Int J Softw Tools Technol Transfer (2013) 15:305–320
DOI 10.1007/s10009-012-0233-2

TACAS 2009

Falsification of LTL safety properties in hybrid systems

Erion Plaku · Lydia E. Kavraki · Moshe Y. Vardi

Published online: 23 May 2012

Abstract This paper develops a novel approach for the falsification of safety properties given by a syntactically safe linear temporal logic (LTL) formula ϕ for hybrid systems with nonlinear dynamics and input controls. When the hybrid system is unsafe, the approach computes a trajectory that indicates violation of ϕ. The approach is based on an effective combination of model checking and motion planning. Model checking searches on-the-fly the automaton of $\neg\phi$ and an abstraction of the hybrid system for a sequence σ of propositional assignments that violates ϕ. Motion planning incrementally extends trajectories that satisfy more and more of the propositional assignments in σ. Model checking and motion planning regularly exchange information to find increasingly useful sequences σ for extending the current trajectories. Experiments that test LTL safety properties on a robot navigation benchmark modeled as a hybrid system with nonlinear dynamics and input controls demonstrate the computational efficiency of the approach. Experiments also indicate significant speedup when using minimized DFA instead of non-minimized NFA for representing $\neg\phi$.

Keywords Hybrid systems · Linear-temporal logic · Safety properties · Motion planning · Model checking

A preliminary version of this work appeared in the Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), LNCS vol. 5505, pp. 368–382.

E. Plaku · L. E. Kavraki (✉) · M. Y. Vardi
Department of Computer Science,
Rice University, Houston, TX 77005, USA
e-mail: kavraki@rice.edu

Present Address:
E. Plaku
Department of Electrical Engineering and Computer Science,
Catholic University of America, Washington, DC 20064, USA

1 Introduction

Hybrid systems, which combine discrete logic and continuous dynamics, provide sophisticated mathematical models used in robotics, 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 been generally specified as a set of unsafe states and verification has been formulated as reachability analysis [1–8]. Many verification methods perform reachability analysis by approximating reachable states [1,4,5] or use abstractions to obtain finite-state models [6–8]. Reachability analysis in hybrid systems is generally undecidable [2,3,9]. Moreover, verification methods have an exponential dependency on the dimension of the state space and are limited in practicality to low-dimensional systems [1,2,5].

To handle more complex hybrid systems, alternative methods [10–16] have been proposed that shift from verification to falsification, which is often the focus of model checking in industrial applications [17]. Even though these falsification methods [10–16] cannot determine that a hybrid system is safe, they can compute witness trajectories when the hybrid system is unsafe (see also [18–20] for work that focuses on resolution completeness). Witness trajectories, similar to error traces in model checking [17], indicate flaws, which can then be corrected. The falsification methods in [10–14] adapt the Rapidly exploring Random Tree (RRT) motion planner [21], which was originally developed for continuous systems. We recently proposed the Hybrid Discrete Continuous Exploration (HyDICE) falsification method [15,16], which also takes advantage of motion planning, but shows significant speedup over RRT-based falsification [12–14].

As more complex hybrid systems are considered, limiting safety properties to a set of unsafe states [1–8, 11–14, 19, 20], as it was also the case in our previous work [15, 16], makes it difficult 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 "once the concentration level of gene A reaches x, then the concentration level of gene B will never reach y" or "the robotic car, after inspecting a contaminated area A should immediately go to the decontamination station B and then eventually go to one of the base stations C or D." Using the temporal connectives $\mathcal{G}$ (globally), $\mathcal{U}$ (until), $\mathcal{X}$ (next), $\mathcal{F}$ (eventually), the safety property in the gene regulatory example is $\mathcal{G}(\neg\pi_A \vee \mathcal{G}(\neg\pi_B))$, where proposition π_A (π_B) is true iff the concentration level of gene A (resp., B) is at least x (resp., y). For the robotic car, the safety property is $\mathcal{G}(\pi_A \rightarrow (\mathcal{X}(\pi_B) \wedge \mathcal{F}(\pi_C \vee \pi_D)))$, where π_i, $i = A, B, C, D$, is true iff the car enters i.

Linear temporal logic has been widely used in model checking of discrete systems in software and hardware [22], and timed systems [23]. LTL has more recently been used in robotics applications. The work in [24, 25] generated trajectories that satisfy LTL constraints on the sequence of triangles visited by a point robot with Newtonian dynamics using a controller that could drive the robot between adjacent triangles. The work in [26] developed an approach for designing controllers for linear systems that satisfy LTL specifications. Other work automatically synthesizes controllers from LTL specifications of tasks that the robot needs to accomplish [27, 28]. LTL in conjunction with hierarchical abstractions has also been used to control robotic swarms [29] and in computational biology to analyze gene networks [30]. The work in [31] developed a method to verify LTL safety properties for robust discrete-time hybrid systems.

Traditional approaches for the verification of LTL safety properties on hybrid systems often cast the problem as reachability analysis via model checking. The idea is to construct an abstraction $\mathcal{M}$ of the hybrid system $\mathcal{H}$ so that checking safety properties expressed by an LTL formula ϕ on $\mathcal{H}$ can be accomplished by checking ϕ on $\mathcal{M}$ [6]. Moreover, since safety properties have finite counterexamples, an NFA $\mathcal{A}$ can be constructed to represent $\neg\phi$, where the size of $\mathcal{A}$ is at most exponential compared to the size of ϕ [32]. This allows for checking ϕ on $\mathcal{H}$ via model checking on $\mathcal{M} \times \mathcal{A}$. The computation of a suitable abstraction $\mathcal{M}$ is a non-trivial issue [6].

Alternative approaches based on motion planning [10–16] focus on hybrid-system falsification, but are limited to safety properties given as a set of unsafe states. Applying these alternative approaches to falsify LTL safety properties is challenging due to intricacies of motion planning. During the search, motion planning extends a tree $\mathcal{T}$ in the state space of the hybrid system $\mathcal{H}$ by adding valid trajectories as new branches. Consider the trajectory ζ from the root of $\mathcal{T}$ to a vertex v. In reachability analysis [10–16], a witness trajectory is found when the state associated with v is unsafe. This is not sufficient when considering a safety property given by an LTL formula ϕ, 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 safety properties, one can consider a straightforward extension of the work in [10–16] to use $\mathcal{A}$ as an external monitor to keep track of the automaton states associated with each tree trajectory and to determine from this information when a tree trajectory satisfies $\neg\phi$. As shown in this work, however, such an approach is computationally inefficient.

The main contribution of this paper is to effectively incorporate LTL safety properties into hybrid-system falsification by combining model checking with motion planning. This extends our previous work [15, 16] which limited safety properties to a set of unsafe states. The proposed approach, Temporal Hybrid Discrete and Continuous Exploration (`TemporalHyDICE`), can be used to compute witness trajectories that indicate violation of safety properties specified via syntactically safe LTL formulas. The reason for using syntactically safe LTL is that determining whether an LTL formula is safe is PSPACE-complete [33]. The work in [33] also provides sufficient syntactic requirements for safe LTL formulas. In particular, it shows that an LTL formula that uses only the temporal connectives $\mathcal{X}$ (next), $\mathcal{R}$ (release) and $\mathcal{G}$ (globally) when put in positive-normal form (by pushing negations all the way to leaves) is safe. Such formulas, when negated, can be translated to NFAs.

`TemporalHyDICE` allows for hybrid systems with nonlinear continuous dynamics and input controls. In fact, `TemporalHyDICE` only requires the availability of a simulator, which, when given a state s, an input control u, and a time step Δt, computes the new state that results from following the dynamics and the discrete transitions of $\mathcal{H}$. Furthermore, `TemporalHyDICE` does not require explicit representations of invariants, guards, and propositions. These are treated as black-box functions used to answer membership queries. However, `TemporalHyDICE` needs an input abstraction $\mathcal{M}$ of the hybrid system $\mathcal{H}$.

In its core, `TemporalHyDICE` draws from research in traditional and alternative approaches in hybrid systems to combine model checking and motion planning. This presents significant challenges, as it requires dealing with 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 discrete witnesses along which to extend $\mathcal{T}$. A discrete witness is a sequence $[\tau_i]_{i=0}^{n-1}$ of propositional assignments that violates ϕ, which is computed by searching on-the-fly the abstraction $\mathcal{M}$ and the automaton $\mathcal{A}$. Motion planning extends $\mathcal{T}$ as guided by the discrete witness $[\tau_i]_{i=0}^{n-1}$ so that more and more of $\tau_0, \ldots, \tau_{n-1}$ 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 alternative discrete witnesses that could expand the search along new directions. This interplay between model checking and motion planning is a crucial component that allows `TemporalHyDICE` to effectively search for a witness trajectory.

`TemporalHyDICE` is tested on a hybrid system which models a vehicle driving over different terrains, similar to the navigation benchmark proposed in [34] and used in [15,16]. The vehicle dynamics vary from one terrain to another and are selected from second-order models of cars, differential drives, and unicycles. Each terrain is subdivided into a uniform grid and some grid cells are labeled as guards, jumps, or propositions. Guards provide conditions that can trigger discrete transitions that enable the vehicle to move from one terrain to another. Syntactically safe LTL formulas are provided over the propositions. Experiments demonstrate the computational efficiency of `TemporalHyDICE` and show the importance of combining model checking and motion planning. Experiments also indicate significant speedup when using minimized DFA instead of non-minimized NFA for representing $\neg\phi$. This agrees with the work in [35], which shows significant speedup when using DFAs instead of NFAs in model checking.

The paper is organized as follows. Section 2 provides the mathematical framework. Section 3 describes a straightforward approach of incorporating LTL into related work [10–16] using $\mathcal{A}$ as an external monitor. Section 4 describes `TemporalHyDICE`, which effectively incorporates LTL. Section 5 describes the experiments. A discussion concludes the paper in Sect. 6.

2 Mathematical framework

This section defines hybrid automata, LTL syntax and semantics, the automaton for $\neg\phi$, LTL over hybrid-system trajectories, and the problem statement.

2.1 Hybrid systems

Hybrid systems are modeled by hybrid automata [2]. This paper extends the definition of hybrid automata to allow for hybrid systems with nonlinear dynamics and input controls, and for invariants, guards, and jumps to be specified as black-box functions.

Definition 1 (*Hybrid automaton*) A hybrid automaton is a tuple

$\mathcal{H} = (S, \text{INVARIANT}, E, \text{GUARD}, \text{JUMP}, U, f)$, where

- $S = Q \times X$ is the Cartesian product of the discrete and continuous state spaces;
- Q is a finite set;
- X maps $q \in Q$ to X_q, where X_q is the continuous state space associated with q;
- INVARIANT maps $q \in Q$ to INVARIANT_q, where

  $$\text{INVARIANT}_q : X_q \rightarrow \{\texttt{true}, \texttt{false}\}$$

  represents constraints that any $x \in X_q$ should satisfy;
- $E \subseteq Q \times Q$ is the set of discrete transitions;
- GUARD maps $(q_i, q_j) \in E$ to $\text{GUARD}_{(q_i,q_j)}$, where

  $$\text{GUARD}_{(q_i,q_j)} : X_{q_i} \rightarrow \{\texttt{true}, \texttt{false}\}$$

  is the guard function associated with (q_i, q_j);
- JUMP maps $(q_i, q_j) \in E$ to $\text{JUMP}_{(q_i,q_j)}$, where

  $$\text{JUMP}_{(q_i,q_j)} : X_{q_i} \rightarrow X_{q_j}$$

  is the jump function associated with (q_i, q_j);
- U maps $q \in Q$ to U_q, where $U_q \subseteq \mathbb{R}^{\dim(U_q)}$ is the set of input controls associated with q;
- f maps $q \in Q$ to f_q, where

  $$f_q : X_q \times U_q \rightarrow \dot{X}_q$$

  is a set of differential equations that describes the continuous dynamics associated with q.

