Falsification of LTL Safety Properties in Hybrid Systems

Erion Plaku, Lydia E. Kavraki, and Moshe Y. Vardi

Dept. of Computer Science, Rice University, Houston TX 77005
{plakue, kavraki, vardi}@cs.rice.edu

Abstract. This paper develops a novel computational method for the falsification of safety properties specified by syntactically safe linear temporal logic (LTL) formulas ϕ for hybrid systems with general nonlinear dynamics and input controls. The method is based on an effective combination of robot motion planning and model checking. Experiments on a hybrid robotic system benchmark with nonlinear dynamics show significant speedup over related work. The experiments also indicate significant speedup when using minimized DFA instead of non-minimized NFA, as obtained by standard tools, for representing the violating prefixes of ϕ.

1 Introduction

Hybrid systems, which combine discrete and continuous dynamics, provide sophisticated mathematical models being used in robotics, automated highway systems, air-traffic management, computational biology, and other areas [1]. An important problem in hybrid systems is the verification of safety properties [1,2], which assert that nothing "bad" happens, e.g., "the car avoids obstacles." A hybrid system is safe if there are no witness trajectories indicating a safety violation. Safety properties have traditionally been specified in terms of a set of unsafe states and verification has been formulated as reachability analysis [1–7]. Reachability analysis in hybrid systems is in general undecidable [2,3]. Moreover, complete algorithms have an exponential dependency on the dimension of the state space and are limited in practicality to low-dimensional systems [1, 2, 4].

To handle more complex hybrid systems, alternative methods [8–12] have been proposed that shift from verification to falsification, which is often the focus of model checking in industrial applications [13]. Even though they are unable to determine that a system is safe, these methods may compute witness trajectories when the system is not safe. Witness trajectories, similar to error traces in model checking [13], indicate modeling flaws, which designers can then correct. The falsification methods in [8–10] adapt the Rapidly-exploring Random Tree (`RRT`) motion planner [14], which was originally developed for reachability analysis in continuous systems. We recently proposed the Hybrid Discrete Continuous Exploration (`HyDICE`) falsification method [11, 12], which also takes advantage of motion planning, but shows significant speedup over related work [9, 10].

As more complex hybrid systems are considered, limiting safety properties to a set of unsafe states, as in current methods [1–12], considerably restricts the

ability of designers to adequately express the desired safe behavior of the system. To allow for more sophisticated properties, researchers have advocated the use of linear temporal logic (LTL), which makes it possible to express safety properties with respect to time, such as "if the concentration level of gene A reaches x, then the concentration level of gene B will never reach y." LTL has been widely used in model checking of discrete systems in software and hardware [15], and timed systems [16]. The work in [17] generated trajectories that satisfy LTL constraints on the sequence of triangles visited by a point robot with Newtonian dynamics by using a controller that could drive the robot between adjacent triangles. The work in [18] used LTL to analyze gene networks. The work in [19] developed a method to verify LTL safety properties for robust discrete-time hybrid systems.

Traditional approaches for verification of an LTL property ϕ on a hybrid system $\mathcal{H}$ often cast the problem as reachability analysis via model checking. Abstractions are typically used to obtain a discrete transition model $\mathcal{M}$ that simulates $\mathcal{H}$, so that checking ϕ on $\mathcal{M}$ is sufficient to checking ϕ on $\mathcal{H}$ [5]. Moreover, with an exponential blow-up at most, a nondeterministic finite automaton (NFA) $\mathcal{A}$ can be constructed that describes all prefixes violating ϕ [20]. This allows for checking ϕ on $\mathcal{H}$ via model checking on $\mathcal{M} \times \mathcal{A}$. The challenge lies in the computation of $\mathcal{M}$, which is limited in practicality to low-dimensional hybrid systems due to the exponential dependency on the state-space dimension [1,2,4].

Applying alternative approaches [8–12] to falsify LTL safety properties by reachability analysis is also challenging due to intricacies of motion planning. During the search, motion planning extends a tree $\mathcal{T}$ in the state space of $\mathcal{H}$ by adding valid trajectories as new branches. Consider a vertex v and the trajectory ζ from the root of $\mathcal{T}$ to v. In reachability analysis [8–12], a witness trajectory is found when the state associated with v is unsafe. When considering LTL, such criteria is not sufficient, since ζ needs to satisfy $\neg\phi$. It then becomes necessary to maintain the propositional assignments satisfied by ζ and to effectively extend $\mathcal{T}$ so that more and more of the propositional assignments of $\neg\phi$ are satisfied.

To handle LTL, one can consider a naïve extension of the work in [8–12] by using $\mathcal{A}$ as an external monitor to determine when a tree trajectory ζ satisfies $\neg\phi$ by keeping track of the automaton states associated with each ζ. As shown in this work, however, such an approach is computationally very inefficient.

The main contribution of this work is to extend `HyDICE` [11, 12] in order to effectively incorporate LTL safety properties into hybrid-system falsification. The proposed approach, termed `TemporalHyDICE`, can be used to compute witness trajectories for the falsification of properties specified by syntactically safe LTL formulas for hybrid systems with

– external inputs, which could represent controls, uncertainties; and
– general nonlinear dynamics, where $\textsc{Flow}_q(x, u, t)$ is treated as a black box that outputs a state x_{new} obtained by following the hybrid-system dynamics when at state (q, x) and applying the input u for t time units.

When differential equations describe the dynamics, closed-form solutions (if available) or numerical integrations can be used for the black-box simulation. When differential equations become too cumbersome to describe the dynamics, other computer programs can be used for the simulation.

In its core, TemporalHyDICE draws from research in traditional and alternative approaches in hybrid systems to synergistically combine model checking and motion planning. This combination presents significant challenges, as it requires dealing with important issues, such as state-space search, memory usage, scalability, and passing of information between model checking and motion planning. In TemporalHyDICE, model checking guides motion planning by providing feasible directions along which to extend $\mathcal{T}$. A feasible direction consists of a sequence $[\tau_i]_{i=1}^n$ of propositional assignments that violates ϕ, which is computed by searching on-the-fly a discrete transition model $\mathcal{M}$ of $\mathcal{H}$ and the automaton $\mathcal{A}$ of $\neg\phi$. By not computing $\mathcal{M} \times \mathcal{A}$ explicitly, TemporalHyDICE considerably reduces the memory used by model checking. Moreover, unlike traditional approaches, TemporalHyDICE does not require $\mathcal{M}$ to simulate $\mathcal{H}$. In fact, $\mathcal{M}$ is based on a simple partition of the state space of $\mathcal{H}$ induced by propositions in ϕ. Motion planning extends $\mathcal{T}$ along directions $[\tau_i]_{i=1}^n$ provided by model checking so that more and more of $\tau_1, \ldots, \tau_n$ are satisfied in succession. As motion planning extends $\mathcal{T}$, it also gathers information to estimate the progress made in the search for a witness trajectory. This information is fed back to model checking to select in future iterations increasingly feasible directions for extending $\mathcal{T}$. This interactive combination of model checking and motion planning is a crucial component that allows TemporalHyDICE to effectively search for a witness trajectory.

An initial validation of TemporalHyDICE is provided by falsifying many properties specified by syntactically safe LTL formulas for a nonlinear hybrid robotic system. Experiments show significant speedup over related work. This work also studies the impact of representing $\neg\phi$ by DFAs or NFAs, as obtained by standard tools. The motivation comes from the work in [21], which shows significant speedup when using DFAs instead of NFAs in model checking. Experiments in this work in the context of falsification of LTL safety properties in hybrid systems also indicate significant speedup when using DFAs instead of NFAs.

The rest is as follows. Section 2 contains preliminaries. A straightforward approach of incorporating LTL into related work [8–12] by using the automaton $\mathcal{A}$ as an external monitor is described in Section 3. As demonstrated by the experiments, such an approach, however, is computationally very inefficient. The proposed approach, TemporalHyDICE, which effectively incorporates LTL into hybrid-system falsification, is described in Section 4. Experiments and results are described in Section 5. The paper concludes in Section 6 with a discussion.

2 Preliminaries

This section defines hybrid automata, LTL, the automata for the complement of LTL formulas, and the problem statement.

