

Mining Hyperproperties using Temporal Logics

EZIO BARTOCCI, TU Wien, Austria

CRISTINEL MATEIS, AIT Austrian Institute of Technology, Austria

ELEONORA NESTERINI, TU Wien, Austria and AIT Austrian Institute of Technology, Austria

DEJAN NIČKOVIĆ, AIT Austrian Institute of Technology, Austria

Formal specifications are essential to express precisely systems, but they are often difficult to define or unavailable. Specification mining aims to automatically infer specifications from system executions. The existing literature mainly focuses on learning properties defined on single system executions. However, many system characteristics, such as security policies and robustness, require relating two or more executions, and hence cannot be captured by properties. Hyperproperties address this limitation by allowing simultaneous reasoning about multiple executions with quantification over system traces.

In this paper, we propose an effective approach for mining Hyper Signal Temporal Logic (HyperSTL) specifications. Our approach is based on the syntax-guided synthesis framework and allows users to control the amount of prior knowledge embedded in the mining procedure. To the best of our knowledge, this is the first mining method for hyperproperties that does not require a pre-defined template as input and allows for quantifier alternation. We implemented our approach and demonstrated its applicability and versatility in several case studies where we showed that we can use the same method to mine specifications both with and without templates, but also to infer subsets of HyperSTL, including STL, HyperLTL, LTL and non-temporal specifications.

CCS Concepts: • **Software and its engineering** → **Specification languages; Software reverse engineering; Computer systems organization** → **Embedded and cyber-physical systems.**

Additional Key Words and Phrases: Specification Mining, Hyperproperties, Cyber-Physical Systems

ACM Reference Format:

Ezio Bartocci, Cristinel Mateis, Eleonora Nesterini, and Dejan Ničković. 2023. Mining Hyperproperties using Temporal Logics. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2023), 25 pages. <https://doi.org/10.1145/3609394>

1 INTRODUCTION

The development of complex systems typically starts from a collection of requirements. Their role is central in the lifecycle of software and embedded systems. In the early stages of the design, specification languages provide the framework to formalize requirements and exchange information between development teams rigorously and unambiguously: formal requirements can precisely capture the expected system behavior. Engineers use these requirements to verify safety-critical properties over a system's model: for example, to test [8] and debug [7] its implementation, and

This article appears as part of the ESWEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2023

Authors' addresses: Ezio Bartocci, TU Wien, Vienna, Austria, ezio.bartocci@tuwien.ac.at; Cristinel Mateis, AIT Austrian Institute of Technology, Vienna, Austria, cristinel.mateis@ait.ac.at; Eleonora Nesterini, TU Wien, Vienna, Austria and AIT Austrian Institute of Technology, Vienna, Austria, eleonora.nesterini@tuwien.ac.at; Dejan Ničković, AIT Austrian Institute of Technology, Vienna, Austria, dejan.nickovic@ait.ac.at.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2023 Copyright held by the owner/author(s).

1539-9087/2023/1-ART1

<https://doi.org/10.1145/3609394>

to localize the root causes of faults [6]. Finally, formal specifications can be used to synthesize monitors [4, 5] that observe the system during its operation and detect violations of requirements, alerting the user or even taking corrective action.

Despite their importance, formal specifications are often partially available. Specification mining [9] refers to a collection of methods that infer potential system properties by observing its executions. The mainstream approaches in specification mining generate formulas in Linear Temporal Logic [25] (LTL) and its continuous-time real-valued extensions such as Signal Temporal Logic [20] (STL). A recent survey in [9] summarized the main techniques for learning STL properties, classifying them according to different characteristics. There are two main approaches to mine specifications [9]: in *template-based* mining [3] (see also the tool Texada [18, 19] for LTL), the specification skeleton is given to the mining procedure, which aims at completing the template with missing elements in a way that is consistent with the observed system executions. The missing parts in the template can be propositions or timing and amplitude parameters. In contrast, *template-free* mining infers the entire structure of the formula [10, 17, 21, 23], including its parameters.

Most approaches for mining requirements focus on learning trace-based properties from a set of traces. However, some critical system characteristics, such as the system robustness or information-flow security requirements, require simultaneous reasoning about multiple traces and cannot be specified as trace properties. Hyperproperties [12] overcome this restraint by allowing specification of properties of trace sets. HyperLTL [11] and HyperSTL [24] are popular specification languages designed to express an essential class of hyperproperties that extend LTL and STL with trace quantifiers respectively.

This paper considers the problem of mining hyperproperties as HyperSTL/HyperLTL formulae. Only a few works in the literature address this problem and only for a very restricted class of hyperproperties. In particular, none of the existing works allows for quantifier alternation. However, in the literature, there are hyperproperties that require quantifier alternation. For example, the information-flow security policies of *noninterference* and of *generalized noninterference* [12] state that for each pair of traces there must exist another trace sharing the same values of secret input variables with the first trace and the same values of public output variables with the second trace. The key idea is that users cannot infer secret information from observing the system's outputs.

Related Work. In [27], the authors proposed an approach to learning the relationship between inputs and outputs of different executions of one or more (non-temporal) functions allowing only one possible (universally quantified) template. Another related work is [26] where the authors present a template-based approach to mine HyperLTL specifications. Our technique differs from this method because it allows for quantifier alternation, while [26] relies on enumerating candidate formulas without this feature. In [15], the authors propose an extension of the L^* algorithm [2] that actively mines the universally-safe fragment of HyperLTL. This fragment consists of HyperLTL formulas that are universally-quantified and until-free. Thus, this method supports a specification language with limited expressivity compared to our approach.

Our contributions. We propose a new procedure for mining hyperproperties from a fixed set of execution traces. The goal is to learn (hyper)properties that this set of positive examples satisfies. The main contributions of our approach are:

- Our work is the first to learn hyperproperties [12], expressed in HyperSTL (including formulas with quantifier alternations), STL, LTL, and HyperLTL as special cases. Thus, our approach represents a step forward to the recent state-of-the-art in [9].
- We have integrated the *syntax-guided synthesis* (SyGuS) [1] in the process of mining temporal logic formulas to guide the search in the space of candidate formulas. More specifically, in addition to the execution traces, our mining procedure takes a grammar that defines the

search space of valid specifications as additional input. We chose HyperSTL, an expressive specification language for hyperproperties, as the most general template. The grammar can also be restrained, e.g. by bounding the number of trace quantifiers, restricting the use of logical and temporal operators, or fixing the predicate parameters. This flexible mechanism for constraining the space of candidate formulas has multiple advantages. First, it allows the engineer to fine-tune the addition of domain knowledge that helps to infer useful formulas while not over-constraining the search. Second, to the best of our knowledge, this approach is the first one that unifies template-free and template-based mining requirements in temporal logics (the closest related attempt is the work in [22] that can handle only quantifier-free first-order logic formulas). Finally, it allows the same method to infer versatile system characteristics such as the correctness properties and the security hyperproperties. In the case of HyperSTL we perform both the parameter and structural syntheses.

- We have extensively evaluated our approach on multiple case studies learning different specifications, including LTL, STL, HyperLTL, HyperSTL, and Hyper-propositional formulas. Furthermore, we compare our results with those obtained from two other state-of-the-art tools: Texada [18, 19] and HyperMiner [26].
- We introduced two heuristics to improve the efficiency of the mining process: (i) we leverage the intrinsic properties of quantifiers to avoid checking all the possible tuples of traces; (ii) we use the correctness properties of HyperSTL/STL to minimize the number of monitoring checks, thus speeding up the overall mining process. As the monitoring of HyperLTL/HyperSTL properties represents an important step in our mining procedure, these heuristics play a crucial step in the applicability of our approach. Authors in [16] proposed an automata-based method to monitor temporal hyperproperties, which is not applicable in our setting as it does not support either HyperSTL or alternating quantifiers.

Outline. In Sec. 2, we provide the necessary background on HyperSTL [24]. In Sec. 3, we present the main steps of our approach to mine HyperLTL formulas, while in Sec. 4, we extend it to learn also HyperSTL specifications. Sec. 5 provides an extensive assessment of our approach to different case studies, comparing the results obtained with our method with those obtained using other tools. We conclude and discuss future work in Sec. 6.

2 PRELIMINARIES

Definition 2.1 (Trace and Set of Traces [24]). Let $\mathbb{T} \subseteq \mathbb{R}_{\geq 0}$ be the time domain. A *trace* with q real-valued variables and j Boolean variables $\omega = (t_1, \omega_1), (t_2, \omega_2), \dots, (t_k, \omega_k)$ is a finite sequence of (time, value) pairs, where $t_i \in \mathbb{T}$, $t_i < t_{i+1}$ for every $i \in \{1, \dots, k-1\}$, and $\omega_i \in \mathbb{D} = \mathbb{R}^q \times \mathbb{B}^j$ is a vector of real and Boolean values. We denote by $|\omega| = k$ the length of ω . We indicate by $\omega[v]$ the projection of trace ω on variable v . We will slightly abuse notation and also denote by $\omega[v]$ the signal $\mathbb{T} \rightarrow \mathbb{R}$ or $\mathbb{T} \rightarrow \mathbb{B}$ (depending on whether v is a real or Boolean variable), such that $\omega[v](t) = \omega_i[v]$ if $t \in [t_i, t_{i+1})$ with $i < k$, or $\omega[v](t) = \omega_k[v]$ if $t \geq t_k$. Given a set of traces T and a set of *trace variables* Π , we denote by $\Pi_T : \Pi \rightarrow T$ a *trace variable assignment*¹ that maps each trace variable to a concrete trace.

Definition 2.2 (HyperSTL [24]). HyperSTL is a hyper-temporal logic interpreted over real- and Boolean-valued traces that is captured by the following syntax:

$$\begin{aligned} \psi &:= \exists \pi. \psi \mid \forall \pi. \psi \mid \varphi \\ \varphi &:= \pi[b] \mid f(\pi[x_1], \dots, \pi[x_q]) < c \mid \neg \varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2 \mid \varphi_1 \mathbf{S}_I \varphi_2 \end{aligned}$$

¹In this work, we assume the *trace variable assignment* is an injective function associating different trace variables with different concrete traces.

where X is a set of real-valued variables, B is a set of Boolean variables, the trace variable $\pi \in \Pi$ ranges over the set $B \cup X$ of variables, $b \in B$, $x_1, \dots, x_q \in X$, $f : \mathbb{R}^q \rightarrow \mathbb{R}$ is a real-valued function, I is an interval in $\mathbb{R}_{\geq 0} \cup +\infty$, and $c \in \mathbb{R}$.

Given $t \in \mathbb{T}$, we define the semantics of HyperSTL in terms of a *robustness* function ρ :

$$\begin{aligned}
\rho(\exists \pi. \psi, \Pi_T, t) &= \max_{\omega \in T} \rho(\psi, \Pi_T[\pi \rightarrow \omega], t) \\
\rho(\forall \pi. \psi, \Pi_T, t) &= \min_{\omega \in T} \rho(\psi, \Pi_T[\pi \rightarrow \omega], t) \\
\rho(\neg \varphi, \Pi_T, t) &= -\rho(\varphi, \Pi_T, t) \\
\rho(\varphi_1 \vee \varphi_2, \Pi_T, t) &= \max(\rho(\varphi_1, \Pi_T, t), \rho(\varphi_2, \Pi_T, t)) \\
\rho(\varphi_1 \mathbf{U}_I \varphi_2, \Pi_T, t) &= \sup_{t' \in (t \oplus I) \cap [0, k]} \min(\inf_{t'' \in [t, t']} \rho(\varphi_1, \Pi_T, t''), \rho(\varphi_2, \Pi_T, t')) \\
\rho(\varphi_1 \mathbf{S}_I \varphi_2, \Pi_T, t) &= \sup_{t' \in (t \ominus I) \cap [0, k]} \min(\inf_{t'' \in (t', t]} \rho(\varphi_1, \Pi_T, t''), \rho(\varphi_2, \Pi_T, t')) \\
\rho(f(\pi[x_1], \dots, \pi[x_q]) < c, \Pi_T, t) &= c - f(\Pi_T(\pi)[x_1](t), \dots, \Pi_T(\pi)[x_q](t)) \\
\rho(\pi[b], \Pi_T, t) &= +\infty \text{ if } \Pi_T(\pi)[b](t), -\infty \text{ otherwise}
\end{aligned}$$

where \oplus and \ominus denote the Minkowski sum and difference, respectively. We note that we can interpret HyperSTL both over dense and discrete-time em (in this case we consider the interval in the discrete domain), and that the latter interpretation will be used notably when considering HyperLTL and LTL fragments.

