Control design for hybrid systems with TuLiP: The temporal logic planning toolbox

Sumanth Dathathri¹, Ioannis Filippidis³, Scott C. Livingston², Richard M. Murray¹, Necmiye Ozay³

Abstract—This tutorial describes TuLiP, the Temporal Logic Planning toolbox. TuLiP is a collection of tools for designing controllers for hybrid systems from specifications in temporal logic. The tools support a workflow that starts from a description of desired behavior in temporal logic, and a description of the system to be controlled. The system can be represented as a discrete transition system, or a hybrid dynamical system with a mixed discrete and continuous state space. The system description can include both discrete and continuous uncontrollable variables that represent disturbances, communication signals, and other environmental factors that affect the system dynamics and controller decisions.

For solving the control design problem, the logic specification is refined, by conjoining it with a discrete description of system dynamics in logic, which is an abstraction of the underlying continuous dynamics, in the case of hybrid systems. For piecewise affine dynamical systems, this abstraction is constructed automatically, guided by the geometry of the dynamics and under logical constraints from the specification. The resulting logic formulae describe admissible discrete behaviors that capture both controlled and environment variables. To find a controller, the toolbox solves a game of infinite duration. Existence of a discrete (winning) strategy for the controlled variables in this game is a proof certificate for the existence of a controller for the original problem that guarantees the satisfaction of the specification. This discrete strategy, refined with continuous controllers when needed, yields a feedback controller for the original hybrid system. The toolbox frontend is written in Python, with backends in C, Python and other languages.

The tutorial starts with an overview of the theory behind TuLiP, and of its software architecture, organized into specification frontends and backends that implement algorithms for abstraction, solving games, and interfaces to other tools. Then, the main elements for writing a specification for input to TuLiP are introduced. These include logic formulae, labeled transition systems, and hybrid dynamical systems, with linear or piecewise affine continuous dynamics. The working principles of the algorithms for predicate abstraction and discrete game solving using nested fixpoints are explained, by following the input specification through the various transformations that compile it to a symbolic representation that scales well to large games. The tutorial concludes with several design examples that demonstrate the toolbox’s capabilities.

I. INTRODUCTION

Before we build a system, we write a description in a suitable language [1]. The language can range from mechanical drawings to matrices. There are two problems involved in writing such a description. First, a description that resembles closely the final implemented system may be difficult to find. Its existence may even be unknown. But, if it is so difficult to describe an implementation, how did we even think about it in the first place?

The disconnect between thinking and implementation arises because we tend to think declaratively, in that we quantify things in our statements. Quantifiers are a convenient means to express lazily what we want, without putting the effort to articulate it in fine detail. It is this disconnect that makes its easier to express the problems we want to solve first in a language that contains quantifiers. Control problems are typically expressed in this way, by asking whether there exists (and if so, to choose one) a controller that constrains sufficiently the system of interest (plant), so that any behavior of their combination be as we desire [2].

The second problem arises because it costs computation to remove the quantifiers from the initial description of a problem (specification). Depending on the type of problem, different methods can eliminate quantifiers with varying computational complexity, or not be able to eliminate them at all. Hybrid systems involve both continuous and discrete variables, and usually both differential and difference equations. The approach reviewed in this tutorial relies on separating the problem, so that different methods be applied to the continuous and discrete subproblems.

A continuous control problem originates from physical reality. Unlike pure software, there are limits to how far we can push physical constraints. Even if possible, demanding control applications can quickly become too expensive to be viable for our purposes. So, it is reasonable to treat the continuous dynamics as a hard constraint, and seek to eliminate quantifiers by designing the discrete, logical behavior of the system. In other words, there can be more freedom in taking decisions, than there is in modifying physical dynamics. In addition, tractable representations are known, and well developed, for discrete descriptions of nonlinear problems that involve both existential ( EXISTS ) and universal ( ALL ) quantifiers. Problems related to hybrid systems are typically nonlinear.

These observations motivate abstraction of continuous behaviors to discrete ones. Not any abstraction works, only those that preserve the properties we care about, those that we write in the starting specification. Preserving a property means that an abstract behavior can be safely refined to a concrete, continuous, behavior that implements the same behavioral objective. This necessity leads to methods for constructing property-preserving abstractions. In the presented framework, these take the form of partition refining algorithms that discretize the continuous flow of a piece-
wise affine dynamical system to a finite directed graph of possible transitions. The possibility of behavior refinement is established by checking for the existence of continuous controllers that decide control inputs in a receding horizon fashion. Universal quantification over continuous variables is present, in the form of external disturbances (noise).

A defining quality of a dynamical system is that it changes in time. Thus, behavior description and quantification are usually over time. We can write such a description by using a time variable $t$, as is customary in physics and control, and quantify over $t$. This is a simple and explicit approach, but, unfortunately, it leads to unreadable descriptions. If few or nobody can read a design, there is little purpose in writing it, because it won’t serve its purpose [3]. Temporal logic has proven useful for reasoning about (discrete) dynamical systems [4]. It trades off explicitness of temporal reference and quantification for readability. We use linear temporal logic to describe the behavior of discrete variables, and the discrete (combinatorial) structure that arises in the continuous problems (discontinuities, switching).

