2016 IEEE 29th Computer Security Foundations Symposium

Runtime Verification of k-Safety Hyperproperties in HyperLTL

Shreya Agrawal
School of Computer Science
University of Waterloo, Canada
Email: s8agrawa@uwaterloo.ca

Borzoo Bonakdarpour
Department of Computing and Software
McMaster University, Canada
Email: borzoo@mcmaster.ca

***Abstract*—This paper introduces a novel *runtime verification* technique for a rich sub-class of Clarkson and Schneider's *hyperproperties*. The primary application of such properties is in expressing security policies (e.g., information flow) that cannot be expressed in trace-based specification languages (e.g., LTL). First, to incorporate syntactic means, we draw connections between safety and co-safety hyperproperties and the temporal logic HYPERLTL, which allows explicit quantification over multiple executions. We also define the notion of *monitorability* in HYPERLTL and identify classes of monitorable HYPERLTL formulas. Then, we introduce an algorithm for monitoring k-safety and co-k-safety hyperproperties expressed in HYPERLTL. Our technique is based on runtime formula progression as well as on-the-fly monitor synthesis across multiple executions. We analyze different performance aspects of our technique by conducting thorough experiments on monitoring security policies for information flow and observational determinism on a real-world location-based service dataset as well as synthetic trace sets.**

I. INTRODUCTION

Cybersecurity is an area of information technology where dependability plays a crucial role. This is because even a short transient violation of security policies may result in leaking private or highly sensitive information, compromising safety, or lead to the interruption of vital public or social services. In order to ensure that computing systems rigorously respect their security policies, numerous formal methods have been developed, most notably, different inference frameworks (e.g., [1]), as well as model checking [2], [3], [4], [5] and theorem proving techniques [6].

While exhaustive verification methods are extremely valuable, they often require developing an abstract model of the system and may suffer from the infamous state-explosion problem. *Runtime verification* (RV) refers to a technique where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. RV complements exhaustive verification techniques as well as under-approximating techniques such as testing and tracing. In the context of cybersecurity, RV is expected to be even more effective as it allows us to detect policy violations due to unanticipated threats that may exploit existing vulnerabilities.

A. Motivating Example

To demonstrate the subtleties of reasoning about security policies especially at run time, consider the anonymized screenshot of one of the second author's EDAS Conference Management[1] web interface in Figure 1. The color-coded table shows the status of submitted papers by the user: accepted (green), rejected (orange), withdrawn (grey), and pending (yellow). This web interface exhibits the following blunt violation of the well-known Goguen and Meseguer's *non-interference* (GMNI) security policy [7], where a low user should not be able to acquire any information about the activities (if any) of the high user. The first two rows show the status of two papers submitted to a conference after their notification: 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 is pending at the time the screenshot is taken. Although the authors (i.e., low users) should not be able to infer the internal decision making activities of the chairs (i.e., high users) before the notification, this table leaks these activities as follows. When a paper is accepted, it is supposed to be assigned to a session in the technical program, while a rejected paper does not need to be assigned to a session. Now, by comparing the first and the fourth 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. This is clearly a violation of non-interference through four independent executions to generate four HTML table rows.

In general, security policies that deal with information flow across multiple executions (e.g., GMNI) cannot be expressed and their evaluation cannot be achieved using a trace-base language such as the linear-time temporal logic (LTL). Although this observation was made in [8] more than a decade ago, little work has been done on the systematic verification of such policies, especially in the area of RV[2]. Monitoring such a policy in EDAS is especially challenging, as the monitor has

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

[2] We note that we have already contacted EDAS. They acknowledged and subsequently fixed the bug.

DOI 10.1109/CSF.2016.24

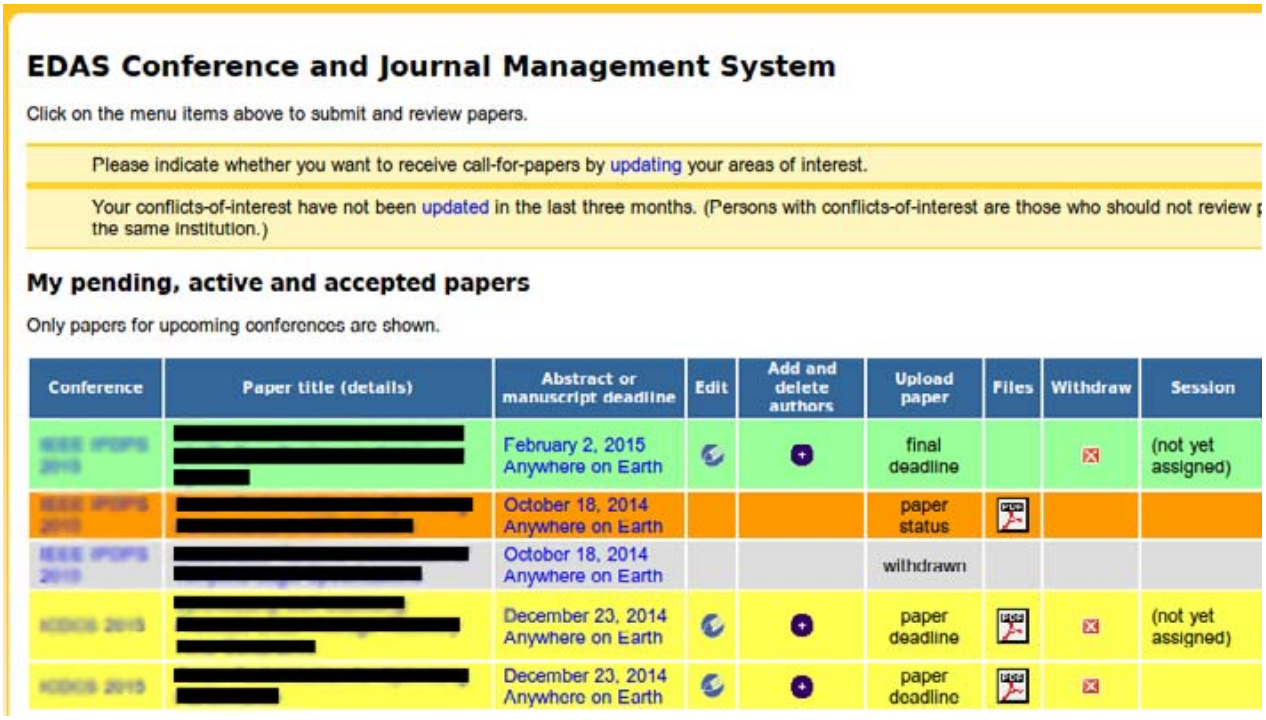


Fig. 1. EDAS conference management website's security violation.

to reason about its observations across independent executions of the procedure that generates each row of the HTML table.

B. Contributions

Clarkson and Schneider [9] proposed the notion of *hyperproperties* as a means to express security policies that cannot be expressed by traditional properties [10]. A hyperproperty is a set of sets of execution traces. Thus, a hyperproperty essentially defines a set of systems that respect a policy. Similar to the traditional concepts of safety and liveness, there are notions of *hypersafety* and *hyperliveness* properties. A hypersafety property can be characterized by a *bad* set of finite sets of finite traces. When the size of each finite set is at most k, it results in a k-safety hyperproperty. For example, GMNI is a 2-safety hyperproperty, as the bad thing can be characterized by pairs of bad executions. One of the very first model checking approaches for a subset of hyperproperties is the work by Terauchi et al. [11], but to our knowledge, there is no work on RV for hyperproperties.

Our focus in this paper is on monitoring k-safety hyperproperties, which represent a rich class of security policies. RV of hyperproperties is especially challenging because the monitor has to reason about the policy across different executions. For example, in Fig. 1 (and in fact in any HTML table generation of this sort) the rows of the table are generated by independent executions of a procedure. Thus, a monitor evaluating a hyperproperty has to (1) deal with the fact that it only observes a finite execution at run time, and (2) implement a mechanism to memorize and reason about its observations across multiple finite executions that occurred in the past and will happen in the future.

In this paper, we introduce the first RV technique for monitoring k-safety hyperproperties with no assistance from a static analyzer. We make the following contributions:

- First, we present a mapping from a subset of k-safety hyperproperties to HyperLTL—a temporal logic that allows quantification over execution traces [4]. We show that a subset of k-safety (respectively, co-k-safety) hyperproperties can be syntactically expressed as a disjunctive (respectively, conjunctive) HyperLTL formula with at most k universal quantifiers.
- Following [12], we define the notion of *monitorability* for HyperLTL and identify k-safety and co-k-safety hyperproperties that are monitorable based on their syntactic representation. We also identify other classes of HyperLTL formulas that are monitorable but are neither k-safety nor co-k-safety.
- We generalize the 3-valued semantics of LTL (LTL_3) [13] to k-safety HyperLTL formulas. We subsequently propose a monitoring algorithm for k-safety and co-k-safety HyperLTL formulas. Our algorithm employs three techniques: (1) a runtime progression logic (which enables us to reason about trace interdependencies), (2) on-the-fly LTL_3 monitor generation, and (3) a procedure that aggregates the progressed formulas and computes runtime verdicts using the generated LTL_3 monitors. The algorithm is appli-

ble to disjunctive (conjunctive) k-hypersafety (respectively, co-k-hypersafety) HYPERLTL formulas.

- We present rigorous experimental results on monitoring three different security policies on a location-based service dataset from Microsoft Research [14] as well as sets of synthetically generated traces. We analyze different metrics such as the length of progressed formulas and the number of generated LTL_3 monitors.

Organization: The rest of the paper is organized as follows. Section II presents the preliminary concepts on hyperproperties and HYPERLTL. In Section III, we establish the connection between k-safety hyperproperties and HYPERLTL. The notion of monitorability is introduced in Section IV. We introduce our RV algorithm and the experimental results in Sections V and VI, respectively. Related work is discussed in Section VII. Finally, we make concluding remarks and discuss future work in Section VIII. All proofs appear in the appendix.