From this basic definition of HyperSTL, we can derive the other operators as usual: **true** $= \varphi \vee \neg \varphi$, **false** $= \neg \mathbf{true}$, $\varphi_1 \wedge \varphi_2 = \neg(\neg \varphi_1 \vee \neg \varphi_2)$, $\varphi_1 \rightarrow \varphi_2 = \neg \varphi_1 \vee \varphi_2$, $\mathbf{F}_I \varphi = \mathbf{true} \mathbf{U}_I \varphi$, $\mathbf{G}_I \varphi = \neg \mathbf{F}_I \neg \varphi$, $\mathbf{O}_I = \mathbf{true} \mathbf{S}_I \varphi$, $\mathbf{H}_I = \neg \mathbf{O}_I \neg \varphi$. The untimed temporal operators **U**, **S**, **F**, **G**, **O** and **H** are obtained by setting the time interval I to $[0, \infty)$. The *next* and *previous* operators are then (in the discrete-time interpretation of the logic) equivalent to $\mathbf{X} \varphi = \mathbf{F}_{[1,1]} \varphi$ and $\mathbf{Y} \varphi = \mathbf{O}_{[1,1]} \varphi$.

By appropriate restrictions of the HyperSTL syntax, we can also obtain HyperLTL, STL and LTL specification languages. HyperLTL is obtained by restricting the temporal operators to untimed ones only. The STL (LTL) formulas correspond to HyperSTL/HyperLTL variants of the form $\forall \pi. \varphi(\pi)$.

PROPOSITION 2.3 (SOUNDNESS [24]). *Given an HyperSTL formula ψ and a trace variable assignment Π_T , when $\rho(\psi, \Pi_T, t) > 0$ the formula is satisfied, while if $\rho(\psi, \Pi_T, t) < 0$ the formula is violated.*

PROPOSITION 2.4 (CORRECTNESS OF STL FRAGMENT [13]). *Given an STL formula expressed as the HyperSTL fragment φ (without trace quantifiers) and two traces ω and ω' over the same time domain, if ω at time t satisfies φ and $\|\omega - \omega'\|_\infty < \rho(\varphi, \omega, t)$, then also ω' satisfies φ at time t .*

PROPOSITION 2.5 (PARAMETRIC FORMULA (ADAPTED FROM [3])). *An HyperSTL formula φ is parametric if at least one of its constants (in the numerical predicates or time bounds of the temporal operator intervals) is replaced by a parameter variable. Given a parametric formula φ with a parameter p , we denote by $\varphi[p \rightarrow v]$ the instantiation of the parameter p in φ with the constant v . A formula φ is said to be monotonically increasing in its parameter p_i if, for every pair of parameter valuations v_i and v'_i such that $v_i \leq v'_i$ and for every trace ω , the following condition on the robustness values is satisfied:*

$$\rho(\varphi[p_i \rightarrow v_i], \omega, t) \leq \rho(\varphi[p_i \rightarrow v'_i], \omega, t).$$

Analogously, φ is said to be monotonically decreasing if the previous condition holds for each pair of parameter valuations v_i and v'_i such that $v_i \geq v'_i$. Monotonicity can be automatically checked using polarity rules in [3].

3 MINING HYPERLTL

In this section, we propose a method for mining HyperLTL that leverages the stochastic search of the syntax-guided synthesis (SyGuS) framework [1] developed in the program synthesis research field. In the original setting, SyGuS aims to find a program that satisfies a given correctness

requirement among all the acceptable programs defined by the grammar. Similarly, we desire to learn a (hyper)property consistent with a set of positive examples chosen from a group of admissible formulas expressed by the user-defined grammar.

This grammar-based approach to specification mining brings many advantages. By allowing the restriction of the grammar, it admits both template-free and template-based mining. In the latter case, the user can control the number of restrictions the grammar imposes. In this section, we assume HyperLTL as the baseline grammar and consider any additional constraints as a step toward building a template.

We illustrate how we can use basic knowledge about the system under investigation to restrict HyperLTL and steer the search toward meaningful characteristics. For many systems, we know their interface and can distinguish between input and output variables. Many significant properties define the temporal relations between the input and the output behavior. The subclass of HyperLTL that characterizes the input/output relations holding throughout the whole execution can be expressed as follows:

$$\begin{aligned}\psi &:= \exists \pi. \psi \mid \forall \pi. \psi \mid \gamma \\ \gamma &:= \mathbf{G}\varphi[I] \rightarrow \mathbf{G}\varphi[O] \\ \varphi[Y] &:= \pi[b] \mid \neg\varphi[Y] \mid \varphi_1[Y] \wedge \varphi_2[Y]\end{aligned}\tag{1}$$

where B is the set of Boolean variables partitioned into input $I \subset B$ and output $O \subseteq B$ variables, $Y \subset B$ is a subset of variables and $b \in Y$. We can further refine the grammar to restrict the scope of mining to the important hyperproperty of *observational determinism*, stating that the system appears deterministic with respect to its inputs:

$$\forall \pi \forall \pi' \quad \mathbf{G}\left(\bigwedge_{\text{in} \in I} \pi[\text{in}] = \pi'[\text{in}]\right) \rightarrow \mathbf{G}\left(\bigwedge_{\text{out} \in O} \pi[\text{out}] = \pi'[\text{out}]\right).\tag{2}$$

In Section 3.1, we describe the algorithm for mining HyperLTL from a set of traces. We propose a measure to quantify the satisfaction of temporal hyperproperties in Section 3.2 and an efficient monitoring technique in Sections 3.3.

3.1 Syntax-Guided Synthesis for Mining Hyperproperties

In Algorithm 1, we describe our adaptation of the SyGuS approach [1] for mining HyperLTL. The procedure takes as input a grammar G that is a subset of HyperLTL, a set of traces (positive examples) T and the target length l of the mined specification².

The algorithm first uniformly samples one candidate specification ψ from the set of all hyperproperties generated by G having m quantifiers and size l (Line 1). Then, ψ is associated with a score s (Line 2), which quantifies the quality of the candidate formula for the dataset T . The score is defined as $s = \exp(0.5 \cdot F(\psi, T))$, where 0.5 is a smoothing value and F is any *fitness function* that gives a (strictly) positive number if ψ is satisfied by T , and a number smaller or equal to zero if it is violated. We propose a concrete fitness function in Section 3.2. Alternatively, the Boolean satisfaction returned by a monitoring algorithm (as the one that we propose in Section 3.3) can be used as *fitness function*. If ψ is consistent with T , i.e. its score is greater than 1, then ψ is a valid solution and we are done. Else, we iteratively mutate ψ with Algorithm 2 (detailed later on) until we either find a variant consistent with T or exceed the maximum number of iterations N_{\max} (Lines 4-12). For the mutated formula ψ' , we compute the so-called *Metropolis-Hastings acceptance ratio*, defined as $M(\psi, \psi') = \min\left(1, \frac{\text{score}(\psi')}{\text{score}(\psi)}\right)$. $M(\psi, \psi')$ corresponds to the probability for ψ' to replace the current candidate ψ . The key idea is to always accept changes that improve the score of the current formula, while changes that decrease the score have a certain probability of being rejected (they are

²The length is affected only by the number of propositions and the operators, but not by the number of quantifiers.

not necessarily rejected to allow for exploration). In the limit, the Metropolis-Hastings procedure is guaranteed to sample formulas with probability proportional to their scores. Consequently, this procedure will eventually draw satisfied formulas (if they exist). However, for practical reasons, we impose a stopping condition after N_{\max} iterations, after which we consider the learning process failed. In our experiments, N_{\max} is set to 500.

Algorithm 1: Stochastic search for HyperLTL mining

Input: HyperLTL (sub)grammar G , set of traces T , formula length l , number of quantifiers m , maximum number of iterations N_{\max}

Output: HyperLTL formula ψ generated from G and satisfied by T

```

1  $\psi \leftarrow \text{sample\_hyperproperty}(G, l, m)$ 
2  $s \leftarrow \text{score}(\psi, T)$ 
3  $i \leftarrow 0$ 
4 while  $s \leq 1$  do
5   if  $i \geq N_{\max}$  then
6     return failure
7    $i \leftarrow i + 1$ 
8    $\psi' \leftarrow \text{mutate}(\psi, G)$ 
9    $s' \leftarrow \text{score}(\psi', T)$ 
10   $p \leftarrow \min\left(1, \frac{s'}{s}\right)$ 
11  if  $\text{uniform}(0, 1) \leq p$  then
12     $(s, \psi) \leftarrow (s', \psi')$ 
13 return  $\psi$ 

```

Algorithm 2: Function $\text{mutate}(\psi, G)$

Input: HyperLTL formula candidate

$\psi = Q_1\pi_1 \dots Q_m\pi_m \varphi(\pi_1, \dots, \pi_m)$,
HyperLTL (sub)grammar G

Output: Mutated formula ψ'

```

1 if  $\text{uniform}(0, 1) < 0.5$  then
2    $i \leftarrow \text{uniform}(\{1, 2, \dots, m\})$ 
3    $Q'_i, Q'_{i+1} \dots Q'_m \leftarrow$ 
4      $\text{sample\_quantifiers}(G, m - i + 1)$ 
5    $\psi' \leftarrow$ 
6      $Q_1\pi_1 \dots Q'_i\pi_i \dots Q'_m\pi_m \varphi(\pi_1, \dots, \pi_m)$ 
7 else
8    $\mathcal{T}_\varphi \leftarrow \text{formula\_to\_tree}(\varphi(\pi_1, \dots, \pi_m))$ 
9    $\mathcal{S} \leftarrow \text{select\_subtree}(\mathcal{T}_\varphi)$ 
10   $\gamma \leftarrow \text{sample\_formula}(G, \text{length}(\mathcal{S}))$ 
11   $\mathcal{T}_\gamma \leftarrow \text{formula\_to\_tree}(\gamma)$ 
12   $\mathcal{T}'_\varphi \leftarrow \text{replace}(\mathcal{T}_\varphi, \mathcal{S}, \mathcal{T}_\gamma)$ 
13   $\varphi'(\pi_1, \dots, \pi_m) \leftarrow$ 
14     $\text{tree\_to\_formula}(\mathcal{T}'_\varphi)$ 
15   $\psi' \leftarrow Q_1\pi_1 \dots Q_m\pi_m \varphi'(\pi_1, \dots, \pi_m)$ 
16 return  $\psi'$ 

```

We now describe the mutation function, summarized in Algorithm 2. Given the current specification candidate ψ , we randomly apply one of two equally-probable changes: (i) we modify the quantifiers (Lines 1-4), or (ii) we change the formula structure (Lines 6-12). In the first case, we replace a randomly selected sequence of consecutive quantifiers with a newly sampled sequence of quantifiers of the same length generated according to the grammar rules G . The number of quantifiers must remain unaltered to avoid adding/removing quantified trace variables in the formula structure φ . In the second case, we modify the quantifier-free part of the current formula φ by manipulating its syntax tree \mathcal{T}_φ ; an example of such transformation is depicted in Figure 1. We uniformly randomly select one of its nodes (e.g., the *until* node in Figure 1) and consider the subtree \mathcal{S} originated by this node (highlighted in red in the figure). From the grammar G , we sample one formula γ having as length the number of nodes in \mathcal{S} (3 in the example) and transform γ into its syntax-tree \mathcal{T}_γ . Then, we replace the subtree \mathcal{S} inside the tree of the original formula \mathcal{T}_φ with \mathcal{T}_γ (an example for \mathcal{T}_γ is given by the green nodes in Figure 2) and translate the resulting tree into the formula φ' . Finally, the output specification ψ' has the same quantifiers as the original specification ψ and φ' as quantifier-free formula.

We observe that, in this formulation of the algorithm, the length l and the number of quantifiers m remain fixed during the whole execution. Since, in general, the user will be interested in a set of hyperproperties satisfied by the set of data rather than in a single one, a natural approach is to use

different lengths and numbers of quantifiers in different runs, either in a systematic, incremental way or by sampling the values of l and m in intervals of interest.

