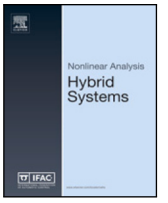
Nonlinear Analysis: Hybrid Systems 62 (2026) 101745

Contents lists available at ScienceDirect

Nonlinear Analysis: Hybrid Systems

journal homepage: www.elsevier.com/locate/nahs



Task and motion planning of dynamic systems using hyperproperties for signal temporal logic☆,☆☆

Jianing Zhao [a,c], Bowen Ye [a], Xinyi Yu [b], Rupak Majumdar [c], Xiang Yin [a,*]

[a] School of Automation and Intelligent Sensing, Shanghai Jiao Tong University, Shanghai, 200240, China
[b] Thomas Lord Department of Computer Science, University of Southern California, Los Angeles, 90089, CA, USA
[c] Max Planck Institute for Software Systems, Kaiserslautern, 67663, Germany

ARTICLE INFO

Keywords:
Signal temporal logic
Hyperproperties
Task and motion planning
Cyber–physical systems

ABSTRACT

In this paper, we investigate the task and motion planning problem for dynamic systems under signal temporal logic (STL) specifications. Existing works on STL control synthesis mainly focus on generating plans that satisfy properties over a single executed trajectory. In this work, we consider the planning problem for *hyperproperties* evaluated over a set of possible trajectories, which naturally arise in information-flow control problems. Specifically, we study discrete-time dynamic systems and employ the recently developed temporal logic HyperSTL as the new objective for planning. To solve this problem, we propose a novel recursive counterexample-guided synthesis approach capable of effectively handling HyperSTL specifications with multiple alternating quantifiers. The proposed method is not only applicable to planning but also extends to HyperSTL model checking for discrete-time dynamic systems. Finally, we present case studies on security-preserving planning and ambiguity-free planning to demonstrate the effectiveness of the proposed HyperSTL planning framework.

1. Introduction

Planning and decision-making are fundamental problems in autonomous robotics and *cyber–physical systems* (CPS). In recent years, there has been growing interest in task and motion planning for *high-level specifications* [1]. To specify objectives for CPS, various temporal logics have been developed, offering expressive and user-friendly tools for formally describing and synthesizing complex tasks. Particularly, signal temporal logic (STL) [2] provides a systematic language for describing complex tasks in continuous dynamic systems with real-valued signals. It can express specifications such as "remain in a region for at least two minutes and then reach another region within three minutes". Recently, STL-based motion planning has been extensively studied and successfully applied in various engineering CPS, including autonomous robots [3], power systems [4], and traffic management [5].

In the context of temporal logic planning for dynamic systems, a central problem is to find a sequence of control inputs such that the system trajectory satisfies a given STL specification. To solve the STL task and motion planning problem, different techniques are proposed including the mixed integer programming [6,7], control barrier functions [8] and gradient-based optimizations [9]. When accounting for system uncertainties or disturbances, the planning problem can be integrated into a model predictive control framework to design feedback controllers that dynamically compensate for real-time perturbations [6,10–12]. Recent work has

☆ This article is part of a Special issue entitled: 'TC1.3 Discrete-event and hybrid systems(IFAC WC 2026)' published in Nonlinear Analysis: Hybrid Systems.

☆☆ This work was supported by the National Natural Science Foundation of China (62573291, 62533017, 62173226).

* Corresponding author.
E-mail addresses: jnzhao@sjtu.edu.cn, jzhao@mpi-sws.org (J. Zhao), yebowen1025@sjtu.edu.cn (B. Ye), xinyi.yu12@usc.edu (X. Yu), rupak@mpi-sws.org (R. Majumdar), yinxiang@sjtu.edu.cn (X. Yin).

https://doi.org/10.1016/j.nahs.2026.101745
Received 1 November 2025; Received in revised form 1 March 2026; Accepted 14 May 2026
Available online 13 June 2026
1751-570X/

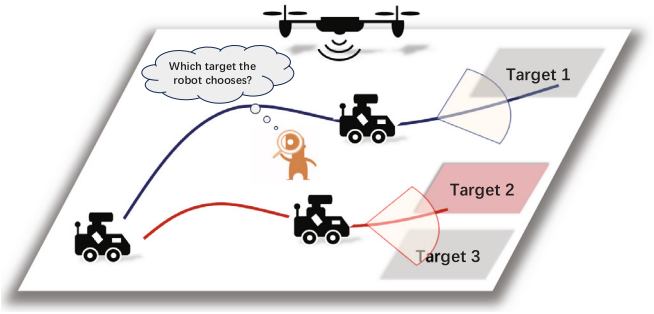


Fig. 1. A motivating example, where a mobile robot aims to reach one of the target regions while securing its real destination from being inferred by a malicious UAV prematurely. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

extended these methods to multi-agent settings, where the goal is to coordinate multiple trajectories to satisfy a global STL specification [13,14].

However, all the aforementioned works focus solely on ensuring the correctness of one system execution. In many practical applications, it is also essential to reason about the system's behavior across a set of possible executions, a concept known as *hyperproperties* [1,15–19]. To illustrate this, we consider a motivating scenario depicted in Fig. 1. Suppose a robot navigates in a workspace aiming to reach one of three possible destinations. Meanwhile, a malicious agent attempts to infer the destination of the robot, using the knowledge of (i) the robot's dynamics, (ii) its planning objective, and (iii) its real-time state trajectory. From a security perspective, the robot aims to avoid revealing its true destination prematurely. If the robot optimizes only for task completion, it could take either the blue or red trajectory shown in the figure, where the shaded sectors represent its reachable set within the next two time steps. However, choosing the blue trajectory would allow the intruder to conclusively determine that the robot is heading to Target 1 two steps in advance. In contrast, the red trajectory leaves the intruder uncertain until arrival as the robot could be moving toward either Target 2 or Target 3. Thus, the red trajectory not only fulfills the task but also preserves security by concealing critical information.

In this paper, we address the motion planning problem for dynamic systems under hyperproperties specified in HyperSTL. We focus on a fragment called existential HyperSTL, which is suitable for trajectory planning. Particularly, in the context of task and motion planning, [20] investigated the problem of synthesizing open-loop plans satisfying HyperLTL specifications. This work was later extended in [21], where the authors studied planning under HyperTWTL specifications, which is an extension of TWTL that incorporates explicit timing constraints for hyperproperties. More recently, [22] investigated the decentralized planning problem of stochastic systems subject to security requirements which are specified by PHyperLTL. However, all the aforementioned works are limited to discrete and logical specifications. For many CPS applications, we require specifications that can directly reason about continuous system dynamics, such as the nonholonomic constraints of mobile robots. While HyperSTL has been demonstrated as a powerful framework for quantitative evaluation of real-valued signals over multiple traces, the planning problem for continuous dynamic systems under HyperSTL specifications remains, to the best of our knowledge, an open challenge in the field.

Our approach builds upon the mixed integer programming-based optimization for STL trajectory planning, and counterexample-guided synthesis for STL reactive synthesis. We integrate these methods into a novel recursive counterexample-guided synthesis framework capable of handling the quantifier alternations over multiple traces inherent in HyperSTL specifications. The main contributions of this work are threefold. First, in contrast to prior works on hyperproperty synthesis in purely logical settings [20,21], our solution can handle continuous systems with real-valued signals. Second, unlike existing HyperSTL model checking approaches [23,24] that operate on finite sets of enumerated signals, our method can synthesize plans for dynamic systems that generate signals. Finally, we present the simulation results that demonstrate the applicability of our framework through case studies involving information-flow security properties in robot motion planning scenarios.

The remaining parts are organized as follows. In Section 2, we review some basic preliminaries and formulate the HyperSTL planning problem. The main synthesis algorithms are presented in Section 3. In Section 4, several case studies are presented motivated by information-flow properties for dynamic systems under STL specifications. Finally, we conclude the paper in Section 5.

2. Problem formulation

2.1. System model

We consider a dynamic system in the form of

$$\Sigma: \ x_{t+1} = f(x_t, u_t), \tag{1}$$

where $x_t \in \mathcal{X} \subseteq \mathbb{R}^n$ and $u_t \in \mathcal{U} \subseteq \mathbb{R}^m$ are the system state and control input at instant t, respectively, and $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ is the transition function describing the dynamic of the system, which is assumed to be continuous in $\mathcal{X} \times \mathcal{U}$. We assume the initial state $x_0 \in \mathcal{X}$ is given.

Suppose at time instant $t \in \mathbb{N}$, system Σ is in state $x_t \in \mathcal{X}$. Then given a sequence of control inputs $\mathbf{u}_{t:N-1} = u_t u_{t+1} \cdots u_{N-1} \in \mathcal{U}^{N-t}$, the solution of the system is $\xi_f(x_t, \mathbf{u}_{t:N-1}) = x_{t+1} \cdots x_N \in \mathcal{X}^{N-t}$ such that $x_{i+1} = f(x_i, u_i), \forall i \geq t$. A *finite* state sequence $x_0 x_1 \cdots x_N$ is said to be a *trace* of system Σ if it is a solution from x_0 under some control inputs. We denote by $\mathcal{T}_\Sigma(x_0)$ and $\mathcal{T}_\Sigma$ the set of all traces generated from the initial state $x_0 \in \mathcal{X}_0$ and the set of all traces generated by system Σ from any states.