II. BACKGROUND

Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{AP}$ be the finite *alphabet*. We call each element of Σ a *letter* (or an *event*). Throughout the paper, Σ^ω denotes the set of all infinite sequences (called *traces*) over Σ, and Σ^* denotes the set of all finite traces over Σ. For a trace $t \in \Sigma^\omega$, $t[i]$ denotes the i^{th} element of t, where $i \in \mathbb{Z}_{\geq 0}$. Also, $t[0,i]$ denotes the prefix of t up to and including i, and $t[i,\infty]$ is written to denote the infinite suffix of t beginning with element i.

Now, let u be a finite trace and v be a finite or infinite trace. We denote the concatenation of u and v by $\sigma = uv$. Also, $u \leq \sigma$ denotes the fact that u is a prefix of σ. Finally, if U is a set of finite traces and V is a finite or infinite set of traces, then the prefix relation $\leq$ on sets of traces is defined as:

$$U \leq V \;\equiv\; \forall u \in U.\, (\exists v \in V.\, u \leq v)$$

Note that V may contain traces that have no prefix in U.

A. Trace Properties

A *trace property* is a set of infinite traces (i.e., a subset of Σ^ω). The set of all trace properties is $\mathcal{P}(\Sigma^\omega)$, where $\mathcal{P}$ denotes the powerset. By $\mathcal{P}^*(X)$, we mean the set of all finite subsets of X. We assume that for a *system* p, $\psi(p)$ is the set of all execution traces of p; i.e., $\psi(p) \subseteq \Sigma^\omega$. We say that a system p *satisfies* a property S (denoted $p \models S$) iff $\psi(p) \subseteq S$.

B. Hyperproperties

It is well known that a large number of interesting security policies, such as non-interference and observational determinism, cannot be expressed by trace properties [8]. To overcome this shortcoming, Clarkson and Schneider [9] introduced the notion of *hyperproperties* to incorporate an additional level of sets to the notion of trace properties [9].

Definition 1 (hyperproperty)**.** A *hyperproperty* is a set of sets of infinite traces, or equivalently a set of trace properties. ■

The set of all hyperproperties is $\mathcal{P}(\mathcal{P}(\Sigma^\omega))$. The interpretation of a hyperproperty as a security policy is that the hyperproperty is the set of systems allowed by that policy. That is, each trace property in a hyperproperty is an allowed system, specifying exactly which executions must be possible for that system. Thus, unlike trace properties, where the notion of satisfaction is based on language *inclusion*, the definition of satisfaction for hyperproperties is based on language *equality*.

Definition 2. A system p *satisfies* a hyperproperty $\boldsymbol{H}$ (denoted, $p \models \boldsymbol{H}$) iff $\psi(p) \in \boldsymbol{H}$. ■

That is, a program satisfies a security policy if and only if its set of traces adheres with one of the entire sets (and not just a subset) of traces of the prescribed policy.

1) Safety Hyperproperties: Safety hyperproperty (or hypersafety) is a generalization of *safety* [10], where the bad thing occurs in a finite set of finite traces. The definition of hypersafety is essentially the same as the definition of safety, except for an additional level of sets.

Definition 3 (k-safety hyperproperty)**.** A hyperproperty $\boldsymbol{S_k}$ is a *k-safety hyperproperty* (is *k-hypersafety*) iff

$$\forall T \in \mathcal{P}(\Sigma^\omega).(T \notin \boldsymbol{S_k}) \;\Rightarrow\; \exists M \in \mathcal{P}^*(\Sigma^*).(M \leq T) \wedge (|M| \leq k) \wedge (\forall T' \in \mathcal{P}(\Sigma^\omega).(M \leq T') \;\Rightarrow (T' \notin \boldsymbol{S_k}))$$ ■

In Definition 3, set M represents a *bad thing* that should never happen. If there is no bound on the cardinality of M, then $\boldsymbol{S_k}$ becomes a safety hyperproperty. Notice that a traditional safety property [10] is synonymous to a 1-safety hyperproperty [9].

Examples:

- A policy that requires 'whenever there is a fail event, then there must not be a login event for at least four time units' is a 1-safety hyperproperty. If one models the passage of every time unit by the event tick, then the bad thing here is a finite trace that contains a fail followed by three or fewer tick events before a login event.
- Goguen and Meseguer's *non-interference* (GMNI) [7], where inputs issued by users holding high variables should be removable without affecting observations of users holding low variables, is a 2-safety hyperproperty.
- The *information leakage* policy is an example of a safety hyperproperty. The bad thing is *some* series of experiments, where the information leaked is more than x bits. Notice that in this example there is no bound on $|M|$.

2) Co-safety Hyperproperties: Intuitively, a *co-safety hyperproperty* (or *co-hypersafety*) stipulates a policy which describes the occurrence of a *good thing* and is a generalization of traditional *co-safety* [15].

Definition 4 (co-k-safety hyperproperty)**.** A hyperproperty $\boldsymbol{C}$ is a *co-k-safety hyperproperty* (or co-k-hypersafety) iff

$$\forall T \in \mathcal{P}(\Sigma^\omega).(T \in \boldsymbol{C}) \Rightarrow \exists M \in \mathcal{P}^*(\Sigma^*).(M \leq T) \wedge (|M| \leq k) \ \wedge \ (\forall T' \in \mathcal{P}(\Sigma^\omega).(M \leq T') \Rightarrow (T' \in \boldsymbol{C})) \quad \blacksquare$$

In Definition 4, set M represents a *good thing* in $\boldsymbol{C}$. Notice that a co-safety property is synonymous to a co-1-safety hyperproperty [9].

Example: The hyperproperty 'for every initial state, there is some terminating trace, but not all traces must terminate' is a co-safety hyperproperty. The good thing here is a set of traces such that for all initial states, a trace in this set terminates. If the number of initial states is restricted to k, then this is a co-k-safety hyperproperty.

C. HyperLTL

HyperLTL is a logic for syntactic representation of hyperproperties. It generalizes LTL by allowing explicit quantification over multiple execution traces simultaneously [4].

1) Syntax:

Definition 5. The set of HyperLTL formulas is inductively defined by the grammar as follows:

$$\varphi ::= \exists \pi.\varphi \mid \forall \pi.\varphi \mid \phi$$
$$\phi ::= a(\pi) \mid \neg\phi \mid \phi \vee \phi \mid \phi \,\mathbf{U}\, \phi \mid \mathbf{X}\phi$$

where $a \in AP$ and π is a trace variable from an infinite supply of variables Γ. $\blacksquare$

Similar to LTL, $\mathbf{U}$ and $\mathbf{X}$ are the 'until' and 'next' operators, respectively. Other standard temporal connectives are defined as syntactic sugar as follows: $\varphi_1 \Rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\texttt{true} = a_\pi \vee \neg a_\pi$, $\texttt{false} = \neg\texttt{true}$, $\mathbf{F}\phi = \texttt{true}\,\mathbf{U}\,\phi$, and $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$. Quantified formulas $\exists\pi$ and $\forall\pi$ are read as 'along some trace π' and 'along all traces π', respectively.

2) Semantics: A formula φ in HyperLTL satisfied by a set of traces T is written as $\Pi \models_T \varphi$, where trace assignment $\Pi : \Gamma \to \Sigma^\omega$ is a partial function mapping trace variables to traces. $\Pi[\pi \to t]$ denotes the same function as Π, except that π is mapped to trace t.

Definition 6. The validity judgment for HyperLTL is defined as follows:

$\Pi \models_T \exists\pi.\varphi$	iff	$\exists t \in T.\, \Pi[\pi \to t] \models_T \varphi$
$\Pi \models_T \forall\pi.\varphi$	iff	$\forall t \in T.\, \Pi[\pi \to t] \models_T \varphi$
$\Pi \models_T a(\pi)$	iff	$a \in \Pi(\pi)[0]$
$\Pi \models_T \neg\phi$	iff	$\Pi \not\models_T \phi$
$\Pi \models_T \phi_1 \vee \phi_2$	iff	$(\Pi \models_T \phi_1) \vee (\Pi \models_T \phi_2)$
$\Pi \models_T \mathbf{X}\phi$	iff	$\Pi[1,\infty] \models_T \phi$
$\Pi \models_T \phi_1 \,\mathbf{U}\, \phi_2$	iff	$\exists i \geq 0.\, (\Pi[i,\infty] \models_T \phi_2 \wedge \forall j \in [0,i).\, \Pi[j,\infty] \models_T \phi_1)$

where the trace assignment suffix $\Pi[i,\infty]$ denotes the trace assignment $\Pi' = \Pi(\pi)[i,\infty]$ for all π. If $\Pi \models_T \phi$ holds for the empty assignment Π, then T satisfies ϕ. $\blacksquare$

Observe that when there is exactly one universal trace quantifier, then LTL and HyperLTL coincide.

Notation: By $\phi(\pi_1, \ldots, \pi_k)$, we mean the formula $\phi(\pi_1) \vee \cdots \vee \phi(\pi_k)$, where $\phi(\pi_i)$ is a syntactic sugar to represent the formula where trace variable π_i is applied to every proposition of ϕ. For example $(a\,\mathbf{U}\,b)(\pi_i)$ means $a(\pi_i)\,\mathbf{U}\,b(\pi_i)$. Obviously, formula $a(\pi)\,\mathbf{U}\,b(\pi')$ is a well-formed formula in HyperLTL, but not in our notation. Also, let LTL_S, and LTL_C be the set of safety and co-safety LTL formulas, respectively. For example, $\mathbf{G}p \in \text{LTL}_S$ and $\mathbf{F}p \in \text{LTL}_C$.