The state of the hybrid automaton is a tuple $(q, x) \in S$, which describes both the discrete and the continuous components. While in mode q, when applying an input control function $\tilde{u} : [0, T] \rightarrow U_q$ for $T \geq 0$ time units, the continuous state changes according to the dynamics expressed by f_q, as shown in the following definition:

Definition 2 (*Continuous trajectory*) A state $s = (q, x) \in S$, a time duration $T \geq 0$, and an input control function $\tilde{u} : [0, T] \rightarrow U_q$ define a continuous trajectory $\chi_{\langle s,\tilde{u},T\rangle} : [0, T] \rightarrow X_q$, such that for all $t \in [0, T]$:

$$\chi_{\langle s,\tilde{u},T\rangle}(t) = x + \int_{h=0}^{t} f_q(\chi_{\langle s,\tilde{u},T\rangle}(h), \tilde{u}(h))\,\mathrm{d}h$$

In this paper, each X_q includes derivatives of different orders, e.g., velocity and acceleration, and, hence, f_q is nonlinear. Numerical integration is used to integrate f_q since closed-form solutions are generally not available.

A discrete transition occurs at $(q, x) \in S$ when a guard is satisfied, i.e., $\text{GUARD}_{(q,q_j)}(x) = \texttt{true}$ for some $(q, q_j) \in E$. In this paper, the discrete transitions are urgent. As a result, the state of the hybrid system is instantaneously changed to (q_j, x_j), where $x_j = \text{JUMP}_{(q,q_j)}(x)$. There is, however, no inherent limitation in dealing with non-urgent discrete transitions. In such cases, the discrete transition can be applied only if the invariant is invalid or when some other user-defined condition is satisfied. The augmentation of a continuous trajectory with a discrete transition is defined as follows.

Definition 3 (*Continuous trajectory + Discrete transition*) Given a continuous trajectory $\chi_{\langle s,\tilde{u},T\rangle} : [0, T] \to X_q$, the trajectory $\Upsilon_{\langle s,\tilde{u},T\rangle} : [0, T] \to S$ follows a discrete transition at time T if it occurs, i.e.,

$$\Upsilon_{\langle s,\tilde{u},T\rangle}(t) = \begin{cases} (q, \chi_{\langle s,\tilde{u},T\rangle}(t)), & t \in [0, T) \text{ and no guard is satisfied,} \\ (q, x_T), & t = T \text{ and no guard is satisfied,} \\ (q_j, \text{JUMP}_{(q,q_j)}(x_T)), & t = T \text{ and } \text{GUARD}_{(q,q_j)}(x_T) = \\ & \quad \texttt{true} \\ & \text{for some } (q, q_j) \in E, \end{cases}$$

where $x_T = \chi_{\langle s,\tilde{u},T\rangle}(T)$.

In Definition 3, when a state satisfies more than one guard, a user-defined procedure can be used to determine which jump to apply.

A trajectory Ψ can be extended by applying an input control function $\tilde{u}$ to the last state of Ψ. More precisely, trajectory extension is defined as follows.

Definition 4 (*Trajectory Extension*) The extension of a trajectory $\Psi : [0, T] \to S$ by applying to $\Psi(T)$ the input control function $\tilde{u} : [0, T'] \to X_{q'}$ is written as

$$\Psi \circ (\tilde{u}, T')$$

and is another trajectory $\Upsilon : [0, T + T'] \to S$ defined as

$$\Upsilon(t) = \begin{cases} \Psi(t), & t \in [0, T] \\ \Upsilon_{\langle \Psi(T),\tilde{u},T'\rangle}(t - T), & t \in (T, T + T']. \end{cases}$$

A hybrid-system trajectory consists of continuous trajectories interleaved with discrete transitions, as indicated by the following definition.

Definition 5 (*Hybrid-system trajectory*) A state $s \in S$ and a sequence of input control functions $\tilde{u}_1 : [0, T_1] \to U_{q_1}, \ldots, \tilde{u}_n : [0, T_n] \to U_{q_n}$ define a hybrid-system trajectory $\zeta : [0, T_1 + \cdots + T_n] \to S$, where

$$\zeta = \Upsilon_{\langle s,\tilde{u}_1,T_1\rangle} \circ (\tilde{u}_2, T_2) \circ \cdots \circ (\tilde{u}_n, T_n).$$

(Valid Hybrid-System Trajectory) ζ is valid iff it satisfies the invariant, i.e., $\forall t \in [0, T_1 + \cdots + T_n]$:

$\textbf{INVARIANT}_{q_t}(x_t) = \texttt{true}$, where $(q_t, x_t) = \zeta(t)$.

2.2 Syntax, semantics, and syntactically safe LTL

Let Π denote a set of propositions. LTL formulas combine propositions with Boolean connectives $\neg, \wedge, \vee$ and temporal connectives $\mathcal{X}$ (next), $\mathcal{U}$ (until), $\mathcal{R}$ (release), $\mathcal{F}$ (future), $\mathcal{G}$ (globally), as defined below.

Definition 6 (*LTL Syntax and semantics* [32]) Every $\pi \in \Pi$ is a formula. If ϕ and ψ are formulas, then

$$\neg\phi,\ \phi \wedge \psi,\ \mathcal{X}\phi,\ \phi\mathcal{U}\psi$$

are also formulas. Let $\sigma = \tau_0, \tau_1, \ldots$ denote an infinite sequence, where each $\tau_i \in 2^{\Pi}$. Let $\sigma^i = \tau_i, \tau_{i+1}, \ldots$ denote the ith postfix of σ. The notation $\sigma \models \phi$ indicates that σ satisfies ϕ and is defined as
$\sigma \models \pi$ if $\pi \in \Pi$ and $\pi \in \tau_0$, $\sigma \models \neg\phi$ if $\sigma \not\models \phi$,
$\sigma \models \phi \wedge \psi$ 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 $\exists\, k \geq 0$ such that
$\sigma^k \models \psi$ and $\forall\, 0 \leq i < k : \sigma^i \models \phi$.
Moreover, we use the abbreviations $\texttt{false} \equiv \pi \wedge \neg\pi$, $\texttt{true} \equiv \neg\texttt{false}$, $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$, $\mathcal{F}\phi \equiv \texttt{true}\,\mathcal{U}\phi$, $\mathcal{G}\phi \equiv \neg\mathcal{F}\neg\phi$, $\phi\mathcal{R}\psi \equiv \neg(\neg\phi\mathcal{U}\neg\psi)$.

Consider a language $L \subseteq \Sigma^\omega$ of infinite words over the alphabet Σ. As defined in [35,36], a finite word $\alpha \in \Sigma^*$ is a violating prefix for L if for all $\beta \in \Sigma^\omega$ it holds that $\alpha \cdot \beta \notin L$, where $\alpha \cdot \beta$ denotes the concatenation of α and β. A language L is a safety language iff every $w \notin L$ has a finite violating prefix [36]. The language defined by an LTL formula ϕ, written as $L(\phi)$, consists of all infinite words over the alphabet $\Sigma = 2^{\Pi}$ that satisfy ϕ. Then, ϕ is a safety formula iff $L(\phi)$ is a safety language [32].

The work in [33], which shows PSPACE-completeness for determining whether an LTL formula is safe, also provides sufficient syntactic requirements for safe LTL formulas. In particular, an LTL formula that uses only the temporal connectives $\mathcal{X}$, $\mathcal{R}$, and $\mathcal{G}$ when put in positive-normal form (by pushing negations all the way to leaves) is safe, as defined below.

Definition 7 (*Positive normal form LTL* [33]) The formulas $\texttt{true}, \texttt{false}, \pi, \neg\pi$ for $\pi \in \Pi$ are in positive normal form. If ϕ and ψ are in positive normal form, then

$$\phi \vee \psi, \phi \wedge \psi, \mathcal{X}\phi, \phi\mathcal{U}\psi, \phi\mathcal{R}\psi$$

are also in positive normal form. ($\mathcal{F}$ and $\mathcal{G}$ can be derived from $\mathcal{U}$ and $\mathcal{R}$ as indicated in Definition 6.)

Definition 8 (*Syntactically safe LTL* [33]) 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.

Since the violating prefixes of a safety LTL formula are of finite length, an NFA can be constructed to represent all the violating prefixes, as defined below.

Definition 9 (*NFA for syntactically safe LTL* [32]) For a syntactically safe LTL formula ϕ, an NFA can be constructed to represent $\neg\phi$ with at most an exponential blow-up on the size of the NFA. An NFA is a tuple $\mathcal{A} = (Z, \Sigma, \delta, z_{\text{init}}, \text{Accept})$, where

- Z is a finite set of states;
- $\Sigma = 2^{\Pi}$ is the input alphabet;
- $\delta : Z \times \Sigma \rightarrow 2^{Z}$ is the transition function;
- $z_{\text{init}} \in Z$ is the initial state; and
- $\text{Accept} \subseteq Z$ is the set of accepting states.

The set of states obtained by running $\mathcal{A}$ on $[\tau_i]_{i=1}^{n}$, $\tau_i \in 2^{\Pi}$, is defined as

$$\mathcal{A}([\tau_i]_{i=1}^{n}) = \begin{cases} z_{\text{init}}, & n = 0 \\ \bigcup_{z \in \mathcal{A}([\tau_i]_{i=1}^{n-1})} \delta(z, \tau_n), & n > 1. \end{cases}$$

$\mathcal{A}$ accepts $[\tau_i]_{i=1}^{n}$ iff $\mathcal{A}([\tau_i]_{i=1}^{n}) \cap \text{Accept} \neq \emptyset$.

2.3 LTL semantics over hybrid-system trajectories

The state space of the hybrid system gives meaning to the propositions $\pi \in \Pi$. Specifically, the truth value of each $\pi \in \Pi$ is determined by a black-box function

$$\text{PROP}_{\pi} : S \rightarrow \{\texttt{true}, \texttt{false}\}.$$

In this way, a state $s \in S$ satisfies π iff $\text{PROP}_{\pi}(s) = \texttt{true}$. Note that s can satisfy more than one proposition. In fact, the map $\tau : S \rightarrow 2^{\Pi}$ maps each $s \in S$ to the propositions satisfied by s, i.e.,

$$\tau(s) = \{\pi : \pi \in \Pi \wedge \text{PROP}_{\pi}(s) = \texttt{true}\}.$$

Moreover, τ makes it possible to contextualize LTL formulas over hybrid-system trajectories. Consider a hybrid-system trajectory $\zeta : [0, T] \rightarrow S$. At time $t_0 = 0$, ζ satisfies the set of propositions $\tau_0 = \tau(\zeta(t_0))$. The trajectory ζ will continue to satisfy τ_0 until some later time t_1, where ζ may satisfy a different set of propositions $\tau_1 = \tau(\zeta(t_1))$, $\tau_0 \neq \tau_1$. In this way, ζ can be broken down into a sequence of propositional assignments $[\tau_i]_{i=0}^{n-1}$ and a sequence of time intervals $[t_0, t_1), \ldots, [t_{n-2}, t_{n-1}), [t_{n-1}, T]$ such that inside the ith time interval ζ satisfies $\tau_i = \tau(\zeta(t_i))$ and that $\tau_i \neq \tau_{i+1}$. As discussed in Sect. 3.1.3, the sequence of propositional assignments $[\tau_i]_{i=0}^{n-1}$ satisfied by ζ is numerically computed based on the forward simulation of the hybrid-system dynamics. The definition of LTL over hybrid-system trajectories follows.