2.2. Signal temporal logics

To describe the high-level specifications of system trajectories, we use signal temporal logic (STL) with bounded-time temporal operators, defined by the following syntax:

$$\phi ::= \top \mid \mu_v \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathbf{U}_{[a,b]} \phi_2, \tag{2}$$

where $\top$ is the true predicate, μ_v is a predicate whose truth value is determined by the sign of its underlying predicate function $v : \mathbb{R}^n \to \mathbb{R}$, i.e., for state $x \in \mathbb{R}$, μ_v is true iff $v(x) \geq 0$. Notations $\neg$ and $\wedge$ are the standard Boolean operators "negation" and "conjunction", respectively, which can further induce "disjunction" $\phi_1 \vee \phi_2$ and "implication" $\phi_1 \to \phi_2$. Notation $\mathbf{U}_{[a,b]}$ is the temporal operator "until", where $a, b \in \mathbb{R}_{\geq 0}$ are time instants, and it can also induce temporal operators "eventually" and "always" by $\mathbf{F}_{[a,b]}\phi := \top \mathbf{U}_{[a,b]} \phi$ and $\mathbf{G}_{[a,b]} := \neg \mathbf{F}_{[a,b]} \neg\phi$.

STL formulae are evaluated on state sequences. We denote by $(\mathbf{x}, t) \vDash \phi$ that sequence $\mathbf{x}$ satisfies ϕ at instant t, and we write $\mathbf{x} \vDash \phi$ whenever $(\mathbf{x}, 0) \vDash \phi$. The reader is referred to [2] for more details on the semantics of STL. Particularly, we have $(\mathbf{x}, t) \vDash \mu_v$ iff $v(x_t) \geq 0$, and $(\mathbf{x}, t) \vDash \phi_1 \mathbf{U}_{[a,b]} \phi_2$ iff there exists $t' \in [t + a, t + b]$ such that $(\mathbf{x}, t') \vDash \phi_2$ and for any $t'' \in [t, t']$, we have $(\mathbf{x}, t'') \vDash \phi_1$. It is also useful to quantitatively evaluate the robustness of STL [25]. We denote by $\rho^\phi(\mathbf{x}, t)$ the robust value of sequence $\mathbf{x}$ at instant t and we have $(\mathbf{x}, t) \vDash \phi$ iff $\rho^\phi(\mathbf{x}, t) \geq 0$.

2.3. HyperSTL

Let $\mathcal{V} = \{\pi_1, \pi_2, \ldots\}$ be a set of trace variables, where each π_i represents an individual trace. The syntax of HyperSTL [23] is given by:

$$\Phi ::= \exists \pi . \Phi \mid \forall \pi . \Phi \mid \phi \tag{3a}$$

$$\phi ::= \top \mid \mu_v^\Theta \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathbf{U}_{[a,b]} \phi_2 \tag{3b}$$

where $\forall$ and $\exists$ are the universal and existential quantifiers, respectively. Note that ϕ is essentially an STL formula, and the only difference is that predicate μ_v^Θ is parameterized by a set of trace variables $\Theta \subseteq \mathcal{V}$. Specifically, μ_v^Θ is a predicate whose value is determined by its underlying predicate function $v : \mathbb{R}^{n \times |\Theta|} \to \mathbb{R}$ such that μ_v^Θ is true iff $v(x^\Theta) > 0$, where $x^\Theta := (x^1, \ldots, x^{|\Theta|}) \in \mathbb{R}^{n \times |\Theta|}$ is a state tuple and each $x^i \in \mathcal{X}$ denotes the corresponding state of trace π_i in Θ.

Therefore, differently from standard STL, ϕ defined in (3b) involves multiple trace variables. Let $\mathbf{x}^\Theta = (\mathbf{x}^1, \ldots, \mathbf{x}^{|\Theta|})$ be a trace tuple, and we denote $\mathbf{x}_t^\Theta := (x_t^1, \ldots, x_t^{|\Theta|})$ as the state tuple at instant t. The semantics of HyperSTL are defined over a (finite or infinite) set of traces $\mathcal{T}$ and a partial mapping (called *trace assignment*) $\Pi : \mathcal{V} \to \mathcal{T}$. We use notation $(\Pi, t) \vDash_\mathcal{T} \Phi$ to denote a HyperSTL formula Φ is satisfied by a set of traces $\mathcal{T}$ at time instant t. The validity judgement [23] of a HyperSTL formula at time instant t is defined recursively by:

$$\begin{array}{lll}
(\Pi, t) \vDash_\mathcal{T} \exists \pi . \Phi & \text{iff} & \exists \xi \in \mathcal{T} \,:\, \Pi(\pi) = \xi \wedge \xi \vDash \Phi \\
(\Pi, t) \vDash_\mathcal{T} \forall \pi . \Phi & \text{iff} & \forall \xi \in \mathcal{T} \,:\, \Pi(\pi) = \xi \wedge \xi \vDash \Phi \\
(\Pi, t) \vDash_\mathcal{T} \mu_v^\Theta & \text{iff} & v(\Pi(\Theta)_t) \geq 0 \\
(\Pi, t) \vDash_\mathcal{T} \neg\phi & \text{iff} & (\Pi, t) \nvDash_\mathcal{T} \phi \\
(\Pi, t) \vDash_\mathcal{T} \phi_1 \wedge \phi_2 & \text{iff} & (\Pi, t) \vDash_\mathcal{T} \phi_1 \wedge (\Pi, t) \vDash_\mathcal{T} \phi_2 \\
(\Pi, t) \vDash_\mathcal{T} \phi_1 \mathbf{U}_{[a,b]} \phi_2 & \text{iff} & \exists t' \in [t + a, t + b] \,:\, (\Pi, t') \vDash_\mathcal{T} \phi_2, \\
 & & \forall t'' \in [t, t'] \,:\, (\Pi, t'') \vDash_\mathcal{T} \phi_1
\end{array}$$

We say a set of traces $\mathcal{T}$ satisfies HyperSTL formula Φ, denoted by $\mathcal{T} \vDash \Phi$, if $(\Pi_\emptyset, 0) \vDash_\mathcal{T} \Phi$. We say a system Σ satisfies Φ, denoted by $\Sigma \vDash \Phi$, if $\mathcal{T}_\Sigma \vDash \Phi$.

2.4. HyperSTL planning problem

In the context of task and motion planning, one needs to find a specific trace to execute. We consider a fragment of HyperSTL called "*Existential HyperSTL*" ($\exists$-HyperSTL) that starts with the existential quantifier $\exists$ in the form of $\Phi ::= \exists \pi . \Phi'$, where Φ' is an arbitrary HyperSTL formula.

Problem 1. Given system Σ in (1) with the initial state $x_0 \in \mathcal{X}$ and the $\exists$-HyperSTL formula $\Phi = \exists \pi . \Phi'$, check whether system $\mathcal{T}_\Sigma(x_0)$ satisfies $\Phi = \exists \pi . \Phi'$. If so, find the control input sequence $\mathbf{u} \in \mathcal{U}^N$ such that $\pi = \xi_f(x_0, \mathbf{u})$ is an instance satisfying Φ.

3. Planning from HyperSTL specifications

In this section, we present our solution approach for the HyperSTL planning problem. We begin by examining two special cases: (i) alternation-free formulae and (ii) formulae with alternation depth 1. Building on these two special cases, we develop a general solution that combines and extends the techniques developed for each case.

3.1. Case of alternation-free HyperSTL

First, we consider the synthesis problem for the special case of alternation-free HyperSTL of the following form

$$\Phi = \exists \pi_1 . \exists \pi_2 . \cdots . \exists \pi_n . \phi. \tag{4}$$

This fragment can be handled by extending the technique for standard STL synthesis. Essentially, it is equivalent to consider an augmented system that consists of n copies of the original system and then to find a n-tuple of control input sequences $\mathbf{u}_1, \ldots, \mathbf{u}_n \in \mathcal{U}^N$, where $\mathbf{u}_i = u_0^i u_1^i \cdots u_{N-1}^i$, such that they jointly satisfy the STL specification ϕ.

To this end, we first encode the STL constraints by binary variable according to the technique in [6]. We denote by $\texttt{constr}(\phi)$ the set of all constraints that encode the satisfaction of STL formula ϕ.

As for the objective function, since we are only interested in the first control input $\mathbf{u}_1$, given the initial state x_0 and the controller $\mathbf{u}_1$, we could define a generic function $J : \mathcal{X}^{N+1} \times \mathcal{U}^N \to \mathbb{R}$ to evaluate the cost incurred by $\mathbf{u}_1$. Therefore, we have the following optimization problem:

$$\underset{\mathbf{u}_1, \ldots, \mathbf{u}_n \in \mathcal{U}^N}{\text{minimize}} \quad J(x_0, \mathbf{u}_1) \tag{5a}$$

$$\text{s.t.} \quad \texttt{constr}(\phi) \tag{5b}$$

$$x_{t+1}^i = f(x_t^i, u_t^i), \ \forall t = 0, \ldots, N-1 \tag{5c}$$

where $x_0^i = x_0$ for any $i = 1, \ldots, n$ and $(x_t^1, x_t^2, \ldots, x_t^n)$ is the augmented system state at time instant t. For the sake of simplicity, we denote the solution of the above optimization problem, when $\mathbf{u}_i \in \mathbb{U}_i$, as a procedure

$$\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n \leftarrow \texttt{SolvePlan}(\phi, \mathbb{U}_1, \mathbb{U}_2, \ldots, \mathbb{U}_n).$$

Hence, if $\texttt{SolvePlan}(\phi, \mathcal{U}^N, \mathcal{U}^N, \ldots, \mathcal{U}^N)$ has a feasible solution, then the first component $\mathbf{u}_1$ is the solution to our problem. For convenience, in what follows, we use notation

$$\xi_f(x_0, \mathbf{u}_1, \ldots, \mathbf{u}_n) := (\xi_f(x_0, \mathbf{u}_1), \ldots, \xi_f(x_0, \mathbf{u}_n))$$

to represent the trace tuple generated by each $\mathbf{u}_i$ from x_0 and we also denote by $\xi_f(x_0, \mathbf{u}_1, \ldots, \mathbf{u}_n) \vDash \phi$ if the STL constraint and system constraint in (5) are satisfied.