D. Specifying Trace Relations

Clarkson et al. [4] introduce the trace relation $=_P$ to ease the representation of equivalence between traces. Let π and π' be two trace variables. For a set $P \subseteq AP$ of atomic propositions, $\pi[0] =_P \pi'[0]$ denotes that the first letter in both π and π' agree on all propositions in P. Further, $\pi =_P \pi'$ means that all the positions in π and π' agree on P:

$$\pi =_P \pi' \ \equiv \ \mathbf{G}(\pi[0] =_P \pi'[0])$$

Example:

- An example of GMNI can be specified as a HyperLTL formula as follows:

  $$\forall\pi.\forall\pi'.\, (\mathbf{G}\lambda_H(\pi') \ \wedge \ \pi \neq_H \pi') \ \Rightarrow \pi =_L \ \pi'$$

  where $\mathbf{G}\lambda_H(\pi')$ denotes that all high variables in π' hold the value λ for all letters, and H and L are the 'high' and 'low' atomic propositions, respectively.

- *Observational Determinism* (OD) requires a system to appear deterministic to a low user (users who only have access to low variables). It is specified as follows:

  $$\forall\pi.\forall\pi'.\, (\pi[0] =_{L,in} \pi'[0]) \ \Rightarrow \ (\pi =_{L,out} \pi')$$

  where $=_{L,in}$ checks for agreement on propositions in L with input values issued by the low user.

Addressing Limitations: While the operator $=_P$ for trace relations allows one to specify properties over a pair of traces that check for equivalence letter by letter, it does not capture comparison of some letter in one trace with one or more letters in another trace, or comparison of letters over temporal formulas specified over P.

To address this limitation, we define a function

$$f : 2^{AP} \to \text{LTL}$$

and extend the trace relation to $\pi \sim_{f,P} \pi'$, for two trace variables π and π', and a set P of atomic propositions. We require that

$$\forall i.(\pi'[i..\infty] \models f(\pi[i] \cap P) \ \wedge \ \pi[i..\infty] \models f(\pi'[i] \cap P))$$

That is, each event maps to some LTL formula and the trace relation between two traces requires that the trace starting from the corresponding event in the other trace should satisfy

this LTL formula. Obviously, when function f is the identity function $\pi =_P \pi' \equiv \pi \sim_{f,P} \pi'$. For example, a variation of GMNI requires that if two traces do not agree on high values and the initial states agree on the low values, then at some point in the future, they should always agree on a subset of low values:

$$\forall\pi.\forall\pi'.((\mathbf{G}\lambda_H(\pi) \wedge \pi \neq_H \pi' \wedge \pi[0] =_L \pi'[0]) \Rightarrow \pi \sim_{f,L} \pi')$$

where $f(x) = \mathbf{F}x$ for $x \subseteq L$.

III. k-Safety/Co-k-Safety Hyperproperties in HyperLTL

In this section, we establish the connection between the set representation of k-safety and co-k-safety hyperproperties with HyperLTL. First, we present a lemma that shows that the complement of a safety (respectively, co-safety) hyperproperty $\boldsymbol{S}$, denoted as $\bar{\boldsymbol{S}}$, is a co-safety (respectively, safety) hyperproperty.

Lemma 1. *The complement of a safety hyperproperty is a co-safety hyperproperty and vice versa. Also, the complement of a k-safety hyperproperty is a co-k-safety hyperproperty and vice versa.*

Clarkson et al. [4] identified HyperLTL$_n$ as the class of HyperLTL formulas in which the sequence of quantifiers at the beginning of the formula involves at most $n-1$ alternations. We now show that a subset of k-safety and co-k-safety hyperproperties can be expressed as a HyperLTL$_1$ formula.

Lemma 2. *Consider a* HyperLTL$_1$ *formula of the following form:*

$$\varphi_{\boldsymbol{C_k}} = \exists\pi_1 \cdots \exists\pi_k.(\phi_1(\pi_1,\ldots,\pi_k) \wedge \cdots \wedge \phi_k(\pi_1,\ldots,\pi_k))$$

where $\phi_1,\ldots,\phi_k \in$ LTL$_C$. Such a formula represents a co-k-safety hyperproperty.

An immediate corollary of Lemma 2 states that a class of k-safety hyperproperties can be expressed in HyperLTL$_1$ formula.

Corollary 1. *Consider a* HyperLTL$_1$ *formula of the following form:*

$$\varphi_{\boldsymbol{S_k}} = \forall\pi_1 \cdots \forall\pi_k.(\phi_1(\pi_1,\ldots,\pi_k) \vee \cdots \vee \phi_k(\pi_1,\ldots,\pi_k))$$

where $\phi_1,\ldots,\phi_k \in$ LTL$_S$. Such a formula represents a k-safety hyperproperty.

Theorem 1. *Conjunction (respectively, disjunction) of* HyperLTL$_1$ *formulas, with at most k quantifiers, given by $\varphi_{\boldsymbol{S_k}}$ in Corollary 1 (respectively, $\varphi_{\boldsymbol{C_k}}$ in Lemma 2), is a k-hypersafety property (respectively, co-k-hypersafety property).*

IV. RV Semantic and Monitorability in HyperLTL

First, we introduce RV semantics and the notion of monitorability for HyperLTL in Subsections IV-A and IV-B, respectively. Then, in Subsection IV-C, we present classes of monitorable HyperLTL$_1$ formulas.

A. RV Semantics

Inspired by the 3-valued semantics of LTL [13], we now define RV semantics for HyperLTL (denoted HyperLTL-3). The semantics utilize three truth values $\mathbb{B}_3 = \{\top,\bot,?\}$, where '?' means that given a formula φ and the current set M of executions at run time, it is not possible to tell whether M satisfies or violates φ; i.e., both cases are possible in this or future executions.

Definition 7 (HyperLTL-3 semantics)**.** Let $M \in \mathcal{P}^*(\Sigma^*)$ be a finite set of finite traces. The truth value of a HyperLTL closed formula φ with respect to M, denoted by $[M \models \varphi]$, is an element of the set $\mathbb{B}_3 = \{\top,\bot,?\}$, and is defined as follows:

$$[M \models \varphi] = \begin{cases} \top & \text{if } \forall T \in \mathcal{P}(\Sigma^\omega).(M \leq T).\Pi \models_T \varphi \\ \bot & \text{if } \forall T \in \mathcal{P}(\Sigma^\omega).(M \leq T).\Pi \not\models_T \varphi \\ ? & \text{otherwise} \end{cases}$$

■

B. Monitorability

Pnueli and Zaks [12] characterize an LTL formula φ as *monitorable* for a finite trace u, if u can be extended to one that can be evaluated with respect to φ at run time. For example, LTL formula $\mathbf{GF}p$ is not monitorable, since there is no way to tell at run time, whether or not in the future, p will be visited infinitely often. On the contrary, formulas in LTL$_S$ (e.g., $\mathbf{G}p$) and in LTL$_C$ (e.g., $\mathbf{F}p$) are monitorable.

We now generalize the same idea to the context of HyperLTL. First, we argue that HyperLTL$_n$ formulas, where $n \geq 2$, (e.g., $\forall\exists\psi$) are not monitorable, as evaluating such formulas requires one to have *all* traces of the system. However, safety and co-safety HyperLTL formulas have a different nature. For instance, consider the following secret sharing scheme (denoted SSS):

A system stores a secret by splitting it into k shares.

A policy that prevents revealing all the k shares can be expressed by the following HyperLTL$_1$ formula:

$$\varphi_{\boldsymbol{SS_k}} = \forall\pi_1 \cdots \forall\pi_k.(\mathbf{G}\neg a_1(\pi_1,\ldots,\pi_k) \vee \ldots \vee \mathbf{G}\neg a_k(\pi_1,\ldots,\pi_k))$$

This formula is monitorable because, if propositions $a_1 \cdots a_k$ become true in at most any k traces, where a_i holds iff share i of the secret has been revealed, then $\varphi_{\boldsymbol{SS_k}}$ can be declared as violated permanently (i.e., $\bot$) for all future executions.

Definition 8 (monitorability)**.** A HyperLTL formula φ is *monitorable* iff

$$\forall M \in \mathcal{P}^*(\Sigma^*).\exists M' \in \mathcal{P}^*(\Sigma^*).[MM' \models \varphi] \in \{\bot,\top\}$$

■

Formula	Property of ϕ	$\top$	$\bot$	Runtime evidence (proof)
$\forall\pi.\,\phi$	$\phi \in \text{LTL}_S$	×	✓	$\exists M.\,\exists u \in M.\,[u \models \phi] = \bot$
$\forall\pi.\,\phi$	$\phi \in \text{LTL}_C - \text{LTL}_S$	×	×	$\nexists M.\,\exists u \in M.\,[u \models \phi] = \bot$
$\forall\pi_1 \ldots \forall\pi_k.\,(\phi_1(\pi_1) \vee \cdots \vee \phi_k(\pi_k))$	$\phi_1, \ldots, \phi_k \in \text{LTL}_S$	×	✓	$\exists M.\,\exists u_1 \ldots u_k \in M.\,[u_1 \models \phi_1] = \bot \wedge \cdots \wedge [u_k \models \phi_k] = \bot$
$\forall\pi_1 \ldots \forall\pi_k.\,(\phi_1(\pi_1) \wedge \cdots \wedge \phi_k(\pi_k))$	$\exists i (1 \leq i \leq k).\,\phi_i \in \text{LTL}_S$	×	✓	$\exists M.\,\exists u_i \in M.\,[u_i \models \phi_i] = \bot$
$\forall\pi_1 \ldots \forall\pi_k.\,(\phi_1(\pi_1) \vee \cdots \vee \phi_k(\pi_k))$	$\exists i (1 \leq i \leq k).\,\phi_i \notin \text{LTL}_S$	×	×	$\nexists M.\,\exists u_i \in M.\,[u_i \models \phi_i] = \bot$
$\forall\pi_1.\forall\pi_2.(\phi_1(\pi_1)\,\mathbf{U}\,\phi_2(\pi_2))$	$\phi_1, \phi_2 \in \text{LTL}_S$	×	✓	$\exists M.\,\exists u_1, u_2 \in M.\,[u_1, u_2 \models \phi_1(u_1)\,\mathbf{U}\,\phi_2(u_2)] = \bot$

TABLE I. MONITORABILITY OF UNIVERSALLY QUANTIFIED HYPERLTL$_1$ FORMULAS.