Hybrid Systems: Hybrid systems are modeled by hybrid automata [2]. A hybrid automaton is a tuple $\mathcal{H} = (S, I, \textsc{Inv}, E, \textsc{Guard}, \textsc{Jump}, U, \textsc{Flow})$, where $S = Q \times X$ is a product of a discrete and finite set Q and continuous spaces $X = \{X_q : q \in Q\}$; $I \subset S$ denotes initial states; $\textsc{Inv} = \{\textsc{Inv}_q : q \in Q\}$, where $\textsc{Inv}_q : X_q \to \{\top, \bot\}$ is the invariant function; $E \subseteq Q \times Q$ denotes discrete transitions;

$\textsc{Guard} = \{\textsc{Guard}_{q_i,q_j} : (q_i,q_j) \in E\}$ and $\textsc{Jump} = \{\textsc{Jump}_{q_i,q_j} : (q_i,q_j) \in E\}$, where $\textsc{Guard}_{q_i,q_j} : X_{q_i} \to \{\top,\bot\}$ and $\textsc{Jump}_{q_i,q_j} : X_{q_i} \to X_{q_j}$ denote guard and jump functions, respectively; $U = \{U_q : q \in Q\}$, where an input in $U_q \subseteq \mathbb{R}^{\dim(U_q)}$ can represent controls, nondeterminism, or uncertainties; and $\textsc{Flow} = \{\textsc{Flow}_q : q \in Q\}$, where $\textsc{Flow}_q : X_q \times U_q \times \mathbb{R}^{\geq 0} \to X_q$ is the flow function. This work treats the dynamics as a black box, where $\textsc{Flow}_q(x,u,t)$ outputs the state obtained by following the dynamics from x when u is applied for t time units. This allows for general nonlinear dynamics. In fact, the only requirement is the ability to simulate the dynamics. $\textsc{Inv}_q : X_q \to \{\top,\bot\}$, $\textsc{Guard}_{q_i,q_j} : X_{q_i} \to \{\top,\bot\}$, and $\textsc{Jump}_{q_i,q_j} : X_{q_i} \to X_{q_j}$ are also treated as black boxes to allow general specifications that do not limit designers to a particular approach, such as polyhedral or ellipsoidal constraints. A hybrid-system trajectory consists of continuous trajectories interleaved with discrete transitions.

Continuous Trajectory: $s = (q,x) \in S$, $T \geq 0$, $u \in U_q$ define a continuous trajectory $\Psi_{s,u,T} : [0,T] \to X_q$, where $\Psi_{s,u,T}(t) = \textsc{Flow}_q(x,u,t)$, $t \in [0,T]$.

Discrete Transition: For any $(q,x) \in S$, let $\chi(q,x) = (q', \textsc{Jump}_{q,q'}(x))$ if $\textsc{Guard}_{q,q'}(x) = \top$ for some $(q,q') \in E$. Otherwise, let $\chi(q,x) = (q,x)$.

Continuous Trajectory + Discrete Transition: $\Upsilon_{s,u,T} : [0,T] \to S$, defined as $\Upsilon_{s,u,T}(t) = (q, \Psi_{s,u,T}(t))$, $0 \leq t < T$ and $\Upsilon_{s,u,T}(T) = \chi(q, \Psi_{s,u,T}(T))$, ensures that a discrete transition at time T, if it occurs, is followed.

Trajectory Extension: Extending $\Phi : [0,T] \to S$ by applying $u' \in U$ to $\Phi(T)$ for $T' \geq 0$ time units, written as $\Phi \circ (u',T')$, is a trajectory $\Xi : [0, T+T'] \to S$ where $\Xi(t) = \Phi(t), t \in [0,T]$ and $\Xi(t) = \Upsilon_{\Phi(T),u',T'}(t-T), t \in (T, T+T']$.

Hybrid-System Trajectory: A state $s \in S$, a sequence $u_1, \ldots, u_k$ of inputs, and a sequence $T_1, \ldots, T_k$ of times define a trajectory $\zeta : [0,T] \to S$, where $T = T_1 + \cdots + T_k$ and $\zeta = \Upsilon_{s,u_1,T_1} \circ (u_2,T_2) \circ \cdots \circ (u_k,T_k)$.

In this work, a discrete transition is taken when a guard condition is satisfied. There is, however, no inherent limitation in dealing with non-urgent discrete transitions. In such cases, enabled discrete transitions could be taken nondeterministically or taken only when the invariant is invalid or a combination of both.

LTL: Let Π denote a set of propositional variables.

LTL Syntax and Semantics [20]: Every $\pi \in \Pi$ is a formula. If ϕ and ψ are formulas, then $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\mathcal{X}\phi$ (next), $\phi \mathcal{U} \psi$ (until), $\phi \mathcal{R} \psi$ (release), $\mathcal{F}\phi$ (future), and $\mathcal{G}\phi$ (globally) are also formulas. Let $\sigma = \tau_0, \tau_1, \ldots \in 2^{\Pi}$. Let $\sigma^i = \tau_i, \tau_{i+1}, \ldots$ We write $\sigma \models \phi$ to indicate that σ satisfies ϕ and define it as
$\sigma \models \top$; $\sigma \not\models \bot$; $\sigma \models \pi$ if $\pi \in \tau_0$; $\sigma \models \phi \wedge \psi$ if $\sigma \models \phi$ and $\sigma \models \psi$;
$\sigma \models \mathcal{X}\phi$ if $\sigma^1 \models \phi$; $\sigma \models \phi \mathcal{U} \psi$ if $\exists k \geq 0$ s.t. $\sigma^k \models \psi$ and $\forall\, 0 \leq i < k : \sigma^i \models \phi$;
$\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$; $\mathcal{F}\phi \equiv \top \mathcal{U} \phi$; $\mathcal{G}\phi \equiv \neg\mathcal{F}\neg\phi$; $\phi \mathcal{R} \psi \equiv \neg(\neg\phi \mathcal{U} \neg\psi)$.

Syntactically Safe LTL [22]*:* An LTL formula ϕ that, when written in positive normal form, uses only the temporals $\mathcal{X}$, $\mathcal{R}$, and $\mathcal{G}$ is syntactically safe. Every syntactically safe formula is a safety formula.

NFA for Syntactically Safe LTL [20]: With an exponential blow-up at most, an NFA can be constructed that describes all prefixes violating a syntactically safe LTL formula. The NFA is a tuple $\mathcal{A} = (Z, \Sigma, \delta, v_0, \text{Acc})$, where Z is a finite set of states; $\Sigma = 2^{\Pi}$ is the input alphabet; $\delta : Z \times \Sigma \to 2^Z$ is the transition

function; $z_0 \in Z$ is the initial state; and $\text{Acc} \subseteq Z$ is the set of accepting states. The set of states on which $[\tau_i]_{i=1}^n$, $\tau_i \in 2^{\Pi}$, ends up when run on $\mathcal{A}$ is defined as

$$\mathcal{A}([\tau_i]_{i=1}^n) = \begin{cases} \delta(z_0, \tau_1), & n=1 \\ \bigcup_{z \in \mathcal{A}([\tau_i]_{i=1}^{n-1})} \delta(z, \tau_n), & n>1. \end{cases} \mathcal{A} \text{ accepts } [\tau_i]_{i=1}^n \text{ iff } \mathcal{A}([\tau_i]_{i=1}^n) \cap \text{Acc} \neq \emptyset.$$

LTL over Hybrid-System Trajectories: Let $\Pi = \{\pi_{q,i} : q \in \mathcal{H}.Q \wedge 1 \leq i \leq n_q\}$, where n_q is the number of propositional variables associated with q. The truth-value of each $\pi_{q,i}$ is determined by a black-box function $\textsc{Prop}_{q,i} : \mathcal{H}.X_q \to \{\top, \bot\}$. The map $\tau : \mathcal{H}.S \to 2^{\Pi}$ maps $(q,x) \in \mathcal{H}.S$ to truth propositions: $\tau((q,x)) = \{\pi_{q,i} : \pi_{q,i} \in \Pi \text{ and } \textsc{Prop}_{q,i}(x) = \top\}$. When interpreted over a hybrid-system trajectory ζ, the notation $\tau(\zeta)$ denotes the sequence of propositional assignments $[\tau_i]_{i=1}^n$ ($\tau_i \in 2^{\Pi}$, $\tau_i \neq \tau_{i+1}$) in the order satisfied by ζ, i.e., $\tau_i = \tau(\zeta(T_i))$ where $0 \leq T_1 < \cdots < T_n \leq |\zeta|$ such that n is as large as possible and $\tau_i \neq \tau_{i+1}$, $1 \leq i < n$. Then, ζ satisfies ϕ, written $\zeta \models \phi$, iff $\tau(\zeta) \models \phi$.