3.2. Case of HyperSTL with alternation depth one

In this subsection, we further consider the following HyperSTL with alternation depth one

$$\Phi = \exists \pi_1 . \exists \pi_2 . \cdots \exists \pi_m . \forall \pi_{m+1} . \forall \pi_{m+2} . \cdots \forall \pi_n . \phi. \tag{6}$$

The key idea is to frame this problem as an STL reactive synthesis task, where the controller must jointly synthesize the first m traces while ensuring robustness against any possible adversarial behaviors in the remaining $n-m$ traces. This problem can be solved using a *counterexample-guided synthesis* approach, as implemented in Algorithm 1.

Specifically, in line 1, we obtain initial input $\mathbf{u}_1, \ldots, \mathbf{u}_n \in \mathcal{U}^N$ by calling `SolvePlan`, where the solution space for each input is the entire input space $\mathcal{U}^N$. In line 2, for each universally quantified input $\mathbf{u}_i$ (where $m+1 \le i \le n$), we initialize candidate domain $\hat{\mathcal{U}}_i$ with the first plan $\mathbf{u}_i$. These sets serve as finite constraints for the optimization problem and are updated whenever new counterexamples are encountered. During the while-loop iteration, the algorithm checks whether the current plan for the first m components satisfies the universality requirement for the last $n-m$ components using procedure `CountCheck`. This procedure attempts to falsify the STL ϕ for the fixed current inputs $\mathbf{u}_1, \ldots, \mathbf{u}_m$ by finding counterexample instances $\mathbf{u}'_{m+1}, \ldots, \mathbf{u}'_n$. If the inputs pass this counterexample check, $\mathbf{u}_1$ constitutes a valid solution. Otherwise, the discovered counterexamples are incorporated into the candidate domains $\hat{\mathcal{U}}_{m+1}, \ldots, \hat{\mathcal{U}}_n$ in line 8. The algorithm then uses these expanded (yet still finite) candidate domains as constraints in line 10 to synthesize a new tuple of inputs $\mathbf{u}_1, \ldots, \mathbf{u}_m$ by solving the optimization problem. If at any point the optimization problem becomes infeasible given the current candidate domains, we can immediately conclude that the entire problem is infeasible. Note that in procedure `CountCheck`, rather than simply performing qualitative falsification of ϕ, we additionally minimize the robustness degree of the STL formula as a quantitative optimization objective. This approach tends to yield more effective counterexamples in practice.

Algorithm 1: Control Synthesis with Depth One

Input: System Σ and HyperSTL Φ in (6)
Output: Control input $\mathbf{u}$

1 $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n \leftarrow$ `SolvePlan`$(\phi, \mathcal{U}^N, \mathcal{U}^N, \dots, \mathcal{U}^N)$;
2 $\hat{\mathcal{U}}_i \leftarrow \{\mathbf{u}_i\}, \forall m+1 \le i \le n$;
3 **while** *True* **do**
4 $\quad$ *res*, $\mathbf{u}'_{m+1}, \dots, \mathbf{u}'_n \leftarrow$ `CountCheck`$(\mathbf{u}_1, \dots, \mathbf{u}_m, \phi)$;
5 $\quad$ **if** *res* = *True* **then**
6 $\quad\quad$ **return** $\mathbf{u}_1$;
7 $\quad$ **else**
8 $\quad\quad$ $\hat{\mathcal{U}}_i \leftarrow \hat{\mathcal{U}}_i \cup \{\mathbf{u}'_i\}, \forall m+1 \le i \le n$;
9 $\quad$ Find $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m \in \mathcal{U}^N$ that

$$\text{minimize} \quad J(x_0, \mathbf{u}_1)$$
$$\text{s.t. } \forall \mathbf{u}_i \in \hat{\mathcal{U}}_i : \xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_n) \models \phi, i = m+1, \dots, n$$

10 $\quad$ **if** the above problem has no solution **then**
11 $\quad\quad$ **return** *"task Φ is infeasible"*

12 **procedure** `CountCheck`$(\mathbf{u}_1, \dots, \mathbf{u}_m, \phi)$
13 Find $\mathbf{u}'_{m+1}, \dots, \mathbf{u}'_n \in \mathcal{U}^N$ that

$$\text{minimize } \rho^{\phi}(\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_m, \mathbf{u}'_{m+1}, \dots, \mathbf{u}'_n))$$

14 **if** $\rho^{\phi}(\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_m, \mathbf{u}'_{m+1}, \dots, \mathbf{u}'_n)) > 0$ **then**
15 $\quad$ **return** *True*, *Null*;
16 **else**
17 $\quad$ **return** *False*, $\mathbf{u}'_{m+1}, \dots, \mathbf{u}'_n$;

Remark 1. Note that Algorithm 1 is sound and complete only when the input domain $\mathcal{U}$ is a finite set. Otherwise, if $\mathcal{U}$ is an infinite set, when executing the algorithm, we could set a maximum number of iterations for the while loop. After the maximum number of iterations, we say HyperSTL Φ is infeasible for the system Σ. In this scenario, Algorithm 1 is sound but not complete. In despite of the incompleteness, we still found, in the following case studies, that if the problem is feasible, then a few number of iterations is sufficient to find a feasible solution. Therefore, the proposed algorithm remains practical to solve the HyperSTL synthesis with alternation depth 1. The readers are referred to [10] for the detailed analysis for the correctness of the counterexample-guided procedure.

3.3. Control synthesis for general HyperSTL formulae

Now, we are ready to tackle the general case of HyperSTL synthesis with arbitrary alternation depth of quantifiers.
Let k be an index of trace variable. For the sake of clarity, we can index the quantifier of each variable in Φ as follows:

$$\Phi = \exists \pi_1. Q_2 \pi_2. \cdots Q_{k-1} \pi_{k-1}. (Q_k \pi_k. Q_k \pi_{k+1}. \cdots Q_k \pi_{m_k}.)(\bar{Q}_k \pi_{m_k+1}. \bar{Q}_k \pi_{m_k+2}. \cdots \bar{Q}_k \pi_{p_k}.) Q_k \pi_{p_k+1}. \cdots Q_n \pi_n. \phi \tag{7}$$

where $\bar{Q} = \forall$ if $Q = \exists$ and $\bar{Q} = \exists$ if $Q = \forall$. Intuitively, m_k represents the last index of the consecutive quantifiers after Q_k that are same with Q_k, i.e., $Q_j = Q_k, \forall k \le j \le m_k$ and $Q_{m_k+1} = \bar{Q}_k$, while p_i denotes the last index of the consecutive quantifiers after Q_{m_k+1} that are same with Q_{m_k+1}, i.e., $Q_j = \bar{Q}_k, \forall m_k + 1 \le j \le p_k$ and $Q_{p_k+1} = Q_k$.

The key idea for solving the general case is to recursively call procedure `CheckHyper` that is designed according to both procedures `SolvePlan` and `CountCheck`. The solution is summarized as Algorithm 2. Specifically, in line 1, we obtain initial input $\mathbf{u}_1, \dots, \mathbf{u}_n \in \mathcal{U}^N$ by `SolvePlan`. If no feasible input is returned, then we claim that task Φ is infeasible for system Σ in lines 2–3. Otherwise, we begin to use the *recursive counterexample guided check* to tackle the multiple quantifier alternation in the general HyperSTL Φ in (7). We fix the obtained inputs $\mathbf{u}_1, \dots, \mathbf{u}_{m_1}$ and view the next $p_1 - m_1$ inputs as the check variables. That is, we use procedure `CheckHyper` to validate whether the traces $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{m_1})$ satisfy

$$\forall \pi_{m_1+1}. \cdots \forall \pi_{p_1}. \exists \pi_{p_1+1}. \cdots Q_n \pi_n. \phi. \tag{8}$$

Thus, we initialize candidate domain $\hat{\mathcal{U}}_i$ with $\mathbf{u}_{m_1+1}, \dots, \mathbf{u}_{p_1}$ in line 4. If satisfied, i.e., `CheckHyper` returns *True*, we know that the inputs $\mathbf{u}_1, \dots, \mathbf{u}_{m_1}$ are feasible and return $\mathbf{u}_1$ as the output in lines 7–8. Otherwise, `CheckHyper` returns *False* together with the counterexample instances $\mathbf{u}'_{m_1+1}, \dots, \mathbf{u}'_{p_1}$. Then, we add them to candidate domains $\hat{\mathcal{U}}_i, \forall i = m_1 + 1, \dots, p_1$ in lines 9–10 and repeat finding new inputs $\mathbf{u}_1, \dots, \mathbf{u}_n$ in line 11 until `CheckHyper` returns *True*. If at any point the optimization problem becomes infeasible given the current candidate domains, we immediately claim that the problem is infeasible in lines 12–13.