In terms of complexity, the monitoring process (and consequently the computation of the score - Lines 2 and 9) represents the only expensive step. As we will detail in Sections 3.2 and 3.3, we require up to $\frac{|T|!}{(|T|-m)!}$ calls to the LTL monitor to compute the score we propose. In the worst-case scenario, Algorithm 1 ends after N_{\max} score evaluations, hence yielding an overall computational complexity of $N_{\max} \cdot \frac{|T|!}{(|T|-m)!}$.

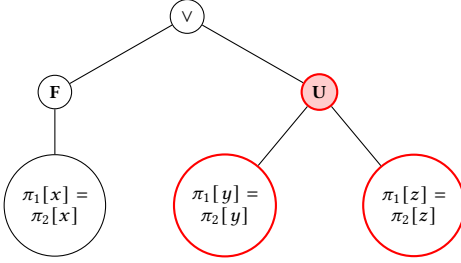


Fig. 1. Syntax tree for the quantifier-free formula: $\varphi(\pi_1, \pi_2) = \mathbf{F}(\pi_1[x] = \pi_2[x]) \vee ((\pi_1[y] = \pi_2[y]) \mathbf{U}(\pi_1[z] = \pi_2[z]))$.

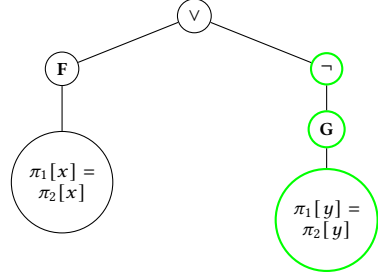


Fig. 2. Syntax tree for the quantifier-free formula: $\varphi'(\pi_1, \pi_2) = \mathbf{F}(\pi_1[x] = \pi_2[x]) \vee \neg \mathbf{G}(\pi_1[y] = \pi_2[y])$.

3.2 Fitness Function

In this section, we propose a *fitness* function $F(\psi, T)$ for quantifying the degree of satisfaction/violation of a HyperLTL specification ψ with respect to T . Consider a HyperLTL formula ψ of the form $Q_1\pi_1 \dots Q_m\pi_m \varphi(\pi_1, \dots, \pi_m)$. We define $F(\psi, T)$ as the minimum number of m -tuples of traces in T whose satisfaction with respect to φ should be changed in order to change the satisfaction of ψ by T . Note that $F(\psi, T)$ is positive when ψ is satisfied by T , and negative otherwise.

Table 1. Example of satisfaction (✓) or violation (×) of $\varphi(\pi, \pi')$ by pairs of traces in T .

$\pi \backslash \pi'$	ω_1	ω_2	ω_3	ω_4
ω_1	-	✓	✓	×
ω_2	×	-	✓	✓
ω_3	✓	×	-	✓
ω_4	×	✓	✓	-

We illustrate this concept with an example. Let $\psi = \forall \pi \exists \pi' \varphi(\pi, \pi')$ where $\varphi(\pi, \pi')$ is an LTL formula whose predicates involve the trace variables π and π' . Let $T = \{\omega_1, \omega_2, \omega_3, \omega_4\}$ and let Table 1 represent the Boolean satisfaction (✓) or violation (×) of $\varphi(\pi, \pi')$ when trace variables π and π' are replaced by concrete pairs of traces in T .

The quantifiers in ψ impose that for every row in the table (i.e., $\forall \pi$) there is at least one column (i.e., $\exists \pi'$) with the satisfaction symbol ✓. It is evident from Table 1 that T satisfies ψ because in every row there are at least two columns marked with ✓. Consequently, at

least two changes from ✓ to × are required to violate ψ . For this reason, we have that $F(\psi, T) = 2$.

We now generalize this example and present the algorithm for computing F . We separate the computation of F into two parts: *positive* fitness F^+ , which gives a positive value when T satisfies ψ and zero otherwise, and *negative* fitness F^- , which gives negative value when T violates ψ and zero otherwise. Since the two cases are symmetric, we present only the procedure for computing F^+ in Algorithm 3.

Algorithm 3: Computation of the *positive* fitness value $F^+(\psi, T)$ **Input:** HyperLTL formula $\psi = Q_1\pi_1 \dots Q_m\pi_m \varphi(\pi_1, \dots, \pi_m)$, set of data T with $|T| = n$;**Output:** $F^+(\psi, T)$ Positive fitness of ψ w.r.t. T ;

```

1 foreach  $(\omega_1, \dots, \omega_m) \in T^m$  s.t.  $\omega_i \neq \omega_j$  if  $i \neq j$  do
2   if  $\varphi(\omega_1, \dots, \omega_m)$  holds then  $O(\psi; \omega_1, \dots, \omega_m) \leftarrow 1$ 
3   else  $O(\psi; \omega_1, \dots, \omega_m) \leftarrow 0$ 
4 for  $k = m, m-1, \dots, 2$  do
5   foreach  $(\omega_1, \dots, \omega_{k-1}) \in T^{k-1}$  s.t.  $\omega_i \neq \omega_j$  if  $i \neq j$  do
6     if  $Q_k = \exists$  then  $O(\psi; \omega_1, \dots, \omega_{k-1}) \leftarrow \sum_{l=1}^n O(\varphi; \omega_1, \dots, \omega_{k-1}, \omega_l)$ 
7     else if  $Q_k = \forall$  then  $O(\psi; \omega_1, \dots, \omega_{k-1}) \leftarrow \min_{l=1, \dots, n} O(\varphi; \omega_1, \dots, \omega_{k-1}, \omega_l)$ 
8 if  $Q_1 = \exists$  then
9    $F^+(\psi, T) \leftarrow \sum_{l=1}^n O(\varphi; \omega_l)$ 
10 else if  $Q_1 = \forall$  then
11    $F^+(\psi, T) \leftarrow \min_{l=1, \dots, n} O(\varphi; \omega_l)$ 
12 return  $F^+(\psi, T)$ 

```

Let $n = |T|$ and $\psi = Q_1\pi_1 \dots Q_m\pi_m \varphi(\pi_1, \dots, \pi_m)$. In the first step, we evaluate for every m -tuple $(\omega_1, \dots, \omega_m)$ of traces in T whether it satisfies the $\varphi(\pi_1, \dots, \pi_m)$ formula, in which every trace variable π_i is instantiated with a concrete trace ω_i . This corresponds to a standard LTL membership check. The membership check results are stored in an m -dimensional tensor O . We then iteratively reduce the dimensions of O until we obtain a scalar value, corresponding to the final outcome of Algorithm 3. To do so, we associate each \exists quantifier with the sum operator and each \forall quantifier with the minimum operator. Then, starting from the innermost quantifier, we apply its corresponding operator to the vectors of values in O sharing all but the last components. In the example of Table 1, since $Q_2 = \exists$, we first sum the number of \checkmark in each row, obtaining the vector $O = (2, 2, 2, 2)$. Then, being $Q_1 = \forall$, we take the minimum value in O , which is 2 and corresponds to the final value of $F^+(\psi, T)$.

If F^+ is equal to zero, the hyperproperty is violated, and we need to compute the *negative* fitness F^- to quantify how robust the violation is. The computation of F^- is symmetric to Algorithm 3.

We observe that Algorithm 3 has high computational complexity – it requires $\frac{n!}{(n-m)!} = n \cdot (n-1) \dots (n-m+1)$ membership checks. Hence, the algorithm has the complexity of $\mathcal{O}(n^m)$, requiring an exponential number of LTL membership checks in the number of the quantifiers.

3.3 Using Quantifier Properties for Efficient Monitoring

The goal of the efficient monitoring algorithm we propose is to reduce the practical complexity of Algorithm 3 by avoiding unnecessary LTL membership checks. Given a HyperLTL formula $\psi = Q_1\pi_1 \dots Q_m\pi_m \varphi(\pi_1, \dots, \pi_m)$ and a set T of traces, we devise a method early-stopping criteria based on the semantics of the quantifiers. Our monitoring approach is summarized in Algorithm 4.

As a preliminary step, we first use a tree \mathcal{T} to enumerate the tuples of traces in T that have m non-repeated components (we assume that different trace variables refer to different execution traces, but slight changes are sufficient to adapt our approach if this hypothesis does not hold). An example for the dataset $T = \{\omega_1, \omega_2, \omega_3, \omega_4\}$ is depicted in Figure 3. We call an *ancestor* of a node p any node encountered when climbing the tree from p to the root, and a *sibling* of a node p any node that shares the same parent as p . The root node of \mathcal{T} is always given by the whole dataset T

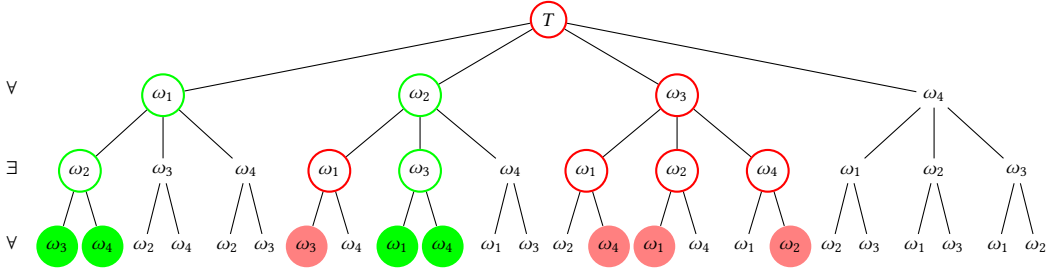


Fig. 3. Tree representation of the 3-tuples of traces for the dataset $T = \{\omega_1, \omega_2, \omega_3, \omega_4\}$. The evaluation of $\psi = \forall\pi\exists\pi'\forall\pi'' \varphi(\pi, \pi', \pi'')$ is represented by the color green to mean satisfaction and red for violation.

Algorithm 4: Efficient Monitoring for Temporal Hyperproperties

Input: HyperLTL formula $\psi = Q_1\pi_1 \dots Q_m\pi_m \varphi(\pi_1, \dots, \pi_m)$, m -tuples of traces of T organized in a tree T ;

Output: Boolean outcome: *satisfaction* or *violation* of ψ with respect to T ;

```

1 node ← SampleLeafNode(root(T))
2 E(node) ← Evaluate(node,  $\varphi$ )
3 depth ← m
4 while depth > 0 do
5   if ( $Q_{\text{depth}} = \forall \wedge E(\text{node}) = \text{unsat}$ )  $\vee$  ( $Q_{\text{depth}} = \exists \wedge E(\text{node}) = \text{sat}$ ) then
6     E(Parent(node)) ← E(node)
7     node ← Parent(node)
8     depth ← depth - 1
9   else if NotEvaluatedSiblings(node) ≠  $\emptyset$  then
10    node ← SampleLeafNode(Sample(NotEvaluatedSiblings(node)))
11    E(node) ← Evaluate(node,  $\varphi$ )
12    depth ← m
13   else if NotEvaluatedSiblings(node) =  $\emptyset$  then
14     node ← Parent(node)
15     if  $Q_{\text{depth}} = \forall$  then E(node) ← sat
16     else if  $Q_{\text{depth}} = \exists$  then E(node) ← unsat
17     depth ← depth - 1
18 return E(root(T))
  
```

(we consider it as the node with depth 0) and has $|T| = n$ children nodes, each one representing a trace in T . These nodes make up depth level 1 in the tree and each of them has $n - 1$ children, one for each trace in T different from the trace of the node itself. By iterating this process and only allowing a node to have children different from its ancestors, we obtain a tree of depth m and with $\frac{n!}{(n-m)!}$ leaves. Each leaf can be interpreted as an m -tuple of traces, obtained as the sequence of its ancestors ordered from the oldest to the youngest. For example, in Figure 3, the leftmost leaf is interpreted as the 3-tuple $(\omega_1, \omega_2, \omega_3)$, while the rightmost leaf as $(\omega_4, \omega_3, \omega_2)$. We conclude the preliminary step by associating each depth level i of the tree with the quantifier Q_i in ψ for all $i = 1, \dots, m$.

