared to other construction methods is that the number of queries the learner needs to pose to the teacher is determined by the size of the smallest deterministic automaton for the target language. Usually, this is significantly smaller than the intermediate automata that occur in a direct construction.

In this paper, we develop an automata representation for the class of regular k-safety hyperproperties. The k-safety hyperproperties are those hyperproperties where every set of traces that violates the hyperproperty contains a set of at most k bad trace prefixes, such that every extension of the bad prefixes also violates the hyperproperty. We represent a k-safety hyperproperty using a bad-prefix automaton, a finite-word automaton that recognizes the bad prefixes as finite words over an alphabet consisting of k-tuples, where each word in the language is interpreted as a set of (at most) k traces. A k-safety hyperproperty may, in principle, have many different representations as such a bad prefix language. Consider, for example, the 2-safety hyperproperty given by the HyperLTL formula $\varphi = \forall\pi\forall\pi'.\ \square(a_\pi \rightarrow a_{\pi'})$ over the set of atomic propositions $\{a\}$, which specifies for each pair of traces π, π', that whenever a holds on π it also holds on π'. A bad prefix for φ is, for example, the set of finite traces $\{t, t'\}$ where $t = \{a\}\{a\}$ and $t' = \{a\}\{\}$. A tuple representation of $\{t, t'\}$ is the sequence $(\{a\},\{a\})(\{a\},\{\})$. Since the set defines no order on t and t', another representation of the bad prefix is the sequence $(\{a\},\{a\})(\{\},\{a\})$.

This work was partially 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).

Just as for bad prefixes for trace properties, the bad prefixes may or may not be minimal; additionally, any ordering of traces in a trace set will lead to a different tuple representation. Using the terminology for trace properties [9], we define a bad-prefix automaton as *tight* if it accepts all bad prefixes; additionally, we say the automaton is *permutation-complete* if it is closed under permutations of the tuples. Minimal deterministic bad-prefix automata that are both tight and permutation-complete provide a canonical representation for k-safety hyperproperties. We provide algorithms for constructing permutation-complete bad-prefix automata for regular k-safety hyperproperties starting from representations in HYPERLTL, nondeterministic bad-prefix automata and deterministic bad-prefix automata.

Based on this automaton representation, we present the first *learning algorithm for hyperproperties*. Our algorithm learns a minimal deterministic tight permutation-complete bad-prefix automaton for some unknown regular k-safety regular hyperproperty and an unknown minimal k.

The remainder of the paper is structured as follows. We give background on hyperproperties and automata in Section II. Section III introduces automata for k-safety hyperproperties and establishes basic facts about tight and permutation-complete bad-prefix automata. In Section IV we present a learning framework for learning k-safety regular hyperproperties and a realization of the framework for HYPERLTL in Section V. With Section VI we conclude with some decidability results on the learnability of k-safety-hyperproperties.

II. BACKGROUND

A. Hyperproperties.

A *trace property* $\mathcal{T}$ over an alphabet Σ is a set of infinite traces from Σ^ω. A trace $t \in \Sigma^\omega$ satisfies the property $\mathcal{T}$ if $t \in T$. The set of all trace properties over the alphabet Σ is denoted by $\mathcal{P}(\Sigma^\omega)$.

A *hyperproperty* over an alphabet Σ is a set $\mathbf{H} \subseteq \mathcal{P}(\Sigma^\omega)$ of sets of infinite traces over Σ [1]. A set of infinite traces $\mathcal{T} \subseteq \Sigma^\omega$ satisfies a hyperproperty $\mathbf{H}$ if $\mathcal{T} \in \mathbf{H}$.

B. HYPERLTL*: A temporal logic for hyperproperties.*

Let $\mathcal{V}$ be an infinite supply of trace variables and let AP be a set of atomic propositions. The syntax of HyperLTL is given by the following grammar:

$$\begin{aligned}\psi \ &::= \ \exists \pi.\, \psi \ \mid \ \forall \pi.\, \psi \ \mid \ \varphi \\ \varphi \ &::= \ a_\pi \ \mid \ \neg\varphi \ \mid \ \varphi \vee \varphi \ \mid \ \bigcirc\varphi \ \mid \ \varphi \,\mathsf{U}\, \varphi\end{aligned}$$

where $a \in \mathrm{AP}$ is an atomic proposition and $\pi \in \mathcal{V}$ is a trace variable. Note that atomic propositions are indexed by trace variables. The quantification over traces makes it possible to express properties like "on all traces ψ must hold", which is expressed by $\forall\pi.\ \psi$. Dually, one can express that "there exists a trace such that ψ holds", which is denoted by $\exists\pi.\ \psi$. The temporal operators are defined as for LTL. The next operator $\bigcirc\psi$ states that the next step along a trace must satisfy ψ. The until operator $\psi_1 \,\mathcal{U}\, \psi_2$ states that ψ_1 must hold along a trace until ψ_2 holds. We also use the derived temporal operators $\Diamond\psi$ eventually ψ holds, $\Box\psi$ the formula ψ holds on all trace positions, and $\psi_1 \,\mathcal{R}\, \psi_2$, the release operator, the dual to $\mathcal{U}$, that states that ψ_2 may not hold only after ψ_1 has been fulfilled otherwise ψ_2 must hold forever.

We abbreviate the formula $\bigwedge_{x\in X}(x_\pi \leftrightarrow x_{\pi'})$, expressing that the traces π and π' are equal with respect to a set of atomic propositions $X \subseteq \mathrm{AP}$ by $\pi =_X \pi'$.

Example 1. The following HYPERLTL formula defines the security policy of reactive noninterference. Let $\mathrm{AP} = I \cup O$, where I and O are sets of low-security inputs and low-security outputs, respectively:

$$\forall\pi.\forall\pi'.\ (\pi \neq_I \pi')\ \mathsf{R}\ (\pi =_O \pi')$$

The formula states that, for every pair of traces, as long as there is no difference in the observed inputs, no difference should be observed in the outputs.

Let $\mathcal{T}$ be a set of traces of some alphabet 2^{AP} for some set of atomic propositions AP. Formally, the semantics of HyperLTL formulas is given with respect to a *trace assignment* Π from $\mathcal{V}$ to $\mathcal{T}$, i.e., a partial function mapping trace variables to actual traces. $\Pi[\pi \mapsto t]$ denotes that π is mapped to t, with everything else mapped according to Π. For a trace $t \in \mathcal{T}$, let $t[i,\infty]$ denote the suffix of t starting at position i. With $\Pi[i,\infty]$ we denote the trace assignment that is equal to $\Pi(\pi)[i,\infty]$ for all π.

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

We say a set of traces T *satisfies* a HyperLTL formula φ if $\Pi \models_T \varphi$, where Π is the empty trace assignment.

We call a HYPERLTL formula φ syntactically-safe if it is of the form $\varphi = \forall^*\pi.\psi$ and ψ is a syntactically-safe LTL formula, i.e., an LTL formula where the only temporal operators are $\bigcirc$ and R.

C. Automata.

A nondeterministic finite automaton (NFA) is defined as a tuple $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$, where Q denotes a finite set of states, Σ denotes a finite alphabet, q_0 denotes a designated initial state, $F \subseteq Q$ denotes the set of accepting states, and $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ denotes the transition relation that maps a state and a letter to the set of successor states. A run in a $\mathcal{A}$ on a finite word $w = w_0 \ldots w_n \in \Sigma^*$ is a sequence of states $r = q_0 \ldots q_{n+1} \in Q^*$ with $q_{i+1} \in \delta(q_i, w_i)$ for all $0 \leq i \leq n$. The run r is accepting if $q_{n+1} \in F$. The set of all accepted words by an automaton $\mathcal{A}$ is called its language and is denoted by $\mathcal{L}(\mathcal{A})$. The size of an automaton is the size of its set of states Q and is denoted by $|\mathcal{A}|$.

Deterministic finite automata (DFA) are a special case of NFAs, where $|\delta(q,a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. The transition relation of a deterministic automaton can be given as a function $\delta : Q \times \Sigma \rightarrow Q$.

A Büchi automaton $\mathcal{B} = (Q, \Sigma, q_0, F, \Delta)$ is an automaton over infinite words. A run of $\mathcal{B}$ on an infinite word $w = w_1 w_2 \cdots \in \Sigma^\omega$ is an infinite sequence $r = q_0 q_1 \cdots \in Q^\omega$ with $q_{i+1} \in \delta(q_i, w_i)$ for all $i \in \mathbb{N}$. A run r is accepting if there exist infinitely many $i \in \mathbb{N}$ such that $q_i \in F$. A Büchi automaton $\mathcal{A} = (Q, \Sigma, q_0, F, \Delta)$ is called safety automaton if $Q = F$, i.e., every run on a safety automaton is accepted. In the rest of the paper we omit the set F from the tuple representation of safety automata.

D. Safety Languages.

A finite word $w = w_1 \dots w_i \in \Sigma^*$ is called a bad prefix for a language $L \subseteq \Sigma^\omega$, if every infinite word $v \in \Sigma^\omega$ with prefix w is not in the language L. A language $L \subseteq \Sigma^\omega$ is called a safety language if every $w \notin L$ has a bad prefix for L. We denote the set of all bad prefixes for a language $\mathcal{L}$ by $\mathsf{BadPref}(L)$. We say $X \subseteq \mathsf{BadPref}(L)$ is a trap for L, if for every $w \notin L$, there exists a prefix of w in X and denote the set of all traps by $\mathsf{Trap}(L)$.

For every ω-regular safety language L, a finite automaton $\mathcal{A}$ that accepts the bad prefixes of L is called a bad-prefix automaton for L. We say that $\mathcal{A}$ is tight if $L(\mathcal{A}) = \mathsf{BadPref}(L)$ and fine if there exists some $X \in \mathsf{Trap}(L)$ and $L(\mathcal{A}) = X$.

E. Notations.

For a sequence $t = \alpha_1 \alpha_2 \dots$ and $i \leq j \in \mathbb{N}$, $t[i] = \alpha_i$, $t[i,j] = \alpha_i \dots \alpha_j$. For $t \in \Sigma^\omega$, $t[i, \infty] = \alpha_i \alpha_{i+1} \dots$.

For $t \in \Sigma^*$ and $\tau \in \Sigma^* \cup \Sigma^\omega$, t is a prefix of τ denoted by $t \leq \tau$ if and only if $|t| \leq |\tau| \wedge \forall i \leq |t|.\ t[i] = \tau[i]$.

III. Automata for k-Safety-Hyperproperties

A. Representations of k-Safety-Hyperproperties

The definition of safety can be generalized to hyperproperties by generalizing the definition of bad-prefixes from a finite trace to a finite set of finite traces [1]. For a set of finite traces $T \subseteq \Sigma^*$ and a set of infinite traces $T' \subseteq \Sigma^\omega$, we say that T is a prefix of T', denoted by $T \leq T'$, if and only if $\forall t \in T. \exists t' \in T'.\ t \leq t'$. A hyperproperty $\mathbf{S}$ over Σ is *hypersafety* if and only if

$$\forall T' \subseteq \Sigma^\omega.\ (T' \notin \mathbf{S} \Rightarrow \exists T \subseteq \Sigma^*.\ (T \leq T' \wedge \forall \widetilde{T} \subseteq \Sigma^\omega.\ (T \leq \widetilde{T} \Rightarrow \widetilde{T} \notin \mathbf{S})))$$

We call T a *bad-prefix* for the hyperproperty $\mathbf{S}$. We denote the set of bad-prefixes for a hypersafety property $\mathbf{S}$ by $\mathsf{BadPref}(\mathbf{S}) = \{T \subseteq \Sigma^* \mid \forall T' \subseteq \Sigma^\omega.\ (T \leq T' \Rightarrow T' \notin \mathbf{S})\}$. We call a bad prefix T for $\mathbf{S}$ minimal, if and only if, there exists no $T' < T$ that is also a bad prefix for $\mathbf{S}$.