Definition 10 (*LTL over hybrid-system trajectories*) Let $\zeta : [0, T] \rightarrow S$ denote a hybrid-system trajectory. Let $0 = t_0 < \cdots < t_{n-1} \leq T$, such that

- $\tau_i \neq \tau_{i+1}$, for all $0 \leq i < n$
- $\tau_i = \tau(\zeta(t))$ for all $0 \leq i < n-1$ and $t \in [t_i, t_{i+1})$
- $\tau_{n-1} = \tau(\zeta(t))$ for all $t \in [t_{n-1}, T]$

where $\tau_i = \tau(\zeta(t_i))$. Then, $\tau(\zeta) \overset{def}{=} [\tau_i]_{i=0}^{n-1}$ denotes the sequence of propositional assignments in the order satisfied by ζ. This mapping of a hybrid-system trajectory to a sequence of propositional assignments allows for the same semantics of LTL as in Definition 6 to carry over to hybrid-system trajectories, e.g., ζ satisfies $\mathcal{X}\phi$ when $\tau(\zeta) \models \mathcal{X}\phi$. In this way, if ϕ denotes an LTL formula, then, it is said that ζ satisfies ϕ iff $\tau(\zeta) \models \phi$.

2.4 Problem statement

Given $\mathcal{P} = (\mathcal{H}, s_{\text{init}}, \phi, \Pi, \tau)$, where
– $\mathcal{H}$ is the hybrid automaton,
– $s_{\text{init}} \in S$ is the initial state,
– Π is the set of propositions, com-
– ϕ is the syntactically safe LTL formula over Π,
– τ is the propositional map, then
pute a sequence of input control functions

$$\tilde{u}_1 : [0, T_1] \rightarrow U_{q_1}, \ldots, \tilde{u}_n : [0, T_n] \rightarrow U_{q_n}$$

such that the resulting hybrid-system trajectory

$$\zeta = \Upsilon_{\langle s_{\text{init}}, \tilde{u}_1, T_1 \rangle} \circ (\tilde{u}_2, T_2) \circ \cdots \circ (\tilde{u}_n, T_n)$$

is valid and satisfies $\neg\phi$, i.e., $\text{INVARIANT}_{q_t}(x_t) = \texttt{true}$, for all $t \in [0, T_1 + \cdots + T_n]$, $(q_t, x_t) = \zeta(t)$; and $\tau(\zeta) \models \neg\phi$.

3 A straightforward approach to incorporate safety LTL into motion planning for hybrid-system falsification

The tree-search framework in motion planning has been adapted for reachability analysis in hybrid systems to compute valid witness trajectories to unsafe states [10–16]. There have been, however, no discussions in the literature [10–16] on how to augment the tree-search framework so that it can also be used for the falsification of LTL safety properties. This section describes a straightforward extension of the tree-search framework in motion planning in order to handle LTL safety properties. The idea is to use the automaton $\mathcal{A}$, which expresses $\neg\phi$, to keep track of the automaton states associated with each trajectory added to the tree and to determine from this information when a tree trajectory witnesses a violation of ϕ. 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 a straightforward approach that uses $\mathcal{A}$ simply as an external monitor is, however, computationally inefficient. Sect. 4, which describes `TemporalHyDICE`, shows how to significantly increase the computational efficiency by effectively combining the LTL tree-search framework in motion planning with model checking.

3.1 Incorporating safety LTL into the tree-search framework using the safety automaton as an external monitor

The search for a witness trajectory is conducted by extending a tree in the state space $\mathcal{H}.S$ and using $\mathcal{A}$ as an external monitor. 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 $(v_i, v_j) \in \mathcal{T}.E$ indicates that a valid hybrid-system trajectory connects $v_i.s$ to $v_j.s$. As the search proceeds iteratively, $\mathcal{T}$ is extended by adding new vertices and new edges. Consider the hybrid-system trajectory $\text{TRAJ}(\mathcal{T}, v)$ from the root of $\mathcal{T}$ to a vertex $v \in \mathcal{T}.V$. Let $\tau(\text{TRAJ}(\mathcal{T}, v))$ denote the sequence of propositional assignments $[\tau_i]_{i=0}^{n-1}$ in the order satisfied by $\text{TRAJ}(\mathcal{T}, v)$, as described in Sect. 2.3. If $\tau(\text{TRAJ}(\mathcal{T}, v)) \models \neg\phi$, then $\text{TRAJ}(\mathcal{T}, v)$ is a witness trajectory, since it indicates a violation of ϕ. Determining whether $\text{TRAJ}(\mathcal{T}, v)$ satisfies $\neg\phi$ can be accomplished by running $\mathcal{A}$ on $\tau(\text{TRAJ}(\mathcal{T}, v))$. As described in Sect. 2.2, $\text{TRAJ}(\mathcal{T}, v)$ satisfies $\neg\phi$ iff an accepting state is reached when $\mathcal{A}$ is run on $\tau(\text{TRAJ}(\mathcal{T}, v))$, i.e.,

$$\tau(\text{TRAJ}(\mathcal{T}, v)) \models \neg\phi \iff \mathcal{A}(\tau(\text{TRAJ}(\mathcal{T}, v))) \cap \mathcal{A}.\text{Accept} \neq \emptyset.$$

For this reason, each vertex $v \in \mathcal{T}.V$ is associated with the automaton states corresponding to $\text{TRAJ}(\mathcal{T}, v)$, written as $v.\alpha$ and defined as

$$v.\alpha = \mathcal{A}(\tau(\text{TRAJ}(\mathcal{T}, v))).$$

Then, $\text{TRAJ}(\mathcal{T}, v)$ satisfies $\neg\phi$ iff $v.\alpha \cap \mathcal{A}.\text{Accept} \neq \emptyset$. Note that $v.\alpha$ can be computed incrementally when v is added to its parent v_{parent} in $\mathcal{T}$. In particular, let ζ denote the hybrid-system trajectory from v_{parent} to v; ζ is associated with the edge $(v_{\text{parent}}, v) \in \mathcal{T}.E$. Then, $v.\alpha$ can be computed by running $\mathcal{A}$ on $\tau(\zeta)$ but starting from the states $v_{\text{parent}}.\alpha$ instead of the initial state of $\mathcal{A}$.

In this way, when a vertex v is added to $\mathcal{T}$ such that $\text{TRAJ}(\mathcal{T}, v)$ satisfies $\neg\phi$, then a witness trajectory is found. Otherwise, new vertices and new edges will continue to be added to $\mathcal{T}$ until an upper bound on the running time is exceeded. Pseudocode for the tree-search framework is given in Algorithm 1 and described below.

Algorithm 1 `LTL-TSF`$(\mathcal{P}, t_{\max})$
Incorporating Safety LTL into the Tree-Search Framework by using the Safety Automaton as an External Monitor

Input
$\mathcal{P} = (\mathcal{H}, s_{\text{init}}, \phi, \Pi, \tau)$: problem specification ◇ 2.4
$t_{\max} \in \mathbb{R}^{>0}$: upper bound on computation time
Output
A valid hybrid-system trajectory that satisfies $\neg\phi$ if one is found or `false` otherwise

```
1: $\mathcal{A} \leftarrow$ construct automaton for $\neg\phi$
2: $\mathcal{T} \leftarrow$ INITIALIZETREE$(\mathcal{P}, \mathcal{A})$                          ◇ 3.1.1
3: while ELAPSEDTIME $< t_{\max}$ do
4:    $v \leftarrow$ SELECTVERTEXFROMTREE$(\mathcal{P}, \mathcal{T})$                   ◇ 3.1.2
5:    $(u, T, s_{\text{new}}, \alpha_{\text{new}}) \leftarrow$ EXTENDTREE$(\mathcal{P}, \mathcal{A}, \mathcal{T}, v)$         ◇ 3.1.3
6:    if $T > 0 \wedge |\alpha_{\text{new}}| > 0$ then
        add a new vertex and edge to $\mathcal{T}$
7:      $v_{\text{new}} \leftarrow$ new vertex; $v_{\text{new}}.s \leftarrow s_{\text{new}}$; $v_{\text{new}}.\alpha \leftarrow \alpha_{\text{new}}$
8:      $(v, v_{\text{new}}) \leftarrow$ new edge; $(v, v_{\text{new}}).\{u, T\} \leftarrow \{u, T\}$
9:      $\mathcal{T}.V \leftarrow \mathcal{T}.V \cup \{v_{\text{new}}\}$; $\mathcal{T}.E \leftarrow \mathcal{T}.E \cup \{(v, v_{\text{new}})\}$
10:   if $\mathcal{A}$.Accept $\cap\, \alpha_{\text{new}} \neq \emptyset$ then
11:     return TRAJ$(\mathcal{T}, v_{\text{new}})$
12: return false
```

3.1.1 *Initializing the tree*

INITIALIZETREE$(\mathcal{P}, \mathcal{A})$ (Algorithm 1:2) associates the root vertex v_{init} with the initial hybrid-system state, i.e., $v_{\text{init}}.s = s_{\text{init}}$, $\mathcal{T}.V = \{v_{\text{init}}\}$, and $\mathcal{T}.E = \emptyset$. The automaton states associated with v_{init} are computed by running $\mathcal{A}$ on the propositions satisfied by $v_{\text{init}}.s$, i.e.,

$$v_{\text{init}}.\alpha = \mathcal{A}.\delta(\mathcal{A}.z_{\text{init}}, \tau(v_{\text{init}}.s)).$$

3.1.2 *Selecting a vertex from which to extend the tree*

SELECTVERTEXFROMTREE$(\mathcal{P}, \mathcal{T})$ (Algorithm 1:4) 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 distance metrics, nearest neighbors, probability distributions, and many other factors, as surveyed in [37,38]. As an example, `RRT` methods [10–14], sample a hybrid-system state $s \in \mathcal{H}.S$ and select the vertex $v \in \mathcal{T}.V$ whose associated hybrid-system state $v.s$ is the closest to s according to a distance metric. Other selection strategies are discussed in Sect. 3.2 when describing how related work [10–14] can use Algorithm 1 to incorporate LTL safety properties.

3.1.3 *Extending the tree by applying input controls and simulating the hybrid system forward in time*

EXTENDTREE$(\mathcal{P}, \mathcal{A}, \mathcal{T}, v)$ (Algorithm 1:5) extends $\mathcal{T}$ from the selected vertex v by computing a valid hybrid-system trajectory $\zeta : [0, T] \rightarrow \mathcal{H}.S$ that starts at $v.s$. A common strategy is to apply some input control function $\tilde{u} : [0, T] \rightarrow \mathcal{H}.U_q$ to $v.s$, where q is the mode of $v.s$, and follow the dynamics $\mathcal{H}.f_q$ until the invariant is no longer satisfied, a guard is satisfied, or a maximum number of steps is exceeded [10–16,37,38]. Note that if a guard is satisfied, the corresponding discrete transition is taken immediately and as a result will be part of the computed trajectory ζ, as described in Definition 3, i.e.,

$$\zeta = \Upsilon_{\langle v.s,\tilde{u},T\rangle}.$$

ExtendTree$(\mathcal{P}, \mathcal{A}, \mathcal{T}, v)$ returns a tuple $(u, T, s_{\text{new}}, \alpha_{\text{new}})$. The control input function $\tilde{u} : [0, T] \rightarrow \mathcal{H}.U_q$ is generally defined by selecting pseudo-uniformly at random an input control $u \in \mathcal{H}.U_q$, i.e.,

$$\tilde{u}(t) = u, \quad \forall t \in [0, T].$$

The random selection of input controls allows for subsequent calls to ExtendTree$(\mathcal{P}, \mathcal{A}, \mathcal{T}, v)$ to extend $\mathcal{T}$ along new directions. Moreover, it has been shown to work well in motion planning for robotic systems with complex continuous dynamics and reachability analysis in hybrid systems [10–16,18,37,38].