Problem Statement: Let $\mathcal{P} = (\mathcal{H}, \mathcal{A}, \tau)$, where $\mathcal{H}$ is a hybrid automaton; $\mathcal{A}$ is an automaton for the complement of a syntactically safe LTL formula ϕ over propositions Π; and τ is a propositional map interpreted both over hybrid-system states and trajectories. Given $\mathcal{P}$, compute a valid trajectory $\zeta : [0,T] \to \mathcal{H}.S$ that satisfies $\neg\phi$, i.e., $(\forall t \in [0,T] : \textsc{Inv}_{q_t}(x_t) = \top$, where $(q_t, x_t) = \zeta(t))$ and $\zeta \models \neg\phi$.

3 Incorporating LTL into Motion-Planning Approaches

Motion planning has been widely used in reachability analysis for continuous robotic systems with dynamics [23,24]. These methods rely on a common framework that iteratively extends a tree in the state space of the system by adding valid trajectories as branches. Recently, the work in [8–10] adapted the tree-search framework for reachability analysis in hybrid systems.

There have been no discussions in the literature on how to augment the tree-search framework with LTL trajectory properties, cf. [8–10]. This section describes a minimal extension of the tree-search framework to handle LTL. The idea is to use $\mathcal{A}$ (DFA or NFA) to keep track of the automaton states associated with each tree trajectory and to determine when a tree trajectory is a witness. In this way, similar to model checking, the tree-search framework searches on-the-fly $\mathcal{H}$ and $\mathcal{A}$. With these modifications, the tree-search framework can be used to falsify LTL safety properties in hybrid systems, and, thus, provide a basis for the experimental comparisons. As demonstrated by the experiments, such an approach, however, is computationally very inefficient. Section 4, which describes `TemporalHyDICE`, then shows how to effectively combine the LTL tree-search framework with model checking on $\mathcal{M}$ and $\mathcal{A}$, where $\mathcal{M}$ is a discrete transition model of $\mathcal{H}$, in order to significantly increase its computational efficiency.

Incorporating LTL into the Tree-Search Framework: The tree is maintained as a graph $\mathcal{T} = (V, E)$. Each vertex $v \in \mathcal{T}.V$ is associated with a state $s \in \mathcal{H}.S$, written as $v.s$. An edge $(v', v'') \in \mathcal{T}.E$ indicates that a valid trajectory connects $v'.s$ to $v''.s$. As the search proceeds iteratively, $\mathcal{T}$ is extended by

Algorithm 3.1 LTL-TSF: Incorporating LTL into the Tree-Search Framework

Input: $\mathcal{P}$: problem specification; $t_{\max} \in \mathbb{R}^{>0}$: upper bound on computation time
Output: A solution trajectory if one is found or $\perp$ otherwise

(a) $\mathcal{T} \leftarrow$ InitalizeTree($\mathcal{P}$)
while ElapsedTime $< t_{\max}$ **do**
  (b) $v \leftarrow$ SelectVertexFromTree($\mathcal{P}, \mathcal{T}$) $\quad \diamondsuit$ *varies from method to method*
  (c) $[u, T, s_{\text{new}}, \alpha_{\text{new}}] \leftarrow$ ExtendTree($\mathcal{P}, \mathcal{T}, v$)
  (d) if $T > 0 \wedge |\alpha_{\text{new}}| > 0$ **then** $v_{\text{new}} \leftarrow$ AddBranchToTree($\mathcal{T}, v, [u, T, s_{\text{new}}, \alpha_{\text{new}}]$)
  (e) if $\mathcal{P}.\mathcal{A}.\text{Acc} \cap \alpha_{\text{new}} \neq \emptyset$ **then return** Traj($\mathcal{T}, v_{\text{new}}$)
return $\perp$

ExtendTree($\mathcal{P}, \mathcal{T}, v$) :=
1: $\epsilon \in \mathbb{R}^{>0} \leftarrow$ time step; $n_{\text{steps}} \in \mathbb{N} \leftarrow$ number of steps
2: $s = (q, x) \leftarrow v.s$; $\alpha \leftarrow v.\alpha$; $x_0 \leftarrow x$; $\alpha_0 \leftarrow \alpha$; $\tau_0 \leftarrow \mathcal{P}.\tau(s)$
3: $u \leftarrow$ sample control from $\mathcal{P}.\mathcal{H}.U_q$
4: **for** $i = 1, 2, \ldots, n_{\text{steps}}$ **do** $\diamondsuit$*simulate the continuous and discrete dynamics of* $\mathcal{P}.\mathcal{H}$
5: $\quad x_i \leftarrow \mathcal{P}.\mathcal{H}.\text{Flow}_q(x_{i-1}, u, \epsilon)$; $\tau_i \leftarrow \mathcal{P}.\tau((q, x_i))$
6: $\quad$ **if** $\tau_{i-1} = \tau_i$ **then** $\alpha_i \leftarrow \alpha_{i-1}$ **else** $\alpha_i \leftarrow \cup_{z \in \alpha_{i-1}} \mathcal{P}.\mathcal{A}.\delta(z, \tau_i)$
7: $\quad$ **if** $\mathcal{P}.\mathcal{H}.\text{Inv}_q(x_i) = \perp$ **then return** $[u, (i-1) * \epsilon, (q, x_{i-1}), \alpha_{i-1}]$
8: $\quad$ **if** $\mathcal{P}.\mathcal{H}.\text{Guard}_{q,q_{\text{new}}}(x_i) = \top$, $(q, q_{\text{new}}) \in \mathcal{P}.\mathcal{H}.E$ **then**
9: $\quad\quad (x_{\text{loc}}, T) \leftarrow$ localize discrete event in $((i-1) * \epsilon, i * \epsilon]$; $\tau_{\text{loc}} \leftarrow \mathcal{P}.\tau((q, x_{\text{loc}}))$
10: $\quad\quad$ **if** $\tau_{i-1} = \tau_{\text{loc}}$ **then** $\alpha_{\text{loc}} \leftarrow \alpha_{i-1}$ **else** $\alpha_{\text{loc}} \leftarrow \cup_{z \in \alpha_{i-1}} \mathcal{P}.\mathcal{A}.\delta(z, \tau_{\text{loc}})$
11: $\quad\quad x_{\text{new}} \leftarrow \mathcal{P}.\mathcal{H}.\text{Jump}_{q,q_{\text{new}}}(x_{\text{loc}})$; $\tau_{\text{new}} \leftarrow \mathcal{P}.\tau((q_{\text{new}}, x_{\text{new}}))$
12: $\quad\quad$ **if** $\tau_{\text{loc}} = \tau_{\text{new}}$ **then** $\alpha_{\text{new}} \leftarrow \alpha_{\text{loc}}$ **else** $\alpha_{\text{new}} \leftarrow \cup_{z \in \alpha_{\text{loc}}} \mathcal{P}.\mathcal{A}.\delta(z, \tau_{\text{new}})$
13: $\quad\quad$ **return** $[u, T, (q_{\text{new}}, x_{\text{new}}), \alpha_{\text{new}}]$
14: **return** $[u, n_{\text{steps}} * \epsilon, (q, x_{n_{\text{steps}}}), \alpha_{n_{\text{steps}}}]$

adding new vertices and edges. Consider the trajectory Traj($\mathcal{T}, v$) from the root of $\mathcal{T}$ to $v \in \mathcal{T}.V$. If Traj($\mathcal{T}, v$) $\models \neg\phi$, then Traj($\mathcal{T}, v$) is a witness. To determine Traj($\mathcal{T}, v$) $\models \neg\phi$, v is associated with the automaton states corresponding to Traj($\mathcal{T}, v$), written as $v.\alpha$ and defined as $v.\alpha = \mathcal{A}(\tau(\text{Traj}(\mathcal{T}, v)))$. Then, Traj($\mathcal{T}, v$) $\models \neg\phi$ iff $\mathcal{A}(v.\alpha) \cap \mathcal{A}.\text{Acc} \neq \emptyset$. Pseudocode is given in Algo. 3.1.