Algorithm 2: Synthesis of General HyperSTL

Input: System Σ and general HyperSTL Φ in (7)
Output: Control input sequence $\mathbf{u}$

$\mathbf{u}_1, \dots, \mathbf{u}_n \leftarrow \texttt{SolvePlan}(\phi, \mathcal{U}^N, \dots, \mathcal{U}^N)$;
if the above problem has no solution **then**
  return *"task Φ is infeasible"*
$\hat{\mathcal{U}}_i \leftarrow \{\mathbf{u}_i\}, \forall m_1 + 1 \le i \le p_1$;
while *True* **do**
  $res, \mathbf{u}'_{m_1+1}, \dots, \mathbf{u}'_{p_1} \leftarrow \texttt{CheckHyper}(\mathbf{u}_1, \dots, \mathbf{u}_{m_1}, \phi)$;
  if $res = True$ **then**
    return $\mathbf{u}_1$;
  else
    $\hat{\mathcal{U}}_i \leftarrow \hat{\mathcal{U}}_i \cup \{\mathbf{u}'_i\}, \forall m_1 + 1 \le i \le p_1$;
  Find $\mathbf{u}_1, \dots, \mathbf{u}_{m_1} \in \mathcal{U}^N$ that

$$\begin{aligned} &\text{minimize} \quad J(x_0, \mathbf{u}_1) \\ &\text{s.t. } \forall \mathbf{u}_i \in \hat{\mathcal{U}}_i, \exists \mathbf{u}_j \in \mathcal{U}^N \ : \ \xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_n) \models \phi, \\ &\quad i = m_1 + 1, \dots, p_1, \ \ j = p_1 + 1, \dots, n \end{aligned}$$

  if the above problem has no solution **then**
    return *"task Φ is infeasible"*

procedure $\texttt{CheckHyper}(\mathbf{u}_1, \dots, \mathbf{u}_{k-1}, \psi)$
if $\psi = \phi, Q_i = \forall, \forall k \le i \le n$ *or* $\psi = \neg\phi, Q_i = \exists, \forall k \le i \le n$ **then**
  $res, \mathbf{w}_k, \dots, \mathbf{w}_n \leftarrow \texttt{CountCheck}(\mathbf{u}_1, \dots, \mathbf{u}_{k-1}, \psi)$;
  if $res = True$ **then**
    return *True, Null*
  else
    return *False*, $\mathbf{w}_k, \dots, \mathbf{w}_n$;
else
  $\mathbf{w}_1, \dots, \mathbf{w}_{k-1}, \mathbf{w}_k, \mathbf{w}_{k+1}, \dots, \mathbf{w}_n \leftarrow \texttt{SolvePlan}(\psi, \{\mathbf{u}_1\}, \dots, \{\mathbf{u}_{k-1}\}, \mathcal{U}^N, \dots, \mathcal{U}^N)$;
  $\hat{\mathcal{U}}'_i \leftarrow \{\mathbf{w}_i\}, \forall m_k + 1 \le i \le p_k$;
  while *True* **do**
    $res, \mathbf{w}'_{m_k+1}, \dots, \mathbf{w}'_{p_k} \leftarrow \texttt{CheckHyper}(\mathbf{w}_1, \dots, \mathbf{w}_{k-1}, \mathbf{w}_k, \dots, \mathbf{w}_{m_k}, \neg\psi)$;
    if $res = True$ **then**
      return *False*, $\mathbf{w}_{m_k+1}, \dots, \mathbf{w}_{p_k}$;
    $\hat{\mathcal{U}}'_i \leftarrow \hat{\mathcal{U}}'_i \cup \{\mathbf{w}'_i\}, \forall m_k + 1 \le i \le p_k$;
    Find $\mathbf{w}_k, \mathbf{w}_{k+1}, \dots, \mathbf{w}_{m_k} \in \mathcal{U}^N$ that

$$\begin{aligned} &\text{minimize} \quad J(x_0, \mathbf{w}_1) \\ &\text{s.t. } \forall \mathbf{w}_i \in \hat{\mathcal{U}}'_i, \exists \mathbf{w}_j \in \mathcal{U}^N \ : \ \xi_f(x_0, \mathbf{w}_1, \dots, \mathbf{w}_n) \models \psi, \\ &\quad i = m_k + 1, \dots, p_k, \ \ j = p_k + 1, \dots, n \end{aligned}$$

    if the above problem has no solution **then**
      return *True, Null*;

Next, we elaborate on the details of procedure `CheckHyper`. At recursion level k, `CheckHyper` takes as input the first $k-1$ control sequences $(\mathbf{u}_1, \dots, \mathbf{u}_{k-1})$, which instantiate the trace variables $\pi_1, \dots, \pi_{k-1}$. Its role is to verify whether the remaining quantified suffix over $\pi_k, \dots, \pi_n$ holds for the induced trajectory tuple $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{k-1})$. More precisely, given $\psi \in \{\phi, \neg\phi\}$, `CheckHyper` checks whether $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{k-1})$ satisfies

$$\forall\pi_k.\forall\pi_{k+1}.\cdots\forall\pi_{m_k}.\exists\pi_{m_k+1}.\cdots\exists\pi_{p_k}.\forall\pi_{p_{k+1}}.\cdots Q_n\pi_n.\psi, \tag{9}$$

where $\psi \in \{\phi, \neg\phi\}$ is the formula to check. The procedure returns *True* if (9) holds; otherwise, it returns *False* together with a counterexample for the next universal block, which is used by Algorithm 2 to update the candidate sets and re-solve. We first determine whether the remaining HyperSTL is already alternation-free. If so, i.e., all the remaining quantifiers are $\forall$ or $\exists$, then we

try to find a counterexample $\mathbf{w}_k, \dots, \mathbf{w}_n$ that falsifies ψ by calling `CountCheck` in line 17. If the counterexample does not exist, then we claim that $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{k-1})$ satisfies (9) and return *True* in lines 18–19. Otherwise, we return *False* together with the counterexample in lines 20–21. On the other hand, if there are still quantifier alternations, then we fix the given inputs $\mathbf{u}_1, \dots, \mathbf{u}_{k-1}$ and begin to check whether there are inputs $\mathbf{w}_k, \mathbf{w}_{k+1}, \dots, \mathbf{w}_{m_k}$ that falsify (9). This is equivalent to check whether or not there are inputs $\mathbf{w}_k, \mathbf{w}_{k+1}, \dots, \mathbf{w}_{m_k}$ such that $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{k-1}, \mathbf{w}_k, \dots, \mathbf{w}_{m_k})$ satisfies

$$\forall \pi_{m_k+1}. \cdots \forall \pi_{p_k}. \exists \pi_{p_k+1}. \cdots \bar{Q}_n \pi_n. \neg \psi. \tag{10}$$

To this end, we first find candidates for $\mathbf{w}_k, \dots, \mathbf{w}_n$ in line 23. Here, we have $\mathbf{w}_i = \mathbf{u}_i, \forall i = 1, \dots, k-1$. Then, for the $m_k + 1$th to p_kth inputs, we add $\mathbf{w}_i$ to candidate domains in line 24 and begin to repeat finding $\mathbf{w}_k, \mathbf{w}_{k+1}, \dots, \mathbf{w}_{m_k}$ until the recursive procedure `CheckHyper` returns *True*, i.e., (10) is satisfied. Specifically, if no feasible solution is found, we declare that $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{k-1})$ satisfies (9), and return *True* in lines 30–33. Otherwise, we declare that the inputs $\mathbf{w}_{m_k+1}, \dots, \mathbf{w}_{p_k}$ falsify (9) and return *False* in lines 27–29. Based on the above explanation and analysis for the Algorithm 2, we obtain the following result.

Theorem 1. *Given system Σ in (1) and a general HyperSTL Φ in (7), the control inputs returned by Algorithm 2 satisfies Φ.*

Proof. We finish the proof by induction. For alternation-free HyperSTL in (4), lines 1–3 imply that Algorithm 2 returns a controller satisfying (4) whenever a feasible solution exists, and returns *"task Φ is infeasible"* otherwise.

For a general HyperSTL in (7), it suffices to establish the correctness of the procedure `CheckHyper`. Specifically, for a fixed tuple of inputs $\mathbf{u}_1, \dots, \mathbf{u}_{k-1}$ (i.e., fixed traces $\pi_1, \dots, \pi_{k-1}$), `CheckHyper` returns *True* if and only if the induced trajectory tuple $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{k-1})$ satisfies the remaining HyperSTL suffix

$$\Psi = Q\pi_k. \cdots Q\pi_{m_k}. \bar{Q}\pi_{m_k+1}. \cdots \bar{Q}\pi_{p_k}. Q\pi_{p_k+1}. \cdots Q_n \pi_n. \psi \tag{11}$$

where Q denotes the quantifier of the current block, $\bar{Q}$ is its dual, and $Q_n = \forall$ if $\psi = \phi$ while $Q_n = \exists$ if $\psi = \neg\phi$.

Base case. When the remaining HyperSTL suffix has no quantifier alternation, e.g.,

$$\Psi_0 = \forall \pi_k. \cdots \forall \pi_n. \phi \tag{12}$$

`CheckHyper` is correct since lines 15–21 reduce Algorithm 2 to Algorithm 1, whose correctness has been established.

Alternation depth 1. Consider a remaining suffix of alternation depth 1 of the form

$$\Psi_1 = \exists \pi_k. \cdots \exists \pi_{m_k}. \forall \pi_{m_k+1}. \cdots \forall \pi_n. \phi \tag{13}$$

In line 23, we set $\mathbf{w}_i = \mathbf{u}_i$ for $i = 1, \dots, k-1$, and then lines 24–31 attempt to synthesize $\mathbf{w}_k, \dots, \mathbf{w}_{m_k}$ that satisfy (13). For a fixed candidate $\mathbf{w}_k, \dots, \mathbf{w}_{m_k}$, the candidate is infeasible if and only if there exists $\mathbf{w}_{m_k+1}, \dots, \mathbf{w}_n$ that falsifies (13), i.e., if and only if $\xi_f(x_0, \mathbf{w}_1, \dots, \mathbf{w}_{m_k})$ satisfies the dual suffix

$$\exists \pi_{m_k+1}. \cdots \exists \pi_n. \neg \phi. \tag{14}$$

This is exactly what the recursive call in line 26 checks via `CheckHyper`$(\mathbf{w}_1, \dots, \mathbf{w}_{m_k}, \neg\phi)$, which is correct by the base case above. Therefore, `CheckHyper`$(\mathbf{w}_1, \dots, \mathbf{w}_{k-1}, \phi)$ returns *True* if and only if no such falsifying witness exists, i.e., if and only if $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{k-1}) \vDash \Psi_1$. Hence, `CheckHyper` is correct for alternation depth 1.