The trajectory ζ computed by ExtendTree is added to $\mathcal{T}$ as a new branch at the vertex v (Algorithm 1:7–9). More precisely, the new vertex v_{new} and the edge (v, v_{new}) are added to the vertices and edges of $\mathcal{T}$, respectively. The new edge (v, v_{new}) is associated with the input control function $\tilde{u}$ and time duration T. The new vertex v_{new} is associated with the state s_{new}, which corresponds to the last state of ζ, i.e., $s_{\text{new}} = \zeta(T)$. The automaton states α_{new} are computed by running $\mathcal{A}$ on $\tau(\zeta)$ but starting from the automaton states $v.\alpha$ instead of the initial state of $\mathcal{A}$. In this way, as discussed in Sect. 3.1, α_{new} corresponds to the automaton states obtained by running $\mathcal{A}$ on $\tau(\text{TRAJ}(\mathcal{T}, v) \circ \zeta)$. If α_{new} includes an accepting state of $\mathcal{A}$, then $\text{TRAJ}(\mathcal{T}, v_{\text{new}})$ constitutes a witness trajectory (Algorithm 1:10). $\text{TRAJ}(\mathcal{T}, v_{\text{new}})$ is computed by concatenating the trajectories associated with the tree edges connecting the root of $\mathcal{T}$ to v_{new} (Algorithm 1:11).

Note that any hybrid-system simulation method can be used to compute the hybrid-system trajectory ζ by simulating the continuous dynamics and the discrete transitions of the hybrid system when applying an input control function $\tilde{u} : [0, T] \rightarrow \mathcal{H}.U_q$ starting from the state $v.s$. For completeness, we describe below a simple iterative procedure. Pseudocode is given in Algorithm 2.

Let n_{steps} denote the number of steps and let $\epsilon > 0$ denote the step size (Algorithm 2:1). Initially, $x_0 = x$ and $\alpha_0 = v.\alpha$, where $v.s = (q, x)$ (Algorithm 2:2). At the ith iteration, x_i is computed by integrating $\mathcal{H}.f_q$ from x_{i-1} for an ϵ time step (Algorithm 2:5). If x_i does not satisfy the invariant, then ExtendTree stops and returns the tuple $[u, (i-1) * \epsilon, (q, x_{i-1}), \alpha_{i-1}]$ (Algorithm 2:6–7). When x_i satisfies the invariant, then ExtendTree checks if a guard is satisfied, which would indicate a discrete event (Algorithm 2:8). In the hybrid-system benchmark in this paper, when a guard condition is satisfied, the corresponding jump is always applied. In the case of non-urgent discrete transitions,

Algorithm 2 ExtendTree$(\mathcal{P}, \mathcal{A}, \mathcal{T}, v)$
Extend Tree by Applying Input Controls and Simulating the Hybrid System Forward in Time

Input
$\mathcal{P} = (\mathcal{H}, s_{\text{init}}, \phi, \Pi, \tau)$: problem specification ◇ 2.4
$\mathcal{A}$: automaton for $\neg\phi$
$\mathcal{T}$: current search tree
v: vertex from which to extend the tree
Output $(u, T, s_{\text{new}}, \alpha_{\text{new}})$

```
1: $\epsilon \in \mathbb{R}^{>0} \leftarrow$ time step; $n_{\text{steps}} \in \mathbb{N} \leftarrow$ number of steps
2: $s = (q, x) \leftarrow v.s$; $\alpha \leftarrow v.\alpha$; $x_0 \leftarrow x$; $\alpha_0 \leftarrow \alpha$; $\tau_0 \leftarrow \tau(s)$
3: $u \leftarrow$ sample control from $\mathcal{H}.U_q$
   simulate the continuous and discrete dynamics of $\mathcal{H}$
4: for $i = 1, 2, \ldots, n_{\text{steps}}$ do
5:   $x_i \leftarrow \chi(\epsilon)$, where $\chi(\epsilon) = x_{i-1} + \int_{h=0}^{\epsilon} f_q(\chi(h), u)\, dh$
6:   if $\mathcal{H}.\text{INVARIANT}_q(x_i) = \texttt{false}$ then
7:     return $(u, (i-1) * \epsilon, (q, x_{i-1}), \alpha_{i-1})$
8:   if for some $\mathcal{H}.\text{GUARD}_{(q,q_{\text{new}})}(x_i) = \texttt{true}$
     and discrete transition should be triggered then
9:     $(x_{\text{loc}}, T) \leftarrow$ localize discrete event in $((i-1) * \epsilon, i * \epsilon]$
10:    $\tau_{\text{loc}} \leftarrow \tau((q, x_{\text{loc}}))$
11:    if $\tau_{i-1} = \tau_{\text{loc}}$ then $\alpha_{\text{loc}} \leftarrow \alpha_{i-1}$
12:    else $\alpha_{\text{loc}} \leftarrow \cup_{z \in \alpha_{i-1}} \mathcal{A}.\delta(z, \tau_{\text{loc}})$
13:    $x_{\text{new}} \leftarrow \mathcal{H}.\text{JUMP}_{(q,q_{\text{new}})}(x_{\text{loc}})$
14:    $\tau_{\text{new}} \leftarrow \tau((q_{\text{new}}, x_{\text{new}}))$
15:    if $\tau_{\text{loc}} = \tau_{\text{new}}$ then $\alpha_{\text{new}} \leftarrow \alpha_{\text{loc}}$
16:    else $\alpha_{\text{new}} \leftarrow \cup_{z \in \alpha_{\text{loc}}} \mathcal{A}.\delta(z, \tau_{\text{new}})$
17:    return $(u, T, (q_{\text{new}}, x_{\text{new}}), \alpha_{\text{new}})$
18:  else
19:    $\tau_i \leftarrow \tau((q, x_i))$
20:    if $\tau_{i-1} = \tau_i$ then $\alpha_i \leftarrow \alpha_{i-1}$
21:    else $\alpha_i \leftarrow \cup_{z \in \alpha_{i-1}} \mathcal{A}.\delta(z, \tau_i)$
22: return $(u, n_{\text{steps}} * \epsilon, (q, x_{n_{\text{steps}}}), \alpha_{n_{\text{steps}}})$
```

as discussed in Sect. 2.1, a user-defined procedure can determine when to apply the discrete transition. Discrete event detection is followed by event localization, which localizes the earliest time $T \in ((i-1)*\epsilon, i*\epsilon]$ where the guard is satisfied (Algorithm 2:9). Bisection or bracketing algorithms are typically used for event localization [39]. Once the discrete event is localized, the discrete transition is then triggered to obtain the new state (Algorithm 2:13). The automaton states are also updated accordingly and $(u, t, (q_{\text{new}}, x_{\text{new}}), \alpha_{\text{new}})$ is returned (Algorithm 2:14–17). If no guard is satisfied, the automaton states α_i associated with (q, x_i) are updated only if $\tau((q, x_i)) \neq \tau((q, x_{i-1}))$ (Algorithm 2:19–21).

Numerical errors in simulation, invariant checking, event localization could in certain cases cause ExtendTree to miss an invariant violation, miss a guard, trigger a different discrete transition, or miss a change in the sequence of propositional assignments satisfied by a hybrid-system trajectory ζ, e.g., $\tau(\zeta(t)) = \tau(\zeta(t+\epsilon))$ but $\tau(\zeta(t+a)) \neq \tau(\zeta(t))$ for some $0 < a < \epsilon$. To minimize such errors, a practical approach is to choose a small ϵ. This approach is the norm in hybrid-system falsification based on motion planning [10–16]. For hybrid systems with guards of the form $\{x : g(x) \geq 0\}$ where $g(x)$ is continuously differentiable, it is also possible to use more accurate event localization algorithms, which come asymptotically close to the guard boundary[39]. In many practical cases, hybrid systems exhibit a degree of robustness [31,40] 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.

3.2 Incorporating safety LTL into related motion-planning approaches for hybrid-system falsification

Using $\mathcal{A}$ as an external monitor, Algorithm 1 provides a straightforward extension of how to incorporate LTL safety properties into the tree-search framework. This makes it possible to adapt related work in hybrid-system falsification [10–16] so that it can handle LTL, as described below.

3.2.1 Incorporating safety LTL into `RRT` methods

LTL can be incorporated into `RRT`-based falsification methods [10–14] using `LTL-TSF` (Algorithm 1) and implementing SelectVertexFromTree($\mathcal{P}$, $\mathcal{T}$) as in [10–14], e.g., sample $s \in \mathcal{H}.S$ 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]` in this paper.

3.2.2 Incorporating LTL safety properties into `HyDICE[NoGuide]`

Similarly to `RRT`, `HyDICE` [15,16] also falls into the broad category of tree-search algorithms. Distinctly from `RRT`, `HyDICE` [15,16] 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 over `RRT`-based falsification [12–14].

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 a 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` [15,16] that does not use the discrete search is referred to as `HyDICE[NoGuide]`. Experiments in [15,16] showed that `HyDICE[NoGuide]` was significantly slower than `HyDICE`, but still considerably faster than `RRT`-based falsification [12–14]. `HyDICE[NoGuide]` corresponds to the tree-search framework, where the function SelectVertexFromTree($\mathcal{P}$, $\mathcal{T}$) is implemented by selecting $v \in \mathcal{T}.V$ according to a probability distribution over $\mathcal{T}.V$, i.e., each $v \in \mathcal{T}.V$ is assigned a weight $w(v)$ based on the number of times v has been selected in the past and the coverage density around v by other vertices in $\mathcal{T}$; then v is selected with probability proportional to $w(v)$. This makes it possible to incorporate LTL safety properties into `HyDICE[NoGuide]` using `LTL-TSF` (Algorithm 1), which is referred to as `HyDICE[NoGuide, LTL-TSF]` in this paper.

4 `TemporalHyDICE`

This section describes `TemporalHyDICE`, which constitutes the main contribution of this paper. `TemporalHyDICE` combines model checking and motion planning to effectively expand $\mathcal{T}$ during the search for a witness trajectory. To speed up the search for witness trajectories, `TemporalHyDICE` also needs as input an abstraction $\mathcal{M}$ of the hybrid system $\mathcal{H}$. The construction of $\mathcal{M}$ is outside the scope of this paper; this paper focuses on how to effectively combine model checking with motion planning. For the purposes of this work the abstraction $\mathcal{M}$ is thought of as a graph $\mathcal{D}$. Generally, a vertex $v_i \in \mathcal{D}$ corresponds to some state-space region whose states satisfy a propositional assignment τ_i and an edge $(v_i, v_j) \in \mathcal{D}$ indicates the possibility of constructing a hybrid-system trajectory ζ that remains in the region associated with v_i until it reaches the region associated with v_j. An example is given in the experiments in Sect. 5. Interestingly, very few requirements are placed on $\mathcal{M}$ (and hence $\mathcal{D}$). `TemporalHyDICE` tries to compensate for any shortcomings by exploiting the synergy of model checking and motion planning as explained in the next section.