(a) InitializeTree($\mathcal{P}$) associates the root vertex v_{init} with the initial hybrid-system state and adds v_{init} to $\mathcal{T}$, i.e., $v_{\text{init}}.s = \mathcal{H}.s_{\text{init}}$, $\mathcal{T}.V = \{v_{\text{init}}\}$, and $\mathcal{T}.E = \emptyset$. The automaton states are computed by running $\mathcal{A}$ on the propositional assignment satisfied by $v_{\text{init}}.s$, i.e., $v_{\text{init}}.\alpha = \mathcal{A}.\delta(\mathcal{A}.z_{\text{init}}, \tau(v_{\text{init}}.s))$.

(b) SelectVertexFromTree($\mathcal{P}, \mathcal{T}$) selects a vertex $v \in \mathcal{T}.V$ from which to extend $\mathcal{T}$. Over the years, numerous strategies have been proposed that rely on distances, nearest neighbors, probability distributions, and much more [23, 24].

(c) ExtendTree($\mathcal{P}, \mathcal{T}, v$) extends $\mathcal{T}$ from v by computing a trajectory $\zeta : \mathbb{R}^{>0} \to \mathcal{H}.S$ that starts at $v.s$ and satisfies the invariant. A common strategy is to apply some input $u \in \mathcal{H}.U$ to $v.s$ and follow the dynamics of $\mathcal{H}$ until the invariant is not satisfied or a maximum number of steps is exceeded [8–12, 23, 24]. The input u is generally selected pseudo-uniformly at random to allow subsequent calls to extend $\mathcal{T}$ along new directions. ExtendTree($\mathcal{P}, \mathcal{T}, v$) returns a tuple $[u, T, s_{\text{new}}, \alpha_{\text{new}}]$, which defines $\zeta = \Upsilon_{v.s,u,T}$ (Section 2), where $s_{\text{new}} = \Upsilon_{v.s,u,T}(T)$ and $\alpha_{\text{new}} = \mathcal{A}(\tau(\text{Traj}(\mathcal{T}, v) \circ \zeta))$. Note that any hybrid-

system simulation method can be used to compute $\zeta = \Upsilon_{v.s,u,T}$. For completeness, we describe a simple iterative procedure. Let n_{steps} denote the number of steps and let $\epsilon > 0$ denote the step size (Algo. 3.1(c):1). Initially, $x_0 = x$ and $\alpha_0 = v.\alpha$, where $v.s = (q, x)$ (Algo. 3.1(c):2). At the i-th iteration, $x_i = \mathcal{H}.\textsc{Flow}_q(x_{i-1}, u, \epsilon)$ (Algo. 3.1(c):5). The automaton states α_i associated with (q, x_i) are updated only if $\tau((q, x_i)) \neq \tau((q, x_{i-1}))$. The update is computed by running $\mathcal{A}$ on $\tau((q, x_i))$ starting from α_{i-1} (Algo. 3.1(c):6). If $\mathcal{H}.\textsc{Inv}_q(x_i) = \bot$, then ExtendTree returns $[u, (i-1) * \epsilon, (q, x_{i-1}), \alpha_{i-1}]$ (Algo. 3.1(c):7). When $\mathcal{H}.\textsc{Inv}_q(x_i) = \top$, ExtendTree checks if a guard is satisfied, which would indicate a discrete event (Algo. 3.1(c):8). Event detection is followed by event localization, which localizes the earliest time $T \in ((i-1) * \epsilon, i * \epsilon]$ where the guard is satisfied (Algo. 3.1(c):9). Bisection or bracketing algorithms are typically used for event localization [25]. The discrete transition is then triggered to obtain the new state (Algo. 3.1(c):11). The automaton states are also updated (Algo. 3.1(c):12).

Numerical errors in simulation, invariant checking, event detection and localization could in certain cases cause ExtendTree to miss an invariant violation, miss a guard, or trigger a different discrete transition. To minimize such errors, a practical approach is to choose a small ϵ. This approach is the norm in hybrid-system falsification methods based on motion planning [8–12]. For hybrid systems with linear guards, it is also possible to use more accurate event detection and localization algorithms, which come asymptotically close to the guard boundary [25]. In many practical cases, hybrid systems exhibit a degree of robustness [19, 26] that minimizes the impact of numerical errors, e.g., small perturbations do not change the mode-switching behavior. As noted, the simple implementation of ExtendTree, presented here for completeness, can be replaced by more sophisticated hybrid-system simulation methods.

(d) AddBranchToTree$(\mathcal{T}, v, [u, T, s_{\text{new}}, \alpha_{\text{new}}])$ adds v_{new} and (v, v_{new}) to $\mathcal{T}$. It also associates s_{new} and α_{new} with v_{new} and u and T with (v, v_{new}).

(e) Traj$(\mathcal{T}, v_{\text{new}})$ computes the trajectory from $v_{\text{init}}.s$ to $v_{\text{new}}.s$ by concatenating the trajectories associated with the tree edges connecting v_{init} to v_{new}.

Incorporating LTL into `RRT`: The work in [8–10] relies on `RRT` [14]. To incorporate LTL into `RRT`, it suffices to use `LTL-TSF` (Algo. 3.1) and implement SelectVertexFromTree$(\mathcal{P}, \mathcal{T})$ as described in [8–10, 14], e.g., sample $s \in \mathcal{H}.S$ pseudo-uniformly at random and select $v \in \mathcal{T}.V$ whose $v.s$ is the closest to s according to a distance metric. This is referred to as `RRT`[`LTL-TSF`].

Incorporating LTL into `HyDICE`[`NoGuide`]: Similarly to `RRT`, `HyDICE` [11,12] also falls into the broad category of tree-search algorithms. Distinctly from `RRT`, `HyDICE` [11, 12] introduced discrete search over $(\mathcal{H}.Q, \mathcal{H}.E)$ to guide the tree search in the context of reachability analysis to a set of unsafe states. At each iteration, the discrete search computed a sequence of discrete transitions from an initial to an unsafe mode. The tree-search framework then extended $\mathcal{T}$ along the direction provided by the discrete search. Experiments showed significant speedup of one to two orders of magnitude over `RRT`-based falsification [9, 10].

Incorporating LTL into `HyDICE` is more involved than in the case of `RRT`, since the discrete search over $(\mathcal{H}.Q, \mathcal{H}.E)$ does not take LTL into account. When

considering LTL, a safety violation is not indicated by an unsafe state, but by an unsafe trajectory that satisfies $\neg\phi$. Therefore, when considering LTL, unsafe states and unsafe modes are not defined. This means that the discrete search over $(\mathcal{H}.Q, \mathcal{H}.E)$ from an initial to an unsafe mode is also not defined. The next section shows how to effectively incorporate LTL into `HyDICE`.

The version of `HyDICE` [11, 12] that does not use the discrete search is referred to in [11, 12] as `HyDICE`[`NoGuide`]. Experiments in [11, 12] showed that `HyDICE`[`NoGuide`] was significantly slower than `HyDICE`, but still faster than `RRT`-based falsification [9, 10]. As described in [11, 12], `HyDICE`[`NoGuide`] corresponds to the tree-search framework, where SelectVertexFromTree$(\mathcal{P}, \mathcal{T})$ is implemented by selecting $v \in \mathcal{T}.V$ according to a probability distribution over $\mathcal{T}.V$. This makes it possible to incorporate LTL into `HyDICE`[`NoGuide`], referred to as `HyDICE`[`NoGuide`, `LTL-TSF`], by using `LTL-TSF` (Algo 3.1).