Definition 1 (k-safety hyperproperty)**.** For any $k' \in \mathbb{N}$, let $\mathsf{BadPref}(\mathbf{S}, k') = \{T \in \mathsf{BadPref}(\mathbf{S}) \mid |T| \leq k'\}$. We call an element of $\mathsf{BadPref}(\mathbf{S}, k')$ a *k'-bad-prefix* for $\mathbf{S}$. A safety hyperproperty $\mathbf{S}$ is a *k-safety hyperproperty*, if every set $T' \notin \mathbf{S}$ has a k-bad-prefix.

In the next section, we define finite automata for k-safety hyperproperties by defining automata that represent their sets of bad-prefixes. Each finite bad prefix of a safety-hyperproperty can be represented by a finite word as follows.

Definition 2 (Representations of k-safety hyperproperties)**.** For a sequence $\sigma = \vec{v}_0 \vec{v}_1 \vec{v}_2 \dots \vec{v}_m \in (\Sigma^k)^*$, let unzip be the mapping defined as $\mathsf{unzip}(\sigma) = \{t_i \in \Sigma^* \mid 1 \leq i \leq k, \forall 0 \leq j \leq m.\ t_i[j] = \vec{v}_j[i]\}$. We call $\sigma \in (\Sigma^k)^*$ a *representation* of $T \subseteq \Sigma^*$ if $\mathsf{unzip}(\sigma) = T$.

For a k-safety-hyperproperty $\mathbf{S}$, a language $L \subseteq (\Sigma^{k'})^*$ is called a *representation* of $\mathbf{S}$ for some $k' \in \mathbb{N}$, when: for all $T \subseteq \Sigma^\omega$, $T \notin \mathbf{S}$, if and only if, there exists $\sigma \in L$, such that, $\mathsf{unzip}(\sigma) \subseteq T$ and $\mathsf{unzip}(\sigma) \in \mathsf{BadPref}(\mathbf{S})$. We call k' the *arity* of the representation and further call L a *k'-representation* of $\mathbf{S}$.

We extend the definition of unzip to languages. For a language $L \subseteq (\Sigma^k)^*$, $\mathsf{unzip}(L) = \{\mathsf{unzip}(\sigma) \mid \sigma \in L\}$.

Notice that a k-safety-hyperproperty has several representations of different arities. It also has several representations of the same arity (by permuting the order on the traces). We denote the set that defines the union of all representations of a k-safety hyperproperty $\mathbf{S}$ of arity k' by $\mathfrak{P}(\mathbf{S}, k')$.

Example 2. The security policy of reactive noninterference given in Example 1 is an example of a 2-safety hyperproperty. In Figure 1 the policy is given by the HyperLTL formula $\mathbf{S}$. A violation of $\mathbf{S}$ along two traces is observed, if up to some position, the traces share the same input sequence and differ in the output values at this position. The set of bad-prefixes for $\mathbf{S}$ is given by the set $\mathsf{BadPref}(\mathbf{S})$. To check whether there is a violation of $\mathbf{S}$ it is sufficient to find two traces that violate $\mathbf{S}$, i.e., any set of traces that violates $\mathbf{S}$ has a bad prefix of size two. The set $\mathsf{BadPref}(\mathbf{S}, 2)$ gives all the bad-prefixes of size two. Two sets of 2-representations of these bad-prefixes are given by the sets Rep_1 and Rep_2[1]. To understand the difference between the representations in Rep_1 and Rep_2 look at the following two traces: Assume w.l.o.g. that $I = \{i\}$ and $O = \{o\}$ and let $t = \{i,o\}\{i,o\}\{i,o\}\dots$ and $t' = \{i,o\}\{i,o\}\{i\}\dots$ be two infinite traces over $2^{O \cup I}$. The set $\{t, t'\}$ violates $\mathbf{S}$ with the bad prefix $T = \{\ \{i,o\}\{i,o\}\{i,o\}\ ,\ \{i,o\}\{i,o\}\{i\}\ \}$. The set T has two 2-representations: the sequence $\sigma_1 = (\{i,o\},\{i,o\})(\{i,o\},\{i,o\})(\{i,o\},\{i\})$, which is in the set Rep_1 but not in Rep_2, and another representation is $\sigma_2 = (\{i,o\},\{i,o\})(\{i,o\},\{i,o\})(\{i\},\{i,o\})$ which belongs to Rep_2 but not to Rep_1. Both σ_1 and σ_2 belong, however, to the set $\mathfrak{P}(\mathbf{S}, 2)$, which contains all representations of 2-bad-prefixes of $\mathbf{S}$.

[1]The sets Rep_1 and Rep_2 are not the only sets with 2-representations of the bad-prefixes of $\mathbf{S}$. For $\mathbf{S}$ there is an infinite number of distinct 2-representations.

$$\mathbf{S} = \forall\pi.\forall\pi'.\ (\pi \neq_I \pi')\ \mathsf{R}\ (\pi =_O \pi')$$
$$\mathsf{BadPref}(\mathbf{S}) = \{T \subseteq \Sigma^* \mid \exists t, t' \in T.\ \exists j.\ t[...j]_I = t'[...j]_I \wedge t[j]_O \neq t'[j]_O\}$$
$$\mathsf{BadPref}(\mathbf{S}, 2) = \{\{t, t'\} \subseteq \Sigma^* \mid \exists j.\ t[...j]_I = t'[...j]_I \wedge t[j]_O \neq t'[j]_O\}$$
$$\mathrm{Rep}_1 = \{(\alpha_0, \alpha'_0) \ldots (\alpha_m, \alpha'_m) \in (\Sigma^2)^* \mid \forall j.(\alpha_j)_I = (\alpha'_j)_I \ \wedge\ \exists i.\exists o \in O.\ o \in \alpha_i \wedge o \notin \alpha'_i\}$$
$$\mathrm{Rep}_2 = \{(\alpha_0, \alpha'_0) \ldots (\alpha_m, \alpha'_m) \in (\Sigma^2)^* \mid \forall j.(\alpha_j)_I = (\alpha'_j)_I \ \wedge\ \exists i.\exists o \in O.\ o \notin \alpha_i \wedge o \in \alpha'_i\}$$
$$\mathfrak{P}(\mathbf{S}, 2) = \{(\alpha_0, \alpha'_0) \ldots (\alpha_m, \alpha'_m) \in (\Sigma^2)^* \mid \forall j.(\alpha_j)_I = (\alpha'_j)_I \ \wedge\ \exists i.\exists o \in O.\ o \in \alpha_i \leftrightarrow o \notin \alpha'_i\}$$

Fig. 1. A 2-safety-hyperproperty given by a HYPERLTL formula $\mathbf{S}$. The formula $\mathbf{S}$ defines the information flow policy of reactive noninterference. The sets $\mathsf{BadPref}(\mathbf{S})$, $\mathsf{BadPref}(\mathbf{S}, 2)$, Rep_1, Rep_2, and $\mathfrak{P}(\mathbf{S}, 2)$ define the sets of bad-prefixes, 2-bad-prefixes, two different 2-representations, and the set of all 2-representation of $\mathbf{S}$, respectively. The set Σ is defined as $\Sigma = 2^{\mathrm{AP}}$ for a set of atomic propositions $\mathrm{AP} = O \cup I$.

In general, for any k-safety hyperproperty $\mathbf{S}$, if a sequence $\sigma \in \mathfrak{P}(\mathbf{S}, k')$ for any $k' \in \mathbb{N}$, then so is any permutation of σ.

Theorem 1. *For every k-safety hyperproperty $\mathbf{S}$, and for $k' \geq k$, there is a k'-representation of $\mathbf{S}$.*

Proof. Clearly, every k-safety hyperproperty has a representation of arity k. Let L be a k-representation for $\mathbf{S}$. Define L' such that each $\sigma' \in L'$ is of the form $\sigma' = (\alpha_0^1, \ldots, \alpha_0^k, \ldots, \alpha_0^{k'})(\alpha_1^1, \ldots, \alpha_1^k, \ldots, \alpha_1^{k'}) \cdots \in (\Sigma^{k'})^*$, where $(\alpha_0^1, \ldots, \alpha_0^k)(\alpha_1^1, \ldots, \alpha_1^k) \cdots \in L$, and for all $i \in \mathbb{N}$ and for all $k < j \leq k'$ we have $\alpha_i^j = \alpha_i^k$. Let the set $\mathsf{unzip}(\sigma') = \{t_1, \ldots, t_k, \ldots, t_{k'}\}$. Clearly, for $k < j \leq k'$, we have $t_j = t_k$. Thus, $\mathsf{unzip}(L') = \mathsf{unzip}(L)$, which makes L' a k'-representation of $\mathbf{S}$. □

In the rest of the paper, the length of a bad prefix T is the length of the longest trace in T. The size of a bad prefix T is the size $|T|$.

B. Bad-prefix automata for k-safety hyperproperties

We now develop a canonical representation for k-safety hyperproperties. We start by defining bad-prefix automata for k-safety hyperproperties. At the end of the section we show that *minimal, deterministic, tight and permutation-complete* bad-prefix automata give a canonical representation for k-safety hyperproperties.

Definition 3 (Regular k-safety hyperproperties)**.** A k-safety hyperproperty $\mathbf{S}$ is called regular if a representation of $\mathbf{S}$ is a regular language.

If a k-safety hyperproperty $\mathbf{S}$ is regular, we can build an automaton that recognizes one of its representations for some arity k'. We call such an automaton a *k'-bad-prefix automaton* for $\mathbf{S}$. An automaton is a bad-prefix automaton for $\mathbf{S}$, if it is a k'-bad-prefix automaton for some arity $k' \in \mathbb{N}$. In the following, we show that we can distinguish different types of bad-prefix automata for k-safety-hyperproprties. The distinction is based on the traditional notions of *tightness* and *fineness* for bad-prefix automata for regular properties [9], and the novel notion of *permutation-completeness* that we define later in this section.

A *tight* bad-prefix automaton for an ω-regular property $\mathcal{T}$ accepts all bad-prefixes of $\mathcal{T}$. The language of a *fine* bad-prefix automaton for $\mathcal{T}$ includes at least one bad prefix for each word $\sigma \notin \mathcal{T}$. Following this tradition we can also make a similar distinction for bad-prefix automata for k-safety-hyperproperty $\mathbf{S}$.

Definition 4 (Tight and fine k-bad-prefix automata)**.** Let $\mathcal{A}$ be a k'-bad-prefix automaton for a k-safety-hyperproperty $\mathbf{S}$ for some $k, k' \in \mathbb{N}$. We call $\mathcal{A}$ *tight* if and only if A accepts a representation for each bad prefixes T of $\mathbf{S}$ with $|T| \leq k'$.

$\mathcal{A}$ is called *fine* if and only if for every word $T \notin \mathbf{S}$ it accepts a representation of at least one bad prefix (not necessarily the minimal one) of T.

Kupfermann and Vardi showed how to construct tight bad-prefix automata for safety-properties [9]. The same constructions cannot be adapted for k-safety hyperproperties, due to the following reasoning. From its definition, a bad-prefix automaton $\mathcal{A}$ for a k-safety-hyperproperty $\mathbf{S}$ that is fine must, for each set T not in $\mathbf{S}$, accept at least one representation of a bad prefix of T. If $\mathcal{A}$ is not tight then either (1) $\mathcal{A}$ is not *vertically tight*:accepts a representation for a bad prefix T, but does not accept any representation for some $T \subset T'$ with $|T'| \leq k'$ which is also a bad prefix for $\mathbf{S}$ or (2) $\mathcal{A}$ is not *horizontally tight*: there is a representation t' of a set $\{w \mid \exists w' \in T, w < w'\}$ that represents a smaller bad prefix for $\mathbf{S}$, i.e., a trace in T is not minimal. The latter case defines tightness according to the traditional definition as in [9].