We now describe how Algorithm 4 works. The key idea is to minimize the number of evaluations of the property with respect to different tuples of traces by introducing early stops as soon as a satisfied witness for the \exists quantifier or a violation for the \forall is found. In particular, we start by evaluating the satisfaction of one uniformly sampled tuple of traces (the function *SampleLeafNode*(x) in Line 1 returns one leaf node of the subtree having as root the node x and the function *Evaluate* in Line 2 calls the LTL monitoring library to compute whether the corresponding tuple of traces satisfies the quantifier-free formula φ). In Figure 3, we color a node with green to indicate satisfaction and with red to indicate violation. Depending on the quantifier associated with the depth level of the current node (i.e., m for the first iteration), we might infer the satisfaction or violation of the parent node. In particular (Line 5), if the current quantifier is \forall and the current node is violated, we also associate its parent node with violation since, regardless of the evaluation of its siblings, we already found a witness of the violation. Conversely, if the quantifier is \exists and the node is satisfied, we consider its parent node as satisfied as well, as the current node already demonstrates that at least one satisfied tuple exists. In Figure 3, we represent the inferred satisfaction and violation of the nodes by coloring only their contour to underline the fact that their evaluation did not require a call to the LTL monitoring library. If it is not yet possible to infer the satisfaction/violation of the parent node, we proceed by evaluating the satisfaction of one of the siblings of the current node that has not been studied yet (Lines 9-12). When all the siblings have been studied, we can finally infer the status of the parent node: violated if the quantifier is \exists and all siblings are violated, or satisfied if the quantifier is \forall and all the siblings are satisfied (Lines 13-17).

Figure 3 shows an example of the execution of Algorithm 4. In this case, the 3-tuples of traces that have been evaluated to conclude the violation of ψ with respect to T are 8 out of 24. Nevertheless, we observe that, in the worst-case, there are $\frac{n!}{(n-m)!}$ LTL membership checks required, provoking an exponential complexity in the number of quantifiers m .

4 EXTENSION TO HYPERSTL AND PARAMETER SYNTHESIS

In this section, we describe the extension of the mining procedure for HyperLTL (presented in Section 3) to HyperSTL. HyperSTL provides two new features that need to be taken into account: bounded temporal operators and numeric predicates over real-valued variables. When the HyperSTL (timing and amplitude) parameters are fixed and part of the template, Algorithm 1 can be used without any adaptation. However, in many common scenarios, this is not the case, as only a range of admissible values for each parameter is known.

We partition the mining procedure into two separate steps: (i) mining the structure of the candidate formula (Algorithm 5), and (ii) inferring the parameters appearing in the formula. In this work, we restrict the first step to finding monotonic parametric HyperSTL formula templates. For this reason, we combine Algorithm 1 with rejection sampling (Lines 1-4 and 11-14) to ensure that the candidate parametric template ψ_p is monotonic with respect to all its parameters. We automatically check monotonicity using the polarity rules described in [3]. Whenever a monotonic formula template is found, we instantiate it (Lines 5 and 15) by replacing every parameter symbol with the value in the respective admissible interval that is most likely to be satisfied, namely the upper bound for monotonically increasing parameters and the lower bound for the decreasing ones. Let us consider for example $\psi_p = \forall \pi \exists \pi'. \mathbf{G}_{[0, p_1]}(\pi[x] + \pi'[x] \leq p_2)$ where p_1 varies in the interval $I_1 = [10, 30]$ and p_2 in $I_2 = [0.5, 3]$. We observe that ψ_p is monotonically decreasing in p_1 and monotonically increasing in p_2 . Hence, the resulting concrete hyperproperty $\psi = \text{instantiate_formula}(\psi_p, [I_1, I_2])$ will be $\psi = \forall \pi \exists \pi'. \mathbf{G}_{[0, 10]}(\pi[x] + \pi'[x] \leq 3)$.

The evaluation of the score is analogous to the HyperLTL case, with the only difference that the quantifier-free STL formulas require an STL monitor instead of an LTL monitor. When we obtain a

candidate HyperSTL formula that is satisfied by the dataset, we perform the second step of the procedure and refine the formula's parameter values.

The parameter synthesis procedure consists in the natural extension to hyperproperties of the combination of monitoring and binary search proposed in [3]. The goal is to find parameter values that provide a tight satisfaction of the hyperproperty, meaning that if their values change slightly, the overall satisfaction outcome of the hyperproperty changes too. The reason for such a choice is to find interesting hyperproperties that do not overgeneralize the features of the given dataset. To achieve this goal, we leverage the monotonicity of ψ_p to approximate the region of parameter values that render ψ_p satisfied by the set of traces T and choose the final parameter values as one point on the satisfaction boundary. We describe in more detail such refinement procedure by presenting its application to the previous example.

We denote by k the number of parameter symbols in the formula ($k = 2$ in the example) and by P the *parameter space*, namely the Cartesian product of the intervals of admissible values for the parameters in ψ_p ($P = [10, 30] \times [0.5, 3]$). The procedure is iteratively repeated on hyper-rectangles in P , starting with the whole P as the first hyper-rectangle. We first consider the point l that is the *least likely to be satisfied* in the current hyper-rectangle, namely the lower bound for monotonically increasing parameters and upper bound for decreasing ones (in the example, $l = [30, 0.5]$). We instantiate ψ_p with l , obtaining $\psi_p[l] = \forall\pi\exists\pi'. \mathbf{G}_{[0,30]}(\pi[x] + \pi'[x] \leq 0.5)$ and we monitor it against the set of traces T . If $\psi_p[l]$ is satisfied - Condition (i) - we could infer the satisfaction of the whole hyper-rectangle. Indeed, since ψ_p is monotonically increasing in p_2 , for any other value $l_2 \geq 0.5$, the formula $\psi_p[30, l_2]$ would be satisfied as well, as any values of $\pi[x]$ and $\pi'[x]$ that realize $\pi[x] + \pi'[x] \leq 0.5$ would also realize $\pi[x] + \pi'[x] \leq l_2$. In an analogous way, since ψ_p is monotonically decreasing in p_1 , we can deduce the satisfaction of $\psi_p[l_1, 0.5]$ for any $l_1 \leq 30$. Conversely, if $\psi_p[l]$ is violated, we monitor $\psi_p[M]$, being M the point that is *most likely to be satisfied*. If $\psi_p[M]$ is violated - Condition (ii) - we can infer the violation of the entire hyper-rectangle with symmetrical considerations (in the first iteration, this cannot happen because we apply such procedure to the output of Algorithm 5, which is guaranteed to be satisfied in at least one point). If none of the conditions (i) and (ii) is realized, we consider the midpoint q on each parameter dimension (for P , $q = [20, 1.75]$), monitor $\psi_p[q]$ and apply the previous reasoning on monotonicity to infer either the satisfaction or violation of one of

Algorithm 5: Mining HyperSTL formula structure

Input: HyperSTL (sub)grammar G , set of traces T , formula length l , number of quantifiers m , maximum number of iterations N_{\max} , list of intervals I for the parameters in G

Output: HyperSTL formula ψ generated from G and satisfied by T

```

1  $mono \leftarrow \text{false}$ 
2 while  $mono = \text{false}$  do
3    $\psi_p \leftarrow \text{sample\_hyperproperty}(G, l, m)$ 
4    $mono \leftarrow \text{check\_monotonicity}(\psi_p)$ 
5  $\psi \leftarrow \text{instantiate\_formula}(\psi_p, I)$ 
6  $s \leftarrow \text{score}(\psi, T)$ 
7  $i \leftarrow 0$ 
8 while  $s \leq 1$  do
9   if  $i \geq N_{\max}$  then return failure
10   $i \leftarrow i + 1$ 
11   $mono \leftarrow \text{false}$ 
12  while  $mono = \text{false}$  do
13     $\psi'_p \leftarrow \text{mutate}(\psi_p, G)$ 
14     $mono \leftarrow \text{check\_monotonicity}(\psi'_p)$ 
15     $\psi' \leftarrow \text{instantiate\_formula}(\psi'_p, I)$ 
16     $s' \leftarrow \text{score}(\psi', T)$ 
17    if  $\text{uniform}(0, 1) \leq \min(1, \frac{s'}{s})$  then
18       $(s, \psi, \psi_p) \leftarrow (s', \psi', \psi'_p)$ 
19 return  $\psi$ 

```

the 2^k generated hyper-rectangles. Figure 4a depicts such process: the blue striped area represents the exact (unknown) satisfaction region, the green area (hyper-rectangle A) is the inferred satisfied region and the white area corresponds to the yet unknown region. The same procedure is then applied to the remaining $2^k - 1 (= 3)$ generated hyper-rectangles (B, C and D in Figure 4a) and iteratively repeated. Figure 4b depicts the status of the satisfaction region approximation after the second iteration of the binary search algorithm: hyper-rectangle B satisfies condition (i); region C violates condition (i) because its *least satisfiable point* $[20, 0.5]$ is violated and violates condition (ii) because its *most satisfiable point* $[0.5, 1.75]$ is satisfied (such point is shared with the satisfied region A, hence does not require explicit monitoring). C is therefore divided into four regions by its midpoint $[15, 1.125]$. Similarly, also D needs to be partitioned.

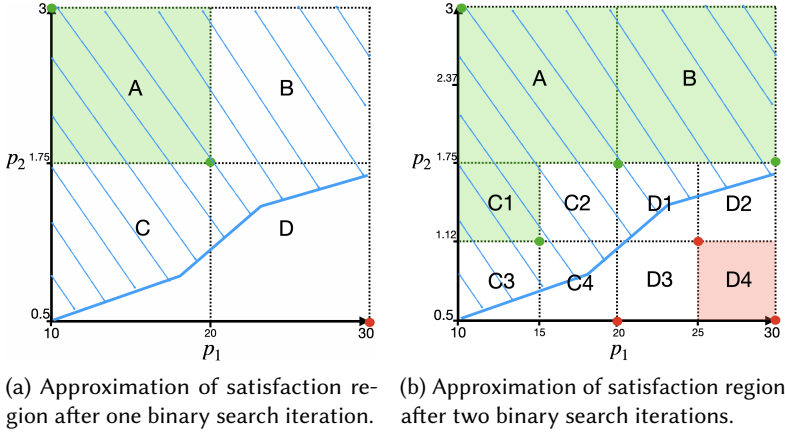


Fig. 4. Example of approximation (green area) of the satisfaction region (blue striped area) of ψ_p through binary search algorithm. The white region represents the unknown region, while the red one is the (known) violated region. Colored points represent monitored parameter values (green for satisfaction and red for violation).

The procedure ends either after the area of the unknown region falls below a predefined threshold or after a maximum number of iterations N_b (in our experiments, $N_b = 10$). The final parameter values correspond to a point sampled on the border of the approximated satisfaction region.

We discuss now the complexity of the proposed approach. Starting from the parameter synthesis, we observe that, given a parametric HyperSTL formula with k parameter symbols, the j -th iteration of the binary search algorithm needs at most $3 \cdot (2^k - 1)^{j-1}$ HyperSTL monitors in the worst-case scenario of all hyper-rectangles requiring to be partitioned. Hence, since N_b is the maximum number of iterations, the whole procedure requires up to $3 \cdot \sum_{j=1}^{N_b} (2^k - 1)^{j-1} = 3 \cdot \frac{(2^k - 1)^{N_b} - 1}{2^k - 2}$ HyperSTL monitors. As stated in Sections 3.2 and 3.3, each HyperSTL monitor corresponds to up to $\frac{|T|!}{(|T| - m)!}$ calls to the STL monitor, hence the parameter refinement requires at most $\frac{|T|!}{(|T| - m)!} \cdot 3 \cdot \frac{(2^k - 1)^{N_b} - 1}{2^k - 2}$ STL membership checks. By summing this value with the complexity of Algorithm 5 (equal to the complexity of Algorithm 1), we conclude that, in the worst-case scenario, our approach needs

$$C = \frac{|T|!}{(|T| - m)!} \cdot \left(N_{\max} + 3 \cdot \frac{(2^K - 1)^{N_b} - 1}{2^K - 2} \right)$$

calls to the STL monitor, being K the total number of parameter symbols in the grammar G .

We observe that our restriction to monotonic formulas is common in the STL specification mining literature, as monotonicity holds for many relevant properties and, at the same time, it renders much more tractable the approximation of the formula's satisfaction boundary.