Induction step. Assume that `CheckHyper` is correct for any remaining suffix whose alternation depth is at most d. Now consider a remaining suffix of alternation depth $d+1$ written as

$$\Psi_{d+1} = Q\pi_k. \cdots Q\pi_{m_k} \Psi_d, \tag{15}$$

where Ψ_d has alternation depth d and the first $k-1$ traces are fixed by $\mathbf{u}_1, \dots, \mathbf{u}_{k-1}$. Without loss of generality, assume $Q = \forall$ (the case $Q = \exists$ is symmetric). In line 23, we again set $\mathbf{w}_i = \mathbf{u}_i$ for $i = 1, \dots, k-1$, and lines 24–31 attempt to synthesize a candidate assignment $\mathbf{w}_k, \dots, \mathbf{w}_{m_k}$ for the current block. Such a candidate is invalid if and only if there exists a witness for the next dual block that falsifies Ψ_{d+1}, which is equivalent to the existence of $\mathbf{w}_{m_k+1}, \dots, \mathbf{w}_{p_k}$ such that $\xi_f(x_0, \mathbf{w}_1, \dots, \mathbf{w}_{m_k})$ satisfies the dual suffix

$$\exists \pi_{m_k+1}. \cdots \exists \pi_{p_k}. \forall \pi_{p_k+1}. \cdots Q_n \pi_n. \neg \psi, \tag{16}$$

whose alternation depth is d. This is precisely checked by the recursive call `CheckHyper`$(\mathbf{w}_1, \dots, \mathbf{w}_{m_k}, \neg\psi)$ in line 26. By the induction hypothesis, this recursive call is correct; hence `CheckHyper`$(\mathbf{u}_1, \dots, \mathbf{u}_{k-1}, \psi)$ returns *True* if and only if no such falsifying witness exists, i.e., if and only if $\xi_f(x_0, \mathbf{u}_1, \dots, \mathbf{u}_{k-1}) \vDash \Psi_{d+1}$.

By induction, `CheckHyper` is correct for any remaining HyperSTL suffix. Finally, lines 5–13 of Algorithm 2 use `CheckHyper` to certify each synthesized candidate: a *True* return value certifies feasibility and terminates the algorithm, whereas a *False* return value provides a violation that is used to update the candidate sets and re-solve. Therefore, Algorithm 2 returns a controller satisfying (7) whenever it is feasible, and returns *"task Φ is infeasible"* otherwise. The proof is thus completed.

Now, we present a simple example for better understanding of the above algorithm for general HyperSTL synthesis.

Example 1. Consider a HyperSTL formula $\exists\pi_1.\forall\pi_2.\exists\pi_3.\phi$ with alternation depth 2. In line 1, the initial $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$ are generated by calling $\texttt{SolvePlan}(\phi, \mathcal{U}^N, \mathcal{U}^N, \mathcal{U}^N)$. Then, our aim is to call $\texttt{CheckHyper}(\mathbf{u}_1, \phi)$ in line 6 to see if $\pi_1 = \xi_f(x_0, \mathbf{u}_1)$ satisfies $\pi_1 \models \forall\pi_2.\exists\pi_3.\phi$. If it holds, then $\mathbf{u}_1$ is the final output; otherwise, another $\mathbf{u}_1$ is synthesized in line 11 according to the current counterexamples in $\hat{\mathcal{U}}_2$. In $\texttt{CheckHyper}(\mathbf{u}_1, \phi)$, since $\forall = Q_2 \neq Q_3 = \exists$, the procedure goes to line 23 where a recursive counterexample check will be started. The general idea is to check if $\pi_1 \models \exists\pi_2.\forall\pi_3.\neg\phi$, i.e., if we can find a counterexample $\mathbf{w}_2$ such that $\xi_f(x_0, \mathbf{u}_1, \mathbf{w}_2) \models \forall\pi_3.\neg\phi$. To be specific, in line 23, initial $\mathbf{u}_1, \mathbf{w}_2, \mathbf{w}_3$ are generated where $\mathbf{w}_2$ serves as the initial counterexample. To check if $\xi_f(x_0, \mathbf{u}_1, \mathbf{w}_2) \models \forall\pi_3.\neg\phi$, we call $\texttt{CheckHyper}(\mathbf{u}_1, \mathbf{w}_2, \neg\phi)$. Since there is only one quantifier Q_3 without alternation, we call $\texttt{CountCheck}(\mathbf{u}_1, \mathbf{w}_2, \neg\phi)$ in line 17. If the check is passed, then we know $\pi_1 \not\models \forall\pi_2.\exists\pi_3.\phi$ with a counterexample $\pi_2 = \xi_f(x_0, \mathbf{w}_2)$. Therefore, the procedure $\texttt{CheckHyper}(\mathbf{u}_1, \phi)$ goes to 27 and the counterexample $\mathbf{w}_2$ in line 28 is returned to line 10 to update the counterexample candidate set. The algorithm will try to find a new $\mathbf{u}_1$ in line 11 until the $\texttt{CheckHyper}(\mathbf{u}_1, \phi)$ returns *True*. On the other hand, if $\texttt{CountCheck}(\mathbf{u}_1, \mathbf{w}_2, \neg\phi)$ returns *False*, then we know $\xi_f(x_0, \mathbf{u}_1, \mathbf{w}_2) \not\models \forall\pi_3.\neg\phi$ with the counterexample $\pi_3 = \xi_f(x_0, \mathbf{w}_3')$. Then the procedure $\texttt{CheckHyper}(\mathbf{u}_1, \phi)$ goes to line 29 to update the counterexample candidate sets and try to find another $\mathbf{w}_2$ according to the current counterexample set of $\mathbf{w}_3'$ in line 30. If there is no such $\mathbf{w}_2$, then it shows $\xi_f(x_0, \mathbf{u}_1) \not\models \exists\pi_2.\forall\pi_3.\neg\phi$. Therefore, the algorithm goes back to line 8 with the correct control input $\mathbf{u}_1$ as the result.

Remark 2. Algorithm 2 follows the same counterexample-guided refinement principle as Algorithm 1 and, in particular, subsumes Algorithm 1 as a special case. Consequently, its termination and correctness properties depend on the cardinality of the input domain $\mathcal{U}$. When $\mathcal{U}$ is finite, the refinement loops in Algorithm 2 are sound and complete. When $\mathcal{U}$ is infinite, the while loops may not terminate in theory; in practice, we impose a maximum number of refinement iterations. Under this setting, Algorithm 2 remains sound but is not complete. Nevertheless, our following case studies indicate that, whenever the instance is feasible, a feasible solution is typically found within a small number of refinement iterations.

Remark 3. Although this paper focuses on controller synthesis, Algorithm 2 can also be used for HyperSTL model checking of discrete-time systems under the same bounded-time semantics, i.e., over finite traces of the HyperSTL length. The only required modification is on the leading quantifier of the specification. If the HyperSTL formula to be checked already starts with an existential quantifier ($\exists\pi_1.\cdots\phi$), then it can be handled by Algorithm 2 without any change. If it starts with a universal quantifier ($\forall\pi_1.\cdots\phi$), we rewrite it using quantifier duality into an equivalent form with an existential leading quantifier, and then apply Algorithm 2 to the rewritten specification.

4. Applications of HyperSTL planning

In this section, we present case studies of the proposed HyperSTL planning problem by examining it from two perspectives. First, we consider the security-preserving planning problem, where a robot must prevent external observers from inferring its critical information. This problem has been explored extensively in the recent literature but in purely logical frameworks [22,26–28]. We will demonstrate that the security-preserving planning can be directly achieved by using HyperSTL specification over continuous systems without any abstraction. Second, we investigate the informed planning problem, in which the robot maintains trajectory indistinguishability from other agents' viewpoints to facilitate collaborative objectives.

4.1. Security-preserving planning

Following the motivating example, let us consider a mobile robot governed by the dynamics in (1), subject to physical constraints $\mathcal{X}$ and $\mathcal{U}$, tasked with satisfying an STL specification ϕ. We assume the presence of an intruder who possesses complete knowledge of the robot's dynamics and its task specification ϕ, but lacks access to the robot's exact control strategy. Consequently, the intruder must infer the robot's intentions solely through trajectory observations.

From the robot's perspective, certain critical states must be protected, represented by a *secret region* $\mathcal{X}_S \subseteq \mathcal{X}$. The security objective requires that whenever the robot enters $\mathcal{X}_S$, the intruder cannot definitively predict this entry before a specified prediction horizon ΔT has elapsed. This requirement can be formally captured through the notion of *pre-opacity* defined as follows:

Definition 1 (*Pre-Opacity*)**.** Given system Σ and STL formula ϕ, we say the system is *ΔT-step opaque* w.r.t. $\mathcal{X}_S$ if $\mathcal{T}_\Sigma$ satisfies

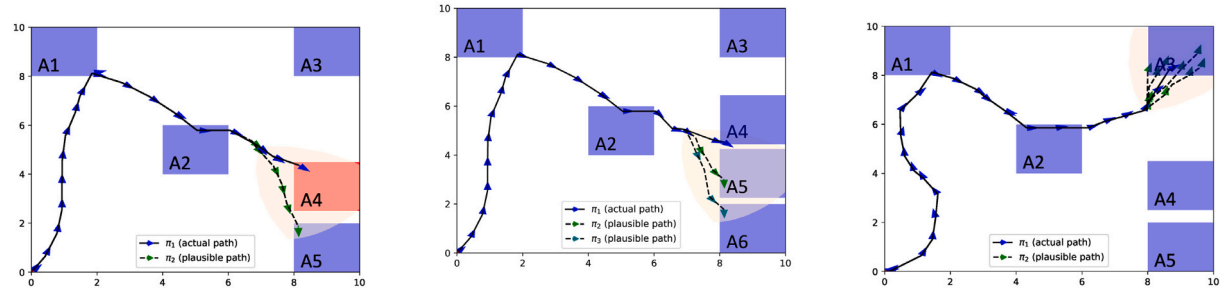
$$\exists\pi_1.\exists\pi_2.\left[\phi^{\pi_1} \wedge \phi^{\pi_2} \wedge \mathbf{G}_{[0,T_\phi]}\left(\mathbf{F}_{[\Delta T,\Delta T]}(x^{\pi_1} \in \mathcal{X}_S) \rightarrow \left[(x^{\pi_1} = x^{\pi_2}) \wedge \mathbf{G}_{[0,\Delta T]}\left(x^{\pi_2} \notin \mathcal{X}_S\right)\right]\right)\right]$$

where T_ϕ is the evaluation horizon of ϕ which is the maximum sum of all nested upper bounds.