Remark 1. Notice that there exists no fine automaton that accepts a representation for a bad prefix T that is not minimal, but accepts no representation for all bad prefixes $T' \subset T$. Assume that no representation of any T' is accepted by $\mathcal{A}$. This means that there is word $\mathbf{S}$ for which no representation of any of its bad-prefixes is accepted by $\mathcal{A}$, namely the set $\widetilde{T}$, where each word in $\widetilde{T}$ is an infinite extension of a word in T'. This contradicts the assumption that $\mathcal{A}$ is a bad-prefix automaton for $\mathbf{S}$.

The next theorem how to construct a bad-prefix automaton that is horizontally tight using the construction presented in [9]. A construction for vertical tightness is left for the theorem that follows.

Theorem 2. *For a k-safety hyperproperty $\mathbf{S}$ over Σ, we can construct a tight bad-prefix automaton for $\mathbf{S}$ of size:*

- *$O(|\mathcal{A}|)$, when $\mathbf{S}$ is represented by a deterministic bad-prefix automaton $\mathcal{A}$.*
- *$2^{O(|\mathcal{A}|)}$, when $\mathbf{S}$ is represented by a nondeterministic bad-prefix automaton $\mathcal{A}$.*

Proof. The proof uses the ideas presented in [10].

- Let $\mathcal{A} = (Q, \Sigma^{k'}, q_0, F, \delta)$ be a deterministic bad-prefix automaton for $\mathbf{S}$ for some $k' \geq k$. To construct a deterministic horizontally tight bad-prefix automaton for $\mathbf{S}$ we replace the set of accepting states F of $\mathcal{A}$ by a set F' which is defined as follows:
$$F' = \{q \in Q \mid \forall \sigma \in Q^{\omega}.\ q < \sigma \rightarrow \exists i \in \mathbb{N}.\sigma[i] \in F\}$$
The set F' defines the set of states q from which there is no infinite run in the automaton that has no accepting state.
- If $\mathcal{A}$ is a nondeterministic bad-prefix automaton for $\mathbf{S}$, we can construct an equivalent deterministic k-bad-prefix automaton $\mathcal{A}'$ of size $2^{|A|}$ and use the construction above. $\square$

Bad-prefix automata for k-safety hyperproperties can also be distinguished according to the representations they accept. A k-bad-prefix automaton $\mathcal{A}$ is called *permutation-complete* if it accepts all representations of every k-bad-prefix it accepts.

In general, the goal is to build a tight and permutation-complete bad-prefix automaton for a k-safety hyperproperty. For tasks such as monitoring a system against a k-safety hyperproperty, such automata are of major importance. With tight automata violations are detected as early as possible. A permutation-complete automaton does not depend on the ordering of the traces and therefore detects a violation no matter in what order the traces are observed.

In the next theorem we show how to construct a permutation-complete and tight k-bad-prefix automaton for a k-safety hyperproperty.

Theorem 3. *For a deterministic k-bad-prefix automaton $\mathcal{A}$ of some k-safety hyperproperty $\mathbf{S}$ over Σ, we can construct a deterministic, tight and permutation-complete k-bad-prefix automaton of size $|A|^{k^k}$.*

Proof. Let $\mathcal{A} = (Q, \Sigma^k, q_0, F, \delta)$ be a deterministic k-bad-prefix automaton for $\mathbf{S}$. We construct a permutation-complete and vertically tight automaton $\mathcal{A}_{\mathfrak{P}}$ for $\mathbf{S}$ that accepts a word $\sigma \in (\Sigma^k)^*$ if any of its permutations [2]is accepted by $\mathcal{A}$. We define these permutations as follows. Let $\varsigma_1, \ldots, \varsigma_{k^k} : \{1, \ldots, k\} \rightarrow \{1, \ldots, k\}$ be pairwise different functions. A permutation of a tuple $(t_1, \ldots, t_k)$ with respect to one function ς_i for $1 \leq i \leq k^k$ is a tuple $(t_{\varsigma_i(1)}, \ldots, t_{\varsigma_i(k)})$. The deterministic bad-prefix automaton $\mathcal{A}_{\mathfrak{P}}$ is defined by the tuple $(Q_{\mathfrak{P}}, \Sigma^k, q_{0,\mathfrak{P}}, F_{\mathfrak{P}}, \delta_{\mathfrak{P}})$, where:

- $Q_{\mathfrak{P}} = (Q_1 \times \cdots \times Q_{k^k})$ where $Q_i = \{(q, i) \mid q \in Q\}$ for $1 \leq i \leq k^k$. A set of states Q_i resembles a copy of the automaton $\mathcal{A}$ that accepts a word σ if it is a permutation of a word σ' accepted by $\mathcal{A}$ with respect to the permutation function ς_i. The initial state $q_{0,\mathfrak{P}} = ((q_0, 1), \ldots, (q_0, k^k))$.
- A word is accepted if one of its permutations is accepted. We define the set of accepting states as $F_{\mathfrak{P}} = \{((q_1, 1), \ldots, (q_{k^k}, k^k)) \mid \exists i.\ q_i \in F\}$.
- The transition relation $\delta_{\mathfrak{P}}$ is defined as follows:
$$((q_1, 1), \ldots, (q_{k^k}, k^k)) \xrightarrow{(t_1, \ldots, t_k)} ((q_1', 1), \ldots, (q_{k^k}', k^k))$$
when $(q_i \xrightarrow{(t_{\varsigma_i(1)}, \ldots, t_{\varsigma_i(k)})} q_i') \in \delta$ for all $1 \leq i \leq k^k$. For each q_i the successor state q_i' for a letter $(t_1, \ldots, t_k)$ is determined by the the transition of its permutation $(t_{\varsigma_i(1)}, \ldots, t_{\varsigma_i(k)})$ in the automaton $\mathcal{A}$.

The automaton $\mathcal{A}_{\mathfrak{P}}$ is a deterministic, permutation-complete and vertically tight. If the automaton $\mathcal{A}_{\mathfrak{P}}$ is not horizontally tight, it can then be translated to on by redefining the set $F_{\mathfrak{P}}$ using the construction in Theorem 2. The size of $A_{\mathfrak{P}}$ is $|Q|^{k^k}$. $\square$

Corollary 1. *For a nondeterministic k-bad-prefix automaton $\mathcal{A}$ of some k-safety hyperproperty $\mathbf{S}$ over Σ, we can construct a permutation-complete and tight k-bad-prefix automaton of size $2^{|A| \cdot k^k}$.*

The exponential blow-up in the size of the automaton in the last corollary results from the translation of nondeterministic automata to deterministic automata.

Remark 2. Notice that the complexity in k is independent of the representation of the k-safety-hyperproperty.

C. Equivalence of k-bad-prefix automata

From the last section we know that the language of bad-prefixes for a safety hyperproperty is superset-closed. Thus every k-safety hyperproperty is also a k'-safety hyperproperty for all $k \leq k'$. This means that $\mathbf{S}$ can be represented by different bad-prefix automata of different arities k'. In the following we show, given a k'-bad-prefix automaton and a k''-bad-prefix automaton with $k' \leq k''$, how to check whether they are bad-prefix automata for the same k-safety hyperproperty $\mathbf{S}$.

Definition 5 (Representation-equivalence of bad-prefix automata)**.** Let $A_{k'}$ be a finite automaton over $\Sigma^{k'}$, and $A_{k''}$ be a finite automaton over $\Sigma^{k''}$ for some alphabet Σ, where $k' \leq k''$. We say that $A_{k'}$ and $A_{k''}$ are *representation-equivalent*, denoted by $A_{k'} \equiv A_{k''}$ if and only if both $A_{k'}$ and $A_{k''}$ are bad-prefix automata for the same k-safety hyperproperty $\mathbf{S}$ for some $k \leq k', k''$.

An algorithm for checking equivalence of bad-prefix automata is given in the next theorem.

Theorem 4. *Let $\mathcal{A}_k$ be a deterministic finite automata over Σ^k, and $\mathcal{A}_{k'}$ a deterministic finite automaton over $\Sigma^{k'}$ for an alphabet Σ and $k, k' \in \mathbb{N}$. Checking whether $\mathcal{A}_k \equiv \mathcal{A}_{k'}$*

[2]From now on, if not stated otherwise, we use the word permutation to mean permutation with repetition.

can be done in time $O(poly(|\mathcal{A}_k| + |\mathcal{A}_{k'}|))$ and in space $2^{O(\max(\log(k),\log(k'))\cdot\max\{k,k'\})}$.

Proof. To check whether $A_k \equiv A_{k'}$ we have to check that:

1) For every representation t accepted by A_k and for every infinite extension $\widetilde{t}$ of t, there is a representation t' accepted by $\mathcal{A}_{k'}$, such that, $t' \leq \widetilde{t}$:
$\forall T \in \mathsf{unzip}(L(A_k)).\ \forall \widetilde{T} \subseteq \Sigma^\omega.$
$T \leq \widetilde{T} \rightarrow\ \exists T' \in \mathsf{unzip}(L(A_{k'})).\ T' \leq \widetilde{T}$

2) For every representation t' accepted by $A_{k'}$ and for every infinite extension $\widetilde{t}$ of t', there is a representation t accepted by A_k, such that, $t \leq \widetilde{t}$:
$\forall T' \in \mathsf{unzip}(L(A_{k'})).\ \forall \widetilde{T} \subseteq \Sigma^\omega.$
$T' \leq \widetilde{T} \rightarrow .\ \exists T \in \mathsf{unzip}(L(A_k)).\ T \leq \widetilde{T}$

W.l.o.g. assume that $k < k'$. Let $\mathcal{A}_k = (Q_k, \Sigma^k, q_{0,k}, \delta_k, F_k)$ and $\mathcal{A}_{k'} = (Q_{k'}, \Sigma^{k'}, q_{0,k'}, \delta_{k'}, F_{k'})$.

1) To check the first direction, we first transform $\mathcal{A}_{k'}$ to a tight and permutation-complete automaton $\mathcal{A}^{\mathfrak{P}}_{k'}$ using the construction in Theorem 3. To be able to compare $\mathcal{A}_k$ with $\mathcal{A}^{\mathfrak{P}}_{k'}$ we first expand the alphabet of $\mathcal{A}_k$ to $\Sigma^{k'}$ by constructing an automaton $A_k^{\uparrow k'}$ that preserves the language of $\mathcal{A}_k$ up to $\mathsf{unzip}(A_k)$. The automaton $A_k^{\uparrow k'}$ is defined by the tuple $(Q_k, \Sigma^{k'}, q_{0,k}, \delta_k^{\uparrow k'}, F_k)$, where $\delta_k^{\uparrow k'}(q, (t_1, \ldots, t_k, \ldots, t'_k)) = q'$ if and only if $t_i = t_k$ for all $k < i \leq k'$ and $\delta_k(q, (t_1, \ldots, t_k)) = q'$, otherwise there is no transition. Clearly, $\mathsf{unzip}(A_k) = \mathsf{unzip}(A_k^{\uparrow k'})$.
We build the product automaton $\mathcal{A}_\otimes$ of $A_k^{\uparrow k'}$ and $\mathcal{A}^{\mathfrak{P}}_{k'}$. If $\mathcal{A}_\otimes$ has a lasso run[3], where there is an accepting state of $A_k^{\uparrow k'}$ but no accepting states of $A^{\mathfrak{P}}_{k'}$, then condition (1) is violated and thus $A_k \not\equiv A_{k'}$. If no such run is found, then $A_k \equiv A_{k'}$.
The size of the product automaton is $|Q_k| \cdot |Q_{k'}|^{k'^{k'}}$. To check the equivalence there is no need to construct the automaton $\mathcal{A}_\otimes$ in fully. Using the same trick as in the polynomial-space model checking algorithm for LTL [11], we can guess a lasso run in $\mathcal{A}_\otimes$ of size at most $|\mathcal{A}_\otimes|$. The lasso can be guessed one position at a time and in each position on can further guess if it is the beginning of the period of the lasso. In each step we check if the guessed next position of the lasso satisfies the transition relation as given in the construction above. Finding a lasso in $\mathcal{A}_\otimes$ can thus be done in time polynomial in the sizes of A_k and $A_{k'}$ and in space exponential in k'.