4 `TemporalHyDICE`

The computational efficiency of `LTL-TSF` (Algo. 3.1) depends on the ability of the approach to quickly extend $\mathcal{T}$ along those directions that lead to the computation of witness trajectories. Motivated by [11, 12], `TemporalHyDICE` uses a discrete transition model $\mathcal{M}$ of $\mathcal{H}$ and effectively combines `LTL-TSF` with model checking over $\mathcal{M}$ and $\mathcal{A}$ to identify and extend $\mathcal{T}$ along such useful directions.

Consider a *discrete witness* $[\tau_i]_{i=1}^n$, i.e., a sequence of propositional assignments accepted by $\mathcal{A}$. Let $\Gamma(\tau_i) = \{s \in \mathcal{H}.S : \tau(s) = \tau_i\}$. If $\mathcal{T}$ can be extended so that a trajectory Traj$(\mathcal{T}, v)$ starts at $\Gamma(\tau_1)$ and enters $\Gamma(\tau_2), \ldots, \Gamma(\tau_n)$ in succession, then Traj$(\mathcal{T}, v)$ would be a witness trajectory. In this way, the discrete witness provides a *feasible* direction along which motion planning in `TemporalHyDICE` can attempt to extend $\mathcal{T}$ in the search for a witness trajectory.

Model checking can be effectively employed for the computation of discrete witnesses. A discrete transition model is constructed as a graph $\mathcal{M} = (V, E)$ in order to capture the partition of $\mathcal{H}.S$ induced by τ, where a vertex $v(\tau_i) \in \mathcal{M}.V$ corresponds to $\Gamma(\tau_i)$ and an edge $(v(\tau_i), v(\tau_j)) \in \mathcal{M}.E$ indicates that it may be possible to enter directly from $\Gamma(\tau_i)$ to $\Gamma(\tau_j)$. Model checking can then compute discrete witnesses by simultaneously searching $\mathcal{A}$ and $\mathcal{M}$.

An issue that arises is which discrete witnesses motion planning can actually follow. Since it is not known a priori which discrete witnesses are feasible, `TemporalHyDICE` maintains a running weight estimate $w([\tau_i]_{i=1}^n)$ on the feasibility of $[\tau_i]_{i=1}^n$. A high weight indicates significant progress is made in extending $\mathcal{T}$ toward $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$, while a low weight indicates little or no progress.

The core loop consists of using model checking to select at each iteration a discrete witness $[\tau_i]_{i=1}^n$ based on $w([\tau_i]_{i=1}^n)$ and then using motion planning to extend $\mathcal{T}$ toward $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ in succession.

Combining Model Checking and Motion Planning: A crucial property of `TemporalHyDICE`, distinctive from earlier work [17], is that model checking and motion planning work in tandem. Information gathered by motion planning (such as coverage, $\Gamma(\tau_i)$'s that have been reached, and time spent) is used to update

Algorithm 4.1 `TemporalHyDICE`

Input: $\mathcal{P}$: problem specification; $t_{\max} \in \mathbb{R}^{>0}$: upper bound on computation time
Output: A witness trajectory if one is found or $\perp$ otherwise

(a) $\mathcal{T} \leftarrow$ InitializeTree($\mathcal{P}$)
(b) $\mathcal{M} = (V, E) \leftarrow$ DiscreteTransitionModel($\mathcal{P}$)
(c) InitializeFeasibilityEstimate($\mathcal{P}, \mathcal{M}, w$)
while ElapsedTime $< t_{\max}$ **do**
  (d) $\sigma \stackrel{def}{=} [(z_i, \tau_i)]_{i=1}^{n} \leftarrow$ DiscreteWitness($\mathcal{P}, \mathcal{M}, w$)
  (e) $\zeta \leftarrow$ ExtendTreeAlongDiscreteWitness($\mathcal{P}, \mathcal{T}, \mathcal{M}, w, \sigma$)
  (f) if $\zeta \neq$ `NIL` **return** ζ
return $\perp$

(e) ExtendTreeAlongDiscreteWitness($\mathcal{P}, \mathcal{T}, \mathcal{M}, w, \sigma$) :=
1: $\sigma_{\text{avail}} \leftarrow \{(z_i, \tau_i) \in \sigma : (z_i, \tau_i).\text{vertices} \neq \emptyset\}$
2: **for** several times **do**
3: $(z_i, \tau_i) \leftarrow$ SelectAvailablePair(w, σ_{avail})
4: $v \leftarrow$ SelectVertexFromAvailablePair($w, (z_i, \tau_i).\text{vertices}$)
5: $[u, T, s_{\text{new}}, \alpha_{\text{new}}] \leftarrow$ ExtendTree($\mathcal{P}, \mathcal{T}, v$)
6: **if** $T > 0 \wedge |\alpha_{\text{new}}| > 0$ **then** $v_{\text{new}} \leftarrow$ AddBranchToTree($\mathcal{T}, v, [u, T, s_{\text{new}}, \alpha_{\text{new}}]$)
7: **if** $\mathcal{P}.\mathcal{A}.\text{Acc} \cap \alpha_{\text{new}} \neq \emptyset$ **then return** Traj($\mathcal{T}, v_{\text{new}}$)
8: UpdateFeasibilityEstimates($\mathcal{P}, \mathcal{T}, \mathcal{M}, w, (z_i, \tau_i)$)
9: $\tau_{\text{new}} \leftarrow \mathcal{P}.\tau(v_{\text{new}}.s)$
10: **for** $z_{\text{new}} \in \alpha_{\text{new}}$ **do**
11: $\sigma_{\text{avail}} \leftarrow \{(z_{\text{new}}, \tau_{\text{new}})\} \cup \sigma_{\text{avail}}$
12: $(z_{\text{new}}, \tau_{\text{new}}).\text{vertices} \leftarrow \{v_{\text{new}}\} \cup (z_{\text{new}}, \tau_{\text{new}}).\text{vertices}$
13: UpdateFeasibilityEstimates($\mathcal{P}, \mathcal{T}, \mathcal{M}, w, (z_{\text{new}}, \tau_{\text{new}})$)
14: **return** `NIL`

the feasibility estimates $w([\tau_i]_{i=1}^{n})$. As a result, a new discrete witness, associated with a high weight, could be selected in the next iteration by model checking. In turn, by using highly feasible discrete witnesses $[\tau_i]_{i=1}^{n}$ as guides, motion planning is able to make progress and extend $\mathcal{T}$ toward $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ until it successfully computes a witness trajectory. Pseudocode is given in Algo. 4.1.

Algo. 4.1(b) DiscreteTransitionModel($\mathcal{P}$): As discussed, $\mathcal{M}$ captures the partition of $\mathcal{H}.S$ induced by τ and serves to eliminate from consideration certain infeasible discrete witnesses. Region $\Gamma(\tau_j)$ is considered unable to directly reach $\Gamma(\tau_k)$, written $\Gamma(\tau_j) \not\rightarrow \Gamma(\tau_k)$, if $\Gamma(\tau_j)$ and $\Gamma(\tau_k)$ do not share a boundary and there is no discrete transition from some $s' \in \Gamma(\tau_j)$ to some $s'' \in \Gamma(\tau_k)$. A discrete witness $[\tau_i]_{i=1}^{n}$ is indeed infeasible if $\Gamma(\tau_k) \not\rightarrow \Gamma(\tau_{k+1})$ for some $1 \leq k < n$, since no trajectory can enter $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ in succession. To eliminate such infeasible discrete witnesses from consideration, $\mathcal{M}$ is constructed as a graph $\mathcal{M} = (V, E)$. A vertex $v(\tau_i)$ is added to $\mathcal{M}.V$ for each $\Gamma(\tau_i)$. An edge $(v(\tau_i), v(\tau_j))$ is added to $\mathcal{M}.E$ if it cannot be determined that $\Gamma(\tau_i) \not\rightarrow \Gamma(\tau_j)$.