Intuitively, this definition states that we need to find a trajectory π_1 satisfying STL task ϕ and there exists at least one trajectory π_2 satisfying ϕ such that if π_1 reaches the secret region within ΔT steps, then π_2 must not reach the secret region within ΔT steps. Thus, π_2 plays as a plausible trajectory such that the intruder cannot determine that the robot will reach the secret region within ΔT steps.

As a concrete case study, we consider a mobile robot described by the following unicycle-type model:

$$\dot{p}_x = v\cos\theta,\ \dot{p}_y = v\sin\theta,\ \dot{\theta} = \omega, \tag{17}$$



(a) Example for pre-opacity. (b) Example for anonymity. (c) Example for distinguishability.

Fig. 2. Case studies with alternation-free and alternation-depth-1 HyperSTL specifications. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

where the physical constraints are given by $x = [p_x, p_y, \theta]^T \in \mathcal{X} = [0, 10]^2 \times [-\pi, \pi]$, $[v, \omega]^T \in \mathcal{U} = [0, 1.0] \times [-\pi/15, \pi/15]$. For convenience, we denote that $\boldsymbol{p} := [p_x, p_y]^T$. To implement the proposed HyperSTL planning algorithm, we discretize system (17) by a sampling time of 0.5 s. All STL time bounds in this section are specified in discrete time steps. For example, $F_{[1,3]}$ means "eventually between steps 1 and 3", i.e, between 0.5 s and 1.5 s.

The robot operates in the workspace shown in Fig. 2(a) and the objective is to collect two kinds of packages and then transport them to the final destinations. Formally, we consider the following STL formula:

$$\phi = \mathbf{F}_{[9,11]}(\boldsymbol{p} \in A_1) \wedge \mathbf{G}_{[22,23]}(\boldsymbol{p} \in A_2) \wedge \mathbf{F}_{[40,41]}(\boldsymbol{p} \in (A_3 \cup A_4 \cup A_5)) \tag{18}$$

where $A_1 = [0, 2]\times[8, 10]$, $A_2 = [4, 6]\times[4, 6]$, are two warehouses for two kinds of packages, respectively, and $A_3 = [8, 10]\times[8, 10]$, $A_4 = [8, 10]\times[2.5, 4.5]$, $A_5 = [8, 10]\times[0, 2]$ are three available destinations. To improve the performance of the robot trajectory, we define the cost function as

$$J(x_0, \mathbf{u}_1) = -\alpha\rho^{\phi}(\xi_f(x_0, \mathbf{u}_1)) + (1 - \alpha)\sum_{i=1}^{T}(\|\boldsymbol{p}_i - \boldsymbol{p}_{i-1}\|^2), \tag{19}$$

where α is the trade-off between the robustness of the task and the length of the trajectory. We set $\alpha = 0.2$.

Now, we assume that destination A_4 serves as the secret region ($\mathcal{X}_S = A_4$) with an intruder prediction horizon of $\Delta T = 4$. We synthesize a secure control plan $\mathbf{u}_1$ based on the system dynamics and the HyperSTL formulation of pre-opacity. Fig. 2(a) illustrates both the generated trajectory $\pi_1 = \xi_f(x_0, \mathbf{u}_1)$ and its companion trajectory that collectively satisfy the pre-opacity condition. The solution trajectory π_1 successfully accomplishes the STL task ϕ from (18). Moreover, Δ-steps before the robot enters $\mathcal{X}_S$, there still exists an alternative path π_2 that avoids $\mathcal{X}_S$ yet still satisfies the STL task ϕ (by navigating to A_5). Thus, even though the intruder observes the robot's exact position at every time step, it cannot conclusively determine whether the robot will enter the secret region Δ-steps in advance when the robot follows trajectory π_1.

In the notion of pre-opacity, it is required that whenever the robot enters the secret region, there must exist an alternative trajectory leading to a non-secret region. However, in some security problems, there is no predefined partition of secret and non-secret regions. Instead, the security requirement is that the robot must maintain at least K plausible alternative intentions. This requirement aligns with the concept of K-anonymity in security literature.

Formally, we assume there exists a set of disjoint regions $\mathcal{X}_A = \mathcal{X}_1 \dot\cup \cdots \dot\cup \mathcal{X}_n$, and the robot aims to keep its destination within $\mathcal{X}_A$ indistinguishable to an intruder. Specifically, if the robot is predicted to enter one of these regions within a given time horizon, the intruder should remain uncertain about which specific region will be visited. We formalize this requirement as follows.

Definition 2 (*Anonymity*)**.** Given system Σ and STL formula ϕ, we say the system is *K-anonymous* w.r.t. $\mathcal{X}_A$ if $\mathcal{T}_\Sigma$ satisfies

$$\exists\pi_1.\exists\pi_2.\cdots\exists\pi_K.\mathbf{G}_{[0,T_\phi]}\left(\mathbf{F}_{[\Delta T,\Delta T]}(x^{\pi_1} \in \mathcal{X}_A) \rightarrow \left[(x^{\pi_1} = \cdots = x^{\pi_K}) \wedge \bigvee_{\{m_1,\ldots,m_K\}\in I_K} \bigwedge_{i=1,\ldots,K} \mathbf{F}_{[0,\Delta T]}(x^{\pi_i} \in \mathcal{X}_{m_i})\right]\right)$$

where $I_K \subseteq 2^{\{1,\ldots,n\}}$ is the set of all index sets with cardinality K.

As an illustrative example, let us still consider the nonholonomic mobile robot (17) with the same physical constraints $\mathcal{X}$ and $\mathcal{U}$. The STL task ϕ is modified to

$$\phi = \mathbf{F}_{[9,11]}(\boldsymbol{p} \in A_1) \wedge \mathbf{G}_{[22,23]}(\boldsymbol{p} \in A_2) \wedge \mathbf{F}_{[40,41]}(\boldsymbol{p} \in \mathcal{X}_A),$$

where $\mathcal{X}_A = A_3 \dot\cup A_4 \dot\cup A_5 \dot\cup A_6$ as shown in Fig. 2(b). Here the critical regions are the four possible destinations and we set $K = 3$ for anonymity. Using the system dynamics in (17) and the HyperSTL specification for anonymity, we compute control input $\mathbf{u}_1$.

The resulting trajectory is illustrated in Fig. 2(b). When the robot follows path π_1, it maintains three plausible destinations before reaching the actual target A_4: A_4 itself along with alternatives A_5 and A_6. This satisfies the 3-anonymity requirement by preserving ambiguity about the final destination.

4.2. Informed planning without ambiguity

While the previous case studies addressed security concerns, ∃-HyperSTL can also model collaborative scenarios under implicit communication. We still consider the robot planning setting. Rather than an intruder observing the trajectory, we now assume a UAV monitors the robot's state in real-time for cooperative purposes. Without direct communication, the robot must ensure its plan remains unambiguous to prevent misinterpretation by the UAV.

Specifically, we define a critical region $\mathcal{X}_C \subseteq \mathcal{X}$ that the robot must visit. The UAV must be able to predict each visit to $\mathcal{X}_C$ with sufficient advance notice (a certain number of steps) to prepare appropriate assistance. This requirement can be formally characterized as follows.

Definition 3 (*Distinguishability*)**.** Given system Σ and STL formula ϕ, we say the system is *ΔT-step distinguishable* w.r.t. $\mathcal{X}_C$ if $\mathcal{T}_\Sigma$ satisfies

$$\exists \pi_1 . \forall \pi_2 . \left[\phi^{\pi_1} \wedge \left(\phi^{\pi_2} \rightarrow \mathbf{G}_{[0,T_\phi]} \left(\mathbf{F}_{[\Delta T, \Delta T]} (x^{\pi_1} \in \mathcal{X}_C) \rightarrow \left[(x^{\pi_1} = x^{\pi_2}) \rightarrow \mathbf{F}_{[0,\Delta T]} (x^{\pi_2} \in \mathcal{X}_C) \right] \right) \right) \right]$$

Intuitively, this definition states that we need to find a trajectory π_1 satisfying the STL task ϕ such that for any other trajectory π_2 also satisfying ϕ, if π_2 reaches the critical region within ΔT steps, then π_1 must also reach the critical region within ΔT steps. Thus, there is no ambiguity about when the critical region must be visited.