2) For the other direction it does not suffice to construct the tight and permutation-complete automaton for $\mathcal{A}_k$ and check the condition (2) on the product automaton with $A_{k'}$ as we did in the last case. The reason why this construction does not work, is due to the different arities k and k'. Intuitively, A_k and $A_{k'}$ are equivalent, if for each representation accepted by $A_{k'}$, a permutation of one of its k-projections satisfies the condition (2). To this aim we construct an automaton $\mathcal{A}_k^{\# k'} = (Q^{\# k'}, \Sigma^{k'}, q_0^{\# k'}, \delta^{\# k'}, F^{\# k'})$ as follows:
Let $\varsigma_1, \ldots, \varsigma_{k'^k} : \{1, \ldots, k\} \rightarrow \{1, \ldots, k'\}$ be pairwise different functions. We call $\varsigma_1, \ldots, \varsigma_{k'^k}$ k-permuted-projection functions.
   - $Q^{\# k'} = (Q_{k,1} \times \cdots \times Q_{k,k'^k})$ where $Q_{k,i} = \{(q, i) \mid q \in Q_k\}$ for $1 \leq i \leq k'^k$. A set of states $Q_{k,i}$ resembles a copy of the automaton $\mathcal{A}$ that accepts a word σ if one of its k-permutated-projections is accepted by $\mathcal{A}$ with respect to the permuted-projection function ς_i. The initial state is defined by $q_0^{\# k'} = ((q_0, 1), \ldots, (q_0, k'^k))$.
   - A word is accepted, if one of its permuted-projections is accepted. We define the set of accepting states as $F_{\mathfrak{P}} = \{((q_1, 1), \ldots, (q_{k'^k}, k'^k)) \mid \exists i.\ q_i \in F\}$.
   - The transition relation $\delta_k^{\# k'}$ is defined as follows:
$((q_1, 1), \ldots, (q_{k'^k}, k'^k))$
$\xrightarrow{(t_1,\ldots,t_{k'})} ((q'_1, 1), \ldots, (q'_{k'^k}, k^k))$
when $(q_i \xrightarrow{(t_{\varsigma_i(1)},\ldots,t_{\varsigma_i(k)})} q'_i) \in \delta$ for $1 \leq i \leq k'^k$. For each q_i the successor state q'_i for a letter $(t_1, \ldots, t_{k'})$ is determined by the transition of its permuted-projection $(t_{\varsigma_i(1)}, \ldots, t_{\varsigma_i(k)})$ in the automaton $\mathcal{A}$.

We build the product automaton $\mathcal{A}_\otimes$ of $A_k^{\# k'}$ and $\mathcal{A}_{k'}$. If $\mathcal{A}_\otimes$ has a lasso run, where there is an accepting state of $A_{k'}$ but no accepting states of $A_k^{\# k'}$, then condition (2) is violated and thus $A_k \not\equiv A_{k'}$. If no such run is found, then $A_k \equiv A_{k'}$.
Again, To check the equivalence we can guess a lasso in $\mathcal{A}_\otimes$ that has an accepting state of $\mathcal{A}_{k'}$ but no accepting state from $A_k^{\# k'}$. Finding a lasso in $\mathcal{A}_\otimes$ can thus be done in time polynomial in the sizes of A_k and $A_{k'}$ and in space exponential in k.

□

Corollary 2. *Checking the representation-equivalence of two non-deterministic finite automata A_k and $A_{k'}$ of arity k, k' can be done in space $O(poly(|A_k| + |A_{k'}|))$ and $2^{O(\max(\log(k),\log(k'))\cdot\max\{k,k'\})}$.*

D. Minimal k-bad-prefix automata

In this section, we complete our search for a canonical representation of regular k-safety hyperproperties and prove that minimal deterministic, tight and permutation-complete bad-prefix automata provide such a representation.

Definition 6 (Minimal Bad-prefix Automaton)**.** A deterministic tight permutation-complete k-bad-prefix automaton $\mathcal{A}$ for some k'-safety hyperproperty $\mathbf{S}$ is called *minimal*, if there is no k''-bad-prefix automaton for $\mathbf{S}$, with $k'' < k$, and $\mathcal{A}$ is the minimal automaton in size for k.

[3]This is an infinite run in the automaton that can be represented by a sequence of states that reach a loop in the automaton.

Lemma 5. *Two safety hyperproperties* $\mathbf{S}$ *and* $\mathbf{S}'$ *are equivalent if and only if* $\mathsf{BadPref}(\mathbf{S}) = \mathsf{BadPref}(\mathbf{S}')$.

Proof. ($\Rightarrow$) Let $T \in \mathsf{BadPref}(\mathbf{S})$. Thus for all $T \leq T'$, it follows, that $T' \not\models \mathbf{S}$ and therefore $T' \not\models \mathbf{S}'$, by assumption. Hence, $T \in \mathsf{BadPref}(\mathbf{S}')$. The same proof holds when exchanging $\mathbf{S}$ and $\mathbf{S}'$ yielding $\mathsf{BadPref}(\mathbf{S}) = \mathsf{BadPref}(\mathbf{S}')$.

($\Leftarrow$) Let $T \notin \mathbf{S}$. Thus there exists some $T' \in \mathsf{BadPref}(\mathbf{S})$ such that $T' \leq T$. According to the assumption we know, that $T' \in \mathsf{BadPref}(\mathbf{S}')$ and thus $T \notin \mathbf{S}'$. The same proof holds when exchanging $\mathbf{S}$ and $\mathbf{S}'$ yielding $\mathbf{S} = \mathbf{S}'$. □

Theorem 6. *Minimal, tight, permutation-complete, deterministic k-bad-prefix automata are a canonical representation for regular k-safety hyperproperties.*

Proof. Let $\mathbf{S}$ and $\mathbf{S}'$ be two regular k-safety hyperproperties. We show that they are equal if and only if they have the same minimal deterministic tight permutation-complete bad-prefix automaton.

($\Rightarrow$) Let $\mathbf{S} = \mathbf{S}'$. From Lemma 5 we know that $\mathsf{BadPref}(\mathbf{S}) = \mathsf{BadPref}(\mathbf{S}')$. This means that any representation L for $\mathbf{S}$ is also a representation for $\mathbf{S}'$. We conclude that any deterministic tight permutation-complete bad-prefix automaton for $\mathbf{S}$ is also a deterministic tight permutation-complete bad-prefix automaton for $\mathbf{S}'$.

($\Leftarrow$) Let $\mathcal{A}$ be a k-bad-prefix automaton for $\mathbf{S}$ and $\mathbf{S}'$. This means that $\mathsf{BadPref}(\mathbf{S}) = \mathsf{BadPref}(\mathbf{S}')$. From Theorem 5 it follows that $\mathbf{S} \equiv \mathbf{S}'$.

It remains to show that k-bad-prefix automata are unique for a k-safety hyperproperty $\mathbf{S}$. Clearly, the minimal arity k is unique. The language of all bad-prefixes of size k is also unique for $\mathbf{S}$. Thus the set of all k-representations is unique and this language is regular by assumption. Further, from the Myhill-Nerode Theorem. it is well-known that minimal deterministic automata are a unique representation for regular languages [12] Hence, the claimed uniqueness follows. □

Based on this canonical representation, we provide, in the next section, a framework for learning automata for regular k-safety hyperproperties.

IV. Learning Automata for k-Safety Hyperproperties

We present a framework for learning *minimal tight, permutation-complete, deterministic* bad-prefix automata for some unknown k-safety hyperproperty $\mathbf{S}$ over an alphabet Σ and an unknown minimal k. The algorithm extends Dana Angluin's L^* algorithm for learning minimal deterministic finite automata from queries and counterexamples [3], to learn minimal bad-prefix automata for a minimal arity k.

A. L^*: A framework for learning regular languages

We give a high-level recap of the L^* framework as presented in Figure 2. We leave some of the technical details for the next section when explaining the extended framework for k-safety-hyperproperties.

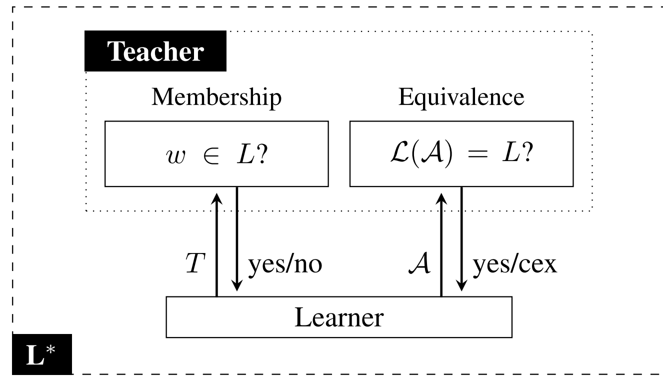

Fig. 2. L^*: A framework for learning minimal deterministic finite automata [3].

The L^* framework consists of two components, a *learner*, that learns an automaton for the unknown language, and a *teacher*, that answers questions about the language. The learner can pose two types of queries to the teacher: *membership-queries*, where the learner asks whether a word is in the target language, and *equivalence-queries*, where the learner asks whether the language of a conjectured automaton is equivalent to the target language.

The learner starts by posing membership queries for words of increasing length. The answers of the teacher are organized in a so-called *observation table*. The observation table represents a so-far constructed automaton, the accepts at least the valid words queried using membership queries. After each membership query, the learner performs two checks in the observation table: (1) a consistency check, certifying that the observation table defines a deterministic automaton; the table contains no two transitions from a state for the same letter, and (2) a closedness check, that tests that it defines a complete automaton, i.e., for each state and for each letter there is a transition from that state for this letter. If one of these checks fails, the observation table can be repaired with the appropriate extension and membership queries (We show how these checks are performed in the case of k-safety hyperproperties in the next section. For the traditional checks for regular languages we refer the reader to [3]).

If the table is both consistent and closed, then the learner can construct a deterministic automaton out of the observation table and queries the teacher on whether the conjectured automaton defines the target language. If the automaton is not equivalent to the target language, the teacher returns a counterexample. This is either a word in the language that is not accepted by the conjectured automaton, or a word that is wrongly accepted by the automaton and is not a member of the target language. The counterexample is added to the observation table, and the learning process continues with the new table.

Angluin showed that, for a *minimal adequate teacher*, i.e., a teacher that answers membership and equivalence queries, that L* terminates after a number of membership queries that is polynomial in the size of the minimal deterministic finite automaton for the target language.

B. L^*_{Hyper}: A framework for learning k-safety hyperproperties

We extend the L* framework given in Figure 2 to a new framework L^*_{Hyper} for learning minimal deterministic tight permutation-complete bad-prefix automaton for k-safety-hyperproperties. In contrast to learning minimal automata for regular properties, the bad-prefix automata learned in L^*_{Hyper} must be minimal both in the arity and in the size. The workflow of L^*_{Hyper} is given in Figure 3.