4.1 Using Robustness for Efficient Monitoring

In this section, we use the correctness property of STL robustness function (Proposition 2.4) to reduce the number of STL monitoring calls during the evaluation of an HyperSTL formula. Every time we need to call the STL monitor for a tuple of traces (Algorithm 4 Line 11), we first check if there exists another already evaluated tuple that differs from the current tuple for just one trace and such that the different traces have distance smaller than the robustness value of the evaluated tuple. If so, we can skip the evaluation of the current tuple, inferring it from the other tuple. To clarify this statement, let us consider an example. Let us suppose we have already monitored the tuple of traces $\Omega = (\omega_1, \omega_2, \omega_3)$ with respect to the quantifier-free formula $\varphi(\pi_1, \pi_2, \pi_3)$ and that Ω satisfies φ with (positive) robustness value $\hat{\rho}$. Whenever we need to monitor another tuple of traces that differs from Ω for only one trace, for example $\Omega' = (\omega_1, \omega_2, \omega_4)$, we compute the infinity-norm distance between the two traces that are different; in this case, $g = \|\omega_3 - \omega_4\|_\infty$. If $g < \hat{\rho}$, we leverage Proposition 2.4 to infer that also Ω' satisfies φ , skipping its monitoring. The analogous reasoning applies if Ω violates φ with (negative) robustness $\bar{\rho}$: if $g < |\bar{\rho}|$, then Ω' violates φ as well. The vice versa does not hold and requires the explicit monitoring of Ω' with respect to φ .

In terms of complexity, we observe that, similarly to the case of the efficient monitoring algorithm, the proposed heuristic effectively reduces the number of STL monitors in practice, but the worst-case remains unchanged.

5 EVALUATION

In this section, we start by defining our research questions. We then describe our experimental setup, our case studies and experimental results.

5.1 Research Questions

We aim to address these research questions:

RQ1[Performance Evaluation] *To what extent are the mined specifications a good characterization of the system properties according to the following metrics?* We introduce several case studies where traces are generated by different systems and run systematic experiments with two different settings: free and constrained grammar. We analyze the results regarding the ratio of false positive examples (i.e., ratio of mined formulas holding on random traces that do not come from the system), the ratio of false negative examples (i.e., ratio of mined formulas that do not hold on new traces coming from the system), the variability of the mined formulas, and computational time. We show how embedding knowledge through restrictions on the space of admissible formulas generally improves all the mentioned performance metrics.

RQ2[Ability to mine useful specifications] *How useful are the specifications that we infer from the traces?* We perform a qualitative analysis of the mined specifications. More specifically, for each subclass of HyperSTL covered by our case studies, we present and discuss learned formulas representing (i) interesting, (ii) trivial or non-relevant, or (iii) incorrect properties of the system.

RQ3[Flexibility of the mining procedure] *Can we effectively use the same method to mine not only HyperSTL properties but also its subsets, such as HyperLTL, STL, LTL, and even non-temporal properties?* To answer this question, we select several case studies, each requiring a different type of specifications: (i) addition and trigonometric (non-temporal) functions, (ii) LTL temporal tester transducers and dining philosophers, (iii) serial adder and microcontroller systems, and (iv) autonomous parking valet scenario.

RQ4[Scalability] *How does our method scale in terms of the complexity of the formulas and concerning different grammar restrictions?* We analyze the computation time our approach needs when varying the length of the target formula and the constraints imposed on the grammar. We also evaluate the effectiveness of our monitoring algorithm.

RQ5[Comparison with the state-of-the-art] *How does our approach compare to other existing specification mining methods?* To answer this research question, we first identify two appropriate specification mining tools, mainly Texada [18] for template-based learning of LTL and HyperMiner [26] for template-based learning of HyperLTL formulas. Since these tools can mine only subsets of specification languages handled by our approach, we select case studies where the comparison is possible and appropriate. We also study whether our more flexible method can infer additional specifications after relaxing the restrictions on the grammar imposed by the other tools.

5.2 Experimental Setup and Case Studies

We implemented our approach in the publicly available tool HyTeM³ (Hyper-Temporal Miner). We also provide the scripts and the seeds to reproduce the results we present. We run the experiments on a scientific cluster based on Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz and Ubuntu 20.04.

In the following, we describe the case studies on which we evaluate our method and the relative data generation. For each case study, we also identify one essential and relevant property of the system-under-test, which we call *target formula*. In some experiments, we will restrict the grammar to steer the mining procedure towards learning such property.

Addition Function takes as input two integers x and y and outputs another integer $z = x + y$. Hence, a trace w is stateless and consists of a single valuation (x, y, z) . We generated a dataset of 100 traces by sampling 50 random pairs of positive integers (x, y) in the range $[0, 1000]$ and for each such pair computing two traces (x, y, z) and (y, x, z) . The target formula for this case study is represented by the *commutativity*:

$$\forall \pi \forall \pi'. (\pi[x] = \pi'[y] \wedge \pi[y] = \pi'[x]) \rightarrow (\pi[z] = \pi'[z]).$$

Trigonometric Functions that we consider in this case study are $\sin x$, $\cos x$, $\arcsin x$ and $\arccos x$, where the input x is a real value bounded to the interval $(-1, 1)$. We generated a dataset of 100 stateless traces of the form $(x, \sin x, \cos x, \arcsin x, \arccos x)$ where x is uniformly sampled from the interval $(-1, 1)$. In this case, one interesting hyperproperty is represented by the *increasing monotonicity* of the arcsin function:

$$\forall x \forall y. (x < y) \leftrightarrow (\arcsin x < \arcsin y).$$

An *LTL Temporal Tester* T_φ for an LTL formula φ is a transducer that realizes the semantics of φ . It reads a (potentially multi-dimensional) trace w and outputs another trace u such that for every $i \geq 0$, $u[i] = 1$ if and only if $(w, i) \models \varphi$. In this case study, we selected five LTL formulas Xp , XXp , Gp , Fp and pUq . For each formula, we created a temporal tester in the form of a Mealy machine and collected a dataset of 50 traces, where each trace is a sequence of 1000 valuations generated by randomly exploring the underlying temporal tester. In other words, a temporal tester reproduces the outcome of the monitoring process of a specific LTL specification and we use temporal testers to generate traces that satisfy predetermined LTL operators. For example, for the *next* operator, given a Boolean trace w , we use T_X to generate the Boolean trace u such that, at every time step, $u \leftrightarrow X(w)$. Hence, for each LTL formula φ , the target formula is the characterization of the temporal tester via the equivalence with the original specification, that is: $\forall \pi. G(\pi[u] \leftrightarrow \varphi(\pi[w]))$.

³<https://github.com/eleonoranesterini/HyTeM>

Dining Philosophers is a well-known concurrency problem. It consists of philosophers sitting at a round table and sharing one chopstick with each one of the two philosophers sitting next to them. Each philosopher can do exactly one of the following three actions: thinking, being hungry or eating. Since two chopsticks are required to be able to eat, two adjacent philosophers cannot eat at the same time. We consider the log trace of one solution of the dining philosophers' problem for five philosophers, available in the repository of the Texada specification mining tool⁴. Every trace is a sequence of 1088 5-tuple valuations encoding the state of five philosophers. We consider as our target formula one of the templates studied in [18]:

$$\forall \pi. \mathbf{G}(\pi[\text{Predicate}_{i,j}] \rightarrow \neg \pi[\text{Predicate}_{i',j'}]), \quad (3)$$

where $\pi[\text{Predicate}_{i,j}]$ stands for “*philosopher i performing action j*”, with $i \in \{1, 2, \dots, 5\}$ and $j \in \{\text{thinking, hungry, eating}\}$ in trace π .

Serial Adder with Carry-On is a sequential machine in which the output of the transducer corresponds to the element-wise binary addition with carry-on of two given inputs. Hence, a trace w of the serial adder with carry-on is a sequence of valuations of the form (x, y, c, z) , where x and y denote the inputs, c denotes the carry-on bit, z denotes the output, and for every time step $i \geq 1$,

$$w[z][i] = (w[x][i] + w[y][i] + w[c][i-1]) \bmod 2,$$

where $w[c][0] = 0$ and, for $i \geq 1$,

$$w[c][i] = \begin{cases} 0 & \text{if } (w[x][i] + w[y][i] + w[c][i-1] \leq 1), \\ 1 & \text{otherwise.} \end{cases}$$

We generated a dataset of 50 traces of 1000 time steps each by randomly creating the sequences of x and y inputs. In this case, the target formula is the *observational determinism* of variable z with respect to x and y :

$$\forall \pi \forall \pi'. (\mathbf{G}(\pi[x] = \pi'[x] \wedge \pi[y] = \pi'[y]) \rightarrow \mathbf{G}(\pi[z] = \pi'[z])).$$

Microcontroller is a system-on-chip (SoC) design studied in the context of mining hyperproperties [26]. We use the publicly available dataset⁵ that consists of 100 traces generated from a microcontroller design, where each trace consists of a sequence of 15884 35-variable valuations. Since we expect the variables to affect each other deterministically, the target formula for this case study is the *observational determinism* (2).

Autonomous Driving Parking is a case study involving an autonomous car driving through a parking lot with a pedestrian that walks out from behind a car onto the path of the vehicle. The car is equipped with an RGB camera and uses YOLOv7, a state-of-the-art image detection deep neural network [28], to detect the pedestrian. As soon as the pedestrian is recognized, the car brakes as hard as possible to avoid the collision. We simulated such scenario with CARLA 9.13 [14] and collected a sequence of 185 samples containing the coordinates of the velocity v of the car and its distance d to the pedestrian. For each execution, the car's target velocity before braking is fixed to a value sampled between 5.55 and 11.11 mps (20 – 40 kph). The length of each execution is of 171 time units (representing 0.05 seconds each). Figure 5 shows an example of an image from the vehicle's onboard camera. For this case study, the target formula is the *robustness* hyperproperty:

$\forall \pi \forall \pi'. \mathbf{G}(\|\pi[\text{in}] - \pi'[\text{in}]\| < \varepsilon_1 \rightarrow \|\pi[\text{out}] - \pi'[\text{out}]\| < \varepsilon_2)$. This hyperproperty relates the magnitude ε_1 of the perturbations in the input variables *in* (in this case study, the velocity) with the

⁴<https://github.com/ModelInference/texada/tree/master/traces/dining-philosophers>

⁵<https://github.com/skmuduli92/TraceExamplesCustomCov/tree/51e3e73d0e507ab1ac44e85a0510118202f73c74>



Fig. 5. Example of the scene as seen from the camera of the autonomous car. To illustrate how the pedestrian crosses the parking lot, we depicted four pedestrians, although only one is the actual scenario.

magnitude ε_2 of the corresponding changes in the output variables *out* (the distance).

Experimental Setup. For each case study, we perform the following steps:

- *Training data generation:* We generate three sets of 50 traces (L_1, L_2, L_3) from the system.
- *Positive test data generation:* We generate another three sets of 50 traces (P_1, P_2, P_3) from the system.
- *Negative test data generation:* We generate three sets of random 50 traces (N_1, N_2, N_3), hence not generated from the system.
- *Mining:* We learn three sets of 50 specifications: S_1, S_2 , and S_3 from L_1, L_2 and L_3 , and another two sets of specifications S'_1 and S''_1 from L_1 . We repeat the mining procedure twice, using (i) an unrestricted grammar with up to three quantifiers, and (ii) a template for the target formula.
- *Evaluation:* Since the mined properties cannot be formally verified against the system, the quality of the learned formulas is assessed through sampling and monitoring. More specifically, we evaluate the procedure according to five criteria: (i) the ratio of false positives, (ii) the ratio of false negatives, (iii) the variability of the mined formulas induced by the randomness of the procedure on the same training dataset, (iv) the variability induced by the differences between datasets, and (v) the computational time.