Note that the computation of $\mathcal{M}$ is problem specific and depends on the black-box definitions of propositional, guards, and reset functions (Section 2). For this reason, DiscreteTransitionModel($\mathcal{P}$) is an external function supplied by the user. Since there is no requirement that $\mathcal{M}$ should simulate $\mathcal{H}$, it is

generally a straightforward process for the user to obtain $\mathcal{M}$ from $\mathcal{P}$. This is the case for the experiments in this work. Moreover, the definition of $\mathcal{M}$ allows for spurious edges, i.e., $(v(\tau_j), v(\tau_k)) \in \mathcal{M}.E$ even when $\Gamma(\tau_j) \not\rightarrow \Gamma(\tau_k)$. This further facilitates the computation of $\mathcal{M}$ since the user can add spurious edges when it is computationally difficult to determine that $\Gamma(\tau_j) \not\rightarrow \Gamma(\tau_k)$. A spurious edge may cause model checking to compute at some iterations infeasible discrete witnesses, since it is impossible to enter directly from $\Gamma(\tau_j)$ to $\Gamma(\tau_k)$. The interplay between model checking and motion planning will cause feasibility estimates associated with spurious edges to decrease rapidly, since motion planning will fail to extend $\mathcal{T}$ from $\Gamma(\tau_j)$ to $\Gamma(\tau_k)$. As a result, model checking will reduce the likelihood of including spurious edges in future computations of discrete witnesses.

Algo. 4.1(d) **DiscreteWitness**$(\mathcal{P}, \mathcal{M}, w)$ uses model checking to compute discrete witnesses by searching on-the-fly $\mathcal{A}$ and $\mathcal{M}$. The search produces a sequence $[(z_i, \tau_i)]_{i=1}^{n}$, where $(z_i, \tau_i) \in \mathcal{A}.Z \times 2^{\Pi}$ and $z_n \in \mathcal{A}.\text{Acc}$. A critical issue is which discrete witness to select from combinatorially many possibilities. To address this issue, `TemporalHyDICE` associates a running estimate $w(z_i, \tau_i)$ on the feasibility of including (z_i, τ_i) in the current discrete witness. Let $(z_i, \tau_i).\text{vertices} = \{v \in \mathcal{T}.V : z_i \in v.\alpha \wedge \tau_i = \tau(v.s)\}$, i.e., v is associated with (z_i, τ_i) iff $v.s$ satisfies τ_i and z_i is included in the automaton states $v.\alpha$ obtained by running $\tau(\textsc{Traj}(\mathcal{T}, v))$ on $\mathcal{A}$. Then,

$$w(z_i, \tau_i) = \text{cov}^{a_1}(z_i, \tau_i) * \text{vol}^{a_2}(\Gamma(\tau_i))/\text{time}(z_i, \tau_i), \tag{1}$$

where $\text{cov}(z_i, \tau_i)$ estimates the coverage of $\Gamma(\tau_i)$ by the states associated with $(z_i, \tau_i).\text{vertices}$; $\text{vol}(\Gamma(\tau_i))$ is the volume of $\Gamma(\tau_i)$; $\text{time}(z_i, \tau_i)$ is the time motion planning has spent extending $\mathcal{T}$ from $(z_i, \tau_i).\text{vertices}$; and a_1, a_2 are normalization constants. The combination of coverage, volume, and computational time is motivated by motion planners for continuous and hybrid systems [8–12, 27]. As in [11,12], $\text{cov}(z_i, \tau_i)$ is computed by imposing an implicit uniform grid on a low-dimensional projection of $\mathcal{H}.S$ and counting the number of grid cells that have at least one state from the states associated with $(z_i, \tau_i).\text{vertices}$. The volume $\text{vol}(\Gamma(\tau_i))$ is a user-supplied value, since it depends on the black-box definitions of the proposition functions $\textsc{Prop}_{q,i}$ (Section 2). In the experiments in this work, $\textsc{Prop}_{q,i}$ define polygons and $\text{vol}(\Gamma(\tau_i))$ is computed as the corresponding polygonal area. `TemporalHyDICE` associates a high weight $w(z_i, \tau_i)$ with (z_i, τ_i) if motion planning has extended $\mathcal{T}$ toward a region $\Gamma(\tau_i)$ with a large volume, and states associated with $(z_i, \tau_i).\text{vertices}$ quickly cover $\Gamma(\tau_i)$.

The discrete witness is computed as the shortest path from initial to accepting states by using Dijkstra's algorithm, where an edge $((z_i, \tau_i), (z_j, \tau_j))$ is assigned the weight $1/(w(z_i, \tau_i) * w(z_j, \tau_j))$. This allows to select highly feasible discrete witnesses. With small probability, the discrete witness is also computed as a random path using a variation of the depth-first-search, where the frontier nodes are visited in a random order. This randomness provides a way to correct for errors inherent with the weight estimates by ensuring that each discrete witness that is not determined as infeasible is selected with non-zero probability.

`TemporalHyDICE` does not explicitly construct $\mathcal{A} \times \mathcal{M}$. During the search for a discrete witness, the outgoing edges of (z_i, τ_i) are computed implicitly

as $\textsc{Edges}(z_i, \tau_i) = \{(z_j, \tau_j) : (v(\tau_i), v(\tau_j)) \in \mathcal{M}.E \wedge z_j \in \mathcal{A}.\delta(z_i, \tau_j)\}$. This allows `TemporalHyDICE` to considerably reduce the memory requirements of model checking. Note that the largest memory requirements in $\mathcal{A}$ are imposed by $\mathcal{A}.\delta$, which can be viewed as a ternary relation, subset of $\mathcal{A}.Z \times \Sigma \times \mathcal{A}.Z$, where $\Sigma = 2^{\Pi}$. On the other hand, $\mathcal{M}$ can be viewed as a binary relation, subset of $\Sigma \times \Sigma$. Explicitly constructing $\mathcal{A} \times \mathcal{M}$ would produce a 4-ary relation, subset of $\mathcal{A}.Z \times \Sigma^2 \times \mathcal{A}.Z$. For this reason, `TemporalHyDICE` does not compute $\mathcal{A} \times \mathcal{M}$ explicitly. In addition, the data structure that stores information about a pair (z_i, τ_i) is created only when a vertex v is added to $\mathcal{T}.V$ such that $z_i \in v.\alpha$ and $\tau_i = \tau(v.s)$. Reducing memory requirements is important for `TemporalHyDICE`, since it allows motion planning to extend $\mathcal{T}$ by adding more vertices and edges.

Algo. 4.1(e) ExtendTreeAlongDiscreteWitness$(\mathcal{P}, \mathcal{T}, \mathcal{M}, w, \sigma)$**:** Let $\sigma = [(z_i, \tau_i)]_{i=1}^{n}$ denote the current discrete witness. The objective is to extend $\mathcal{T}$ so that it reaches $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ in succession. To achieve this objective, the method proceeds by extending $\mathcal{T}$ from vertices associated with pairs (z_i, τ_i).

(*Algo. 4.1(e):1*) Only pairs $(z_i, \tau_i) \in \sigma$ reached by $\mathcal{T}$, i.e., $(z_i, \tau_i).\text{vertices} \neq \emptyset$, can be considered for selecting a vertex v from which to extend $\mathcal{T}$.

(*Algo. 4.1(e):3*) $\textsc{SelectAvailablePair}(w, \sigma_{\text{avail}})$ selects a pair (z_i, τ_i) from σ_{avail} with probability $w(z_i, \tau_i) / \sum_{(z_j, \tau_j) \in \sigma_{\text{avail}}} w(z_j, \tau_j)$, where $w(z_i, \tau_i)$ is defined in Eqn. 1. This selection, thus, favors highly feasible pairs.

(*Algo. 4.1(e):4*) $\textsc{SelectVertexFromAvailablePair}(w, (z_i, \tau_i).\text{vertices})$ selects a vertex v from $(z_i, \tau_i).\text{vertices}$ with probability $\frac{1}{\text{nsel}(v)} / \sum_{v' \in (z_i, \tau_i).\text{vertices}} \frac{1}{\text{nsel}(v')}$, where $\text{nsel}(v)$ is one plus the number of times v has been selected in the past from $(z_i, \tau_i).\text{vertices}$. This is based on well-established strategies in motion planning that favor those vertices selected less frequently in the past [23, 24].