Let $\mathbf{S}$ be the unknown k-safety-hyperproperty over some alphabet Σ. The learner starts with membership queries, and fills the observation table until it is closed and consistent. Because the minimal arity k is initially unknown the learner starts by posing questions over sets of arity 1. During the learning process the alphabet changes to larger arity k', when the teacher returns a counterexample of this arity.

Assume the current arity in the learning process is k' for some $k' \leq k$. In membership queries, the learner asks whether a finite set of finite sequences of equal length $T = \{t_1, \ldots, t_{k'}\} \subseteq \Sigma^n$ (given by some representation) for some $n \in \mathbb{N}$ is a bad-prefix for $\mathbf{S}$. The answers of the teacher are organized in an observation table $\mathcal{O} = (S, E, \Delta)$ where: $S \subseteq (\Sigma^{k'})^*$ is a non-empty finite prefix-closed set of *accessing sequences*, $E \subseteq (\Sigma^{k'})^*$ is a non-empty finite suffix-closed set of *separating sequences*, and $\Delta : (S \cup S \cdot \Sigma) \cdot E \to \{0, 1\}$ a mapping defined as $\Delta(s \cdot e) = 1$ if and only if $s \cdot e$ is a representation of a bad-prefix for $\mathbf{S}$.

Consider the observation table given in Figure 4. The set S includes the words $\epsilon, \neg a, a, a \cdot \neg a$ in the first four rows of the table. The set E includes the words $\epsilon, \neg a$ defining the columns. The words in the remaining columns define the set $S \cdot \Sigma$ (as we will see later, these rows are necessary for the closedness and consistency checks). The value of an entry in the table is 1 if the word $s \cdot e$, where s is a word of the row and e the word of the column, is a representation of a bad-prefix for the hyperproperty given by the formula $\forall \pi, \pi'.\ a_\pi \wedge \square(a_\pi \leftrightarrow a_{\pi'})$. Otherwise the value of the entry is 0.

For $t \in S \cdot \Sigma$ we denote by $\mathsf{row}(t)$ a finite function from E to $\{0, 1\}$ defined by $\mathsf{row}(t)(e) = \Delta(t \cdot e)$. An observation table $\mathcal{O} = (S, E, T)$ is called closed if for all $t \in S \cdot \Sigma$ there exists $s \in S$ with $\mathsf{row}(t) = \mathsf{row}(s)$. The table $\mathcal{O}$ is called consistent, if for all $t, t' \in S$ with an equal function $\mathsf{row}(t) = \mathsf{row}(t') \Rightarrow \mathsf{row}(t \cdot e) = \mathsf{row}(t' \cdot e)$ for all $e \in \Sigma$. We define $\mathsf{row}(S) = \{\mathsf{row}(s) \mid s \in S\}$. Closedness guarantees that every transition is defined, i.e., for each state $q \in Q$ and label $a \in \Sigma$, $\delta(q, a) \in Q$, and consistency guarantees that $\mathcal{A}$ is deterministic.

For a closed and consistent observation table $\mathcal{O}$ over an arity k' we can construct an DFA $\mathcal{A} = (Q, \Sigma^{k'}, q_0, F, \delta)$ that accepts all the k'-bad-prefixes that have been confirmed by the teacher so far. We define $Q = \{\mathsf{row}(s) \mid s \in S\}$, $q_0 = \mathsf{row}(\epsilon)$, $F = \{\mathsf{row}(s) \mid s \in S \text{ and } \Delta(s) = 1\}$ and $\delta(\mathsf{row}(s), a) = \mathsf{row}(s \cdot a)$ for all $s \in S$ and $a \in \Sigma$. The table in Figure 4 is closed and consistent, and defines the automaton given to its right. For $k' \in \mathbb{N}$, we call an automaton $\mathcal{A}$ over Σ^k consistent with an observation table $\mathcal{O} = (S, E, \Delta)$ over Σ^k if for all $s \in (S \cup S \cdot \Sigma^k), e \in E\ \Delta(s, e) = 1 \Leftrightarrow s \cdot e \in L(\mathcal{A})$.

To check whether the learned automaton $\mathcal{A}$ is a k'-bad-prefix automaton for $\mathbf{S}$, the learner poses an equivalence query to the teacher. In equivalence queries, the teacher answers whether the proposed automaton $\mathcal{A} = (Q, \Sigma^{k'}, q_0, F, \delta)$ is a k'-bad-prefix automaton for $\mathbf{S}$. In case $\mathcal{A}$ is not, the teacher provides a counterexample.

C. Handling counterexamples of equivalence queries

If the equivalence test fails, then the teacher returns a counterexample. A counterexample is either a bad prefix for which no representation is accepted by the conjectured automaton, or a representation accepted by the automaton that is no representation of a bad prefix for $\mathbf{S}$.

Handling bad prefixes depends on their arity. We distinguish between two types of counterexamples with respect to the current considered arity k', namely, counterexamples with arity $k'' \leq k'$ and counterexamples with arity $k'' > k'$.

If the counterexample has arity $k'' \leq k$, the counterexample is treated as for the traditional L* learner, by extending the table with a representation of this counterexample and querying all its prefixes.

If the counterexample has arity $k'' > k'$ then the arity of the target automaton is increased to k''. The sets S and E are extended to sequences over the alphabet $\Sigma^{k''}$ by replacing every element $t = v_0 \ldots v_n \in (\Sigma^{k'})^*$ in S and E by $t' = v'_0 \ldots v'_n \in (\Sigma^{k''})^*$, such that, for all $0 \leq i \leq n$ and $0 \leq j \leq k'$, $v_i[j] = v'_i[j]$ and for $k' < j \leq k''$, $v_i[k'] = v'_i[j]$. Notice that the size of the table increases by the number of prefixes of the counterexample.

Consider again our example in Figure 4. The conjecture automaton is not a bad-prefix automaton for the language $\forall \pi, \pi'.\ a_\pi \wedge \square(a_\pi \leftrightarrow a_{\pi'})$. A counterexamples of arity 2 is given by the set $C = \{\{a \cdot \neg a\}, \{a \cdot a\}\}$. The observation table must be extended to Σ^2. Having extended the observation table, we must add a representation of the counterexample C to the set of accessing sequences S. C is a bad prefix, which can be verified using a membership query. As no 2-representation of C are in the target language. We choose the 2-representation $(a, a)(\neg a, a)$ and add it to the set of accessing sequences resolves the current counterexample. The new observation table is closed and consistent, and it represents a 2-bad-prefix automaton that represents the target language. The final table and its automaton are depicted in Figure 5.

In some case, the resulting automaton might pass the equivalence check but is not necessarily permutation complete. If the goal is to construct a permutation-complete automaton, we additionally check the automaton for permutation-completeness.

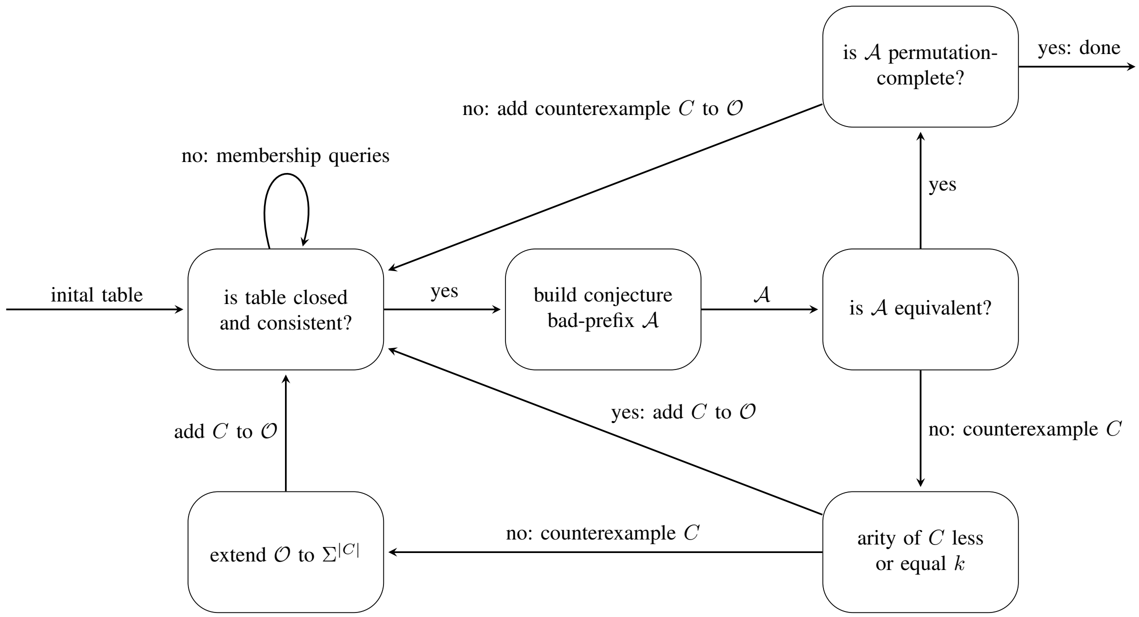

Fig. 3. L^*_{Hyper}: A framework for learning k-safety-hyperproperties.

	ϵ	$\neg a$
ϵ	0	1
$\epsilon \cdot \neg a$	1	1
$\epsilon \cdot a$	0	0
$a \cdot \neg a$	0	0
$\neg a \cdot a$	1	1
$\neg a \cdot \neg a$	1	1
$a \cdot a$	0	0
$a \cdot \neg a \cdot a$	0	0
$a \cdot \neg a \cdot \neg a$	0	0

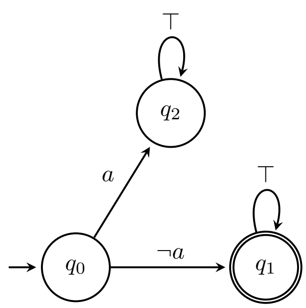

Fig. 4. The observation table (on the left) for the 4^{th} iteration in the learning process for the language $\forall \pi, \pi'.\, a_\pi \wedge \Box(a_\pi \leftrightarrow a_{\pi'})$, and the corresponding DFA (on the right).

D. Termination of L^*_{Hyper}

As for L^*, to guarantee that our learning framework terminates with a minimal automaton, it must rely on a minimal-adequate teacher. For L^*_{Hyper} we define such a teacher as follows.

Definition 7 (Minimal-Adequate Teacher)**.** A teacher is called minimal-adequate, if the counterexamples provided are of minimal length and every counterexample has an arity of at most the minimal target-arity.

For a minimal adequate teacher we show that for an unknown k-safety hyperproperty $\mathbf{S}$, L^*_{Hyper} terminates after at most n equivalence queries, where n is the size of the minimal deterministic tight permutation-complete k-bad-prefix automaton for $\mathbf{S}$.

Let $\mathcal{O} = (S, E, \Delta)$ be the observation table. Following the ideas in the termination proof for L^*, we need to show that

1) the set $\mathsf{row}(S)$ does not grow beyond n.
2) with each counterexample, the size of the set $\mathsf{row}(S)$ must be strictly monotonically increasing.

In the following lemma we give a proof for condition 1). With Lemmas 8, 9, 10 and 11, we show that L^*_{Hyper} also satisfies condition 2).