Games arise as a name for alternation, meaning the nesting of universal and existential quantifiers. Universal quantifiers represent the part of the world that we don’t design, nor control at the time of deployment. Existential quantifiers express the design candidates that we consider choosing among. Having hidden continuity within a discrete abstraction, the game thus obtained is discrete. Moreover, in engineering applications, typically this game represents the useful life of the product, which is of unspecified length. Thus, the resulting game is of infinite duration. We employ symbolic methods for solving games, in order to find a controlling strategy that steers the system so that any resulting behavior satisfy the temporal formula we wrote as specification. Symbolic algorithms rely on data structures that can remain small, even for problems that are large, as measured by the number of states relevant to reasoning about their behaviors. Solving a game means exactly eliminating quantifiers.

II. PROBLEM INSTANCES

As mentioned in Section I, in TuLiP, design problems are described using temporal logic, difference equations, and linear inequalities that describe constraints and geometry. All these elements are simply mathematical formulae. There are many ways to write mathematics within a computer. In TuLiP, a programmatic interface is used. Each aspect of a problem is expressed separately, corresponding to the subsections that follow.

We use some notation and terminology from TLA$^+$ [5], with some details omitted. In TLA$^+$, the notation $f[x]$ denotes the value of function $f$ at $x$. A function $f$ with domain $S$ is written as $f \in [x \in S \mapsto f[x]]$. The set of all functions with domain $S$ that take values in $T$ is written $[S \rightarrow T]$ [5, p.48]. Let $\mathbb{N}$ denote the set of natural numbers, and $\mathbb{B}$ the Boolean values TRUE, FALSE. Negation, conjunction, and disjunction are denoted with $\neg, \land, \lor$. The operator $a \equiv b$ denotes equality of Boolean values, and is FALSE if any of $a, b$ takes a non-Boolean value. A set of consecutive natural numbers is denoted $i,j \triangleq \{ k : (k \in \mathbb{N}) \land (i \leq k) \land (k \leq j) \}$

A function $f$ with domain $f = 0..j$, for some $j \in \mathbb{N}$, is a tuple, also written as $(f[0], f[1], \ldots, f[j])$.

A. Temporal logic

1) As a shorthand: When thinking about a system that changes in discrete time steps, its evolution in time can be modeled as a sequence, of infinite length. A sequence is a function $\sigma \in [\mathbb{N} \rightarrow D]$, from the natural numbers $\mathbb{N}$ to some set $D$.

Suppose that we want to mention in the design definition [1] some sequences that we are interested in, for example those $\sigma$ such that $x \in R$ all the time, for some desired set $R$. We can simply quantify over time to express the collection of these sequences as $\forall t \in \mathbb{N} : x[t] \in R$.

A slightly more interesting property we might want to specify is repeated visits to the set $R$ (perhaps to reset our system for using it the next day). Again, by quantifying time, we can write $\forall t \in \mathbb{N} : \exists r \in \mathbb{N}_{\geq 1} : x[r] \in R$.

Imagine writing similar formulae for hundreds of properties that we specify, and reasoning mathematically about them. This approach quickly becomes unreadable, thus unmanageable by humans. For this reason [6], Amir Pnueli introduced temporal logic [4], as a convenient shorthand that allows for simpler formulas, and more readable proofs of theorems about system properties.

A (linear) temporal logic (LTL) formula describes sequences of states, called behaviors. Each state is an assignment of values to all variables$^1$ A pair $(s_i, s_{i+1})$ of consecutive states within a behavior is called a step. Fig. 1. A formula that contains primed variables (for example $x'$) and defines a Boolean function over steps is called an action. Given some step, $(s_0, s_1)$ and an action, $(x = 1) \land (x' = 2)$, the unprimed letter $x$ denotes the value $s_0[x]$ of variable $x$ in state $s_0$, and the primed letter $x'$ denotes the value $s_1[x]$. So, priming a variable denotes its value in the “next” state.

An LTL formula can contain Boolean and temporal operators. In LTL, the previous statements that mentioned $\forall t$ can be written as $\square (x \in R)$ ($\square$ is the “always” operator), and $\lozenge (x \in R)$ ($\lozenge$ is the “eventually” operator).

Using a variety of temporal operators and nesting them leads to unreadable formulas that defy why we want to use LTL in the first place. For this reason, only a few operators are used, and in very specific ways. There is another important reason for restricting how temporal operators are to be used to write an LTL formula. Later, we will

$^1$A state assigns to all variables mentioned in a specification, as well as unmentioned variables. This definition simplifies composition of components that were developed starting with separate specification formulae.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig1.png}
\caption{A behavior is an infinite sequence of states, and a step is a pair of states $(s_i, s_{i+1})$.}
\end{figure}
synthesize controllers for some of the variables, in order to satisfy a given LTL formula. LTL formulae with arbitrarily nested operators are computationally very hard to synthesize (exponential time in number of relevant states).