Formula	Property of ϕ	$\top$	Runtime evidence (proof)	$\bot$
$\exists\pi.\,\phi$	$\phi \in \text{LTL}_C$	✓	$\exists M.\,\exists u \in M.\,[u \models \phi] = \top$	×
$\exists\pi.\,\phi$	$\phi \in \text{LTL}_S - \text{LTL}_C$	×	$\nexists M.\,\exists u \in M.\,[u \models \phi] = \top$	×
$\exists\pi_1 \ldots \exists\pi_k.\,(\phi_1(\pi_1) \wedge \cdots \wedge \phi_k(\pi_k))$	$\phi_1, \ldots, \phi_k \in \text{LTL}_C$	✓	$\exists M.\,\exists u_1 \ldots u_k \in M.\,[u_1 \models \phi_1] = \top \wedge \cdots \wedge [u_k \models \phi_k] = \top$	×
$\exists\pi_1 \ldots \exists\pi_k.\,(\phi_1(\pi_1) \vee \cdots \vee \phi_k(\pi_k))$	$\exists i (1 \leq i \leq k).\,\phi_i \in \text{LTL}_C$	✓	$\exists M.\,\exists u_i \in M.\,[u_i \models \phi_i] = \top$	×
$\exists\pi_1 \ldots \exists\pi_k.\,(\phi_1(\pi_1) \wedge \cdots \wedge \phi_k(\pi_k))$	$\exists i (1 \leq i \leq k).\,\phi_i \notin \text{LTL}_C$	×	$\nexists M.\,\exists u_i \in M.\,[u_i \models \phi_i] = \top$	×
$\exists\pi_1.\exists\pi_2.(\phi_1(\pi_1)\,\mathbf{U}\,\phi_2(\pi_2))$	$\phi_1, \phi_2 \in \text{LTL}_C$	✓	$\exists M.\,\exists u_1, u_2 \in M.\,[u_1, u_2 \models \phi_1(u_1)\,\mathbf{U}\,\phi_2(u_2)] = \top$	×

TABLE II. MONITORABILITY OF EXISTENTIALLY QUANTIFIED HYPERLTL$_1$ FORMULAS.

C. Monitorables Classes in HYPERLTL$_1$

Table I (respectively, Table II) refers to universally (respectively, existentially) quantified HYPERLTL$_1$ formulas and summarizes their monitorability. The tables also provide the evidence that can show whether a formula can be truthified or falsified at run time, based on Definition 8. Observe that this evidence is, in fact, the proof of monitorability of the formula as well. For example:

- For the formula $\varphi = \forall\pi.\,\phi$, where $\phi \in \text{LTL}_S$ (first row of Table I), is monitorable, as any finite trace can be extended to one that falsifies ϕ. For instance, if $\phi = \mathbf{G}p$, every finite trace can be extended to one that looks like $u = p^*\neg p$, which violates ϕ and, hence, φ. Such a formula, however, cannot be declared satisfied at run time since that would require monitoring every infinite trace in the infinite domain of π.
- For the third formula in Table I, the runtime evidence that violates the formula is a finite set of finite traces where every formula $\phi_1, \ldots, \phi_k$ is violated by a finite trace in the set. The SSS policy corresponds to this formula which is violated if a finite set of finite traces reveal each of the k shares. Also, GMNI and OD fall in this category of formulas.

Observe that the highlighted formulas in Tables I and II are not monitorable. The third row formulas, correspond to monitorable k-safety and co-k-safety hyperproperties, respectively. The fourth formula, in Table I (respectively, Table II) is an example of a monitorable formula that is neither a k-hypersafety nor a co-k-hypersafety property if $\exists i, j.\,\phi_i \in \text{LTL}_S$ and $\phi_j \notin \text{LTL}_S$ (respectively, $\exists i, j.\,\phi_i \in \text{LTL}_C$ and $\phi_j \notin \text{LTL}_C$). Note that, these tables do not capture all formulas in HYPERLTL$_1$, and only shows some relevant ones pertaining to monitorability of k-safety hyperproperties. However, the monitorability of all other formulas can be derived from Definition 8 and the given tables.

Formula	Property of ϕ	$\top$	$\bot$
$\forall\pi_1.\,\phi_1 \;\vee\; \exists\pi_2.\,\phi_2$	$\phi_1 \in \text{LTL}_S, \phi_2 \in \text{LTL}_C$	✓	×
$\forall\pi_1.\,\phi_1 \;\wedge\; \exists\pi_2.\,\phi_2$	$\phi_1 \in \text{LTL}_S, \phi_2 \in \text{LTL}_C$	×	✓
$\forall\pi_1.\,\phi_1 \;\vee\; \exists\pi_2.\,\phi_2$	$\phi_1 \in \text{LTL}_S, \phi_2 \notin \text{LTL}_C$	×	×
$\forall\pi_1.\,\phi_1 \;\wedge\; \exists\pi_2.\,\phi_2$	$\phi_1 \notin \text{LTL}_S, \phi_2 \in \text{LTL}_C$	×	×

TABLE III. MONITORABILITY OF HYPERLTL FORMULAS WITH CONJUNCTION OR DISJUNCTION OF FORMULAS FROM TABLES I AND II.

In Table III, we take disjunctions and conjunctions of formulas from Tables I and II. The runtime evidence for whether a formula of the given syntactic form can be declared satisfied or violated at run time follows trivially.

Theorem 2. *Every k-safety hyperproperty and every co-k-safety hyperproperty that satisfies Theorem 1 is monitorable.*

By exploring the monitorability of various formulas in HYPERLTL$_1$, we see that the set of monitorable formulas in HYPERLTL$_1$ includes properties outside of k-safety and co-k-safety hyperproperties. For example, the formula $\forall\pi_1.\,\forall\pi_2.\,(\mathbf{G}p(\pi_1) \wedge \mathbf{F}q(\pi_2))$ is neither a safety hyperproperty nor a co-safety hyperproperty. However, it is monitorable and can be declared violated at run time. We note that the above classification also includes HYPERLTL$_1$ formulas that are monitorable, but are neither k-hypersafety nor co-k-hypersafety.

Theorem 3. *The set of all monitorable* HYPERLTL$_1$ *formulas includes properties that are neither k-hypersafety nor co-k-hypersafety properties.*

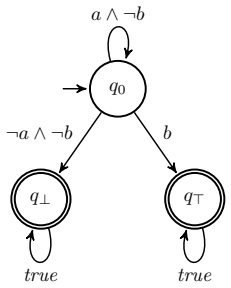


Fig. 2. LTL$_3$ monitor for formula $a \,\mathbf{U}\, b$.

V. Monitoring Algorithm

In this section, we present our algorithm for monitoring k-safety and co-k-safety hyperproperties given by Theorem 1.

A. Algorithm Sketch

Consider formula

$$\varphi_{\boldsymbol{S_k}} = \forall \pi_1 \cdots \forall \pi_k. (\phi_1(\pi_1, \ldots, \pi_k) \vee \cdots \vee \phi_k(\pi_1, \ldots, \pi_k)).$$

Our algorithm has three key elements:

- In order to monitor such a formula, due to the existence of trace quantifiers, each sub-formula ϕ_i, where $1 \leq i \leq k$, needs to be monitored independently, possibly across different executions. For example, in OD, if the initial state of any two pairs of executions correspond to a low input, then the monitor must be able to identify the initial states, so that it can analyze the rest of both executions to ensure that only low outputs are produced. Thus, we assume that our monitoring algorithm is notified when an execution terminates and when a new execution commences.
- In order for the monitor to memorize and combine the independent evaluation of each sub-formula across different executions at run time, we utilize a *Petri net* (see Fig. 3). That is, on-the-fly evaluation of each ϕ_i is achieved by a net whose verdict contribute to determining the verdict of $\varphi_{\boldsymbol{S_k}}$.
- If there exists a trace relation $\sim_{f,p}$ in the formula, then monitoring an execution may depend on evaluation of past and future executions. To tackle this problem, we propose a formula *progression* technique, which constructs a formula to be further progressed or monitored in the future executions. In such cases, the monitor structure evolves over time (see Fig. 4).

In the remainder of this section, we first describe our progression technique in Section V-B1. Section V-B2 introduces our monitoring algorithm. We generalize our technique for formulas of Tables I-III that are neither k-safety nor co-k-safety hyperproperties, in Subsection V-C.

B. Algorithm Details

We now describe different aspects of the algorithm in detail.

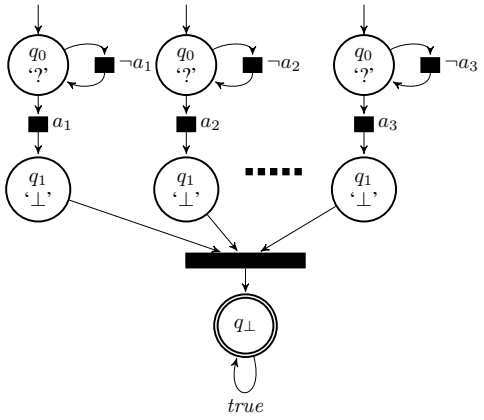


Fig. 3. Monitor for SSS.