4.1 Combining model checking and motion planning

Consider a sequence $[\tau_i]_{i=0}^{n-1}$ of propositional assignments that satisfies $\neg\phi$. Let

$$\Gamma(\tau_i) = \{s \in \mathcal{H}.S : \tau(s) = \tau_i\}.$$

If $\mathcal{T}$ can be extended so that a hybrid-system trajectory $\text{TRAJ}(\mathcal{T}, v)$ starts at $\Gamma(\tau_0)$ and reaches $\Gamma(\tau_1), \ldots, \Gamma(\tau_{n-1})$ in succession, i.e.,

$$\tau(\text{TRAJ}(\mathcal{T}, v)) = [\tau_i]_{i=0}^{n-1},$$

then $\text{TRAJ}(\mathcal{T}, v)$ would be a witness trajectory. In this way, $[\tau_i]_{i=0}^{n-1}$ provides a sequence $\Gamma(\tau_0), \ldots, \Gamma(\tau_{n-1})$ of regions along which motion planning can attempt to extend $\mathcal{T}$ to compute a witness trajectory.

TemporalHyDICE relies on model checking to effectively compute sequences of propositional assignments that satisfy $\neg\phi$. Specifically, model checking searches on-the-fly $\mathcal{A}$ and the graph $\mathcal{D}$ to compute a discrete witness, i.e., a sequence

$$[(z_i, \tau_i)]_{i=0}^{n-1}$$

where $(z_i, \tau_i) \in \mathcal{A}.Z \times 2^{\Pi}$, $z_0 = \mathcal{A}.z_{\text{init}}$, $z_{n-1} \in \mathcal{A}.\text{Accept}$. By not computing $\mathcal{D} \times \mathcal{A}$ explicitly, as described in Sect. 4.2, TemporalHyDICE considerably reduces the memory used by model checking. Note that model checking can provide many alternative discrete witnesses. Due to hybrid-system dynamics, invariants, guards, and jumps, in some cases, it may be easy for motion planning to extend $\mathcal{T}$ from some $\Gamma(\tau_i)$ to $\Gamma(\tau_{i+1})$ as indicated in the discrete witness, while in other cases, it may be difficult or even impossible. This raises the issue of which discrete witness to select among the many available alternatives. To address this issue, TemporalHyDICE maintains a running estimate

$$\text{COST}(z_i, \tau_i)$$

on the cost of having the motion planner spend additional computational time attempting to extend $\mathcal{T}$ from (z_i, τ_i).vertices to $\Gamma(\tau_{i+1})$, where

$$(z_i, \tau_i).\text{vertices} = \{v \in \mathcal{T}.V : z_i \in v.\alpha \wedge \tau_i = \tau(v.s)\}.$$

Note that $\tau_i = \tau(v.s)$ means that $\text{TRAJ}(\mathcal{T}, v)$ has reached $\Gamma(\tau_i)$, and $z_i \in v.\alpha$ means that z_i is an automaton state obtained when running $\mathcal{A}$ on $\tau(\text{TRAJ}(\mathcal{T}, v))$. Therefore, (z_i, τ_i).vertices indicate that TemporalHyDICE has extended $\mathcal{T}$ in succession from

$$\Gamma(\tau_0), \ldots, \Gamma(\tau_i).$$

In this way, when $\text{COST}(z_i, \tau_i)$ is low, TemporalHyDICE estimates that more computational time should be dedicated to extending $\mathcal{T}$ from (z_i, τ_i).vertices toward $\Gamma(\tau_{i+1})$ so that $\mathcal{T}$ reaches $\Gamma(\tau_0), \ldots, \Gamma(\tau_{i+1})$ in succession. As described in Sect. 4.3, $\text{COST}(z_i, \tau_i)$ is low when (z_i, τ_i) is underexplored, since additional exploration by motion planning could add new connections from (z_i, τ_i).vertices to $\Gamma(\tau_{i+1})$ and advance the search further. When (z_i, τ_i) is overexplored, then $\text{COST}(z_i, \tau_i)$ is high, since overexploration does not bring much new information and wastes valuable computational time. The cost of a discrete witness is then defined as

$$\text{COST}\left([(z_i, \tau_i)]_{i=0}^{n-1}\right) = \sum_{i=0}^{n-2} \text{COST}(z_i, \tau_i)\, \text{COST}(z_{i+1}, \tau_{i+1}).$$

Since when exploring (z_i, τ_i) the objective of motion planning is to reach $\Gamma(\tau_{i+1})$ from (z_i, τ_i).vertices, then $\text{COST}(z_i, \tau_i)\,\text{COST}(z_{i+1}, \tau_{i+1})$ is used in the summation instead of just $\text{COST}(z_i, \tau_i)$. Note that cost estimates in this paper, drawing from our earlier work [15,16] and extensive experiments, are designed to be computed efficiently and are shown to work well in practice.

Algorithm 3 TemporalHyDICE($\mathcal{P}, \mathcal{D}, t_{\max}$)
Combining Model Checking and Motion Planning

```
Input
  $\mathcal{P} = (\mathcal{H}, s_{\text{init}}, \phi, \Pi, \tau)$: problem specification                ◇ 2.4
  $\mathcal{D} = (V, E)$: abstraction
  $t_{\max} \in \mathbb{R}^{>0}$: upper bound on computation time
Output
  A valid hybrid-system trajectory that satisfies $\neg\phi$ if one is found or
  false otherwise
1: $\mathcal{A} \leftarrow$ construct automaton for $\neg\phi$
2: $\mathcal{T} \leftarrow$ InitializeTree($\mathcal{P}, \mathcal{A}$)                                   ◇3.1.1
3: while ElapsedTime $< t_{\max}$ do
     model checking: search on-the-fly $\mathcal{D}$ and $\mathcal{A}$
     bias search toward low-cost discrete witnesses
4:   $\sigma \overset{def}{=} [(z_i, \tau_i)]_{i=1}^{n} \leftarrow$ DiscreteWitness($\mathcal{P}, \mathcal{D}, \mathcal{A}$)           ◇4.2
     motion planning: extend $\mathcal{T}$ guided by discrete witness
5:   $\sigma_{\text{avail}} \leftarrow \{(z_i, \tau_i) \in \sigma : (z_i, \tau_i).\text{vertices} \neq \emptyset\}$          ◇4.4
6:   for several times do
7:     $(z_i, \tau_i) \leftarrow$ select pair from $\sigma_{\text{avail}}$                          ◇4.4
8:     $v \leftarrow$ select vertex from $(z_i, \tau_i)$.vertices                       ◇4.4
9:     $(u, T, s_{\text{new}}, \alpha_{\text{new}}) \leftarrow$ ExtendTree($\mathcal{P}, \mathcal{A}, \mathcal{T}, v$)           ◇3.1.3
10:    if $T > 0 \wedge |\alpha_{\text{new}}| > 0$ then
         add a new vertex and edge to $\mathcal{T}$
11:      $v_{\text{new}} \leftarrow$ new vertex; $v_{\text{new}}.s \leftarrow s_{\text{new}}$; $v_{\text{new}}.\alpha \leftarrow \alpha_{\text{new}}$
12:      $(v, v_{\text{new}}) \leftarrow$ new edge; $(v, v_{\text{new}}).\{u, T\} \leftarrow \{u, T\}$
13:      $\mathcal{T}.V \leftarrow \mathcal{T}.V \cup \{v_{\text{new}}\}$; $\mathcal{T}.E \leftarrow \mathcal{T}.E \cup \{(v, v_{\text{new}})\}$
14:    if $\mathcal{A}.\text{Accept} \cap \alpha_{\text{new}} \neq \emptyset$ then
15:      return TRAJ($\mathcal{T}, v_{\text{new}}$)
       update cost estimates based on new info gathered from motion
       planning
16:    UpdateCost($z_i, \tau_i$)                                            ◇4.3
17:    $\tau_{\text{new}} \leftarrow \mathcal{P}.\tau(v_{\text{new}}.s)$
18:    for $z_{\text{new}} \in \alpha_{\text{new}}$ do
19:      $\sigma_{\text{avail}} \leftarrow \{(z_{\text{new}}, \tau_{\text{new}})\} \cup \sigma_{\text{avail}}$                      ◇4.4
20:      $(z_{\text{new}}, \tau_{\text{new}})$.vertices $\leftarrow \{v_{\text{new}}\} \cup (z_{\text{new}}, \tau_{\text{new}})$.vertices
21:      UpdateCost($z_{\text{new}}, \tau_{\text{new}}$)                                  ◇4.3
22: return false
```

Model checking in TemporalHyDICE biases the computation toward discrete witnesses associated with low cost. TemporalHyDICE estimates that additional exploration of these discrete witnesses can advance the search further by increasing exploration in underexplored regions and avoiding spending computational time in overexplored regions. TemporalHyDICE does not completely ignore high-cost discrete witnesses. In particular, in addition to low-cost discrete witnesses, random discrete witnesses are selected, although less frequently, as a way to correct for errors inherent with the cost estimates.

The computation of the current discrete witness, expansion of $\mathcal{T}$ by motion planning as guided by the discrete witness, and updates to cost estimates based on new information gathered by motion planning constitute the core loop of TemporalHyDICE. As a result of the updated cost estimates, a new discrete witness can be selected by model checking in the next iteration of the core loop. In this way, information gathered by motion planning leads `TemporalHyDICE` to consider alternative discrete witnesses that could expand the search for a witness trajectory along new directions. This interplay between model checking and motion planning through cost estimates, as demonstrated by the experiments, allows `TemporalHyDICE` to efficiently compute witness trajectories. Pseudocode is given in Algorithm 3. Sections describing the main steps in Algorithm 3 are referenced after each line.

4.2 Computation of discrete witnesses

DiscreteWitness($\mathcal{P}, \mathcal{A}, \mathcal{D}$) (Algorithm 3:4) uses model checking to compute discrete witnesses by searching on-the-fly $\mathcal{A}$ and $\mathcal{D}$. The search produces a sequence $[(z_i, \tau_i)]_{i=0}^{n-1}$, where $(z_i, \tau_i) \in \mathcal{A}.Z \times 2^{\Pi}$, $z_0 = \mathcal{A}.z_{\text{init}}$, and $z_{n-1} \in \mathcal{A}$.Accept.

With high probability, the discrete witness is computed as the shortest path from initial to accepting states in $\mathcal{A} \times \mathcal{D}$ where an edge $((z_i, \tau_i), (z_j, \tau_j))$ is assigned the weight $\text{COST}(z_i, \tau_i)\,\text{COST}(z_j, \tau_j)$. This allows to select low-cost discrete witnesses. With small probability, the discrete witness is also computed as a random path by using a variation of depth-first-search which visits frontier nodes in a random order. This randomness provides a way to correct for errors inherent with the cost estimates by ensuring that each discrete witness is selected with non-zero probability.

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

$$\text{EDGES}(z_i, \tau_i) = \{(z_j, \tau_j) : (v(\tau_i), v(\tau_j)) \in \mathcal{D}.E \wedge z_j \in \mathcal{A}.\delta(z_i, \tau_j)\}.$$

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 $\Sigma = 2^{\Pi}$. The graph $\mathcal{D}$ can be viewed as a binary relation, subset of $\Sigma \times \Sigma$. Explicitly constructing $\mathcal{A} \times \mathcal{D}$ 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 $\mathcal{A} \times \mathcal{D}$ explicitly. Reducing memory requirements is important, since it allows motion planning to extend $\mathcal{T}$ by adding more vertices and edges.