(*Algo.4.1(e):5–7*) As described in Section 3, $\textsc{ExtendTree}(\mathcal{P}, \mathcal{T}, v)$ and $\textsc{AddBranchToTree}(\mathcal{P}, \mathcal{T}, v, [u, T, s_{\text{new}}, \alpha_{\text{new}}])$ extend $\mathcal{T}$ from v by computing and adding to $\mathcal{T}$ a valid trajectory that starts at $v.s$. If any of the automaton states α_{new} is an accepting state, then $\textsc{Traj}(\mathcal{T}, v_{\text{new}})$ is a witness trajectory.

(*Algo.4.1(e):8–13*) The feasibility estimate associated with (z_i, τ_i) is updated to reflect the extension of $\mathcal{T}$ from v. The vertex v_{new} is associated with each $(z_{\text{new}}, \tau_{\text{new}})$, where $z_{\text{new}} \in \alpha_{\text{new}}$ and $\tau_{\text{new}} = \tau(v_{\text{new}}.s)$. The feasibility estimate $w(z_{\text{new}}, \tau_{\text{new}})$ is also updated to reflect the addition of v_{new} to $(z_{\text{new}}, \tau_{\text{new}}).\text{vertices}$. Each $(z_{\text{new}}, \tau_{\text{new}})$ is also added to σ_{avail}, so that it becomes available for selection in the next iteration. The updated weights better estimate the feasibility of each discrete witness, and thus improve the selection of discrete witnesses for the next iteration. This in turn allows motion planning to make more progress in extending $\mathcal{T}$ toward $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ and eventually compute a witness trajectory.

5 Experiments and Results

The experiments provide an initial validation of `TemporalHyDICE` for the falsification of safety properties expressed by syntactically safe LTL formulas for hybrid systems with nonlinear dynamics. `TemporalHyDICE` is shown to be significantly more efficient than the straightforward extensions of related work [8–12],

which use the automaton $\mathcal{A}$ as an external monitor (see Section 3). The experiments also demonstrate the importance of model checking and the discrete transition model in the computational efficiency of `TemporalHyDICE`. This paper also studies the impact of $\mathcal{A}$ (NFA or DFA) on the efficiency of `TemporalHyDICE`.

The hybrid system $\mathcal{H}$ models an autonomous vehicle driving over different terrains, similar to the navigation benchmark proposed in [28] and used in [11, 12]. Each terrain corresponds to a mode $q \in \mathcal{H}.Q$. The dynamics, velocity, and acceleration vary from one terrain to another. Second-order dynamics (with 5 dimensions) for modeling cars, differential drives, and unicycles (see [11,23,24] for model details) are associated with each mode. In each terrain, several polygons are marked as propositions $\textsc{Prop}_{q_i,k}$ and guards $\textsc{Guard}_{q_i,q_j}$. A state $s = (q, x) \in \mathcal{H}.S$ satisfies $\textsc{Prop}_{q_i,k}$ (resp., $\textsc{Guard}_{q_i,q_j}$) iff $q = q_i$ and the position-component of x is inside $\textsc{Prop}_{q_i,k}$ (resp., $\textsc{Guard}_{q_i,q_j}$). When $\textsc{Guard}_{q_i,q_j}$ is satisfied, a discrete transition occurs. The mode is then set to q_j and velocity is set to zero.

The choice of this specific system is to provide a concrete benchmark that is easily scalable to test `TemporalHyDICE` as the complexity of LTL formulas is increased. For the experiments, 12 safety properties and 100 instances of the benchmark were created. Syntactically safe LTL formulas were manually designed in order to provide meaningful properties. Benchmark instances were generated at random in order to test `TemporalHyDICE` over many problems and obtain statistically significant results. Experimental data is publicly available.[1]

Problem Instances: In each problem instance, number of modes is $n_Q = 10$, number of propositions per mode is $n_P = 15$, and number of guards per mode is $n_G = 5$. A random problem instance is generated as follows. First, the second-order dynamics associated with each mode is selected pseudo-uniformly at random from those of a car, unicycle, or differential drive. Second, velocity is bounded by $v_{\max}$, where $v_{\max}$ is selected pseudo-uniformly at random from $[3, 6]$m/s. Third, for each mode, n_P propositions and n_G guards are generated as random polygons. Let $\pi_1, \ldots, \pi_{150}$ denote the generated propositions.

Syntactically-Safe LTL Formulas: Let $\beta_0 = \neg(\pi_1 \vee \cdots \vee \pi_{150})$.
– sequencing ($n = 3, 4, 5, 6$): Witness trajectory will reach $\pi_1, \ldots, \pi_n$ in order: $\phi_1^n = \neg\left(\beta_0 \mathcal{U}\left(\pi_1 \wedge (\pi_1 \mathcal{U}\left(\pi_2 \wedge (\pi_2 \mathcal{U}\left(\ldots \pi_{n-1} \wedge (\pi_{n-1} \mathcal{U}\left(\beta_0 \mathcal{U} \pi_n\right))\right)\right)\right)\right)\right)\right)$;
– counting ($n = 1, 2, 3, 4$): Witness trajectory will reach π_2, π_3, π_4 n-times in order, and then it will reach π_5: $\phi_2^n = \neg(\varsigma_1 \mathcal{U}(\pi_1 \wedge \Xi_1(\Xi_2 \cdots (\Xi_n(\varsigma_1 \mathcal{U} \pi_5)))))$, $\Xi_j(\psi) \stackrel{def}{=} \varsigma_1 \mathcal{U}\left(\pi_2 \wedge (\varsigma_2 \mathcal{U}\left(\pi_3 \wedge (\varsigma_3 \mathcal{U}\left(\pi_4 \wedge (\pi_4 \mathcal{U}\left(\varsigma_1 \wedge \psi\right))\right)\right)\right)\right)\right)$; $\varsigma_i \stackrel{def}{=} \beta_0 \vee \pi_i$;
– coverage ($n = 4, 5, 6, 7$): Witness trajectory reacheas each π_i: $\phi_3^n = \bigvee_{i=1}^{n} \mathcal{G}(\neg \pi_i)$.

Results: Experiments were run on Rice Cray XD1 ADA and PBC clusters. Each run uses a single processor (2.2Ghz, 8GB RAM), i.e., no parallelism. The automata for each $\neg\phi$ are computed by standard tools (scheck [29]). Comparisons of `TemporalHyDICE` to `RRT`[`LTL-TSF`] in Table 1(a) provide a basis for the results. While `TemporalHyDICE` solved all problem instances, `RRT`[`LTL-TSF`] timed out in almost every instance. `RRT`[`LTL-TSF`] relies on distance metrics and nearest neighbors to guide the search. By relying on such limited information, as shown in [11, 12] in the context of reachability analysis, it quickly becomes difficult to

[1] `http://www.kavrakilab.org/data/TACAS2009/`

(a) Comparison of different methods.

LTL safety formula	ϕ_1^3	ϕ_1^4	ϕ_1^5	ϕ_1^6	ϕ_2^1	ϕ_2^2	ϕ_2^3	ϕ_2^4	ϕ_3^4	ϕ_3^5	ϕ_3^6	ϕ_3^7
nr. states minimized DFA	10	21	46	105	23	76	164	287	16	32	64	128
`TemporalHyDICE`	18.6	25.5	27.2	40.4	22.2	40.4	63.3	88.3	14.6	40.9	127.9	293.2
`RRT[LTL-TSF]`	267.2	X	X	X	X	X	X	X	X	X	X	X
`HyDICE[NoGuide,LTL-TSF]`	245.3	X	X	X	X	X	X	X	X	X	X	X
`TemporalHyDICE`[no $\mathcal{M}$]	19.2	55.7	X	X	203.8	X	X	X	76.2	367.5	X	X

(b) Comparison of `TemporalHyDICE` when using a minimal DFA, a minimal NFA constructed by hand, or an NFA constructed by standard tools for ϕ_2^n, $n = 1, 2, 3, 4$.