1) Progression for Trace Relations: Recall that in Section II-D, we introduced relation $\sim_{f,P}$ to express inter-trace relations. Unlike existing techniques for formula rewriting [16] and progression [17], which split a formula into goals for the current state and future goals, our formula progression method constructs a formula for execution traces based on the goals satisfied by the currently seen executions. Formally, let $U = \{u_1, u_2, \ldots, u_m\}$ be a finite set of finite traces (representing a set of program executions at run time) and

$$\pi_1 \sim_{f,P} \pi_2 \sim_{f,P} \cdots \sim_{f,P} \pi_n$$

be a trace relation in some HyperLTL$_1$ formula φ to be monitored. We define the *progression function*

$$\boldsymbol{Pg} : 2^P \times \mathbb{Z} \times \mathbb{Z} \to \text{LTL}$$

inductively in Equation 1. On observing an event of any trace, the progression function is applied according to one of the three cases:

- The first case handles the very first event in the very first trace at run time.
- The second case handles progression within a trace, adding $i+1$ number of next operators applied on f.
- The third case shows how progression is transferred from one execution to the next (up to n times for each trace). Thus, the progression of every new trace depends upon the progression of all the previously observed traces.

For example, for a policy that requires no two traces reach the same state (i.e., $f(x) = \neg x$), if a trace 'abc' is observed then $\boldsymbol{Pg}(a, 1, 0)$ results in $\neg a$, $\boldsymbol{Pg}(b, 1, 1) = \neg a \wedge \mathbf{X}\neg b$, and $\boldsymbol{Pg}(c, 1, 2) = \neg a \wedge \mathbf{X}\neg b \wedge \mathbf{XX}\neg c$. Every other trace is checked against the progressed formula $\neg a \wedge \mathbf{X}\neg b \wedge \mathbf{XX}\neg c$.

Essentially, since the trace relation involves a set of n trace variables in φ, the progression function is applied to every subset of size n of the set of program executions. For instance, for monitoring a 2-safety hyperproperty over m execution traces, $\boldsymbol{Pg}$ needs to be applied for each pair of traces in U; i.e., $\binom{m}{2}$ times.

$$\begin{cases} \boldsymbol{Pg}(u_1[0],1,0) = f(u_1[0] \cap P) \\ \boldsymbol{Pg}(u_j[i+1],j,i+1) = \boldsymbol{Pg}(u_j[i],j,i) \wedge \mathbf{X}^{i+1} f(u_j[i+1] \cap P) \quad \text{if } (1 \leq i) \wedge (1 \leq j \leq n-1) \\ \boldsymbol{Pg}(u_{j+1}[0],j+1,0) = \boldsymbol{Pg}(u_j[|u_j|],j,|u_j|) \wedge f(u_{j+1}[0] \cap P) \quad \text{if } (1 < j < n-1) \end{cases} \tag{1}$$

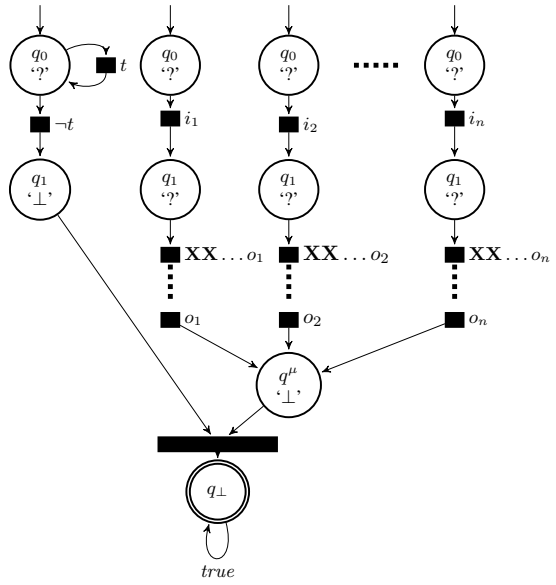


Fig. 4. Petri net for property termination-sensitive

2) LTL$_3$ *Monitors:* Let $\varphi = \forall\pi_1 \cdots \forall\pi_k.\ \phi_1 \vee \cdots \vee \phi_k$ be the HYPERLTL$_1$ formula to be monitored. We first categorize formulas that require the progression logic and those that do not, as they will be handled differently. A sub-formula ϕ_i, $1 \leq i \leq k$, is called *observation independent* if it does not contain the $\sim_{f,P}$ relation. Otherwise, the sub-formula is called *observation dependent*. For example, all sub-formulas in SSS are observation independent. And, formulas OD and GMNI have observation dependent sub-formulas. An observation independent formula is monitored by an LTL$_3$ monitor. We denote the set of observation independent and observation dependent sub-formulas of φ by α and β, respectively.

Definition 9 (LTL$_3$ Monitor [13])**.** Let ϕ be an LTL formula over alphabet Σ. The *monitor* $\mathcal{M}^\phi$ of ϕ is the deterministic finite-state automaton $(\Sigma, Q, q_0, \delta, \lambda)$, where Q is a set of states, q_0 is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and λ is the function $\lambda : Q \to \mathbb{B}_3$, such that for any $u \in \Sigma^*$:

$$[u \models \phi] = \lambda(\delta(q_0, u)) \quad \blacksquare$$

Bauer et al. [13] propose an automated technique to construct such a monitor for a given LTL formula. For example, Fig. 2 shows the LTL$_3$ monitor for formula $a\,\mathbf{U}\,b$. As mentioned in the algorithm sketch, we use a Petri net to build a network of LTL$_3$ monitors.

Definition 10. A (1-Safe) *Petri net* is defined by a triple $S = (L, \Sigma, \delta)$, where L is a set of places, Σ is the alphabet, and $\delta \subseteq 2^L \times \Sigma \times 2^L$ is a set of transitions. A transition τ is a triple $({}^\bullet\tau, \sigma, \tau^\bullet)$, where ${}^\bullet\tau$ is the set of input places of τ and $\tau^\bullet$ is the set of output places of τ. $\blacksquare$

3) Step-by-step Description: We utilize the 'termination-sensitive' policy as a running example to demonstrate the steps of Algorithm 1:

> *'If any execution of a machine reaches a terminating state, then no two other executions starting from the same initial state should reach differing accepting state.'*

This policy is the following 3-safety hyperproperty:

$$\varphi = \forall\pi_1\forall\pi_2\forall\pi_3\,.\,\mathbf{G}t(\pi_1) \ \vee (\pi_2[0] =_P \pi_3[0] \Rightarrow \pi_2 \sim_{f,P} \pi_3)$$

where t is a proposition for 'not in terminating state' and function f is such that $f(o) = o$, where o is a proposition indicating an accepting state, captures the latter part of our policy. In this formula, the set of observation independent sub-formulas is $\alpha = \{\mathbf{G}t\}$ and the set of observation dependent sub-formulas is $\beta = \{\pi_2 \sim_{f,P} \pi_3\}$. Fig. 4 shows the Petri net that would be created on observing n independent executions. Also, Fig. 3 shows the monitor for SSS, for which the corresponding HYPERLTL$_1$ formula only includes observation independent sub-formulas.

We now describe the algorithm in detail which leads to the final construction of Fig. 4. For intuition, one can think of the places in the Petri net as the states a monitor goes through. The nets are simply the collection of LTL$_3$ monitors constructed during the process and the value λ of the monitor indicates whether the formula is satisfied, violated or still unknown.

1) (Line 4) Initially, the set of places in the Petri net (which will be constructed on the fly) contains a final place $q_\perp$ denoting the violation of φ (recall that a hypersafety formula cannot be declared satisfied at run time) and the final output of our algorithm (λ) is unknown ('?').
2) (Lines 5-9) For every observation independent sub-formula, an LTL$_3$ monitor is constructed using `construct_net` which becomes a net of the Petri net. For example, the leftmost net in Fig. 4 (for $\mathbf{G}t$) and all the nets in Fig. 3 (for each $\mathbf{G}\neg a_i$) are constructed in this step. Then, for every observation

Algorithm 1: k-safety monitoring algorithm

Input: $\varphi = \forall\pi_1 \cdots \forall\pi_k.\, \phi_1 \vee \cdots \vee \phi_k, \alpha, \beta$
Output: $\lambda \in \{?, \bot\}$

```
1  L := {q_⊥};
2  T := {(q_⊥, true, q_⊥)};
3  λ :=?;
4  nets := {};
5  foreach φ ∈ α do
6      construct_net(φ, true);
7  foreach μ ∈ β do
8      L := L ∪ {q^μ} ;
9      T := T ∪ {(q^μ, true, q_⊥)} ;
10 while true do
11     get_input(e^i_j, new_trace);
12     if new_trace = true then
13         reset (nets);
14         β' := β;
15     foreach M^φ ∈ nets do
16         q := evaluate(M^φ, e^i_j) ;
17         if λ^φ(q) = ⊥ then
18             if ∃μ ∈ β . q = q^μ then
19                 nets := nets − {M^μ};
20             else
21                 nets := nets − {M^φ};
22     foreach μ' ∈ β' and μ ∈ β do
23         μ' := Pg(e^i_j, j, i);
24         if progression_complete(μ', μ) then
25             construct_net(μ', μ);
26     if (•τ, true, {q_⊥}•) is enabled then
27         λ := ⊥; return λ ;
28 construct_net(Formula φ, Formula μ) {
29 M^φ := (Σ, Q^φ, q_0^φ, δ^φ, λ^φ) (see Definition 9);
30 L := L ∪ Q^φ;
31 T := T ∪ δ^φ;
32 Let q ∈ Q^φ, where λ^φ(q) = ⊥ (only one such state
   exists is in Q);
33 if μ ≠ true then
34     merge(q, q^μ);
35     T := T − {(q^μ, q^μ)};
36 else
37     T := T ∪ (q, true, q_⊥) ;
38     T := T − {(q, q)};
39 nets := nets ∪ {M^φ};
40 }
```

dependent sub-formula μ, a new state q^μ is added to the Petri net, which is also an input place for a transition to the output place $q_\perp$. For example, this results in place q^μ for sub-formula $\mu = \pi_2 \sim_{f,P} \pi_3$ in Fig. 4.