Lemma 7. *Let $\mathbf{S}$ be a regular k-safety hyperproperty, let $\mathcal{O} = (S, E, \Delta)$ be an observation table, and let n be the size of the minimal k-bad-prefix automaton for $\mathbf{S}$. Then, the size of $\mathsf{row}(S)$ is bounded by n in any iteration of L^*_{Hyper}.*

Proof. The proof follows from the fact that, for any closed and consistent observation table, an automaton that is consistent with this table has at least n states [3]. $\square$

To prove condition 2), we need to show that after every counterexample, the set $\mathsf{row}(S)$ must increase by at least one. This requires us to show that

- resolving consistency and closedness of observation tables increases the size of $\mathsf{row}(S)$ by at least one. (Lemma 8 and Lemma 9)
- adding a counterexample makes the observation table $\mathcal{O}$ inconsistent or not closed (Lemma 10)
- extending the observation table from an arity k' to $k'' > k'$ preserves the size of $\mathsf{row}(S)$ (Lemma 11)

We define a witness of inconsistency in $\mathcal{O}$ as a triple $(s, t, a) \in S \times S \times \Sigma$ such that $\mathsf{row}(s) = \mathsf{row}(t)$ and $\mathsf{row}(s \cdot a) \neq \mathsf{row}(t \cdot a)$ and a witness of non-closedness in $\mathcal{O}$ as a word $w \in S \cdot \Sigma$ such that $\mathsf{row}(w) \neq \mathsf{row}(s)$ for all $s \in S$.

	ϵ	$(\neg a, \neg a)$
ϵ	0	1
$\epsilon \cdot (\neg a, \neg a)$	1	1
$\epsilon \cdot (a, a)$	0	0
$(a, a) \cdot (\neg a, \neg a)$	0	0
$(a, a) \cdot (\neg a, a)$	1	1
$(a, \neg a)$	1	1
$(\neg a, a)$	1	1
$(\neg a, \neg a) \cdot (*)$	1	1
$(a, a) \cdot (a, a)$	0	0
$(a, a) \cdot (\neg a, a)$	1	1
$(a, a) \cdot (\neg a, \neg a) \cdot (a, a)$	0	0
$(a, a) \cdot (\neg a, \neg a) \cdot (\neg a, a)$	1	1
$(a, a) \cdot (\neg a, \neg a) \cdot (a, \neg a)$	1	1
$(a, a) \cdot (\neg a, \neg a) \cdot (\neg a, \neg a)$	0	0
$(a, a) \cdot (\neg a, a) \cdot (*)$	1	1

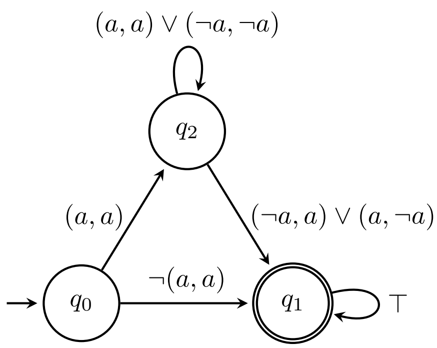

Fig. 5. Final observation table for learning $\varphi = \forall \pi, \pi'. \, a_\pi \wedge \square(a_\pi \leftrightarrow a_{\pi'})$, and 2-bad-prefix automaton for φ. We employ the notation $x \cdot (*)$ to denote all extensions of x.

The proofs for the following three lemmas are given in Dana Angluin's termination proof for L^* [3].

Lemma 8. *Let $\mathcal{O} = (S, E, \Delta)$ be an inconsistent observation table over Σ with a witness (s, t, a). Resolving this witness increases the size of $\mathsf{row}(S)$.*

Lemma 9. *Let $\mathcal{O} = (S, E, \Delta)$ be an observation table over Σ that is not closed with a witness $w \in S \cdot \Sigma$. Resolving this witness increases the size of $\mathsf{row}(S)$.*

The next lemma states that amending an observation table with a counterexample results in an inconsistent observation table.

Lemma 10. *Let $\mathcal{O} = (S, E, \Delta)$ be a consistent observation table over an alphabet Σ. Let $w \in \Sigma^*$, be a counterexample resulting from an equivalence check or from a check of permutation-completeness. Then the resulting observation table $\mathcal{O}' = (S', E', \Delta')$ that results from $\mathcal{O}$ by amending it with w is either inconsistent or not closed.*

Obtaining a counterexample containing k' many traces, which can not be represented by the learner's current arity k_L, i.e., $k' > k_L$, the observation table needs to be extended to k' before adding the counterexample. We want to achieve this extension without losing information obtained by earlier queries, i.e., without changing the size of $\mathsf{row}(S)$. Therefore, we extend each representation in $S \cup E$ to a representation in $\Sigma^{k'}$ by repeating its last position. After this extension every representation still represents the same set of traces, since we only repeat one of the traces in trace set. The procedure of extending an observation table from arity k_L to a larger arity k is given in Algorithm 1. The function extend is defined as follows: For a tuple $(t_1, \ldots, t_{k_L}) \in \Sigma^{k_L}$, $\mathsf{extend}(t_1, \ldots, t_{k_L}, k') = (t_1, \ldots, t_{k_L}, \ldots, t_{k'})$, where $t_i = t_{k_L}$ for all $k_L < i \leq k'$.

Algorithm 1 EXTEND

Input Observation Table $\mathcal{O} = (S, E, \Delta)$ over Σ^{k_L}, $k' > k_L$
Output Observation table $\mathcal{O}'$ over $\Sigma^{k'}$

```
O' = (S', E', Δ') = ({ε}, {ε}, {((ε, ε), Δ(ε, ε))})
for s ∈ S do
    S' = S' ∪ {extend(s, k')}
end for
for e ∈ E do
    E' = E' ∪ {extend(e, k')}
end for
fill Δ' for (S' ∪ S' · Σ^{k'}) · E' using Membership queries
return O'.
```

Lemma 11. *Let $\mathcal{O} = (S, E, \Delta)$ be an observation table over an alphabet $\Sigma^{k'}$ for some $k' \in \mathbb{N}$. Let $\mathcal{O}' = (S', E', \Delta')$ be the observation table obtained by applying* EXTEND *(Algorithm 1) to $\mathcal{O}$ for some $k'' > k'$. Then, $|\mathsf{row}(S)| = |\mathsf{row}(S')|$.*

Proof. Prove by contradiction. Assume $|\mathsf{row}(S)| \neq |\mathsf{row}(S')|$. We only treat one direction the other direction can be proven equivalently. Let $|\mathsf{row}(S)| > |\mathsf{row}(S')|$, i.e., there exist $s, t \in S$ and $e \in E$ such that the following (in-)equalities hold

$$\Delta(s, e) \neq \Delta(t, e)$$
$$\Delta(\mathsf{extend}_{k''}(s), \mathsf{extend}_{k''}(e)) = \Delta(\mathsf{extend}_{k''}(t), \mathsf{extend}_{k''}(e)).$$

W.l.o.g., let $\Delta(\mathsf{extend}_{k''}(t), \mathsf{extend}_{k''}(e)) \neq \Delta(t, e)$, i.e., the result of a membership queries for $\mathsf{extend}_{k''}(t \cdot e)$ and $t \cdot e$ differ. Since $\mathsf{extend}_{k''}$ does not alter the represented set of words, such words s, t, and e cannot exist. A contradiction to our assumption. $\square$

Using the lemmas above, we are now able to prove the termination of L^*_{Hyper}.

Theorem 12. *L^*_{Hyper} terminates after at most n equivalence queries, where n is the size of the minimal deterministic tight permutation-complete k-bad-prefix automaton for* **S**.

E. Complexity of L^*_{Hyper}

In the last section we showed that for a minimal-adequate teacher, L^*_{Hyper} terminates after at most n equivalence queries, where n is the size of the minimal bad-prefix automaton for the target k-safety hyperproperty $\mathbf{S}$. Preceding every equivalence query, the learner checks the consistency and closedness of the observation table, and after the equivalence check the learner may need to check the permutation-completeness of the conjectured automaton or extend the observation table to a higher arity, and then amend the observation table with a new counterexample.

After each operation, the number of rows in the observation table is increased by at least one, but cannot increase beyond n, as we have shown in Lemma 7. This means that we can perform at most n many of these operations.

For L^*, Angluin showed that the learning algorithm is polynomial in n and in the length of the largest returned counterexample. This complexity also holds for L^*_{Hyper}. The runtime of L^*_{Hyper} also depends on the goal arity k. The runtime complexity in k can be derived by studying the runtime of the procedures for checking permutation-completeness and extending the observation table

Lemma 13. *[3] Checking the observation table for consistency and closedness can be done in time polynomial in the size of the observation table.*

The complexity of Algorithm 1 is given in the following lemma.

Lemma 14. *Extending an observation table over arity k to an equivalent observation table of arity $k' > k$ can be done in time polynomial in the size of the observation table and exponential in k'.*

Proof. This follows from the runtime of the procedure EXTEND, where we have to perform $|S' \cup S' \cdot \Sigma^{k'}| \cdot E'$ membership queries. As $|S'| = |S|$ and $|E'| = |E|$, the runtime of EXTEND is polynomial in S and E and exponential in k'. □

A k-bad-prefix automaton, that is not permutation-complete, accepts one representation of every bad prefix but not every representation of it. Algorithm 2 provides a procedure for deciding permutation-completeness. For $\mathcal{A} = (Q, \Sigma^{k'}, q_0, F, \delta)$ a k-bad-prefix automaton and $\varsigma : \{1, \dots, k'\} \to \{1, \dots, k'\}$, we define $A^\varsigma = (Q, \Sigma^{k'}, q_0, F, \delta^\varsigma)$ where $\delta^\varsigma(s, (a_1, \dots, a_{k'}) = \delta(s, (a_{\varsigma(1)}, \dots, a_{\varsigma(k')}))$. If $\mathcal{A}$ is permutation-complete, then $L(\mathcal{A}^\varsigma) \subseteq L(\mathcal{A})$ for every ς.

Lemma 15. *Checking whether a k-bad-prefix automaton $\mathcal{A}$ is permutation-complete for a k-safety hyperproperty $\mathbf{S}$ can be done in time exponential in k and polynomial in $\mathcal{A}$.*

Building on Lemmas 13, 14 and 15 the overall complexity of the algorithm L^*_{Hyper} is given by the following theorem.

Theorem 16 (Learning k-Bad-Prefix Automata)**.** *Provided a minimal-adequate teacher, L^*_{Hyper} learns a minimal, deterministic, tight and permutation-complete bad-prefix automaton $\mathcal{A}$ for a k-safety property $\mathbf{S}$ over Σ in time polynomial in n and m, where n is the size $\mathcal{A}$, m is the length of the longest counterexample provided, and in time exponential in k.*

Algorithm 2 ISCOMPLETE

Input k-bad-prefix automaton $\mathcal{A}$ over Σ^k
Output $\mathcal{A}$ permutation-complete, or
  $w \in \Sigma^k$ s.t. $w \notin L(\mathcal{A})$

for $\varsigma \in \{1, \dots, k\}^{\{1, \dots, k\}}$ **do**
  if $L(\mathcal{A}^\varsigma) \not\subseteq L(\mathcal{A})$ **then**
    return $w \in (L(\mathcal{A}^\varsigma - \mathcal{A}))$
  end if
end for
return Closed

V. LEARNING AUTOMATA FOR HYPERLTL

The next natural step is to instantiate the L^*_{Hyper} framework to learn automata for HYPERLTL. We dedicate this section to instantiating the L^*_{Hyper} framework for learning minimal permutation-complete automata for universally-safe HYPERLTL formulas.

Definition 8 (Universally-safe HYPERLTL formulas)**.** A universally-safe HYPERLTL formula φ is of the form $\forall \pi_1 \dots \forall \pi_k . \psi$, where ψ is a safety LTL formula.

In the following we show the complexity of deciding membership and equivalence queries for universally-safe HYPERLTL formulas.