4.3 Cost estimates

Cost estimates are based on information gathered during motion planning. Specifically,

$$\text{COST}(z_i, \tau_i) = \frac{\text{TIME}^2(z_i, \tau_i)}{\text{COV}(z_i, \tau_i)}$$

where $\text{TIME}(z_i, \tau_i)$ is the number of times motion planning has attempted to extend $\mathcal{T}$ from (z_i, τ_i).vertices and $\text{COV}(z_i, \tau_i)$ estimates the coverage of $\Gamma(\tau_i)$ by states associated with (z_i, τ_i).vertices. In this way, `TemporalHyDICE` associates a low $\text{COST}(z_i, \tau_i)$ with (z_i, τ_i) when motion planning has made rapid progress in covering $\Gamma(\tau_i)$ with states associated with (z_i, τ_i).vertices. The component $\text{TIME}(z_i, \tau_i)$ gives priority to (z_i, τ_i)'s that have not been selected frequently in the past. These cost estimates are designed to be computed efficiently and are motivated by related work [13–16,41] and extensive experiments, which show that it works well in practice.

As in [15,16], $\text{COV}(z_i, \tau_i)$ 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, τ_i).vertices. In the experiments in this paper, where the hybrid system models a robotic vehicle navigating in different terrains, the low-dimensional projection corresponds to the vehicle position, i.e., $\text{POSITION}(x)$ (Sect. 5.1). Note that $\text{COV}(z_i, \tau_i)$ needs to be updated only when a vertex $v \in \mathcal{T}.V$ is added to (z_i, τ_i).vertices. To make this update efficient, each (z_i, τ_i) maintains its own list of grid cells, (z_i, τ_i).cells. Initially, (z_i, τ_i).cells is empty. When a vertex v is added to (z_i, τ_i).vertices, the low-dimensional projection of $v.s = (q, x)$ is computed as $p = \text{POSITION}(x)$, which is then used to determine the grid cell c that should contain p. The cell c is then added to (z_i, τ_i).cells if it is not already there. A hash-map is used to efficiently search if $c \in (z_i, \tau_i)$.cells. In this way, $\text{COV}(z_i, \tau_i)$ is efficiently updated as the number of covered cells, i.e., $\text{COV}(z_i, \tau_i) = |(z_i, \tau_i).\text{cells}|$.

As evidenced by the experiments, the cost estimates allow model checking to compute low-cost discrete witnesses that effectively guide motion planning as it expands $\mathcal{T}$ during the search for a witness trajectory. Since the cost estimates are updated efficiently there is also little computational overhead.

4.4 Motion planning

Let $\sigma = [(z_i, \tau_i)]_{i=0}^{n-1}$ denote the current discrete witness. The objective is to extend $\mathcal{T}$ so that it reaches $\Gamma(\tau_0), \ldots, \Gamma(\tau_{n-1})$ in succession. Motion planning proceeds by extending $\mathcal{T}$ from vertices associated with pairs (z_i, τ_i) (Algorithm 3:5–15). Note that only pairs $(z_i, \tau_i) \in \sigma$ reached by $\mathcal{T}$, i.e., (z_i, τ_i).vertices $\neq \emptyset$, have vertices from which to extend $\mathcal{T}$. Let σ_{avail} contain all such pairs, i.e.,

$$\sigma_{\text{avail}} = \{(z_i, \tau_i) \in \sigma : (z_i, \tau_i).\text{vertices} \neq 0\} \quad \text{(Algorithm 3:5)}$$

At each iteration, a pair (z_i, τ_i) is selected from σ_{avail} to be further explored with probability

$$\frac{1/\text{COST}(z_i, \tau_i)}{\sum_{(z_j, \tau_j) \in \sigma_{\text{avail}}} 1/\text{COST}(z_j, \tau_j)} \quad \text{(Algorithm 3:7)}$$

This selection favors pairs (z_i, τ_i) associated with low cost, which indicate that additional progress can be made by further exploring (z_i, τ_i). The exploration consists of extending several new branches from (z_i, τ_i).vertices (Algorithm 3:9–15). Specifically, a vertex v is selected from (z_i, τ_i).vertices with probability

$$\frac{1/\text{TIME}(v)}{\sum_{v' \in (z_i, \tau_i).\text{vertices}} 1/\text{TIME}(v')}, \qquad \text{(Algorithm 3:8)}$$

where TIME(v) is one plus the number of times v has been selected in the past from (z_i, τ_i).vertices. This is based on well-established strategies in motion planning that favor those vertices selected less frequently in the past [37,38]. After a vertex v has been selected, as described in Sect. 3, $\mathcal{T}$ is extended from v by adding to $\mathcal{T}$ a valid hybrid-system trajectory that starts at $v.s$ and ends at a new vertex, $v_{\text{new}}.s$ (Algorithm 3:9–13). If any of the automaton states $v_{\text{new}}.\alpha$ is an accepting state, then TRAJ$(\mathcal{T}, v_{\text{new}})$ is a witness trajectory (Algorithm 3:14–15).

As a result of the exploration, cost estimates need to be updated (Algorithm 3:16–21) as described in Sect. 4.3. Moreover, v_{new} is associated with each $(z_{\text{new}}, \tau_{\text{new}})$, where $z_{\text{new}} \in v_{\text{new}}.\alpha$ and $\tau_{\text{new}} = \tau(v_{\text{new}}.s)$. COST$(z_{\text{new}}, \tau_{\text{new}})$ is also updated to reflect adding v_{new} to $(z_{\text{new}}, \tau_{\text{new}})$.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.

Updates to the cost estimates allow motion planning to select other pairs $(z_j, \tau_j) \in \sigma_{\text{avail}}$ to explore during the remaining iterations. Moreover, the updates provide better cost estimates for the discrete witnesses, which improves the selection of discrete witnesses in the next iteration of the core loop of `TemporalHyDICE`. This in turn allows motion planning to make more progress in extending $\mathcal{T}$ as guided by the discrete witness and eventually compute a witness trajectory.

5 Experiments and results

Experiments in this paper show that `TemporalHyDICE` provides a promising framework for the falsification of safety properties expressed by syntactically safe LTL formulas for hybrid systems with nonlinear continuous dynamics. `TemporalHyDICE` is shown to be significantly more efficient than the straightforward extensions of related work [10–16], which use the automaton $\mathcal{A}$ as an external monitor (Sect. 3). The experiments also demonstrate the importance of model checking and of the discrete transition model in the computational efficiency of `TemporalHyDICE`. This paper also studies the impact on the computational efficiency of `TemporalHyDICE` when using a minimized DFA, minimized NFA constructed by hand, or NFA constructed by standard tools [42].

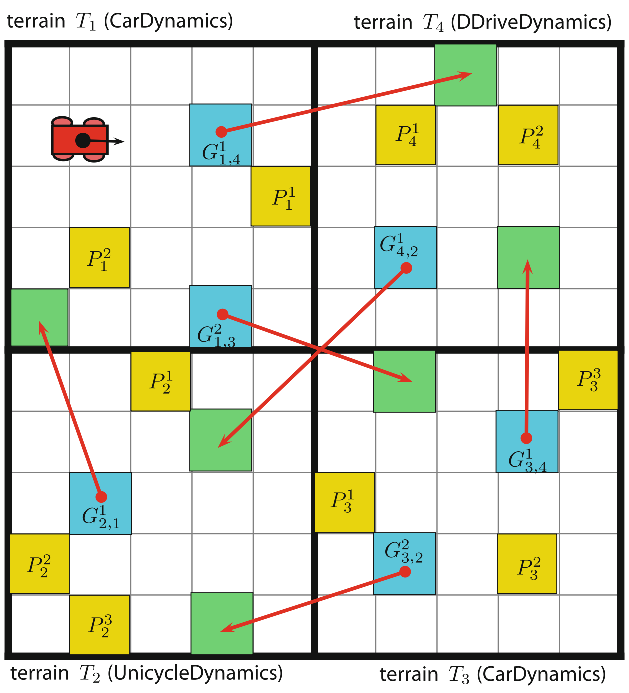

Fig. 1 In the hybrid-system benchmark, a vehicle drives over different terrains. The vehicle dynamics vary from terrain to terrain, i.e., the vehicle dynamics in each terrain T_i are selected from those of a car, unicycle, or differential drive. Each T_i is subdivided into a uniform grid, and some grid cells are labeled as guards, jumps, or propositions. Grid cells labeled with $G^k_{i,j}$ (in *blue*) denote guards. Discrete transitions are indicated by straight arrows. A discrete transition occurs when the vehicle center reaches a guard. Only one discrete transition per guard is shown. The corresponding jump cells are in *green*. Grid cells labeled with P^k_i (in *yellow*) denote propositions (color figure online)

5.1 Hybrid-system robot navigation benchmark

The hybrid system $\mathcal{H}$ models an autonomous vehicle driving over different terrains, similar to the navigation benchmark proposed in [34] and used in [15,16]. Figure 1 provides an illustration. Benchmark instances, as described below, are generated at random in order to test falsification methods over a large number of problems and obtain statistically significant results.

Terrains The number of terrains is $n_T = 10$. Each terrain T_i is a unit square. The vehicle dynamics vary from terrain to terrain. Specifically, the dynamics in each T_i are selected pseudo-uniformly at random from those of a car, unicycle, or differential drive, which are described at the end of this section.

To compute guards, jumps, and propositions, each terrain T_i is subdivided into a 35×35 uniform grid. Let $c_1, \ldots, c_m$ denote a random permutation of all the grid cells from all the terrains ($m = n_T \times 35 \times 35$).

Guards and jumps $c_1, \ldots, c_{n_G}$ are labeled as guards, where $n_G = 50$. For each guard cell, the corresponding jump cell is selected uniformly at random from the remaining grid cells, i.e., $\{c_{n_G+1}, \ldots, c_m\}$. A grid cell labeled as a guard $G^k_{i,j}$ defines a guard function as follows:

$$\text{GUARD}^k_{(q_i,q_j)}(x) = \texttt{true} \iff \text{POSITION}(x) \in G^k_{i,j},$$

where POSITION(x) denotes the position of the vehicle center when the continuous state is x. In other words, GUARD$^k_{(q_i,q_j)}(x)$ is satisfied when the vehicle center reaches $G^k_{i,j}$. In that case, a discrete transition is triggered, which instantaneously changes the vehicle dynamics to the dynamics associated with terrain T_j and resets the continuous state according to JUMP$^k_{(q_i,q_j)}$, where[1]

$$\text{JUMP}^k_{(q_i,q_j)}(x) = (\text{POSITION}(x) + \text{CENTER}(T_j) \\ -\text{CENTER}(T_i), \text{ORIENTATION}(x), 0, \ldots, 0)$$

and CENTER(T_i) denotes the center of terrain T_i. Thus, the vehicle maintains the same position and orientation with respect to the origin of T_j as it did with respect to the origin of T_i immediately before the discrete transition. In this way, guards and jumps can be thought as providing overpasses that allow the vehicle to immediately move from one terrain to another.

Propositions The number of propositions is $n_P = 150$. Then, a random subset of n_P grid cells is selected from $\{c_{2n_G+1}, \ldots, c_m\}$ and each selected grid cell is labeled as proposition. A grid cell labeled as a proposition P^k_i defines a proposition function as follows:

$$\text{PROP}_{\pi_{i,k}}(x) = \texttt{true} \iff \text{POSITION}(x) \in P^k_i.$$

In this way, PROP$^k_{\pi_{i,k}}(x)$ is satisfied when the vehicle center reaches P^k_i.

User-defined graph $\mathcal{D}$ For the hybrid-system benchmark in this paper, $\mathcal{D}$ is defined as follows. A vertex is added to $\mathcal{D}$ for each grid cell. If a grid cell c is not labeled as a guard, then an edge is added to $\mathcal{D}$ from c to its left, right, up, and down neighboring cells. If a grid cell c is labeled as a guard $G^k_{i,j}$, then an edge is added from c to the cell c', where c' is the grid cell associated with the jump for $G^k_{i,j}$.