3) (Lines 28–39) The procedure `construct_net` creates nets of the Petri net (Lines 30-31). Each net is essentially an LTL$_3$ monitor. When an LTL$_3$ monitor is created, if the second argument μ is `true`, then the state q of this monitor, where $\lambda(q) = \perp$ becomes an input place for a transition whose output place is $q_\perp$ (e.g., in the net that corresponds to $\mathbf{G}t$ in Fig. 4) (Lines 36-38). Otherwise, q is *merged* with the existing q^μ state (if there is one), making any incoming (respectively, outgoing) transitions to (respectively, from) q now point to (respectively, leave) q^μ. The state q is then removed (Lines 33-35). Finally, the created net is added to `nets`. In our example, after this step `nets` contains only $\mathcal{M}^{\mathbf{G}t}$ (Line 39).

4) (Lines 11-14) The monitor continuously gets an event e for evaluation (using `get_input`) from the system under inspection. If the current event marks the beginning of a new trace, then function `reset` re-initializes every LTL$_3$ monitor in the set `nets` by moving the token to their initial state. We note that in Lines 19 and 21, nets whose current state evaluates to $\perp$' are removed from `nets`. This is so that they are not re-initialized again. Next, a new set β' is initialized to β (the set of observation dependent formulas) to perform progression on formulas in β' without modifying the original formulas.

5) (Lines 15-21) On getting our first event (and thereafter on every event), the function `evaluate` performs transitions on every net in `nets` and moves the token to a new place q. If $\lambda(q) = \perp$, then we check for whether q is one of the q^μ states. If this is the case, we remove from `nets` any net whose '$\perp$' state is reached. In our example, since we currently have only monitor $\mathcal{M}^{\mathbf{G}t}$ in `nets`, if our event was indeed $\neg t$, then we remove this net from `nets`.

6) (Lines 22-23) Next, we iterate over all observation dependent sub-formulas in β' and β, such that $\mu' \in \beta'$ is the corresponding formula for $\mu \in \beta$. Formula μ' is progressed over e (and stored within β' itself) by applying the progression function $\boldsymbol{Pg}$ on μ'. By doing this, we are able to capture according to our running example, the initial state (proposition i_j) and subsequently the accepting state (proposition o_j) for an execution.

7) (Lines 24-25) Next, `progression_complete()` function returns whether or not the trace relation involving μ' has progressed for all the trace variables it was defined over. If so, an LTL$_3$ monitor is constructed for the progressed formula and added to the Petri net. In the example, when progression completes for the second trace (similarly for subsequent traces), then a new LTL$_3$ monitor net is created for the formula $i_2 \wedge \mathbf{XX}\ldots o_2$, whose state q, where $\lambda(q) = \perp$, is merged with q^μ, where $\mu = \pi_2 \sim_{f,P} \pi_3$ (recall that this state was introduced initially, for every $\mu \in \beta$).

8) (Lines 26-27) If all the input places of the transition to the output place $q_\perp$ contain a token which means that all sub-formuals were violated, the transition executes

and the monitor returns with the final evaluation $\bot$, meaning that the formula has been falsified.

Observe that, monitoring a co-k-hypersafety follows an identical algorithm except that (1) the state $q_\bot$ is replaced by $q_\top$, denoting satisfaction, (2) all $\bot$'s become $\top$'s, and (2) the token is placed in the final state of a net if it evaluates to $\top$.

Theorem 4. *Let φ be a k-safety* HYPERLTL$_1$ *formula. Algorithm 1 returns $\bot$ for an input set $M \in \mathcal{P}^*(\Sigma^*)$ iff $[M \models \varphi] = \bot$.*

Observation 1. *The complexity of Algorithm 1 to monitor a k-safety* HYPERLTL$_1$ *formula φ is*

$$O\left(\binom{n}{k} + \sum_{\phi \in \varphi} x^\phi\right)$$

where n is the number of finite executions and x^ϕ is the complexity of synthesizing a monitor for LTL *sub-formula ϕ in φ.* ■

Note that in Observation 1, the sub-formula ϕ can be an observation independant or observation dependant formula. In case it is the latter, the size of ϕ becomes directly dependent on the observed traces and function f in the trace relation. Thus, in some cases the size of ϕ can increase with an increasing length of the observed trace and with that there is an increase in the memory required for storing the net in the monitor for such a formula.

C. Monitoring beyond k-hypersafety

Monitoring of other monitorable HYPERLTL$_1$ formulas, such as the ones described in Table III, is straightforward using Algorithm 1:

- Formulas that are conjunctions or disjunctions of a safety hyperproperty with a co-safety hyperproperty, should be first reduced to monitoring of the monitorable part of the formula. For example, the formula $\forall \pi_1.\, \phi_1 \vee \exists \pi_2.\, \phi_2$, where ϕ_2 is a co-safety property, can be reduced to monitoring of only $\exists \pi_2.\, \phi_2$. This is because a formula with a $\forall$ quantifier can never be declared satisfied and due to the disjunction it reduces to checking for satisfaction of ϕ_2 by some trace. Hence, Algorithm 1 will monitor the reduced part only. Similarly, for the second row of Table III, since ϕ_1 is a safety property, its violation by a trace can be detected, which is sufficient to falsify the complete formula.
- Notice that, conjunctions or disjunctions of monitorable k-safety HYPERLTL$_1$ formulas can be monitored by enabling the transition to the final state of the Petri net either when all of the input places contain a token or at least one input place contains a token, respectively. Examples of such formulas are in row 4 of Tables I and II.

VI. IMPLEMENTATION AND RESULTS

A. Experimental Settings

Data sets: We use a dataset collected for a study at Microsoft Research [14]. The dataset involves GPS location data of 21 users taken over a period of eight weeks in the region of Seattle, USA. We also create synthetically generated datasets using Poisson, normal, and uniform distributions to ensure the robustness of our experiments. Each trace, in all of these datasets, corresponds to the continuous movement of a single user on a single day. The rationale behind using such traces is that a server might log locations of a user from the time a user opens an application until the time the user closes it. The traces are anonymized, that is, the trace itself does not reveal the identity of the user it belongs to. Each dataset consists of up to 200 finite traces with different lengths.

Security policies: We experiment with is three k-safety hyperproperties that specify the security, privacy, and anonymity of a user's GPS location data:

- *Anonymity* (GMNI) - If the initial location for a set of anonymized traces is the same, all traces must eventually reach the final location reported by any trace:

$$\forall \pi_1.\, \forall \pi_2.\, ((\mathbf{G}\lambda_H(\pi_1) \wedge \pi_1 \neq_H \pi_2 \wedge \pi_1[0] =_L \pi_2[0]) \Rightarrow \pi_1 \sim_{f,L} \pi_2)$$

  where f maps x, the final location in a trace, to LTL formula $\mathbf{F}x$.
- *Privacy* (OD) - Assuming the traces are deanonymized, the locations visited by a user must be the same in all the traces of the same user.

$$\forall \pi_1.\, \forall \pi_2.\, (\pi_1 \sim_{f,L} \pi_2)$$

  where function f maps every location x to $\mathbf{F}x$.
- *Security* - Suppose that visiting a set of k locations, uniquely identifies some secure information about a user. Then, over all traces the user must not report having visited all of these k locations.

$$\forall \pi_1 \cdots \forall \pi_k.\, (\neg \mathbf{F} l_1(\pi_1) \wedge \cdots \wedge \neg \mathbf{F} l_k(\pi_k))$$

  where $l_i(\pi_j)$ means that location l_i is revealed in trace π_j.

Metrics: The metrics used for evaluation are (1) the total number of generated LTL$_3$ monitor nets, and (2) the length of the progressed formulas. While both metrics directly represent memory overhead, they also indirectly characterize time time overhead as well.

B. Results and Analysis

For each of the distributions, we generate 100 synthetic datasets for evaluation. The plotted values are the means of the results obtained from all the datasets, for each distribution along with their error bars.

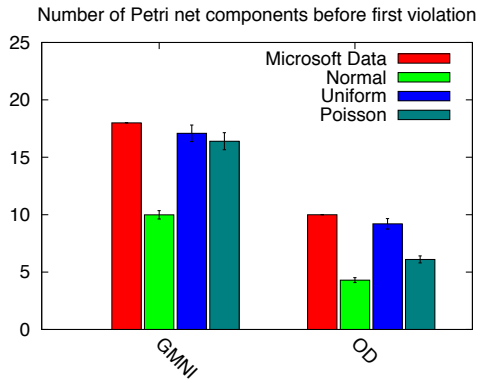


Fig. 5. Number of monitor nets generated before detection of first violation

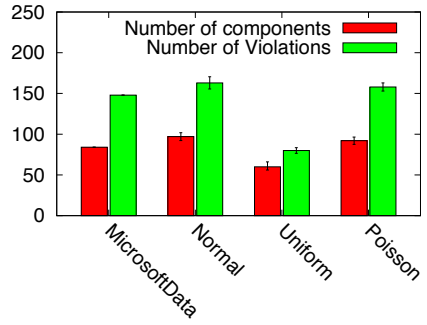
Number of monitor nets: Fig. 5 shows that the number of nets generated for GMNI before the first violation is detected is greater than OD. This is because GMNI is less strict than OD since the dependency is only on the first and last observed locations, whereas OD requires every location visited by a user in one trace to be visited in every other trace. This results in the property OD being violated in fewer observed traces. For the security property which consists of only observation-independent sub-formulas, the number of components remains a fixed number k for any number of traces and violations. Normal distribution shows lower number of nets generated in GMNI due to reduced probability of seeing the last location of a trace in another trace. For the OD property, the probability of seeing all the locations of one trace in another trace is reduced further for both normal and Poisson distributions. Hence, we see that a violation is detected sooner, resulting in fewer number of components being created.

We also let the algorithm report all the violations instead of terminating at the first violation. We evaluated the violation of property GMNI. Figure 6(a) shows that the number of violations reported were close to twice the number of nets created—which was less than 50% of the traces evaluated. Here we see that for the normal distribution the number of components created is much greater than the uniform distribution, as the probability of the same locations being visited among traces is higher for the uniform distribution due to which the number of unique progressed formulas are fewer. The total number of violations and components for evaluation of OD was significantly greater due to the property being more strict as explained earlier.