A. Deciding membership queries

Theorem 17. *Let T be a set of traces, with each trace being of length n, and let a universally-safe HYPERLTL formula $\varphi = \forall \pi_1 \dots \forall \pi_{k_T} .\ \psi$. The problem of deciding whether T is a bad prefix for φ can be solved in time polynomial in n and space polynomial in $|\psi|$ and $k_T \cdot \log(|T|)$.*

Proof. Deciding whether $t \in \Sigma$ is a bad prefix for a safety LTL formula φ can be done in space polynomial in $|\varphi|$ and time polynomial in $|t|$ by guessing whether t allows an accepting run in the Büchi automaton for ψ. If no such run is found, then t is a bad prefix for ψ.

For a set T of traces of length n we need to check whether one of the representations is a bad prefix for φ. There are at most $|T|^{k_T}$ many different representations for T that can be encoded by $\log(|T|^{k_T}) = O(k_T \cdot \log(|T|))$ bits. Checking whether T is a bad prefix for ψ can be done in space polynomial in $k_T \cdot \log(|T|)$ and in ψ, by guessing the representation and applying the bad prefix check for ψ. □

For the purpose of completeness, we present a second algorithm solving membership queries symbolically. To this end, we employ the decidability results of HYPERLTL [13]. In practice, the second algorithm is expected to outperform the first one due to its dependence on SAT solving and the efficiency of state-of-the-art SAT solvers.

Theorem 18. *Let $T = \{t_1, \ldots, t_n\}$ be a set of finite traces and let a universally-safe* HyperLTL *formula $\varphi = \forall\pi_1 \ldots \forall\pi_{k_T}.\ \psi$. T is a bad prefix of φ if and only if the following* HyperLTL *formula is unsatisfiable:*

$$\varphi' := \exists\pi'_1 \ldots \exists\pi'_n \\ \forall\pi_1 \ldots \forall\pi_{k_T}.\ \pi'_1 \geq t_1 \wedge \cdots \wedge \pi'_n \geq t_n \wedge \psi$$

Proof. ($\Rightarrow$) Let $T = \{t_1, \ldots t_n\} \subseteq \Sigma^*$ be a bad prefix of φ. Thus, for all $T' \subseteq \Sigma^\omega$ with $T \leq T'$ it holds: $T' \not\models \varphi$. Therefore, no traces $\pi'_1, \ldots, \pi'_n$ exist that satisfy ψ and φ' is unsatisfiable.

($\Leftarrow$) Let $T = \{t_1, \ldots t_n\} \subseteq \Sigma^*$ be a set of traces and let φ' be unsatisfiable. We distinguish the following two cases:

1) φ is unsatisfiable:
   Thus, no set of infinite traces can satisfy φ and T is, like every other non-empty set of traces, a bad prefix of φ.
2) φ is satisfiable:
   Then, the conjunction of $\pi'_1 \geq t_1 \wedge \cdots \wedge \pi'_n \geq t_n$ and ψ is unsatisfiable in the context of the given quantifiers. Thus, there does not exists a set of traces $T' \subseteq \Sigma^\omega$ having n traces that extend $t_1, \ldots, t_n$ and $T' \models \varphi$. Therefore, T satisfies the definition of a bad prefix of φ.

□

According to the results in [13] and since φ' is in the bounded $\exists^*\forall^*$ fragment of HyperLTL, Theorem 18 grants us an algorithm deciding membership queries in space exponential in $|\varphi'| = n \cdot m \cdot |\Sigma| + k_t + |\varphi|$ where m is the length of the traces in T.

B. Deciding equivalence queries

In this section, we focus on the resolution of equivalence queries. Given an automaton $\mathcal{A}$ and a universally-safe HyperLTL formula φ, to check whether $\mathcal{A}$ is a bad-prefix automaton for φ we need to check that:

1) Every word accepted by $\mathcal{A}$ is a representation of a bad prefix of φ.
2) The automaton $\mathcal{A}$ accepts a representation of a bad prefix, for every set of traces violating φ.

The next theorem give an algorithms for deciding problem (1). Problem (2) is solve in the theorem that follows.

Theorem 19. *Given a universally-safe* HyperLTL *formula $\varphi = \forall\pi_1 \ldots \forall\pi_{k_T}.\ \psi$ and a deterministic automaton $\mathcal{A} = (Q, \Sigma^{k_L}, q_0, F, \delta)$. Checking whether every word w accepted by $\mathcal{A}$ is a k_L-representation of a bad prefix of φ can be done in time polynomial in $|\mathcal{A}|$, exponential in $|\psi|$ and doubly exponential in k_T.*

Proof. We transform φ into a HyperLTL formula $\varphi' = \forall\pi_1 \ldots \forall\pi_{k_T}.\ \psi'(\pi_1, \ldots, \pi_{k_T})$ where

$$\psi'(\pi_1, \ldots, \pi_{k_T}) = \bigwedge_{\varsigma:\{1,\ldots,k_T\}\to\{1,\ldots,k_T\}} \psi(\pi_{\varsigma(1)}, \ldots, \pi_{\varsigma(k_T)})$$

Note that the trace property described by ψ' is permutation-complete with respect to Σ^{k_T}, i.e., for all $t \in (\Sigma^{k_T})^\omega$, it holds $t \models_{\mathsf{LTL}} \psi' \Leftrightarrow \emptyset \models_{\mathsf{unzip}(t)} \varphi$ where the LTL semantic is adjusted such that a_{π_i} holds if a holds in the i-th component. The size of φ' is exponential in k_T and we can construct a nondeterministic safety automaton $\mathcal{N}_{\varphi'}$ accepting all infinite sequences that represent a set T of at most k_T traces such that $T \models \varphi$. The size of $\mathcal{N}_{\varphi'}$ is exponential in the size of ψ' [14].

$\mathcal{A}$ and $\mathcal{N}_{\varphi'}$ are of different arities, i.e., we need to extend $\mathcal{A}$ to k_T Let $\mathcal{A}' = (Q', q'_0, \Sigma^k, F', \delta')$ be an automaton, we define the extension $\mathsf{extend}(\mathcal{A}, k')$ of $\mathcal{A}'$ onto $\Sigma^{k'}$ for $k' > k$ as follows: $\mathsf{extend}(\mathcal{A}, k') = (Q', q'_0, \Sigma^{k'}, F', \delta'')$ with $\delta'(q, \mathsf{extend}(s, k')) = q'$ iff $\delta(q, s) = q'$ for $q, q' \in Q'$ and $s \in \Sigma^k$. In order to check whether $\mathcal{A}$ accepts any sequence that does not represent a bad prefix, we then construct the nondeterministic product automaton of $\mathsf{extend}(\mathcal{A}, k_T)$ and $\mathcal{N}_{\varphi'}$, and check the emptiness of the product automaton. □

In general, for a HyperLTL formula $\varphi = \forall\pi_1 \ldots \forall\pi_{k_T}.\ \psi$ we can make the assumption that L^*_{Hyper} never constructs an automaton for an arity larger than k_T. With this assumption, problem (2) can be solved by checking whether for every set of traces T violating φ, if $\mathcal{A}$ accepts a representation of a k_L-bad prefix of T.

Theorem 20. *Let $\varphi = \forall\pi_1 \ldots \forall\pi_{k_T}.\ \psi$ be a universally-safe* HyperLTL *formula and let $\mathcal{A}_L$ be a deterministic bad-prefix automaton over Σ^{k_L} for some alphabet Σ. The problem of deciding whether $\mathcal{A}_L$ recognizes a k_L-representation of a bad prefix for every $T \notin \mathcal{L}(\varphi)$ can be solved in time polynomial in $|\mathcal{A}_L|$, exponential in $|\psi|$, and space exponential in k_T.*

Proof. For φ, we can construct a fine, nondeterministic k-bad-prefix automaton $\mathcal{N}_\varphi = (Q_\varphi, \Sigma^{k_T}, q_{0,\varphi}, F_\varphi, \Delta_\varphi)$ for ψ on the adapted alphabet over Σ^{k_L} in time exponential in $|\psi|$ [10]. In Theorem 4, we provided an equivalence check which solves the above problem with respect to two deterministic k-bad-prefix automata. However, since only one of the two automata is complemented only $\mathcal{A}_L$ has to be deterministic whereas $\mathcal{N}_\varphi$ can be nondeterministic as well. Then the complexity bound follows from Theorem 4.

In case the equivalence does not hold, a counterexample of minimal length can easily be produced in the size of the cross-product automaton, similar to Theorem 19. This counterexample can be reduced to minimal size by repeated deleting of traces as long as the answer to membership with it do not change. This takes linear time in k_T in addition to the complexity of membership queries. □

VI. Learnability of Hyperproperties

We conclude our study with an investigation of the landscape of learnable hyperproperties. We point out results regarding the theoretical boundaries of learning trace properties, as well as, hyperproperties. These findings do not directly affect L^*_{Hyper}, proposed in the last section, but rather affect possible future extensions.

We start with an elementary result regarding the class of k-safety hyperproperties. This result even holds for 1-safety hyperproperties, i.e., safety trace properties.

Theorem 21. *Answering membership queries for safety languages is undecidable in general.*

The previous theorem can be proved by a reduction to the halting problem. Therefore, one constructs an undecidable safety language. Membership for this language is undecidable. Thus, the theorem follows. Since the class of 1-safety hyperproperties is a equal to the set of safety languages [1], the subsequent corollary immediately follows from Theorem 21.

Corollary 3. *Membership queries for k-safety hyperproperties are, in general, undecidable.*

Despite the undecidability result, there are many important classes of safety languages for which membership queries are decidable. One important example is the class of safety languages defined in LTL. Here the problem can be decided by constructing the conjugation between the LTL formula at hand and the given prefix, expressed in LTL. Afterwards checking the satisfiability of the obtained formula decides the membership [15]. In addition, equivalence queries for the safety fragment of LTL can be decided: Let $\mathcal{A}$ be a deterministic bad-prefix automaton and ψ an LTL formula. We can construct a deterministic bad-prefix automaton $\mathcal{A}_\psi$ for ψ accepting all bad-prefixes of ψ [9]. $\mathcal{A}$ is a bad-prefix automaton for ψ if $\mathcal{A}$ and $\mathcal{A}_\psi$ accept the same language, which can be easily checked. Hence, the decidability of queries of safety languages defined in LTL is completely solved. Thus, automated approaches using the learning of safety properties in LTL are applicable, in general.

Looking at our learning framework for hypersafety properties expressed in HYPERLTL, it becomes of interest to ask, to what extend queries can be decided? Regarding equivalence queries for HYPERLTL, we obtain the following negative result. The proof is independent of the representation model for HYPERLTL.

Proposition 22. *Equivalence queries are undecidable for the full class of hyperproperties expressed in* HYPERLTL*.*

Proof. Assume that we can decide membership and equivalence queries for φ. Then, we can answer the satisfiability of φ by asking whether $\varphi \equiv \mathsf{False}$. This contradicts the undecidability of HyperLTL-SAT [13]. $\square$

Thus, it becomes important to consider subclasses of HYPERLTL, like the universal-safe HYPERLTL fragment. In order to use such subclasses in automated learning environments, it is necessary to decide whether a HYPERLTL formulas is an element of this subclass. For LTL, the corresponding question—whether a formula expresses a safety property—is decidable [16]. It is even possible to decompose the formula into its pure safety and liveness parts. Clarkson and Schneider showed that such a separation into a pure hypersafety and a pure hyperliveness part always exists for hyperproperties [1]. Thus, it looks promising that such a separation can be computed. In the following theorem we reduce the satisfiability of HYPERLTL to deciding whether a given HYPERLTL formula describes a k-hypersafety property. Thus, according to the undecidability result for the satisfiability of HYPERLTL, our problem is undecidable [13].

Theorem 23. *Whether a* HYPERLTL *formula φ expresses a k-safety hyperproperty is undecidable in general.*

Proof. We start by observing that every unsatisfiable HYPERLTL formula describes the hyperproperty $\mathcal{L}(\varphi) = \emptyset$ and $\mathcal{L}(\varphi)$ is k-safety for all $k \in \mathbb{N}$.