As a case study, we still consider the nonholonomic mobile robot in (17) operating in the workspace shown in Fig. 2(c). The STL task ϕ remains the same as in (18), and we use the cost function defined in (19). Let the critical region be $\mathcal{X}_C = A_3$, and set the prediction horizon to $\Delta T = 2$. Using Algorithm 1, we synthesize an unambiguous control plan $\mathbf{u}_1$ based on the system dynamics and the HyperSTL formulation of distinguishability. The resulting trajectory $\pi_1 = \xi_f(x_0, \mathbf{u}_1)$ is depicted as the blue line in Fig. 2(c). Observe that all other plausible trajectories satisfying ϕ (shown as green lines) also reach the critical region within two steps. This ensures that the UAV can uniquely determine the robot's intended destination, enabling appropriate assistance actions to be taken.

4.3. Robust security-preserving planning

While the previous case studies focused on security under partial observations, we now consider a robust security-preserving planning setting with exogenous disturbances. To capture robustness and opacity simultaneously, we formulate the requirement as a higher-alternation HyperSTL.

We consider a disturbed linear system

$$z_{t+1} = Az_t + Bu_t + w_t \tag{20}$$

where $z_t \in \mathcal{X} \subseteq \mathbb{R}^2$ is the state and $w_t \in \mathcal{W} \subseteq \mathbb{R}^2$ is the disturbance generated from a compact set $\mathcal{W}$. Given an input sequence $\mathbf{u} = u_0 u_1 \cdots u_{N-1} \in \mathcal{U}^N$ and a disturbance sequence $\mathbf{w} = w_0 w_1 \cdots w_{N-1} \in \mathcal{W}^N$, we denote by $\xi_f(x_0, \mathbf{u}, \mathbf{w})$ the resulting system trajectory. The robot is assigned with an STL task

$$\phi = \mathbf{F}_{[9,11]} (z \in A_1) \wedge \mathbf{G}_{[22,23]} (z \in A_2) \wedge \mathbf{F}_{[40,41]} (z \in (A_3 \cup A_4 \cup A_5)) \tag{21}$$

where A_1, A_2, A_3 are three optional destinations. Our objective is to find a control input sequence $\mathbf{u} \in \mathcal{U}^N$ such that $\xi_f(x_0, \mathbf{u}, \mathbf{w}) \models \phi, \forall \mathbf{w} \in \mathcal{W}^N$.

For convenience, we first consider the nominal system with zero disturbance:

$$x_{t+1} = Ax_t + Bu_t, \ x_0 = z_0. \tag{22}$$

Let $\pi_1 := \xi_f(x_0, \mathbf{u}, 0) = x_0 x_1 \cdots x_N$ denote the corresponding nominal trajectory. Since the disturbance set $\mathcal{W}$ is known, the disturbed state satisfies

$$z_t \in x_t + \bigoplus_{k=0}^{t-1} A^{t-1-k} \mathcal{W} =: \mathcal{X}(t, x_t). \tag{23}$$

Hence, to ensure robust satisfaction of the STL task, it suffices to enforce the specification over the reachable tube $\mathcal{X}(t, x_t)$ at each time t.

On the other hand, we assume the intruder observes only the direction of the robot from a fixed optical camera located at x_c, without any distance measurement. Formally, the observation function is defined as the bearing direction

$$\mathcal{H}(x_t) = R(n_t) \frac{x_t - x_c}{\|x_t - x_c\|},$$

where

$$R(n_t) = \begin{bmatrix} \cos n_t & -\sin n_t \\ \sin n_t & \cos n_t \end{bmatrix}$$

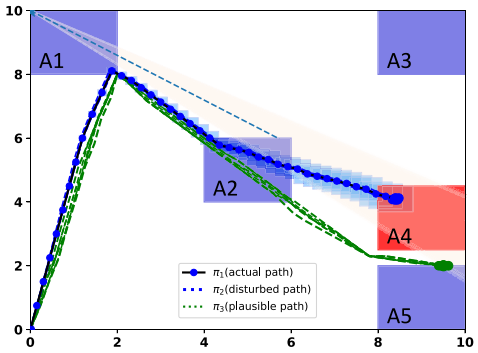


Fig. 3. Case study of disturbance-robust opacity.

Table 1
Runtime and iteration statistics of case studies of HyperSTL planning.

	Pre-opacity	Anonymity	Distinguishability	DR opacity
Iterations	1	1	6	11
Time	294.8	845.9	1985.3	1035.5

is the 2D rotation matrix and $n_t \in [-\bar{\theta}, \bar{\theta}]$ denotes the bounded measurement noise. Consequently, given a measurement y_t, the intruder's state estimate corresponds to the wedge-shaped set

$$\mathcal{H}^{-1}(y_t) = \{\, x_c + r\, R(\delta) y_t \in \mathcal{X} :\ r \geq 0,\ \delta \in [-\bar{\theta}, \bar{\theta}] \,\}.$$

From the robot's perspective, the planning objective should be satisfied robustly under disturbances while preventing an external observer from inferring whether the robot enters the secret region $\mathcal{X}_S$. This joint robustness-security requirement can be formally captured by the following notion of *disturbance-robust opacity*.

Definition 4 (*Disturbance-Robust Opacity*)**.** Given system Σ and STL formula ϕ, we say the system is *robustly opaque* w.r.t. $\mathcal{X}_S$ if $\mathcal{T}_\Sigma$ satisfies

$$\exists \pi_1 . \forall \pi_2 . \exists \pi_3 . \begin{bmatrix} \phi^{\pi_1} \wedge \left(\mathbf{G}_{[0,T_\phi]} [x^{\pi_2} \in \mathcal{X}(t, x^{\pi_1})] \rightarrow \phi^{\pi_2} \right) \wedge \\ \phi^{\pi_3} \wedge \left[\mathbf{F}_{[0,T_\phi]} (x^{\pi_1} \in \mathcal{X}_S) \rightarrow \mathbf{G}_{[0,T_\phi]} \begin{bmatrix} (x^{\pi_3} \in \mathcal{H}^{-1}(\mathcal{H}(x^{\pi_2}))) \wedge \\ (x^{\pi_1} \in \mathcal{X}_S \rightarrow x^{\pi_2} \notin \mathcal{X}_S) \end{bmatrix} \right] \end{bmatrix}$$

Intuitively, this definition requires synthesizing a nominal trajectory π_1 that satisfies the STL task ϕ. It further enforces robustness in the sense that each disturbed trajectory π_2 that remains within the prescribed disturbance tube $\mathcal{X}(t, x^{\pi_1})$ over $[0, T_\phi]$ must also satisfy ϕ. Moreover, to preserve security, for each such disturbed trajectory π_2, there must exist a plausible trajectory π_3 that satisfies ϕ and is observationally equivalent with π_2. In particular, whenever π_2 visits the secret set $\mathcal{X}_S$, the plausible trajectory π_3 is required to stay outside $\mathcal{X}_S$ at the same time, thereby preventing an external observer from inferring whether $\mathcal{X}_S$ has been visited.

As a case study, we consider a single-integrator mobile robot modeled by (20) with $A = I_2$ and $B = [1, 1]^\top$. Using Algorithm 2, we synthesize a secure control plan $\mathbf{u}_1$ based on the nominal dynamics (22) and the HyperSTL formulation of disturbance-robust opacity. The resulting nominal trajectory $\pi_1 = \xi_f(x_0, \mathbf{u}_1)$, together with ten randomly generated disturbed trajectories π_2, is shown in Fig. 3. We observe that all sampled disturbed trajectories π_2 visit the secret region $\mathcal{X}_S = A_4$. To preserve security, Fig. 3 further illustrates that, for each disturbed trajectory π_2, there exists a corresponding plausible trajectory π_3 that is observationally indistinguishable from π_2 and visits a non-secret region at the same time. Consequently, an intruder cannot determine with certainty whether the robot has entered the secret region, and the robot's secret is protected.

4.4. Scalability and computational performance

In this subsection, we report the computational performance of the proposed synthesis method in terms of (i) total synthesis time and (ii) the number of refinement iterations (i.e., the number of times the counterexample candidate sets are updated before termination).

Runtime of HyperSTL planning case studies. We first summarize the synthesis time for the four HyperSTL planning properties considered before, i.e., pre-opacity, anonymity, distinguishability, and disturbance-robust opacity (DR opacity). Note that the first three properties are evaluated on the nonholonomic models, while disturbance-robust opacity is evaluated on the linear system. Table 1 reports the runtime (in seconds) and iterations for each property.

Table 2
Statistics of scalability test.

T	$\exists\exists$		$\exists\forall$		$\exists\forall\exists$		$\exists\forall\exists\forall$	
	Iterations	Time	Iterations	Time	Iterations	Time	Iterations	Time
10	1	0.17	1	0.42	1	0.41	2	4.72
20	1	0.25	1	0.62	4	3.64	2	2.35
30	1	0.34	1	0.73	3	3.01	3	11.84
40	1	0.43	2	2.49	5	7.19	5	41.56
50	1	0.52	3	5.10	4	6.76	7	60.31

We observe that the synthesis time generally increases as the quantifier alternation becomes more involved, since deeper alternations introduce additional counterexample-guided checks. Moreover, the results show that both Algorithms 1 and 2 terminate after a finite number of iterations on all reported case studies, i.e., whenever the HyperSTL is feasible, a control plan can be found within a finite number of counterexample-guided updates. Finally, although disturbance-robust opacity has a higher alternation depth, it is solved faster than distinguishability because it is instantiated on a linear single-integrator system, whose underlying optimization problems are substantially easier than those for the nonlinear unicycle dynamics.