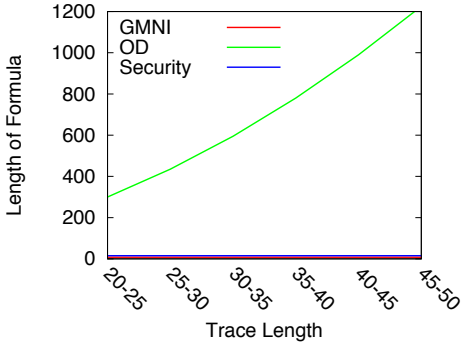
Length of progressed formulas: On comparing the length of formulas for each of the properties, OD formula is much longer as compared to both GMNI and security (see Table IV), as it captures every event observed by a trace. GMNI has a fixed, much smaller length since the formula is dependent only on the first and last observation. Thus, the length of the progressed formula remains the same for all trace lengths. Observation-independent sub-formulas result in a fixed length for security. We analyze the dependency of the length of the formula to be monitored on the length of traces for property OD. As the length of each trace increases the length of the formula increases (see Fig. 6(b)); i.e., a longer trace implies tracking of user location more frequently which results in an increasing length of the formula.

TABLE IV. TRACE LENGTH OF MONITORED FORMULAS BEFORE FIRST VIOLATION

GMNI	OD	security
8	238	12



(a) Total number of nets vs. total number of violations



(b) Length of formulas vs. trace length

Fig. 6. Memory consumption analysis.

VII. RELATED WORK

In this section, we discuss the current state-of-the-art work and techniques used in the past for runtime monitoring for security policies. We illustrate how our contributions improve upon or are different from any other related work.

A. Offline Verification

Basin et al. [2] develop a model checker for security protocols. Since traditional tools and verification methodologies are not equipped to deal with sets of traces, several results introduce new logics or operators to express hyperproperties. SecLTL extends LTL by using an additional *hide* modality [3]. It allows expression of non-interference as well as the instance until a high level data should remain independent of interference from low level data. The modal μ-calculus does not

suffice to express some information flow properties. Epistemic logic has been used to implicitly quantify over traces [18]. However, HYPERLTL and HYPERCTL* [4] subsume epistemic logic and quantified propositional temporal logic [19]. In [5], the authors introduce a model checking algorithm for verifying HYPERLTL formulas.

B. Static Analysis

Sabelfeld et al. [20] survey the literature focusing on static program analysis for enforcement of security policies. In some cases, with compilers using Just-in-time compilation techniques and dynamic inclusion of code at run time in web browsers, static analysis does not guarantee secure execution at run time. Type systems, frameworks for JavaScript [21] and ML [22] are some approaches to monitor information flow. Several tools [23], [24], [25] add extensions such as statically checked information flow annotations to Java language. Clark et al. [26] present verification of information flow for deterministic interactive programs. Our approach, on the other hand, is capable of monitoring a rich subset of k-safety hyperproperties and not just information flow without assistance from static analyzers.

C. Dynamic analysis

Russo et al. [27] concentrate on permissive techniques for the enforcement of information flow under flow-sensitivity. It has been shown that in the flow-insensitive case, a sound purely dynamic monitor is more permissive than static analysis. However, they show the impossibility of such a monitor in the flow-sensitive case. A framework for inlining dynamic information flow monitors has been presented by Magazinius et al. [28]. The approach by Chudnov et al. [29] uses hybrid analysis instead and argues that due to JIT compilation processes, it is no longer possible to mediate every data and control flow event of the native code. They leverage the results of Russo et al. [27] by inlining the security monitors. Chudnov et al. [30] again use hybrid analysis of 2-safety hyperproperties in relational logic. They check for violation on observing a single run that they call a 'major' trace, which is monitored with alternate 'minor' traces. Hybrid analysis uses the goodness of static analysis and combines it with dynamic analysis. However, dynamic languages like JavaScript make such approaches impractical.

Austin et al. [31] implement a purely dynamic monitor, however, restrictions such as "no-sensitive upgrade" were placed. Some techniques deploy taint tracking and labelling of data variables dynamically [32], [33]. Zdancewic et al. [34] verify information flow for concurrent programs. Decker et al. [35] provide verification techniques for first-order theories for reasoning about data that can be applied to check for secure execution of multi-threaded, object oriented systems. Most of the techniques cited above aim to monitor security policies that are 2-safety hyperproperties, on observing a single run, whereas, our work is for any k-safety hyperproperty, when multiple runs are observed.

D. SME

Secure multi-execution [36] is a technique to enforce non-interference. In SME, one executes a program multiple times, once for each security level, using special rules for I/O operations. Outputs are only produced in the execution linked to their security level. Inputs are replaced by default inputs except in executions linked to their security level or higher. Input side effects are supported by making higher-security-level executions reuse inputs obtained in lower-security-level threads. This approach is sound in a deterministic language.

While there are small similarities between SME and our work, there are fundamental differences. Firstly, SME only focuses on non-interference, while k-safety hyperproperties cover a significantly richer class of security policies. Secondly, SME aims at enforcing non-interference while our method monitors k-safety hyperproperties; i.e., there is no enforcement. So, SME enforces a security policy at the cost of restricting what it can enforce, whereas our technique monitors a much larger set of policies.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we focused on runtime verification of a rich class of security policies. Our specification language is a subset of k-safety/co-safety hyperproperties which allows expressing policies that are not trace-based (e.g., information flow). First, in order to have syntactic means, we characterized k-(co)safety hyperproperties in HYPERLTL, a temporal logic that allows explicit quantification over execution traces. Then, we generalized LTL_3 [13] (a logic designed for runtime verification of LTL) to the context of HYPERLTL and defined its notion of monitorability and identified different classes of monitorable HYPERLTL formulas by syntactic means. Finally, we introduced a runtime verification algorithm for monitoring k-safety/co-k-safety hyperproperties and studied its performance with respect to different metrics.

There are numerous interesting research avenues to extend this work. Generalizing this work to a distributed monitoring framework (e.g., [37]) to monitor hyperproperties in a distributed setting is a highly challenging task. Another open problem is to design a technique to monitor general (unbounded) hypersafety as well as hyperliveness properties. Finally, one can monitor hyperproperties by analyzing execution as well as an abstract model of the system at run time. The latter idea is especially beneficial for monitoring hyperliveness properties. Runtime enforcement of hyperproperties is another interesting open problem.

IX. ACKNOWLEDGMENT

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grants 430575-2012 and 463324-2014.

REFERENCES

[1] M. Burrows, M. Abadi, and R. M. Needham, "A logic of authentication," *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 18–36, 1990.

[2] D. A. Basin, S. Mödersheim, and L. Viganò, "Ofmc: A symbolic model checker for security protocols," *International Journal of Information Security*, vol. 4, no. 3, pp. 181–208, 2005.

[3] R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl, "Model checking information flow in reactive systems," in *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2012, pp. 169–185.

[4] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal logics for hyperproperties," in *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, 2014, pp. 265–284.

[5] B. Finkbeiner, M. N. Rabe, and C. Sanchez, "Algorithms for model checking HyperLTL and HyperCTL*," in *Proceedings of the 27th International Conference on Computer-Aided Verification (CAV)*, 2015, to appear.

[6] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, "Applying formal methods to a certifiably secure software system," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 82–98, 2008.

[7] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[8] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, pp. 30–50, February 2000.

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

[10] B. Alpern and F. B. Schneider, "Defining liveness," *Inf. Process. Lett.*, vol. 21, no. 4, pp. 181–185, 1985.

[11] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *SAS*, 2005, pp. 352–367.

[12] A. Pnueli and A. Zaks, "PSL model checking and run-time verification via testers," in *Proceedings of the 14th International Symposium on Formal Methods (FM)*, 2006, pp. 573–586.

[13] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, p. 14, 2011.

[14] J. Krumm and A. Brush, "MSR GPS privacy dataset 2009," 2009. [Online]. Available: Retrievedfromhttp://research.microsoft.com/~jckrumm/GPSData2009

[15] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001.

[16] K. Havelund and G. Rosu, "Monitoring Programs Using Rewriting," in *Automated Software Engineering (ASE)*, 2001, pp. 135–143.

[17] F. Bacchus and F. Kabanza, "Planning for temporally extended goals," *Ann. Math. Artif. Intell.*, vol. 22, no. 1-2, pp. 5–27, 1998.

[18] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, "Reasoning about knowledge: A response by the authors," *Minds and Machines*, vol. 7, no. 1, p. 113, 1997.

[19] A. P. Sistla, M. Y. Vardi, and P. Wolper, "The complementation problem for büchi automata with appplications to temporal logic," *Theoretical Computer Science*, vol. 49, pp. 217–237, 1987.

[20] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[21] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for javascript," in *Proceedings of PLDI*, 2009, pp. 50–62.

[22] F. Pottier and V. Simonet, "Information flow inference for ml," in *Proceedings of Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 2002, pp. 319–330.

[23] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 1999, pp. 228–241.

[24] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, P. L. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*

[25] A. C. Myers and B. Liskov, "Complete, safe information flow with decentralized labels," 1998.

[26] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *Proceedings of Formal Aspects in Security and Trust*, 2008, pp. 50–66.

[27] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proceedings of the XXrd IEEE Computer Security Foundations Symposium (CSF)*, 2010, pp. 186–199.

[28] J. Magazinius, A. Russo, and A. Sabelfeld, "On-the-fly inlining of dynamic security monitors," *Computers & Security*, vol. 31, no. 7, pp. 827–843, 2012.

[29] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *Proceedings of CSF*, 2010, pp. 200–214.