The *ratio of false positives* (RFP) is the proportion of mined formulas that are satisfied by a set of traces not generated from the same system. A high value of RFP (close to 1) indicates that the mined formulas might be too generic and not specific to the given dataset. Similarly, the *ratio of false negatives* (RFN) is the proportion of mined formulas that are violated by a set of traces generated from the same system. A small value of RFN (close to 0) means that the mined hyperproperties do not overfit the learning dataset, as they are valid also on an unseen set of traces. We formalize RFP and RFN as functions of a generic set Q of formulas as follows:

$$\text{RFP}(Q) = \frac{|\{\varphi \mid \varphi \in Q \wedge N \models \varphi\}|}{|Q|}, \quad \text{RFN}(Q) = \frac{|\{\varphi \mid \varphi \in Q \wedge P \not\models \varphi\}|}{|Q|}.$$

We quantify the degree of *variability* V_{same} of the sets of specifications mined using the same set of traces (S_1, S'_1, S''_1) and the *variability* V_{diff} of the sets of formulas when different sets of traces are adopted (S_1, S_2, S_3). The two quantities vary between 0 and 1, where the former value indicates that the three sets are all identical and the latter one that they are completely different, not sharing any formula. By comparing V_{same} with V_{diff} , we aim at studying whether the randomness in the output specifications is intrinsic to the mining process (if $V_{\text{same}} \approx V_{\text{diff}}$) or it depends on the training

Table 2. Summary of the experimental results on eleven case studies.

Case Study	Setting	Target formula	RFP	RFN	V_{same}	V_{diff}	Time (hours)	#Formulas
Addition	Constrained	Commutativity	0.5 ± 0.3	0.5 ± 0.35	1	0.95	2.14 ± 1.77	50 ± 0
	Free		0.72 ± 0.04	0.09 ± 0.07	0.97	0.98	0.53 ± 0.19	50 ± 0
Trigonometry	Constrained	Monotonicity	0.11 ± 0.02	0.02 ± 0.02	0.95	0.98	0.018 ± 0.002	50 ± 0
	Free		0.85 ± 0.009	0.04 ± 0.03	1	0.99	0.03 ± 0.02	50 ± 0
Tester Always	Constrained	Equiv. Always	0.19 ± 0.03	0.0 ± 0.0	0.62	0.64	0.05 ± 0.04	50 ± 0
	Free		0.78 ± 0.01	0.02 ± 0.01	0.98	0.99	10.8 ± 6.8	50 ± 0
Tester Eventually	Constrained	Equiv. Eventually	0.18 ± 0.04	0.0 ± 0.0	0.67	0.63	0.05 ± 0.03	50 ± 0
	Free		0.65 ± 0.02	0.02 ± 0.02	1	1	4.8 ± 3.2	50 ± 0
Tester Next	Constrained	Equiv. Next	0.28 ± 0.06	0.0 ± 0.0	0.74	0.69	0.12 ± 0.05	50 ± 0
	Free		0.96 ± 0.01	0.01 ± 0.009	0.97	0.98	6.3 ± 2.7	50 ± 0
Tester NextNext	Constrained	Equiv. NextNext	0.23 ± 0.03	0.0 ± 0.0	0.75	0.75	0.14 ± 0.04	48 ± 2
	Free		0.98 ± 0.01	0.006 ± 0.009	0.97	1	7.6 ± 5.0	50 ± 0
Tester Until	Constrained	Equiv. Until	0.69 ± 0.07	0.0 ± 0.0	0.7	0.64	0.23 ± 0.05	50 ± 0
	Free		0.96 ± 0.0	0.04 ± 0.0	1	1	9.6 ± 3.8	50 ± 0
Philosophers	Constrained	Template in [18]	0.54 ± 0.08	0.2 ± 0.1	1	1	0.07 ± 0.04	50 ± 0
	Free		0.72 ± 0.02	0.18 ± 0.04	1	1	0.8 ± 0.9	50 ± 0
Serial Adder	Constrained	Obs Det	1.0 ± 0.0	0.0 ± 0.0	0.42	0.45	0.39 ± 0.12	29 ± 2
	Free		0.91 ± 0.01	0.006 ± 0.009	0.98	0.97	4.9 ± 3.6	50 ± 0
Microcontroller	Constrained	Obs Det	0.98 ± 0.01	0.04 ± 0.06	1	1	10 ± 4	50 ± 0
	Free		-	-	-	-	runout	-
AD Parking	Constrained	Robustness	0.0 ± 0.0	0.12 ± 0.09	0.28	0.41	1.25 ± 0.29	30 ± 2
	Free		0.28 ± 0.05	0.05 ± 0.01	0.99	0.95	12.85 ± 2.348	50 ± 0

dataset (if $V_{\text{same}} \ll V_{\text{diff}}$). V_{same} and V_{diff} are defined as:

$$V_{\text{same}} = \frac{|S \setminus S_1| + |S \setminus S'_1| + |S \setminus S''_1|}{6 \cdot \text{average}(|S_1|, |S'_1|, |S''_1|)}, \quad V_{\text{diff}} = \frac{|\hat{S} \setminus S_1| + |\hat{S} \setminus S_2| + |\hat{S} \setminus S_3|}{6 \cdot \text{average}(|S_1|, |S_2|, |S_3|)},$$

where $S = (S_1 \cup S'_1 \cup S''_1)$ and $\hat{S} = (S_1 \cup S_2 \cup S_3)$. In terms of computational time, we set a *runout* after 72 hours for the total computation (i.e., five learning runs of 50 formulas each).

The described experiments and evaluations are repeated twice for each case study: one with free grammar (with the only limitation of having at most 3 quantifiers and a maximum formula length of 7) and one with a restricted grammar to steer the search towards the *target formula* of each case study. In particular, we impose the following grammars:

- $\forall \pi \forall \pi'. \varphi_1(\pi, \pi') \rightarrow \varphi_2(\pi, \pi') \mid \varphi_1(\pi, \pi') \leftrightarrow \varphi_2(\pi, \pi')$ with $\varphi_1(\pi, \pi')$ and $\varphi_2(\pi, \pi')$ Boolean combinations of predicates defined over trace variables π and π' for the *Addition Function* and the *Trigonometric Functions*;
- $\forall \pi. \mathbf{G}(\varphi(\pi))$, with φ a generic LTL specification for the *Temporal Testers* and the *Philosophers*;
- Grammar defined by Eq. (1) but with two universal quantifiers for the *Serial Adder* and the *Microcontroller*;
- $\forall \pi \forall \pi'. \mathbf{G}(\varphi_1(\pi, \pi') \rightarrow \varphi_2(\pi, \pi'))$, where $\varphi_1(\pi, \pi')$ and $\varphi_2(\pi, \pi')$ are two generic STL templates defined over the trace variables π and π' for the *AD Parking*.

5.3 Experimental Results

We present here the experimental results that we have carried out, providing an answer to the research questions.

5.3.1 Quantitative Analysis. For each of the 11 case studies, Table 2 reports the average and standard deviation over RFP(S_i) for $i = 1, 2, 3$ as the ratio of false positives RFP and over RFN(S_i) as the ratio of false negatives RFN, the variability of the formulas mined on the same trace set V_{same} and on different trace sets V_{diff} and the average and standard deviation of the computational time needed to learn a set of specifications. Before analyzing the table, we point out that we could not generate traces for the *Philosophers* case study as the available dataset consists of a single trace log. Hence,

for only this case study, the free setting imposes the use of just one quantifier. Furthermore, to reproduce the experimental setup adopted for the other case studies, we divided the available log trace in three sub-traces of the same length corresponding to $L_1 = P_3$, $L_2 = P_1$, $L_3 = P_2$ such that L_1 , L_2 and L_3 are disjoint sets and so it is each pair (L_i, P_i) for $i = 1, 2, 3$. Moreover, we note that, although the target number of formulas is set to 50 for all executions of the tool, it was not always possible in the constrained setting to learn so many formulas due to the limited number of valid specifications admitted by the restricted grammar. For this reason, the last column of Table 2 reports the average and standard deviation of the number of mined formulas over the five learning runs.

From Table 2, we first observe that the ratio of false positives RFP is quite high in the free setting of all case studies (from a minimum of 0.28 till a maximum of 0.98), meaning that the learned formulas are probably overgeneralizing the characteristics of the training datasets because they are satisfied also by randomly generated sets of traces. Accordingly, the ratio of false negatives RFN is small, as the mined formulas are satisfied also by the other sets of traces generated by the same system used for the training datasets. If we compare these outcomes with the corresponding ones obtained in the constrained setting, we see that both RFP and RFN improve significantly: from the drop-off of RFP we deduce that the mined specifications are now much tighter and they impose stricter requirements. Since the corresponding value of RFN does not increase but actually decreases, we can conclude that the achieved strictness is not an overfitting of the training data, as the mined formulas are also satisfied by unseen sets of positive examples. Therefore, as expected, the presented results show that the embedding of knowledge into the mining process through restrictions on the grammar is helpful in steering the search towards hyperproperties that are consistent with the system that generated the data, without overgeneralizing the system's properties.

By studying the variability, we observe that the values of V_{same} and V_{diff} for each setting of each case study are always extremely similar. This reveals that the variability in the mined formulas is not significantly affected by the employment of different training sets, rather than by the randomness intrinsic in the learning process. The variability values reported in Table 2 are often very high (close to the maximum value of 1), which is undesirable because it implies that different runs may produce highly different results. However, we remark that: (i) The equivalence between different formulas in our study is syntactic and not semantics, hence, for example, $\psi_1 = \forall \pi.(\pi[x] = \pi[y])$ and $\psi_2 = \forall \pi.(\pi[y] = \pi[x])$ are considered as two different specifications. Therefore, it is very likely that the specifications are more homogeneous than they appear and the reported variability values should be considered as upper bounds for the actual semantic variability. (ii) In all case studies (apart from the *Addition* and the *Philosophers*), the constrained setting produces much less variable results than the corresponding free setting. This reveals that one reason behind the high variability in the free learning is the high number of existing satisfied formulas, among which our method draws randomly. Indeed, when the grammar is restricted and the consequent number of admissible formulas is reduced, the variability decreases effectively.

Finally, we observe that the free setting is consistently more computational expensive than the constrained one due to the presence of hyperproperties with three quantifiers. In this case, the monitoring process becomes extremely costly, as the number of LTL/STL monitors in the worst case scenario is exponential in the number of quantifiers. In particular, we registered a *runout* for the free learning in the *Microcontroller* case study. This application is indeed the most expensive one for the high number of variables (35) and the length of the traces (15884 time units).

5.3.2 Qualitative Analysis. In this section, we conduct a qualitative analysis of some of the specifications mined by our approach. For each subclass of HyperSTL, we report (i) one example of interesting specification (possibly different from the target formula of the relative case study); (ii) a trivial or non-relevant formula; (iii) an incorrect formula, which is valid on the training set, but not

on the system in general. The variable names refer to the notation introduced for the corresponding case study. The complete list of mined formulas is available online³.

Hyper-proposition: *Trigonometry*

$$\forall x \forall y. (\sin x > \sin y) \leftrightarrow (\arcsin x > \arcsin y) \quad (4)$$

$$\exists x \forall y. \neg(\arccos y > x \leftrightarrow \arccos y \geq \arccos x) \quad (5)$$

$$\forall x \forall y. (x \leq \sin y) \leftrightarrow (x < \sin y) \quad (6)$$

Specification (4) represents an interesting unexpected hyperproperty stating that \sin and \arcsin are monotonically related in the interval $(-1, 1)$, while (5) is not particularly relevant as it just affirms that a possible equivalence does not hold. Finally, formula (6) is not valid in general as in the case $x = \sin y$ the proposed equivalence is actually false.

LTL: *Always Temporal Tester*

$$\forall \pi. \mathbf{G}(\pi[u] \rightarrow (\mathbf{F}\pi[w])) \quad (7)$$

$$\forall \pi. \mathbf{G}(\pi[w] \mathbf{U}(\pi[u] \leftrightarrow \pi[u])) \quad (8)$$

$$\forall \pi. \neg \mathbf{X} \mathbf{X} \mathbf{G} \pi[w] \quad (9)$$

Reminding that, at each time step t , $\pi[u[t]] = 1$ if and only if $\mathbf{G}(\pi[w], t)$ holds, we observe that the LTL formula (7) informs us that the satisfaction of the temporal operator *globally* \mathbf{G} implies the satisfaction of *finally* \mathbf{F} , which is an important property for LTL semantics. Conversely, (8) is a trivial specification as the second term in the *until* \mathbf{U} operator is a tautology, while (9) is satisfied by the training dataset but, in general, it may exist a trace $\hat{\pi}$ such that, starting (at least) from the third time step, its variable w is always 1.