Scalability test. Since the size of the mixed-integer encoding grows with the specification horizon, it is important to evaluate how the length of HyperSTL formulae affects synthesis time. In order to assess this effect, we adopt a linear single-integrator model and the following HyperSTL templates to test Algorithm 2,

$$\Phi_0 = \exists\pi_1.\exists\pi_2.\mathbf{F}_{[0,T]}\mathsf{reach}_A(\pi_1) \wedge \mathbf{F}_{[0,T]}\mathsf{reach}_B(\pi_2) \wedge \mathbf{G}_{[0,T]}\mathsf{avoid}(\pi_1,\pi_2)$$
$$\Phi_1 = \exists\pi_1.\forall\pi_2.\mathbf{F}_{[0,T]}\mathsf{reach}_A(\pi_1) \wedge (\mathbf{F}_{[0,T]}\mathsf{reach}_B(\pi_2) \rightarrow \mathbf{G}_{[0,T]}\mathsf{avoid}(\pi_1,\pi_2))$$
$$\Phi_2 = \exists\pi_1.\forall\pi_2.\exists\pi_3.\mathbf{F}_{[0,T]}\mathsf{reach}_A(\pi_1) \wedge (\mathbf{F}_{[0,T]}\mathsf{reach}_B(\pi_2) \rightarrow \mathbf{G}_{[0,T]}\mathsf{avoid}(\pi_1,\pi_2)) \wedge \mathbf{F}_{[0,T]}\mathsf{meet}(\pi_2,\pi_3)$$
$$\Phi_3 = \exists\pi_1.\forall\pi_2.\exists\pi_3.\forall\pi_4.\mathbf{F}_{[0,T]}\mathsf{reach}_A(\pi_1) \wedge (\mathbf{F}_{[0,T]}\mathsf{reach}_B(\pi_2) \rightarrow \mathbf{G}_{[0,T]}\mathsf{avoid}(\pi_1,\pi_2)) \wedge \mathbf{F}_{[0,T]}\mathsf{meet}(\pi_2,\pi_3) \wedge (\mathbf{F}_{[0,T]}\mathsf{reach}_C(\pi_4) \rightarrow \mathbf{G}_{[0,T]}\mathsf{avoid}(\pi_3,\pi_4))$$

where $\mathsf{reach}_X(\pi) = \|x_t - x_X\| \leq r_X$, $X = A, B, C$, $\mathsf{avoid}(\pi_i,\pi_j) = \|x_t^i - x_t^j\| \geq d$, $\mathsf{meet}(\pi_i,\pi_j) = \|x_t^i - x_t^j\| \leq r_m$ and we set $x_A = [4,4]^\top$, $x_B = [1,4]^\top$, $x_C = [1,2]^\top$, $r_A = r_B = r_C = 0.3$, $d = 0.8$, $r_m = 1$. For each $\Phi_i, i = 0,1,2,3$, we vary the HyperSTL length T over $\{10,20,30,40,50\}$ and record the runtime and number of iterations. The statistics are summarized in Table 2.

We observe that, whenever the HyperSTL instance is feasible, the recursive counterexample checks terminate after a finite number of iterations. Overall, the results indicate that our approach scales well with increasing specification length and limited quantifier alternation depths.

5. Conclusions

In this paper, we investigate the task and motion planning for dynamic systems under signal temporal logic specifications. Unlike existing STL control synthesis approaches that impose temporal properties solely on individual trajectories, we leverage HyperSTL semantics to characterize inter-relationships between multiple system executions. Our approach integrates mixed-integer programming optimization and counterexample-guided synthesis techniques in a novel manner to solve the control synthesis problem. Through case studies involving information-flow synthesis for dynamic systems, we demonstrate the effectiveness of our approach. Note that this paper focuses on an offline planning setting, where the synthesis runtime is typically less stringent. An important direction for future work is to improve computational efficiency to support online reactive HyperSTL planning. Moreover, we also plan to extend our results to stochastic settings to achieve HyperSTL planning with probabilistic guarantees.

CRediT authorship contribution statement

Jianing Zhao: Formal analysis, Conceptualization. **Bowen Ye:** Writing – original draft, Formal analysis. **Xinyi Yu:** Formal analysis. **Rupak Majumdar:** Writing – review & editing. **Xiang Yin:** Supervision, Funding acquisition, Formal analysis.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Xiang Yin reports financial support was provided by National Natural Science Foundation of China. Xiang Yin reports a relationship with National Natural Science Foundation of China that includes: funding grants. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] S. Liu, A. Trivedi, X. Yin, M. Zamani, Secure-by-construction synthesis of cyber-physical systems, Annu. Rev. Control. 53 (2022) 30–50.
[2] O. Maler, D. Nickovic, Monitoring temporal properties of continuous signals, in: International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, 2004, pp. 152–166.
[3] P. Yu, Y. Gao, F.J. Jiang, K.H. Johansson, D.V. Dimarogonas, Online control synthesis for uncertain systems under signal temporal logic specifications, Int. J. Robot. Res. 43 (6) (2024) 765–790.
[4] B. Park, M.M. Olama, Mitigation of motor stalling and FIDVR via energy storage systems with signal temporal logic, IEEE Trans. Power Syst. 36 (2) (2020) 1164–1174.
[5] N. Arechiga, Specifying safety of autonomous vehicles in signal temporal logic, in: IEEE Intelligent Vehicles Symposium, 2019, pp. 58–63.
[6] V. Raman, A. Donzé, M. Maasoumy, R.M. Murray, A. Sangiovanni-Vincentelli, S.A. Seshia, Model predictive control with signal temporal logic specifications, in: IEEE Conference on Decision and Control, 2014, pp. 81–87.
[7] G.A. Cardona, D. Kamale, C.-I. Vasile, STL and wSTL control synthesis: A disjunction-centric mixed-integer linear programming approach, Nonlinear Anal. Hybrid Syst. 56 (2025) 101576.
[8] L. Lindemann, D.V. Dimarogonas, Control barrier functions for signal temporal logic tasks, IEEE Control. Syst. Lett. 3 (1) (2018) 96–101.
[9] N. Mehdipour, C.-I. Vasile, C. Belta, Arithmetic-geometric mean robustness for control from signal temporal logic specifications, in: American Control Conference, 2019, pp. 1690–1695.
[10] V. Raman, A. Donzé, D. Sadigh, R.M. Murray, S.A. Seshia, Reactive synthesis from signal temporal logic specifications, in: 18th International Conference on Hybrid Systems: Computation and Control, 2015, pp. 239–248.
[11] S.S. Farahani, R. Majumdar, V.S. Prabhu, S. Soudjani, Shrinking horizon model predictive control with signal temporal logic constraints under stochastic disturbances, IEEE Trans. Autom. Control 64 (8) (2018) 3324–3331.
[12] Y. Meng, C. Fan, Signal temporal logic neural predictive control, IEEE Robot. Autom. Lett. 8 (11) (2023) 7719–7726.
[13] D. Sun, J. Chen, S. Mitra, C. Fan, Multi-agent motion planning from signal temporal logic specifications, IEEE Robot. Autom. Lett. 7 (2) (2022) 3451–3458.
[14] S. Liu, A. Saoud, D.V. Dimarogonas, Controller synthesis of collaborative signal temporal logic tasks for multi-agent systems via assume-guarantee contracts, IEEE Trans. Autom. Control (2025).
[15] M.R. Clarkson, F.B. Schneider, Hyperproperties, J. Comput. Secur. 18 (6) (2010) 1157–1210.
[16] M. Anand, V. Murali, A. Trivedi, M. Zamani, Formal verification of hyperproperties for control systems, in: Proceedings of the Workshop on Computation-Aware Algorithmic Design for Cyber-Physical Systems, 2021, pp. 29–30.
[17] J. Zhao, S. Li, X. Yin, A unified framework for verification of observational properties for partially-observed discrete-event systems, IEEE Trans. Autom. Control 69 (7) (2024) 4710–4717.
[18] M. Anand, V. Murali, A. Trivedi, M. Zamani, Verification of hyperproperties for dynamical systems via barrier certificates, IEEE Trans. Autom. Control 69 (10) (2024) 6920–6934.
[19] E. Goubault, S. Putot, Inner and outer approximations of arbitrarily quantified reachability problems, Nonlinear Anal. Hybrid Syst. 59 (2026) 101629.
[20] Y. Wang, S. Nalluri, M. Pajic, Hyperproperties for robotics: Planning via hyperltl, in: IEEE International Conference on Robotics and Automation, 2020, pp. 8462–8468.
[21] E. Bonnah, L. Nguyen, K.A. Hoque, Motion planning using hyperproperties for time window temporal logic, IEEE Robot. Autom. Lett. 8 (8) (2023) 4386–4393.
[22] F. Pontiggia, F. Macák, R. Andriushchenko, M. Chiari, M. Češka, Decentralized planning using probabilistic hyperproperties, 2025, arXiv preprint arXiv:2502.13621.
[23] L.V. Nguyen, J. Kapinski, X. Jin, J.V. Deshmukh, T.T. Johnson, Hyperproperties of real-valued signals, in: International Conference on Formal Methods and Models for System Design, 2017, pp. 104–113.
[24] E. Bartocci, C. Mateis, E. Nesterini, D. Ničković, Mining hyperproperties using temporal logics, ACM Trans. Embed. Comput. Syst. 22 (5s) (2023) 1–26.
[25] A. Donzé, O. Maler, Robust satisfaction of temporal logic over real-valued signals, in: International Conference on Formal Modeling and Analysis of Timed Systems, 2010, pp. 92–106.
[26] C. Shi, A.N. Kulkarni, H. Rahmani, J. Fu, Synthesis of opacity-enforcing winning strategies against colluded opponent, in: IEEE Conference on Decision and Control, 2023, pp. 7240–7246.
[27] Y. Zheng, A. Lai, W. Lan, X. Yu, Optimal path planning with opacity-preserving temporal logic specifications using bipartite synthesizers, in: IEEE Conference on Decision and Control, 2023, pp. 7862–7867.
[28] Z. He, J. Yuan, N. Ran, X. Yin, Security-based path planning of multi-robot systems by partially observed petri nets and integer linear programming, IEEE Control. Syst. Lett. (2024).