	Minimized DFA				Minimized NFA				Standard NFA			
LTL safety formula	ϕ_2^1	ϕ_2^2	ϕ_2^3	ϕ_2^4	ϕ_2^1	ϕ_2^2	ϕ_2^3	ϕ_2^4	ϕ_2^1	ϕ_2^2	ϕ_2^3	ϕ_2^4
nr. states in automaton	23	76	164	287	7	11	15	19	27	176	912	4099
`TemporalHyDICE`	22.2	40.4	63.3	88.3	23.5	37.6	52.5	74.4	86.2	X	X	X

Table 1. Reported is the average time in seconds to solve 100 problem instances for each of the LTL formulas. Times for `TemporalHyDICE` include the construction of $\mathcal{M}$, which took $<$ 1s. Entries marked with X indicate a timeout (set to 400s).

find feasible directions to extend $\mathcal{T}$, causing a rapid decline in the growth of $\mathcal{T}$. The results in Table 1(a) confirm this observation also in the case of applying `RRT[LTL-TSF]` to falsify LTL safety properties in hybrid systems. By combining model checking and motion planning, `TemporalHyDICE` effectively guides the tree search. We also observe that the running time of `TemporalHyDICE` increases sub-linearly (ϕ_1^n and ϕ_2^n) or sub-quadratically (ϕ_3^n) with the number of states in the minimized DFA. These results provide promising initial validation.

Comparisons of `TemporalHyDICE` to `HyDICE[NoGuide,LTL-TSF]` in Table 1(a) demonstrate the importance of combining model checking and motion planning. Without model checking to guide motion planning, `HyDICE[NoGuide,LTL-TSF]`, similar to `RRT[LTL-TSF]`, times out in almost all instances.

Comparisons of `TemporalHyDICE` to `TemporalHyDICE`[no $\mathcal{M}$] in Table 1(a) indicate the importance of computing discrete witnesses by searching $\mathcal{M}$ and $\mathcal{A}$ (as in `TemporalHyDICE`) and not just $\mathcal{A}$ (as in `TemporalHyDICE`[no $\mathcal{M}$]). When searching just $\mathcal{A}$, a discrete witness may contain propositional assignments τ_i and τ_{i+1} that cannot be satisfied consecutively, i.e., $\Gamma(\tau_i) \not\rightarrow \Gamma(\tau_{i+1})$. As discussed in Section 4, $\mathcal{M}$ serves to eliminate from consideration many of these infeasible discrete witnesses. This in turn speeds up the search for a witness trajectory since $\mathcal{T}$ is extended far more frequently toward feasible directions. It is also important to note that, even though the discrete witnesses obtained by searching just $\mathcal{A}$ are not as beneficial as those obtained by searching $\mathcal{M}$ and $\mathcal{A}$, `TemporalHyDICE`[no $\mathcal{M}$] is still considerably faster than methods that do not guide the tree search, cf. `RRT[LTL-TSF]` and `HyDICE[NoGuide,LTL-TSF]`.

Table 1(b) compares `TemporalHyDICE` when using NFAs computed by standard tools (scheck [29]), minimal NFAs constructed by hand, or minimal DFAs. These experiments are motivated by the work in [21], which shows significant speedup when using DFAs instead of NFAs in model checking. As Table 1(b)

shows, `TemporalHyDICE` is only slightly faster when using minimal NFAs instead of minimal DFAs, even though minimal NFAs had significantly fewer states. As concluded in [21], DFA search has a significantly smaller branching factor than NFA search, which allows it to offset the drawbacks of a possibly exponential increase in the size of DFA. This observation is also supported by comparisons of minimal DFAs to standard NFAs, since in such cases there is significant speedup when using minimal DFAs. Therefore, the non-minimized NFA should be determinized and minimized.

6 Discussion

This work developed a novel method, `TemporalHyDICE`, for the falsification of safety properties specified by syntactically safe LTL formulas for hybrid systems with general nonlinear dynamics. By effectively combining model checking and motion planning, when a hybrid system is unsafe, `TemporalHyDICE` may compute a witness trajectory that indicates a violation of the safety property. Experiments show significant speedup over related work. As we consider more complex safety properties and high-dimensional continuous systems, it becomes important to further improve the synergistic combination of model checking and motion planning. Another direction is to extend the theory developed in [30] to show probabilistic completeness for `TemporalHyDICE`.

Acknowledgment

This work is supported by NSF CNS 0615328 (EP, LK, MV), a Sloan Fellowship (LK), and NSF CCF 0613889 (MV). Equipment is supported by NSF CNS 0454333 and NSF CNS 0421109 in partnership with Rice University, AMD, and Cray.

References

1. Tomlin, C.J., Mitchell, I., Bayen, A., Oishi, M.: Computational techniques for the verification and control of hybrid systems. Proc of IEEE **91**(7) (2003) 986–1001
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science **138**(1) (1995) 3–34
3. Henzinger, T., Kopke, P., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: ACM Symp on Theory of Computing. (1995) 373–382
4. Mitchell, I.M.: Comparing forward and backward reachability as tools for safety analysis. LNCS (2007) 4416:428–443
5. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.: Discrete abstractions of hybrid systems. Proc of IEEE **88**(7) (2000) 971–984
6. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and Counterexample-guided Refinement in Model Checking of Hybrid Systems. Intl J of Foundations of Computer Science **14**(4) (2003) 583–604
7. Giorgetti, N., Pappas, G.J., Bemporad, A.: Bounded model checking for hybrid dynamical systems. In: Conf on Decision & Control, Seville, Spain (2005) 672–677

8. Bhatia, A., Frazzoli, E.: Incremental search methods for reachability analysis of continuous and hybrid systems. LNCS (2004) 2993:142–156
9. Kim, J., Esposito, J.M., Kumar, V.: An RRT-based algorithm for testing and validating multi-robot controllers. In: Robotics: Science & Systems, Boston, MA (2005) 249–256
10. Nahhal, T., Dang, T.: Test coverage for continuous and hybrid systems. LNCS (2007) 4590:449–462
11. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: From verification to falsification. LNCS (2007) 4590:468–481
12. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: From verification to falsification by combining motion planning and discrete search. Formal Methods in System Design (2008)
13. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.: Benefits of bounded model checking at an industrial setting. LNCS (2001) 2102:436–453
14. LaValle, S.M., Kuffner, J.J.: Randomized kinodynamic planning. Intl J of Robotics Research **20**(5) (2001) 378–400
15. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
16. Behrmann, G., David, A., Larsen, K.G., Möller, O., Pettersson, P., Yi, W.: Uppaal present and future. In: Conf on Decision & Control, Orlando, FL (2001) 2881–2886
17. Fainekos, G.E., Kress-Gazit, H., Pappas, G.: Temporal logic motion planning for mobile robots. In: IEEE Intl Conf on Robotics & Automation, Barcelona, Spain (2005) 2020–2025
18. Batt, G., Belta, C., Weiss, R.: Temporal logic analysis of gene networks under parameter uncertainty. IEEE Trans of Automatic Control **53** (2008) 215–229
19. Damm, W., Pinto, G., Ratschan, S.: Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems. Intl J of Foundations of Computer Science **18**(1) (2007) 63–86
20. Kupferman, O., Vardi, M.: Model checking of safety properties. Formal methods in System Design **19**(3) (2001) 291–314
21. Armoni, R., Egorov, S., Fraer, R., Korchemny, D., Vardi, M.: Efficient LTL compilation for SAT-based model checking. In: Intl Conf on Computer-Aided Design, San Jose, CA (2005) 877–884
22. Sistla, A.: Safety, liveness and fairness in temporal logic. Formal Aspects of Computing **6** (1994) 495–511
23. Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press, Cambridge, MA (2005)
24. LaValle, S.M.: Planning Algorithms. Cambridge University Press, MA (2006)
25. Esposito, J., Kumar, V., Pappas, G.: Accurate event detection for simulation of hybrid systems. LNCS (2001) 204–217
26. Julius, A.A., Fainekos, G.E., Anand, M., Lee, I., Pappas, G.J.: Robust test generation and coverage for hybrid systems. LNCS (2007) 4416:329–342
27. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Discrete search leading continuous exploration for kinodynamic motion planning. In: Robotics: Science & Systems, Atlanta, GA (2007)
28. Fehnker, A., Ivancic, F.: Benchmarks for hybrid systems verification. LNCS (2004) 2993:326–341
29. Latvala, T.: Efficient model checking of safety properties. LNCS (2003) 2648:74–88
30. Ladd, A.M.: Motion Planning for Physical Simulation. PhD thesis, Rice University, Houston, TX (2006)