HyperLTL: *Serial Adder*

$$\forall \pi \forall \pi'. \mathbf{G}(\pi[x] = \pi'[x] \wedge \pi[y] = \pi'[y]) \rightarrow (\mathbf{G}(\pi[z] = \pi'[z])) \quad (10)$$

$$\forall \pi \exists \pi'. \pi[x] = \pi'[x] \quad (11)$$

$$\forall \pi \forall \pi'. \mathbf{G}(\pi[y] \neq \pi'[y]) \rightarrow \mathbf{G}(\pi[z] = \pi'[z]) \quad (12)$$

Formula (10) is extremely relevant as it represents the *observational determinism* of variable z with respect to the two input variables x and y : if two traces agree on the same values for x and y , they have to agree also on the output variable z . We consider the HyperLTL specification (11) trivial because it states that for every trace there exists another starting with the same value on the x variable. Since x is Boolean, such information is not very informative. Finally, (12) is incorrect, as the Serial Adder can output non-identical traces (z variable), even though their second inputs y are always different.

STL/HyperSTL: *AD Parking*

$$\forall \pi. \mathbf{G}(\pi[v] < 12.75) \quad (13)$$

$$\forall \pi \forall \pi' \forall \pi''. (\mathbf{G}(|\pi[d] - \pi'[d]| < 0.1 \vee \pi[d] > 50)) \mathbf{U}(\pi''[v] < 0.1) \quad (14)$$

$$\exists \pi \forall \pi'. \mathbf{G}(|\pi[d] - \pi'[d]| < 12.57) \quad (15)$$

Specification (13) is an interesting STL property defining the upper bound of the car's velocity in all executions; such value is consistent with the (supposed unknown) set up of the case study where the car's target velocity varies in the interval $[5.55, 11.11]$ mps. To get a tighter bound on the maximum velocity, the user can decrease the threshold for the uncertainty region area in the parameter synthesis phase (in this application, the threshold was set to one tenth of the input parameter interval). (14) is not an interesting HyperSTL formula as at the beginning of every episode the car starts from a standstill, so the velocity is below 0.1 mps. Consequently, the second

term for the *until* operator is immediately satisfied and the condition in the first term is irrelevant. To conclude, (15) is valid for the traces in the learning dataset, but in principle it is not a requirement of the case study set up, so it may exist a trace whose variable d assumes a value whose distance from the d -values of the other traces is greater than 12.57 meters.

To summarize and to answer **RQ2**, we remark that the presence of incorrect formulas represents the main inevitable limitation of the *passive learning* approaches: the absence of guarantees for the mined properties to hold in general for the system that generated the data. For this reason, a manual inspection of the learned formulas is required. Nevertheless, the mined specifications remain useful to infer new interesting insights into the given set of traces.

5.3.3 Flexibility of the Mining Procedure. The experimental results show that our mining procedure can be effectively used to infer useful specifications. We have seen that different case studies have different requirements. For instance, certain systems operate in a stateless manner, while others are sequential. Some have properties that can be described based on single executions, while others necessitate reasoning about multiple executions. We have demonstrated that we can effectively use our procedure in a flexible manner to address this large variety of requirements. Table 3 summarizes this observation and shows that we were able to use our approach to mine (i) stateless hyperproperties, (ii) LTL, (iii) HyperLTL, (iv) STL, and (v) HyperSTL. To the best of our knowledge, we also developed the first approach that is able to mine properties with quantifier alternation. This flexibility is achieved by putting appropriate restrictions on the grammar that is explored by our procedure. To summarize, we give a positive answer to the research question **RQ3**.

Table 3. The applicability of the specification mining procedure.

Case Study	Specification
Addition	Hyper-propositional
Trigonometry	Hyper-propositional
Temporal Tester	LTL
Dining Philosophers	LTL
Serial Adder with Carry-On	HyperLTL
Microcontroller	HyperLTL
AD Parking	STL/HyperSTL

5.3.4 Scalability and Computational Efficiency. In this section, we study the computational efficiency of our mining procedure and how it scales. Consequently, we perform several experiments and report the results.

We first perform an experiment in the Serial Adder with Carry On case study to analyze the time required to mine HyperLTL formulas. Figure 6a shows the comparison of the average time (over 50 runs) needed to learn one formula for fixed formula lengths (from 3 to 7). The plot distinguishes among three different setups: the grammar only imposes the employment of two quantifiers (*free quant.*), the grammar is restricted to the $\forall\forall$ quantifiers (*constrained quant.*) and the search is limited to the input-output relation template given by Eq. (1) but with two universal quantifiers (*template*). For all formula lengths, *constrained quant.* requires an amount of time notably greater than the *free* setup. There are at least two explanations for this effect: (i) Intuitively, it is much easier for a formula to hold for a dataset when existential quantifiers are used rather than when only universal quantifiers are employed. In formulas,

$$\forall\pi\forall\pi' \varphi(\pi, \pi') \rightarrow \forall\pi\exists\pi' \varphi(\pi, \pi'), \exists\pi\forall\pi' \varphi(\pi, \pi') \rightarrow \exists\pi\exists\pi' \varphi(\pi, \pi').$$

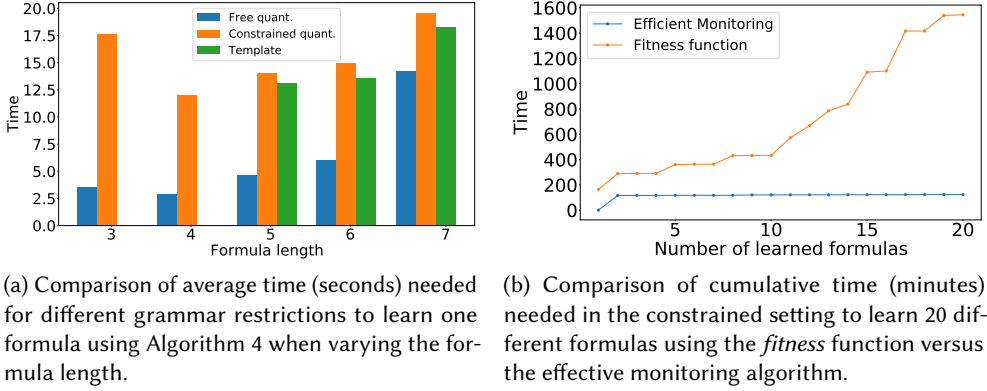


Fig. 6. Analysis of time required by our method in the different setups of the Serial Adder case study.

(ii) With our effective monitoring technique (Algorithm 4), it is faster to monitor formulas with existential quantifiers rather than universal ones. Indeed, in the former case, as soon as a witness satisfying the specification is found, the monitoring algorithm stops. Conversely, for the $\forall\forall$ quantifiers, all the pairs of traces need to be evaluated to conclude that the overall hyperproperty is satisfied by the dataset⁶. From Figure 6a we can also conclude that, as expected, for increasing formula lengths (and consequent complexity), the average time increases as well. Such increase is monotonic in all three settings, with the only exception of length 3 for *constrained quant.* An explanation could be that the relative space of satisfied formulas is very limited and, therefore, it is more time-consuming to sample several formulas in it. Finally, we observe that the *template* setup is comparable to the *constrained quant.* one in terms of time.

The experiments described so far were all carried out using the effective monitoring algorithm (Algorithm 4) to compute the score of the candidate hyperproperties. In the next experiment, we use instead the *fitness* function (Algorithm 3). Figure 6b depicts the comparison in terms of time between the two monitoring algorithms to learn 20 formulas in the constrained-quantifiers scenario, namely when the quantifiers are set to $\forall\forall$. The *fitness* function requires a quantity of time that increases exponentially with the number of learned formulas, while the effective monitoring approach remains almost constant (the mining of the first formulas is the most expensive because at the beginning our method enumerates all admissible formulas for the sampled length; such enumeration is stored and reused in the remaining learning process). The reason for the disparity between *fitness* function and monitoring algorithm is that, for every candidate hyperproperty, the *fitness* function always requires the evaluation of all pairs of traces (that, in this case, are $50 \cdot 49 = 2450$) to return a quantitative value representing the satisfaction or violation of the formula. Conversely, since the effective monitoring algorithm only returns a Boolean outcome, it stops as soon as a violation is detected and monitors 2450 pairs of traces only in the worst case scenarios. Consequently, it is evident that the *fitness* function does not scale properly and it is hardly usable for learning purposes. However, it is very helpful for a different task: the ranking of the learned formulas. In other words, we can use the *fitness* function to establish which ones in the set of learned formulas are the most interesting ones, in the sense of being the tightest satisfied for the dataset under study. For example, we compute the *fitness* value F for specifications (10) and (11) and obtain 1 and 23, respectively. These quantitative values confirm our previous manual

⁶We note that the situation is opposite if we are interested in monitoring the property violations.

inspection: formula (11) appears quite generic, while the $\forall\forall$ specification (10) represents a more specific characterization of the set of traces. Hence, the *fitness* function should be leveraged to replace (or, at least, support) the role played by the user in analyzing the outcomes of the learning method.

Finally, we analyze the performance of the monitoring algorithm (Algorithm 4) and its improvement for HyperSTL formulas based on the STL correctness proposition that we introduced in Section 4.1. We consider the AD Parking application and compare the number of tuples of traces that have to be monitored to evaluate formulas with two quantifiers for (i) a brute-force monitoring algorithm (e.g., the *fitness* function) that always evaluates all the tuples of traces (in this case, $185 \cdot 184 = 34040$); (ii) the effective monitoring algorithm; (iii) the effective monitoring algorithm with the STL correctness proposition improvement. We sample 80 HyperSTL specifications (20 for each pair of quantifiers), both satisfied and violated by the parking scenario dataset. Figure 7 depicts the average number in a logarithmic scale of the number of pair of traces evaluated by each monitoring algorithm. Although in the worst-case scenario the effective monitoring algorithm needs to evaluate all pairs of traces, we observe that, in practice, such number is much smaller and drops of two orders of magnitude in the formulas with alternation of quantifiers. The improvement in alternation-free hyperproperties is more restrained because the satisfied universally-quantified formulas and the violated existentially-quantified ones inevitably require the evaluation of all pairs of traces.

Finally, we observe that the improvement based on the STL robustness correctness proposition further enhances the monitoring algorithm performances, although the most evident speed-up is achieved by the efficient monitoring.

As the answer to **RQ4**, we can observe that we could successfully apply our method to many case studies with reasonable computational effort (Table 2). The limitation bottleneck remains the number of monitoring checks that need to be done and that is exponential in the number of formula quantifiers. This limitation is in practice partially addressed by the two heuristics that explain intrinsic robustness and quantifiers properties to avoid many calls to the LTL/STL monitor. Since such heuristics cannot be applied to the *fitness* function, its computation remains significantly slow and hardly applicable to large datasets. Nevertheless, it represents a valid help for the user when the learned formulas need to be analyzed.

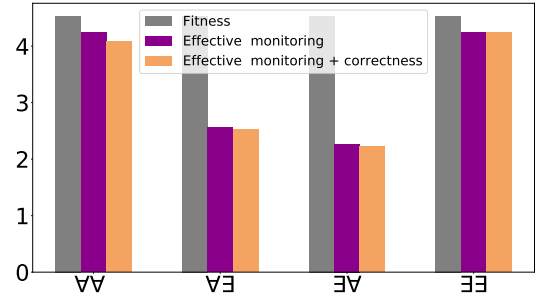


Fig. 7. Average number (logarithmic scale) of pairs of traces monitored in the AD Parking case study.

5.3.5 Comparison with the State-of-the-Art. We note that the other existing methods in the literature can only mine subsets of specifications that can be handled by our approach. We therefore restrict the grammar accordingly to make the comparison meaningful.

Comparison with Texada. Texada [18, 19] is a tool for mining LTL formulas from fixed user-defined templates, where only propositional variables can be inferred.