Let $\varphi = Q_1\pi_1 .\dots Q_n\pi_n.\ \psi$ be HYPERLTL formula over the alphabet Σ. We construct the formula $\varphi' = Q_1\pi_1 .\dots Q_n\pi_n.\ \psi \wedge \Diamond\, a_{\pi_1}$ over the alphabet $\Sigma' = \Sigma \,\dot\cup\, \{a\}$, i.e., $a \notin \Sigma$. We will prove that φ is unsatisfiable if and only if $\mathcal{L}(\varphi')$ expresses a k-safety hyperproperty. Therefore we distinguish the following two cases:

- φ is unsatisfiable: φ' is unsatisfiable as well. Thus, φ' is k-safety for any $k \in \mathbb{N}$.
- φ is satisfiable: Let $T \subseteq \Sigma^\omega$ be a set of traces such that $\emptyset \models_T \varphi$. Construct $T' \subseteq \Sigma'^\omega$ $T' = \{t' \mid \exists t \in T.\ t' \equiv_\Sigma t \wedge \forall i \in \mathbb{N}.\ t'[i] \models a\}$. Since $T \models \varphi$ and $T' \models \forall\pi.a_\pi$ it follows: $T' \models \varphi'$. Thus, φ' is satisfiable and $\mathcal{L}(\varphi')$ is not a k-safety hyperproperty since the extra conjunct enforces every or some trace, depending on Q_1, to satisfy eventually a, a liveness requirement.

$\square$

The same proof provides that checking if a HYPERLTL formula describes a safety hyperproperty is undecidable.

In contrast to the negative decidability results, we give a decision procedure for deciding whether a formula is k-safe for the important $\forall$-fragment of HYPERLTL. This, is a superset of the universal-safe formulas.

Theorem 24. *Let $\varphi = \forall\pi_1 \dots \forall\pi_k.\ \psi$ be a* HYPERLTL *formula. It is decidable if $L(\varphi)$ is a k-safety hyperproperty, in space polynomial in $|\psi|$ and space exponential in k.*

Proof. First, note that for HYPERLTL formulas in the $\forall$-fragment, the described hyperproperty is safe if and only if it is k-safe. By the definition of k-safety hyperproperties, $L(\varphi)$ is k-safe if and only if

$$\forall T.\ \left[T \not\models \varphi \Rightarrow \left[\exists T' \leq T.\ |T'| \leq k \wedge \forall \tilde{T}' \geq T'.\ \tilde{T}' \not\models \varphi\right]\right]$$

Our first claim is: φ is safe for all sets of size at most k if and only if φ is safe.

($\Leftarrow$) Let T be some set of traces that violates φ. There exists a set $T' \leq T$ with $|T'| \leq k$ and thus φ must reject every set of traces extending T' including those of size k. Hence, φ is safe for traces of size at most k.

($\Rightarrow$) Let φ be k-safe for all sets of traces of size at most k. Then, a set of traces violating φ must have a bad-prefix T' of size at most k. Further, every set of traces extending T has this prefix T' and thus φ is safety.

Further, we claim that φ is safe for a set of traces T of size i if and only if the following formula is safe with respect to LTL semantics:

$$\psi_i(\pi_1, \ldots, \pi_i) := \bigwedge_{\varsigma:\{1,\ldots,k\}\to\{1,\ldots,i\}} \psi(\pi_{\varsigma(1)}, \ldots, \pi_{\varsigma(k)})$$

We rephrase the foregoing claim as follows: for all $m \leq k$ and $T = \{t_1, \ldots, t_m\}$:

$$T \models \varphi \text{ if and only if } (t_1, \ldots, t_m) \models \psi_m$$

($\Rightarrow$) Following the semantic of HyperLTL $T \models \varphi$ implies: For all Π with $\mathsf{Traces}(\Pi) \subseteq T$. $\Pi \models_\emptyset \psi$. Thus, $(t_1, \ldots, t_m) \models \psi_m(\pi_1, \ldots, \pi_m)$.

($\Leftarrow$) $(t_1, \ldots, t_m) \models \psi_m$ implies that for all $\varsigma : \{1, \ldots, k\} \to \{1, \ldots, m\}$: $(t_{\varsigma(1)}, \ldots, t_{\varsigma(k)}) \models \psi(\pi_1, \ldots, \pi_k)$. Hence, for all trace assignments Π with $\mathsf{Traces}(\Pi) \subseteq T$. $\Pi \models_\emptyset \psi$. And thus by definition $T \models \varphi$.

Hence, it follows φ is safe for trace sets of size up to m if and only if ψ_m is safe under the semantics of LTL. Therefore, deciding whether ψ_k is reduced to checking LTL safety, which can be computed in space polynomial in $|\psi_k|$ [16]. The result coincides with whether φ is a k-safety hyperproperty. □

In the last two sections, we have shown that for HyperLTL formulas in the universal-safe fragment, we can both decide if a given formula belongs to the fragment and decide membership queries. For the safe hyperproperties in the $\forall^*$-fragment of HyperLTL, we can decide whether a formulas belongs to the fragment, but the decidability of the queries is open. In general, equivalence queries are undecidable.

VII. Related Work

Most verification techniques for k-safety hyperproperties are based on self-composition [17], [18]. Self-composition enables the use of standard techniques for information flow policy verification, such as program logics and model checking. Automata-theoretic approaches for the verification of information-flow policies include model checking algorithms, such as for HyperLTL [14], for Mantel's Basic Security Predicates BSPs [19], and for epistemic logics [20]. There are also related algorithms for synthesis [21], satisfiability [13], and monitoring [22]–[24]. Typically, these approaches rely on automata constructions for trace properties. For example, in automata-based HyperLTL monitoring [23], an automaton is constructed for the underlying LTL formula over an indexed set of atomic propositions. During monitoring, this automaton is then applied to multiple combinations of the observed traces by instantiating the indices in all necessary permutations. None of these approaches define a canonical representation for hyperproperties.

There is a rich body of work on learning from examples, ranging from learning automata [3], [25], [26] to approaches for specification mining on systems [6]–[8]. Further algorithms have been presented for specification mining of information-flow polices [27], [28]. Approaches for learning specifications for monitoring malicious behavior were presented in [29]. Our learning approach provides, to the best of our knowledge, the first general framework for learning information-flow policies.

VIII. Conclusion

We have presented the first canonical representation for k-safety-hyperproperties. We introduced automata for representing k-safety hyperproperties and gave algorithms for constructing permutation-complete automata for such hyperproperties. We also presented the learning framework L^*_{Hyper} that can be used to learn minimal permutation-complete automata for k-safety hyperproperties and gave an instantiation for HyperLTL. The advantage of the algorithm is that it allows us to interactively learn monitors for information-flow polices and automatically construct efficient monitors from HyperLTL specification. It further allows for the simplification of manually specified k-safety-properties and for automatic equivalence checks between hyperproperties.

As a natural next step, we plan to implement the learning algorithm for HyperLTL and investigate further classes of HyperLTL formulas beyond the universally-safe fragment. Moreover, we plan on investigating further possible canonical representations for k-safety hyperproperties. For example, instead of looking for automata that accept the representations of all bad prefixes up to the arity k, we can change the definition of tightness to mean representations of only minimal bad prefixes. One advantage of this definition is, that in the case of HyperLTL, we can skip the additional expensive check that the conjecture automaton is vertically tight. This however, comes with the trade off, that the learner now has to extract minimal bad prefixes out of the counterexamples, which adds exponential costs.

References

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

[2] M. Y. Vardi, "Verification of concurrent programs: The automata-theoretic framework," in *Annals of Pure and Applied Logic*, 1987, pp. 167–176.

[3] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[4] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003, Proceedings*, 2003, pp. 331–346.

[5] B. Finkbeiner and H. Torfah, "Synthesizing skeletons for reactive systems," in *Automated Technology for Verification and Analysis*, C. Artho, A. Legay, and D. Peled, Eds. Cham: Springer International Publishing, 2016, pp. 271–286.

[6] A. van Lamsweerde and L. Willemet, "Inferring declarative requirements specifications from operational scenarios," *IEEE Trans. Softw. Eng.*, vol. 24, no. 12, pp. 1089–1114, Dec. 1998. [Online]. Available: https://doi.org/10.1109/32.738341

[7] A. Fern, S. Yoon, and R. Givan, "Learning domain-specific control knowledge from random walks," in *Proceedings of the Fourteenth International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'04. AAAI Press, 2004, pp. 191–198. [Online]. Available: http://dl.acm.org/citation.cfm?id=3037008.3037033

[8] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke, "User guidance for creating precise and accessible property specifications," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 208–218. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181801

[9] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Form. Methods Syst. Des.*, vol. 19, no. 3, pp. 291–314, Oct. 2001.

[10] O. Kupferman and R. Lampert, "On the construction of fine automata for safety properties," in *Proceedings of the 4th International Conference on Automated Technology for Verification and Analysis*, ser. ATVA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 110–124. [Online]. Available: http://dx.doi.org/10.1007/11901914_11

[11] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[12] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[13] B. Finkbeiner and C. Hahn, "Deciding Hyperproperties," in *27th International Conference on Concurrency Theory (CONCUR 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 59. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 13:1–13:14.

[14] B. Finkbeiner, M. N. Rabe, and C. Sánchez, "Algorithms for model checking hyperltl and hyperctl," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 30–48.

[15] J. Li, L. Zhang, G. Pu, M. Y. Vardi, and J. He, "Ltlf satisfiability checking," in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, vol. 263. IOS Press, 2014, pp. 513–518.

[16] G. P. Maretic, M. T. Dashti, and D. A. Basin, "LTL is closed under topological closure," *Inf. Process. Lett.*, vol. 114, no. 8, pp. 408–413, 2014.

[17] G. Barthe, P. R. D'argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical. Structures in Comp. Sci.*, vol. 21, no. 6, pp. 1207–1252, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1017/S0960129511000193

[18] G. Barthe, J. M. Crespo, and C. Kunz, "Beyond 2-safety: Asymmetric product programs for relational program verification," in *Logical Foundations of Computer Science*, S. Artemov and A. Nerode, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 29–43.

[19] D. D'Souza, R. Holla, K. R. Raghavendra, and B. Sprick, "Model-checking trace-based information flow properties," *J. Comput. Secur.*, vol. 19, no. 1, pp. 101–138, Jan. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2590694.2590698

[20] M. Balliu, "Logics for information flow security:from specification to verification," Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2014.

[21] B. Finkbeiner, C. Hahn, P. Lukert, M. Stenger, and L. Tentrup, "Synthesizing reactive systems from hyperproperties," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 289–306.

[22] S. Agrawal and B. Bonakdarpour, "Runtime verification of k-safety hyperproperties in hyperltl," in *CSF*. IEEE Computer Society, 2016, pp. 239–252.

[23] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup, "Monitoring hyperproperties," in *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, 2017, pp. 190–207.

[24] C. Hahn, M. Stenger, and L. Tentrup, "Constraint-based monitoring of hyperproperties," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 115–131.

[25] D. Angluin, "Queries and concept learning," *Mach. Learn.*, vol. 2, no. 4, pp. 319–342, Apr. 1988. [Online]. Available: https://doi.org/10.1023/A:1022821128753

[26] A. Farzan, Y.-F. Chen, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang, "Extending automated compositional verification to the full class of omega-regular languages," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 2–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792738

[27] L. Clapp, S. Anand, and A. Aiken, "Modelgen: Mining explicit information flow specifications from concrete executions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 129–140. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771810

[28] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," *SIGPLAN Not.*, vol. 44, no. 6, pp. 75–86, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1543135.1542485

[29] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287628