[30] A. Chudnov, G. Kuan, and D. A. Naumann, "Information flow monitoring as abstract interpretation for relational logic," in *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, 2014, pp. 48–62.

[31] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *ACM Transactions on Programming Languages and Systems*, 2009, pp. 113–124.

[32] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Privacy scope: A precise information flow tracking system for finding application leaks," EECS Department, University of California, Berkeley, Tech. Rep., Oct 2009.

[33] S. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum, "A virtual machine based information flow control system for policy enforcement," vol. 197, no. 1, pp. 3–16, 2008.

[34] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *Computer Security Foundations Workshop*, 2003, pp. 29–.

[35] N. Decker, M. Leucker, and D. Thoma, "Monitoring modulo theories," in *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014, pp. 341–356.

[36] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *31st IEEE Symposium on Security and Privacy, S&P*, 2010, pp. 109–124.

[37] M. Mostafa and B. Bonakdarpour, "Decentralized runtime verification of LTL specifications in distributed systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015, pp. 494–503.

APPENDIX

Lemma 1. *The complement of a safety hyperproperty is a co-safety hyperproperty and vice versa. Also, the complement of a k-safety hyperproperty is a co-k-safety hyperproperty and vice versa.*

Proof: Let $\boldsymbol{S}$ be a safety hyperproperty and $\bar{\boldsymbol{S}}$ be its complement set. Let $\boldsymbol{M_h}$ be the bad set of finite sets of finite traces, such that for each $M \in \boldsymbol{M_h}$ and any $T \in \mathcal{P}(\Sigma^\omega)$, where $M \leq T$, we have $T \notin \boldsymbol{S}$. This means that $T \in \bar{\boldsymbol{S}}$ and, hence, every infinite extension of M is in $\bar{\boldsymbol{S}}$. Since any $T \in \bar{\boldsymbol{S}}$ can be associated with such an M, hyperproperty $\bar{\boldsymbol{S}}$ is indeed a co-safety hyperproperty. In fact, the bad thing in $\boldsymbol{S}$ (i.e., set M) becomes the good thing in $\bar{\boldsymbol{S}}$. Finally, if $\boldsymbol{S}$ is a k-safety hyperproperty, since $|M| \leq k$, then $\bar{\boldsymbol{S}}$ trivially becomes a co-k-safety hyperproperty. The other direction from co-hypersafety to hypersafety trivially follows similar proof structure. ■

Lemma 2. *Consider a* HYPERLTL$_1$ *formula of the following form:*

$$\varphi_{\boldsymbol{C_k}} = \exists \pi_1 \cdots \exists \pi_k. (\phi_1(\pi_1, \ldots, \pi_k) \wedge \cdots \wedge \phi_k(\pi_1, \ldots, \pi_k))$$

where $\phi_1, \ldots, \phi_k \in$ LTL$_C$. *Such a formula represents a co-*k*-safety hyperproperty.*

Proof: Let $\boldsymbol{C_k}$ be the hyperproperty (i.e., the set of sets of traces) that represents $\varphi_{\boldsymbol{C_k}}$. We will show that $\boldsymbol{C_k}$ is a co-k-safety hyperproperty. We need to show that for any $T \in \boldsymbol{C_k}$, there is a finite set of finite traces M, such that any infinite continuation of M is in $\boldsymbol{C_k}$. From the semantics of HYPERLTL, we know that $\forall T \in \boldsymbol{C_k}$, there is a Π such that $\Pi \models_T \varphi_{\boldsymbol{C_k}}$. Therefore, there exist infinite traces $t_1, \ldots, t_k \in T$ that satisfy $\phi_1, \ldots, \phi_k$. Since, $\phi_1, \ldots, \phi_k$ are co-safety properties, there exists an m_i $(1 \leq i \leq k)$ for each $t_i \models \phi_i$, such that

$$\forall t \in \Sigma^\omega. (m_i \leq t \;\Rightarrow\; t \models \phi_i)$$

Now observe that for the set M of all such m_i, any infinite continuation T' of M will satisfy $\varphi_{\boldsymbol{C_k}}$ and hence $T' \in \boldsymbol{C_k}$. Hence, $\boldsymbol{C_k}$ is a co-safety hyperproperty. Finally, since $\varphi_{\boldsymbol{C_k}}$ involves only k trace variables, $\boldsymbol{C_k}$ is a co-k-safety hyperproperty. ■

Corollary 1. *Consider a* HYPERLTL$_1$ *formula of the following form:*

$$\varphi_{\boldsymbol{S_k}} = \forall \pi_1 \cdots \forall \pi_k. (\phi_1(\pi_1, \ldots, \pi_k) \vee \cdots \vee \phi_k(\pi_1, \ldots, \pi_k))$$

where $\phi_1, \ldots, \phi_k \in$ LTL$_S$. *Such a formula represents a* k*-safety hyperproperty.*

Proof: First, notice that $\neg\varphi_{\boldsymbol{C_k}}$ in Lemma 2 will exactly have the syntax of $\varphi_{\boldsymbol{S_k}}$, where each $\neg\phi_i$ is a safety property. Now, observe that in Lemma 2, $\varphi_{\boldsymbol{C_k}}$ gives the syntactic representation of the co-k-safety hyperproperty $\boldsymbol{C_k}$. It follows that $\neg\varphi_{\boldsymbol{C_k}}$ will be the syntactic representation of a k-safety hyperproperty $\boldsymbol{S_k}$ (from Lemma 1). ■

Theorem 1. *Conjunction (respectively, disjunction) of* HYPERLTL$_1$ *formulas, with at most* k *quantifiers, given by* $\varphi_{\boldsymbol{S_k}}$ *in Corollary 1 (respectively,* $\varphi_{\boldsymbol{C_k}}$ *in Lemma 2), is a* k*-hypersafety property (respectively, co-*k*-hypersafety property).*

Proof: Let's consider a disjunction of HYPERLTL$_1$ formulas

$$\varphi_{\boldsymbol{C}} = \varphi_1 \;\vee\; \ldots \;\vee\; \varphi_n$$

where φ_i $(1 \leq i \leq n)$ is a closed HYPERLTL$_1$ formula representing a co-k-hypersafety property $\boldsymbol{C_i}$ as given by Lemma 2. The union of a set of co-safety hyperproperties remains a co-safety hyperproperty, which translates to taking disjunction of the HYPERLTL$_1$ representations of these properties. Hence, $\varphi_{\boldsymbol{C}}$ is a co-k-hypersafety.

Similarly, for k-hypersafety consider a conjunction of HYPERLTL$_1$ formulas

$$\varphi_{\boldsymbol{S}} = \varphi_1 \;\wedge\; \ldots \;\wedge\; \varphi_n$$

where φ_i $(1 \leq i \leq n)$ is a closed HYPERLTL$_1$ formula representing a k-hypersafety property $\boldsymbol{S_i}$ as given by Corollary 1. The intersection of a set of safety hyperproperties remains a safety hyperproperty, which translates to taking conjunction of the HYPERLTL$_1$ representations of these properties. Hence, $\varphi_{\boldsymbol{S}}$ is a k-hypersafety. ■

Theorem 2. *Every* k*-safety hyperproperty and every co-*k*-safety hyperproperty that satisfies Theorem 1 is monitorable.*

Proof: The proof follows from whether $\exists M \in \mathcal{P}^*(\Sigma^*)$ satisfies Definition 8 for a k-safety or co-k-safety hyperproperty, that is syntactically represented in HYPERLTL$_1$ by formulas given in Theorem 1. The runtime evidence for these hyperproperties in Tables I and II shows that such an M indeed exists for every k-safety and co-k-safety hyperproperty. ■

Theorem 4. *Let* φ *be a* k*-safety* HYPERLTL$_1$ *formula. Algorithm 1 returns* $\perp$ *for an input set* $M \in \mathcal{P}^*(\Sigma^*)$ *iff* $[M \models \varphi] = \perp$.

Proof: Let $\varphi = \forall \pi_1 \ldots \forall \pi_k. \phi_1(\pi_1) \vee \cdots \vee \phi_k(\pi_k)$.

- ($\Rightarrow$) For an input set $M \in \mathcal{P}^*(\Sigma^*)$, where $[M \models \varphi] = \perp$, by contradiction, let us assume that Algorithm 1 returns '?'. The antecedent implies that for all ϕ_i where $(1 \leq i \leq k)$, there exists $m \in M$ such that any extension of m violates ϕ_i. If the algorithm has not yet returned $\perp$, then there exists at least one component M^ϕ of the Petri net that has not yet reached state q such that $\lambda^\phi(q) = \perp$. From Definition 9, we know that the component M^ϕ in the Petri net reports violation if $[m \models \phi_i] = \perp$ contradicting that even though m is observed, component M^ϕ does not report violation. Therefore, if $[M \models \varphi] = \perp$, then Algorithm 1 returns $\perp$.
- ($\Leftarrow$) If Algorithm 1 returns $\perp$, by contradiction, let us assume that $[M \models \varphi] \neq \perp$. The antecedent implies that all input places q^ϕ that have a transition to $q_\perp$, i.e., $(q, true, q_\perp)$, contain a token. By construction of component M^ϕ (Definition 9), on running m over M^ϕ state q^ϕ, such that $(q^\phi, true, q_\perp)$, is reached iff $[m \models \phi] = \perp$. Therefore, for each ϕ_i, where $(1 \leq i \leq k)$, there exists $m_i \in M$, such that $[m \models \phi_i] = \perp$. Therefore, for a trace assignment Π and $\forall T \in \mathcal{P}(\Sigma^\omega)$ such that $\pi_i \rightarrow m_i t$ for some $t \in \Sigma^\omega$, we have $m_i t \in T$, and for all $1 \leq i \leq k$, we have $\Pi \not\models_T \varphi$. This contradicts the assumption that $[M \models \varphi] \neq \perp$ from Definition 7. Hence, if Algorithm 1 returns $\perp$ then $[M \models \varphi] = \perp$.

■