We run experiments for the *Philosophers* case study reproducing the experimental setting in [18]. More specifically, we first restrict the grammar to template (3) and learn all the intended LTL formulas reported in [18] in 4.6 seconds, while Texada required 0.035 seconds.. Such properties

express the intrinsic requirements of the Dining Philosophers problem: (i) Two adjacent philosophers cannot eat at the same time (e.g., $\mathbf{G}(\text{philosopher 1 is eating} \rightarrow \neg \text{philosopher 2 is eating})$) and (ii) Each philosopher can only perform one action at a time (e.g., $\mathbf{G}(\text{philosopher 1 is thinking} \rightarrow \neg \text{philosopher 1 is hungry})$). In the second step, we restrict the grammar to the other template reported in [18]: $\mathbf{F}(\text{Predicate}_{i,j} \wedge \text{Predicate}_{i',j'})$ and effectively mine the five LTL formulas representing the intended property: each pair of non-adjacent philosophers is allowed to eat at the same time. Our method needed 1.08s, while Texada 0.044s.

Finally, to infer new insights from the traces, we run a new experiment with the less restrictive grammar of the form $\forall \pi. \mathbf{G}\varphi(\pi)$, where $\varphi(\pi)$ is an arbitrary LTL formula. The time required to learn 50 formulas for this run is 133s. Among the other mined properties, we find formulas in the form of template (3), meaning that our approach can successfully learn this property even without imposing its template. Moreover, we learn another interesting formula: $\gamma = \mathbf{G}(\text{philosopher 2 is eating} \rightarrow \neg \mathbf{X}(\text{philosopher 2 is hungry}))$. In words, γ states that, for the second philosopher, it is not possible to change state from eating to being hungry. Inspired by this result, we repeat the experiments by imposing the template of γ and, in 7.6s, we discover that, for any philosopher, the only allowed changes of actions are: $\text{thinking} \rightarrow \text{hungry} \rightarrow \text{eating} \rightarrow \text{thinking}$. This final outcome shows that a template-based approach, on the one hand, allows for a focused search on known features, but that, on the other hand, it is not capable of inferring new knowledge. This is one limitation of Texada that we overcome by allowing the user to run experiments both with and without prior knowledge.

Comparison with HyperMiner. HyperMiner [26] is a template-based alternation-free approach to learn HyperLTL specifications from traces, targeting mainly systems-on-chip (SoC) design. The authors show the applicability of their approach on several SoC designs, including the *Microcontroller* case study considered in this paper. HyperMiner is run using two formula templates: the observational determinism and a similar deterministic dependency that has to hold for each time step. In [26], authors report they need nearly 10 minutes to learn 239 formulas. In this experiment, we qualitatively compare our approach to HyperMiner, by demonstrating that we are able to infer useful specifications beyond the above two templates. Indeed, contrary to [26], our method can learn interesting formulas with quantifier alternation and without a predetermined template. Some examples of such formulas mined with free grammar include:

$$\begin{aligned} & \exists \pi \forall \pi'. (\pi[x] = \pi'[x]) \\ & \forall \pi \forall \pi' \exists \pi''. (\pi[y] = \pi''[y]) \leftrightarrow (\pi'[z] = \pi''[z]) \\ & \forall \pi \forall \pi' \forall \pi''. (\pi'[p] = \pi''[p]) \cup \pi[q] = \pi'[q] \end{aligned}$$

where the variables x is *mem_intrfc_inc_pc*, y is *mem_intrfc_op2_buff*, z is *mem_intrfc_ddat_ir*, p is *mem_intrfc_pcs_source*, and q is *mem_intrfc_imm_r*.

We then consider the experiments with grammar constrained to equations (1) and verify that the observational determinism relationship holds between many pairs of variables. In some cases, a variable appears a deterministic function of two variables (instead of only one), such as

$$\forall \pi \forall \pi'. \mathbf{G}(\pi[r] = \pi'[r] \wedge \pi[s] = \pi'[s]) \rightarrow \mathbf{G}(\pi[t] = \pi'[t]),$$

where variables r , s and t correspond to the design variables *mem_intrfc_imm2_r*, *decoder_state* and *mem_intrfc_dmem_wait*. We remark that such hyperproperty could not be learned by [26], as it only supports the one-variable observational determinism template. However, the time required by our approach to learn 50 formulas in this setting is significantly higher: around three hours.

To summarize, our method can reproduce the results of other mining tools while providing much more flexibility and expressiveness. More specifically, we are unaware of any different approach capable of covering both template-based and template-free mining, with arbitrary control of domain

knowledge that can be embedded between these two extremes. As expected, our approach is quite slower because its broad applicability prevents it from benefiting from optimizations used by the other tools for specific templates expressed in a fixed specification language. Overall, the presented experiments give a positive answer to **RQ5**.

6 CONCLUSION AND FUTURE WORK

We introduced the first method that mines HyperSTL and quantifier-alternated HyperLTL specifications from system executions. The method leverages the syntax-guided synthesis allowing the user to embed as much prior knowledge as they want through the definition of the input grammar.

In summary, the main novelties of the paper are:

- Adaptation of SyGuS stochastic search approach to the mining of HyperLTL (Algorithm 1) and HyperSTL (Algorithm 5);
- Introduction of two heuristics to improve the monitoring of hyperproperties: the efficient monitoring algorithm (Algorithm 4) based on the quantifiers semantics and the robustness-based HyperSTL monitoring improvement (Section 4.1), which relies on the correctness property of STL quantitative semantics;
- Introduction of the fitness function described by Algorithm 3 to quantify the degree of satisfaction/violation of hyperproperties. Although its expensive monitoring renders impractical its implementation in the mining process, it can be leveraged to help the user in assessing the quality of the learned formulas;
- Extensive evaluation on several case studies characterized by different specification languages and analysis of the impact of the domain-knowledge on the quality of the mined formulas.

In future works, we aim to address the limitation of the passive mining approach by developing an active learning method that can steer the generation of new traces from the system. Finally, we are interested in improving the exploration of the formula space by employing different fitness functions, such as the STL robustness, to drive the search.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956123 and it is partially funded by the TU Wien-funded Doctoral College for SecInt: Secure and Intelligent Human-Centric Digital Technologies. The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme. Finally, the authors would like to thank Leonardo Mariani for his insightful comments and suggestions on the earlier draft of the paper.

REFERENCES

- [1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proc. of FMCAD 2013*. IEEE, 1–8. <https://doi.org/10.1109/FMCAD.2013.6679385>
- [2] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106. [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
- [3] Eugene Asarin, Alexandre Donzé, Oded Maler, and Dejan Nickovic. 2011. Parametric Identification of Temporal Properties. In *Proc. of RV 2011 (LNCS, Vol. 7186)*. Springer, 147–160. https://doi.org/10.1007/978-3-642-29860-8_12
- [4] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. 2018. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications. In *Lectures on Runtime Verification - Introductory and Advanced Topics*. LNCS, Vol. 10457. Springer, 135–175. https://doi.org/10.1007/978-3-319-75632-5_5
- [5] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). LNCS, Vol. 10457. Springer, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

- [6] Ezio Bartocci, Thomas Ferrère, Niveditha Manjunath, and Dejan Nickovic. 2018. Localizing Faults in Simulink/Stateflow Models with STL. In *Proc. of HSCC 2018*, Maria Prandini and Jyotirmoy V. Deshmukh (Eds.). ACM, 197–206. <https://doi.org/10.1145/3178126.3178131>
- [7] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Nickovic. 2021. CPSDebug: Automatic failure explanation in CPS models. *Int. J. Softw. Tools Technol. Transf.* 23, 5 (2021), 783–796. <https://doi.org/10.1007/s10009-020-00599-4>
- [8] Ezio Bartocci, Leonardo Mariani, Dejan Ničković, and Drishti Yadav. 2023. Property-Based Mutation Testing. In *Proc. of ICST 2023*. 222–233. <https://doi.org/10.1109/ICST57152.2023.00029>
- [9] Ezio Bartocci, Cristinel Mateis, Eleonora Nesterini, and Dejan Nickovic. 2022. Survey on mining signal temporal logic specifications. *Inf. Comput.* 289, Part (2022), 104957. <https://doi.org/10.1016/j.ic.2022.104957>
- [10] Giuseppe Bombara, Cristian Ioan Vasile, Francisco Penedo, Hirotoshi Yasuoka, and Calin Belta. 2016. A Decision Tree Approach to Data Classification using Signal Temporal Logic. In *Proc. of HSCC 2016*. ACM, 1–10. <https://doi.org/10.1145/2883817.2883843>
- [11] Michael R. Clarkson, Bernd Finkbeiner, Masoud Kolehini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *Proc. of POST 2014 (LNCS, Vol. 8414)*. Springer, 265–284. https://doi.org/10.1007/978-3-642-54792-8_15
- [12] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (2010), 1157–1210. <https://doi.org/10.3233/JCS-2009-0393>
- [13] Alexandre Donzé, Thomas Ferrère, and Oded Maler. 2013. Efficient Robust Monitoring for STL. In *Proc. of CAV 2013 (LNCS, Vol. 8044)*. Springer, 264–279. https://doi.org/10.1007/978-3-642-39799-8_19
- [14] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16. <https://proceedings.mlr.press/v78/dosovitskiy17a.html>
- [15] Bernd Finkbeiner, Lennart Haas, and Hazem Torfah. 2019. Canonical Representations of k-Safety Hyperproperties. In *Proc. of CSF 2019*. IEEE 32nd, 17–1714. <https://doi.org/10.1109/CSF.2019.00009>
- [16] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2020. Efficient monitoring of hyperproperties using prefix trees. *Int. J. Softw. Tools Technol. Transf.* 22, 6 (2020), 729–740. <https://doi.org/10.1007/s10009-020-00552-5>
- [17] Susmit Jha, Ashish Tiwari, Sanjit A. Seshia, Tuhin Sahai, and Natarajan Shankar. 2019. TeLEx: learning signal temporal logic from positive examples using tightness metric. *Formal Methods Syst. Des.* 54, 3 (2019), 364–387. <https://doi.org/10.1007/s10703-019-00332-1>
- [18] Caroline Lemieux and Ivan Beschastnikh. 2015. Investigating Program Behavior Using the Texada LTL Specifications Miner. In *Proc. of ASE*. 870–875. <https://doi.org/10.1109/ASE.2015.94>
- [19] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL Specification Mining (T). In *Proc. of ASE*. 81–92. <https://doi.org/10.1109/ASE.2015.71>
- [20] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *Proc. of FORMATS 2004 and FTRTFT 2004*. LNCS, Vol. 3253. Springer, 152–166. https://doi.org/10.1007/978-3-540-30206-3_12
- [21] Sara Mohammadinejad, Jyotirmoy V. Deshmukh, Aniruddh Gopinath Puranic, Marcell Vazquez-Chanlatte, and Alexandre Donzé. 2020. Interpretable classification of time-series data using efficient enumerative techniques. In *Proc. of HSCC 2020*. ACM, 9:1–9:10. <https://doi.org/10.1145/3365365.3382218>
- [22] Daniel Neider and Bishwamitra Ghosh. 2020. Probably Approximately Correct Explanations of Machine Learning Models via Syntax-Guided Synthesis. (2020). <https://doi.org/10.48550/arXiv.2009.08770>
- [23] Laura Nenzi, Simone Silveti, Ezio Bartocci, and Luca Bortolussi. 2018. A Robust Genetic Algorithm for Learning Temporal Specifications from Data. In *Proc. of QEST 2018 (LNCS, Vol. 11024)*, Annabelle McIver and András Horváth (Eds.). Springer, 323–338. https://doi.org/10.1007/978-3-319-99154-2_20
- [24] Luan Viet Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V. Deshmukh, and Taylor T. Johnson. 2017. Hyperproperties of Real-Valued Signals. In *Proc. of MEMOCODE 2017*. ACM, 104–113. <https://doi.org/10.1145/3127041.3127058>
- [25] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 46–57. <https://doi.org/10.1109/SFCS.1977.32>
- [26] Mayank Rawat, Sujit Kumar Muduli, and Pramod Subramanyan. 2020. Mining Hyperproperties from Behavioral Traces. In *Proc. VLSI-SOC 2020*. IEEE, 88–93. <https://doi.org/10.1109/VLSI-SOC46417.2020.9344106>
- [27] Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. 2017. Discovering Relational Specifications. In *Proc. of ESEC/FSE 2017*. ACM, 616–626. <https://doi.org/10.1145/3106237.3106279>
- [28] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2022. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. (2022). <https://doi.org/10.48550/arXiv.2207.02696>

Received 23 March 2023; revised 05 May 2023; accepted 30 June 2023