Second-order models of vehicle dynamics Vehicle break dynamics associated with each $q_i \in Q$ are selected uniformly at random from second-order models of cars, differential drives, and unicycles. Details of these models can be found in [15,37,38]. For completeness, these models are summarized below.

Car State $x = (p, \theta, v, \psi)$ consists of position $p \in \mathbb{R}^2$, orientation $\theta \in [-\pi, \pi)$, velocity ($|v| \leq 3\,\text{m/s}$), and steering angle ($|\psi| \leq 40°$). The distance between front and rear axles is $L = 0.8\,\text{m}$. Controls consist of the acceleration ($|u_0| \leq 0.8\,\text{m/s}^2$) and rotational velocity of the steering angle ($|u_1| \leq 25°/\text{s}$). Dynamics equations are $\dot{p}_0 = v\cos(\theta)$, $\dot{p}_1 = v\sin(\theta)$, $\dot{\theta} = v\tan(\psi)/L$, $\dot{v} = u_0$, $\dot{\psi} = u_1$.

Unicycle State $x = (p, \theta, v, \omega)$ consists of position $p \in \mathbb{R}^2$, orientation $\theta \in [-\pi, \pi)$, translational velocity ($|v| \leq 3\,\text{m/s}$), and rotational velocity ($|\omega| \leq 20°/\text{s}$). Controls consist of the translational ($|u_0| \leq 0.3\,\text{m/s}^2$) and rotational u_1 ($|u_1| \leq 10°/\text{s}^2$) accelerations. Dynamics equations are $\dot{p}_0 = v\cos(\theta)$, $\dot{p}_1 = v\sin(\theta)$, $\dot{\theta} = \omega$, $\dot{v} = u_0$, $\dot{\omega} = u_1$.

Differential drive State $x = (p, \theta, \omega_\ell, \omega_r)$ consists of position $p \in \mathbb{R}^2$, orientation $\theta \in [-\pi, \pi)$, and rotational velocities ($|\omega_\ell|, |\omega_r| \leq 5°/\text{s}$) of the left and right wheels. Wheel radius is $r = 0.2\,\text{m}$ and axis length is $L = 0.8\,\text{m}$. Controls consist of the left and right wheel rotational accelerations ($|u_0|, |u_1| \leq 10°/s^2$). Dynamics equations are $\dot{p}_0 = 0.5r(\omega_\ell + \omega_r)\cos(\theta)$, $\dot{p}_1 = 0.5r(\omega_\ell + \omega_r)\sin(\theta)$, $\dot{\theta} = r(\omega_r - \omega_\ell)/L$, $\dot{\omega}_\ell = u_0$, $\dot{\omega}_r = u_1$.

5.2 Syntactically safe LTL formulas

Syntactically safe LTL formulas were manually designed in order to provide meaningful properties. These formulas are defined over the $n_P = 150$ propositions generated for each benchmark instance. Let $\pi_1, \ldots, \pi_{150}$ denote a random permutation of the propositions PROP$_{\pi_{i,k}}$. Let β_0 denote a special proposition, which is true iff none of the propositions $\pi_1, \ldots, \pi_{150}$ is true. Note that β_0 geometrically corresponds to the grid cells not labeled as propositions, as illustrated in Fig. 1. The syntactically safe LTL formulas are defined as follows.

Sequencing ($n = 3, 4, 5, 6$): A witness trajectory ζ reaches $\pi_1, \ldots, \pi_n$ in that order via β_0. More specifically, ζ starts at β_0 and remains in β_0 until it reaches π_1; then remains in π_1 until it reaches β_0; then remains in β_0 until it reaches π_2; ...; then remains in π_{n-1} until it reaches β_0; then remains in β_0 until it reaches π_n. Formally, the formulas are as follows:

$$\phi^3_1 = \neg(\beta_0\mathcal{U}(\pi_1 \wedge (\pi_1\mathcal{U}(\beta_0\mathcal{U}(\pi_2 \wedge (\pi_2\mathcal{U}(\beta_0\mathcal{U}\pi_3)))))))$$
$$\phi^4_1 = \neg(\beta_0\mathcal{U}(\pi_1 \wedge (\pi_1\mathcal{U}(\beta_0\mathcal{U}(\pi_2 \wedge (\pi_2\mathcal{U}(\beta_0\mathcal{U} \\ (\pi_3 \wedge (\pi_3\mathcal{U}(\beta_0\mathcal{U}\pi_4)))))))))$$
$$\ldots$$
$$\phi^n_1 = \neg(\beta_0\mathcal{U}(\pi_1 \wedge (\pi_1\mathcal{U}(\beta_0\mathcal{U}(\ldots(\pi_{n-1}\wedge \\ (\pi_{n-1}\mathcal{U}(\beta_0\mathcal{U}\pi_n))))))))$$

Note that ζ is never allowed to reach any of the propositions $\pi_{n+1}, \ldots, \pi_{150}$. In this way, $\pi_{n+1}, \ldots, \pi_{150}$ serve as obstacles that must be avoided by ζ at all times.

Counting ($n = 1, 2, 3, 4$): A witness trajectory ζ starts at $\beta_0 \vee \pi_1$ and remains there until it reaches π_1. Then, ζ repeats the following n-times: remains in $\pi_1 \vee \beta_0$ until it reaches π_2;

[1] After a discrete transition, components of the continuous state except position and orientation are reset to zero to account for the fact that state components can vary from mode to mode, e.g., the car has translational velocity and steering angle in the continuous state, but differential-drive has rotational velocities of the wheels.

then remains in $\pi_2 \vee \beta_0$ until it reaches π_3; then remains in $\pi_3 \vee \beta_0$ until it reaches π_4; then remains in π_4 until it reaches $\pi_1 \vee \beta_0$. After repeating these steps n times, ζ remains in $\pi_1 \vee \beta_0$ until it reaches π_5. Formally, the formulas are as follows:

$$\phi_2^n = \neg(\varsigma_1 \mathcal{U}(\pi_1 \wedge \Xi_n)),$$

where $\varsigma_i = \beta_0 \vee \pi_i$;

$$f(\psi) = \varsigma_1 \mathcal{U}(\pi_2 \wedge (\varsigma_2 \mathcal{U}(\pi_3 \wedge (\varsigma_3 \mathcal{U}(\pi_4 \wedge (\pi_4 \mathcal{U}(\varsigma_1 \wedge \psi)))))))$$

(see note[2]); and

$$\Xi_1 = f(\varsigma_1 \mathcal{U} \pi_5), \Xi_2 = f(\Xi_1), \ldots, \Xi_n = f(\Xi_{n-1}).$$

Note that ζ is never allowed to reach any of the propositions $\pi_6, \ldots, \pi_{150}$, which, in this way, serve as obstacles that must be avoided at all times.

Coverage ($n = 4, 5, 6, 7$): A witness trajectory eventually reaches each $\pi_1, \ldots, \pi_n$, i.e.,

$$\phi_3^n = \neg\left(\bigwedge_{i=1}^{n} \mathcal{F}(\pi_i)\right).$$

The automata $\mathcal{A}$ for the complement of each safety LTL formula are computed by standard tools [42]. The table below shows the number of states of the minimized DFA.

LTL safety formula (nr. states minimized DFA)			
ϕ_1^3 (10)	ϕ_1^4 (21)	ϕ_1^5 (46)	ϕ_1^6 (105)
ϕ_2^1 (23)	ϕ_2^2 (76)	ϕ_2^3 (164)	ϕ_2^4 (287)
ϕ_3^4 (16)	ϕ_3^5 (32)	ϕ_3^6 (64)	ϕ_3^7 (128)

5.3 Results

Experiments were run on Rice Cray XD1 PBS and ADA clusters, where each processor runs at 2.2 GHz and has up to 8GB RAM. Each run uses a single processor and a single thread, i.e., no parallelism.

For the experiments, 100 benchmark instances were generated at random as described in Sect. 5.1. For each combination of a method M_i and an LTL safety formula ϕ, Fig. 2 reports the average time in seconds obtained by method M_i when attempting to compute a witness trajectory for ϕ for each of the 100 benchmark instances. Timeout for each run was set to 400 s. Computational times for `TemporalHyDICE` include the construction of the graph $\mathcal{D}$, which took < 1 s.

[2] The notation $f(\psi)$ is used for convenience as a function that takes the argument ψ and places it as indicated in the expression.

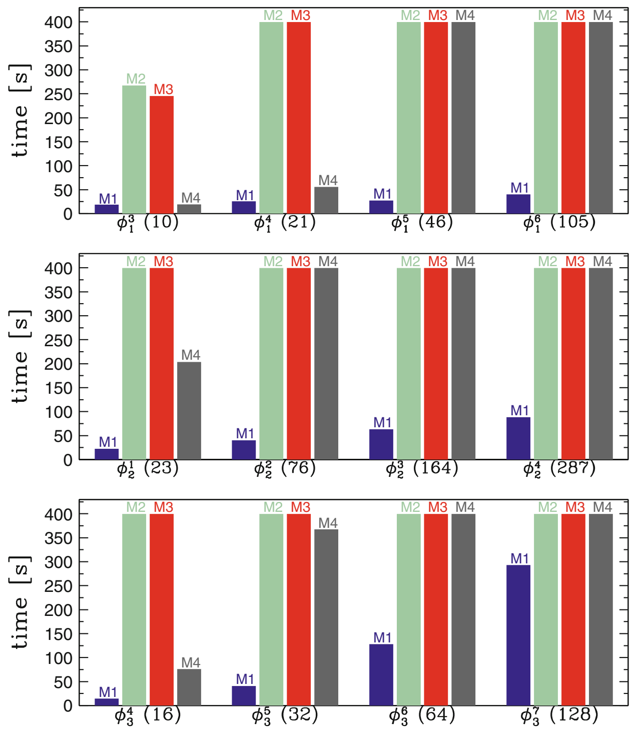

Fig. 2 Comparison of `TemporalHyDICE` (denoted M1), `RRT[LTL-TSF]` (denoted M2), `HyDICE[NoGuide, LTL-TSF]` (denoted M3), `TemporalHyDICE`[no $\mathcal{D}$] (denoted M4) when computing witness trajectories for various LTL safety properties of the hybrid-system model. Number inside parentheses after ϕ_i^n indicates number of states in minimized DFA. 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 $\mathcal{D}$, which took <1 s. Timeout was set to 400 s (color figure online)

5.3.1 *Computational efficiency of* `TemporalHyDICE`

Figure 2 shows that `TemporalHyDICE` is significantly more efficient than `RRT[LTL-TSF]` and `HyDICE[NoGuide LTL-TSF]`. Recall that `RRT[LTL-TSF]` and `HyDICE [NoGuide, LTL-TSF]` correspond to extensions of related work [10–16], which make it possible for the related work to handle LTL specifications (since related work in its original formulation could not handle LTL specifications). As described in Sect. 3, these extensions are obtained by using the safety automaton $\mathcal{A}$ as an external monitor.

The results in Fig. 2 indicate that `TemporalHyDICE` efficiently computed witness trajectories for all problem instances. Even for the most challenging problem in the experiments (instances where the LTL safety property is specified by ϕ_3^7), the average running time of `TemporalHyDICE` was less than five minutes. In the other cases, `TemporalHyDICE` was even faster. As an example, for the LTL safety formulas $\phi_1^3, \phi_1^4, \phi_1^5, \phi_1^6$, the average running times of `TemporalHyDICE` are 18.6s, 25.5s, 27.2s, and 40.4s, respectively.

As shown in Fig. 2, while TemporalHyDICE efficiently 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 [15,16] in the context of reachability analysis, it quickly becomes difficult for RRT-based methods to find feasible directions along which to extend $\mathcal{T}$, causing a rapid decline in the growth of $\mathcal{T}$. The results in Fig. 2 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 tends to increase sub-linearly (ϕ_1^n and ϕ_2^n) or sub-quadratically (ϕ_3^n) with the number of states in the minimized DFA.

5.3.2 *Impact of combining model checking and motion planning*

Comparisons in Fig. 2 between TemporalHyDICE and HyDICE[NoGuide, LTL-TSF] show the importance of combining model checking and motion planning. Recall that, as described in Sect. 3, HyDICE[NoGuide, LTL-TSF] denotes the extension of our earlier work [15,16] to use the automaton $\mathcal{A}$ as an external monitor so that it can be applied for the falsification of LTL safety properties. Without model checking to guide motion planning, however, HyDICE[NoGuide, LTL-TSF], similar to RRT[LTL-TSF], times out in almost all instances.

Comparisons in Fig. 2 between TemporalHyDICE and TemporalHyDICE[no $\mathcal{D}$] indicate the importance of computing discrete witnesses by searching both the graph $\mathcal{D}$ and the automaton $\mathcal{A}$ (as in TemporalHyDICE) and not just $\mathcal{A}$ (as in TemporalHyDICE[no $\mathcal{D}$]). 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., $\Gamma(\tau_i)$ is not directly connected to $\Gamma(\tau_{i+1})$. The graph $\mathcal{D}$ 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 $\mathcal{D}$ and $\mathcal{A}$, TemporalHyDICE[no $\mathcal{D}$] is still considerably faster than falsification methods that do not guide the tree search, such as RRT[LTL-TSF] and HyDICE[NoGuide, LTL-TSF].

5.3.3 *Impact of automata minimization and determinization*

Figure 3 compares TemporalHyDICE when using for the automaton $\mathcal{A}$ NFAs computed by standard tools [42], minimal NFAs constructed by hand, or minimal DFAs obtained by determinizing and minimizing the NFAs. These experiments are motivated by the work in [35], which shows significant speedup when using DFAs instead of NFAs in the context of model checking.

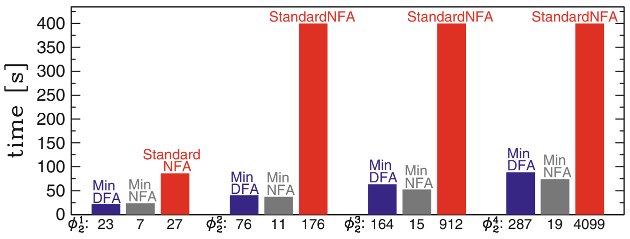

Fig. 3 Comparison of the computational efficiency of TemporalHyDICE when using a minimal DFA, a minimal NFA constructed by hand, or an NFA constructed by standard tools [42] for the safety properties specified by the various ϕ_2^n formulas. Reported is the average time in seconds to solve 100 problem instances for each of the LTL formulas. Timeout was set to 400 s (color figure online)

As Fig. 3 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 [35], DFAs offer computational advantages that can offset the drawbacks of a possibly exponential increase in the number of states. In particular, a DFA search has a significantly smaller branching factor, since there is exactly one transition that can be followed from each state for each propositional assignment, while when using an NFA there are generally many more. This observation is also supported by comparisons of minimal DFAs to standard NFAs in Fig. 3, 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 paper presented a novel method, TemporalHyDICE, for the falsification of safety properties specified by syntactically safe LTL formulas for hybrid systems with nonlinear dynamics and input controls. By effectively combining model checking and motion planning, when a hybrid system is unsafe, TemporalHyDICE computes a witness trajectory that indicates a violation of the safety property. Model checking in TemporalHyDICE computes discrete witnesses by searching on-the-fly an automaton $\mathcal{A}$ for the complement of the LTL safety formula and the user-defined abstraction of the hybrid system. Motion planning explores the state space of the hybrid system by extending a search tree, consisting of feasible hybrid-system trajectories, along the directions specified by the discrete witness. Information gathered during exploration is fed back to model checking to improve the discrete witnesses computed in future iterations. Experiments that test LTL safety properties on a robot navigation benchmark modeled as a hybrid system with nonlinear dynamics and input controls provide promising validation. Results show significant speedup over extensions of RRT-based falsification and demonstrate the importance of combining model checking and motion planning for the falsification of LTL safety properties.

TemporalHyDICE opens up several venues for future research. As we consider increasingly challenging problems, it becomes important to further improve the synergistic combination of model checking and motion planning. Moreover, the development of algorithms that make use of parallel or multi-threaded computational resources provides an important venue to significantly improve the computational efficiency of the framework. Another research direction is to extend the theory developed in [43] to show probabilistic completeness for TemporalHyDICE.

Acknowledgments This work is supported in part by NSF CNS 0615328 (EP, LK, MV), NSF CCF 1018798 (LK, MV), 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. IEEE **91**(7), 986–1001 (2003)
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. Theor. Comput. Sci. **138**(1), 3–34 (1995)
3. Henzinger, T., Kopke, P., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: ACM Symp on Theory of Computing, pp. 373–382 (1995)
4. Chutinan, C., Krogh, B.H.: Computational techniques for hybrid system verification. IEEE Trans. Autom. Control **48**(1), 64–75 (2003)
5. Mitchell, I.M.: Comparing forward and backward reachability as tools for safety analysis. In: Hybrid Systems: Computation and Control. LNCS, vol. 4416, pp. 428–443 (2007)
6. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.: Discrete abstractions of hybrid systems. Proc. IEEE **88**(7), 971–984 (2000)
7. 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. Int. J. Found. Comput. Sci. **14**(4), 583–604 (2003)
8. Giorgetti, N., Pappas, G.J., Bemporad, A.: Bounded model checking for hybrid dynamical systems. In: Conference on Decision and Control, Seville, Spain, pp. 672–677 (2005)
9. Branicky, M.: Universal computation and other capabilities of continuous and hybrid systems. Theor. Comput. Sci. **138**(1), 67–100 (1995)
10. Branicky, M.S., Curtiss, M.M., Levine, J., Morgan, S.: Sampling-based planning, control, and verification of hybrid systems. IEE Proc. Control Theory Appl. **153**(5), 575–590 (2006)
11. Bhatia, A., Frazzoli, E.: Incremental search methods for reachability analysis of continuous and hybrid systems. In: Hybrid Systems: Computation and Control. LNCS, vol. 2993, pp. 142–156 (2004)
12. Esposito, J.M., Kim, J., Kumar, V.: Adaptive RRTs for validating hybrid robotic control systems. In: Workshop on Algorithmic Foundations of Robotics, Zeist, Netherlands, pp. 107–132 (2004)
13. Kim, J., Esposito, J.M., Kumar, V.: An RRT-based algorithm for testing and validating multi-robot controllers. In: Robotics: Science and Systems, Boston, pp. 249–256 (2005)
14. Nahhal, T., Dang, T.: Coverage-guided test generation for continuous and hybrid systems. Formal Methods Syst. Des. **34**(2), 183–213 (2009)
15. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: from verification to falsification. In: International Conference on Computer Aided Verification. LNCS, vol. 4590, pp. 468–481 (2007)
16. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: from verification to falsification by combining motion planning and discrete search. Formal Methods Syst. Des. **34**(2), 157–182 (2009)
17. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.: Benefits of bounded model checking at an industrial setting. In: International Conference on Computer Aided Verification. LNCS, vol. 2102, pp. 436–453 (2001)
18. Cheng, P., Kumar, V.: Sampling-based falsification and verification of controllers for continuous dynamic systems. Int. J. Robot. Res. **27**(11–12), 1232–1245 (2008)
19. Bhatia, A., Frazzoli, E.: Sampling-based resolution-complete safety falsification of linear hybrid systems. In: IEEE Conference on Decision and Control, New Orleans, pp. 3405–3411 (2007)
20. Bhatia, A., Frazzoli, E.: Sampling-based resolution-complete algorithms for safety falsification of linear systems. In: Hybrid Systems: Computation and Control. LNCS, vol. 4981, pp. 606–609 (2008)
21. LaValle, S.M., Kuffner, J.J.: Randomized kinodynamic planning. Int. J. Robot. Res. **20**(5), 378–400 (2001)
22. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
23. Behrmann, G., David, A., Larsen, K.G., Mller, O., Pettersson, P., Yi, W.: Uppaal—present and future. In: Conference on Decision and Control, Orlando, Florida, pp. 2881–2886 (2001)
24. Fainekos, G.E., Girard, A., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for dynamic mobile robots. Automatica **45**(2), 343–352 (2009)
25. Fainekos, G.E., Kress-Gazit, H., Pappas, G.: Temporal logic motion planning for mobile robots. In: IEEE International Conference on Robotics and Automation, Barcelona, Spain, pp. 2020–2025 (2005)
26. Kloetzer, M., Belta, C.: A fully automated framework for control of linear systems from temporal logic specifications. IEEE Trans. Autom. Control **53**(1), 287–297 (2008)
27. Kress-Gazit, H., Fainekos, G., Pappas, G.J.: Temporal-logic-based reactive mission and motion planning. IEEE Trans. Robot. **25**(6), 1370–1381 (2009)
28. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Receding horizon temporal logic planning for dynamical systems. In: IEEE Conference on Decision and Control, Shanghai, China, pp. 5997–6004 (2009)
29. Kloetzer, M., Belta, C.: Temporal logic planning and control of robotic swarms by hierarchical abstractions. IEEE Trans. Robot. **23**(2), 320–331 (2007)
30. Batt, G., Belta, C., Weiss, R.: Temporal logic analysis of gene networks under parameter uncertainty. IEEE Trans. Autom. Control **53**, 215–229 (2008)
31. Damm, W., Pinto, G., Ratschan, S.: Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems. Int. J. Found. Comput. Sci. **18**(1), 63–86 (2007)
32. Kupferman, O., Vardi, M.: Model checking of safety properties. Formal Methods Syst. Des. **19**(3), 291–314 (2001)
33. Sistla, A.: Safety, liveness and fairness in temporal logic. Formal Aspects Comput. **6**, 495–511 (1994)

34. Fehnker, A., Ivancic, F.: Benchmarks for hybrid systems verification. In: Hybrid Systems: Computation and Control. LNCS, vol. 2993, pp. 326–341 (2004)
35. Armoni, R., Egorov, S., Fraer, R., Korchemny, D., Vardi, M.: Efficient LTL compilation for SAT-based model checking. In: International Conference on Computer-Aided Design, San Jose, pp. 877–884 (2005)
36. Alpern, B., Schneider, F.: Recognizing safety and liveness. Distrib. Comput. **2**, 117–126 (1987)
37. 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 (2005)
38. LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006)
39. Esposito, J., Kumar, V., Pappas, G.: Accurate event detection for simulation of hybrid systems. In: Hybrid Systems: Computation and Control. LNCS, pp. 204–217 (2001)
40. Julius, A.A., Fainekos, G.E., Anand, M., Lee, I., Pappas, G.J.: Robust test generation and coverage for hybrid systems. In: Hybrid Systems: Computation and Control. LNCS, vol. 4416, pp. 329–342 (2007)
41. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Discrete search leading continuous exploration for kinodynamic motion planning. In: Robotics: Science and Systems, Atlanta, Georgia, pp. 326–333 (2007)
42. Latvala, T.: Efficient model checking of safety properties. In Ball, T., Rajamani, S., (eds.) Model Checking Software. LNCS, vol. 2648, pp. 74–88 (2003)
43. Ladd, A.M.: Motion planning for physical simulation. PhD thesis, Rice University, Houston (2006)