2) State machines in temporal logic: The traditional way that a system is described in engineering and physics is by writing some difference equations, and an initial condition. What these equations and the initial condition define is a state machine. This way of writing a state machine can be expressed in temporal logic, as follows [5]. We define the initial condition $Init$ (a state predicate that mentions only unprimed variables), and an action $Next$ (that can mention primed variables) that describes what steps the state machine can take.

Perhaps the simplest example is a digital clock $c$ that starts at $Init \equiv (c = 0)$ and alternates between 1 and 0, $Next \equiv (c' = 1 - c)$. Assembling these into an LTL formula, we have $Init \land \neg Next$. Notice that a state machine description talks about how only variable $c$ changes in a behavior. It doesn’t say what other variables are doing. Recall that each state in a behavior assigns values to all variables, including those unmentioned by the formulae we write, so the unmentioned variables are free to change in arbitrary ways. This detail becomes important when composing systems.

Going one step further, we can apply assume-guarantee thinking to our state machines. Motivation for doing so is that our state machine doesn’t live in its own world. It usually depends on other state machines, and unless they behave as assumed, our state machine cannot behave as we want. This is formalized by defining two state machines, one for the rest of the world (the “environment”), and another for the system we are designing, by writing the formulas $Init_e$, $Next_e$ and $Init_s$, $Next_s$.

In addition, we usually want our state machines to exhibit some behavior over the future, which we cannot express using a step-by-step constraint $Next$. Usually, this is some liveness property, expressed with the formulae $\Box \diamond Recur$. So, we need to define the predicates $Recur_e$ and $Recur_s$, one for each of the (environment and system) state machines. To summarize informally, we require that the commitment/guarantee property

$$Init_s \land \Box Next_s \land \bigwedge_i \Box \diamond Recur_{s,i}$$

is not violated before the assumption/rely property

$$Init_e \land \Box Next_e \land \bigwedge_j \Box \diamond Recur_{e,j}$$

is. Thus, if the assumption holds throughout time, so should the commitment. We will formalize this notion of precedence in the next section.

The pair of an $\Diamond \Box$ with an $\Box \Diamond$ [10], [11] is called a Streett pair. Due to the disjunction $\lor_j$ and conjunction $\land_i$, it is called a generalized pair [12]. Generalized Reactivity(1) [11], [12], [9] is the name given to games (Section III-A) with one generalized Streett pair (Streett(1)) as control objective. GR(1) games can be solved in time polynomial in the number of relevant states, whereas GR(k) games ($k$ Streett pairs) have complexity factorial in $k$ [13], [14]. “Solving” refers to finding a controller for some variables that ensures satisfaction of a GR(k) formula, for any behavior of the uncontrolled variables.

Streett (Rabin) formulae have the form $\land \left( (\Diamond \Box \lor \Box \Diamond) \land (\Diamond e \lor \Box e) \right)$. Thus, they are the conjunctive (disjunctive) normal form (CNF) for temporal formulae expressing liveness, in analogy to the propositional CNF and DNF [15].

---

3) A tractable fragment: We can make the previous precise by writing it as follows²:

$$\varphi \equiv Init \Rightarrow \land \left( (\Box \diamond Next_e) \Rightarrow Next_s \right) \land \left( (\Box \diamond Next_e) \Rightarrow v \land j=0 \diamond \Box P_j \lor \land j=0 \Box \diamond R_i \right)$$

where $P_j \equiv \neg Recur_{e,j}$ and $R_i \equiv Recur_{s,i}$. This formula describes the interaction of two entities, usually called an environment and system. The action formula $Next_s$ describes steps that the system is allowed to take (transition relation), called $Next_s$-steps. For example, a step satisfies $Next_s \equiv (x = 5) \Rightarrow (y' = 2)$ if either variable $y$ takes the value 2 in the second state, or variable $x$ does not take the value 5 in the first state of the step.

The formula $(\Box \diamond Next_e) \Rightarrow Next_s$ requires that the system take a $Next_s$-step next, unless the environment has taken some $\neg Next_e$-step up to now. This expresses a requirement on system changes, under the assumption that the environment hasn’t deviated from $Next_e$. It also prevents the controller from deviating from $Next_s$ with the aim of forcing a later deviation from $Next_e$, a notion called strict realizability [8], [9]. So, $Next_e$ can contain contain constraints that arise from physical modeling and conventions about interface protocols. The “historically” operator $\Box$ is not used elsewhere, so we do not discuss it more.

The formula $\lor_{j=0}^n \diamond \Box P_j \lor \land_{i=0}^n \Box \diamond R_i$ describes a liveness goal, namely to either eventually satisfy some $P_j$ forever ($\diamond \Box$), or satisfy repeatedly ($\Box \diamond$) all the $R_i$. More details are given in Fig. 2.

B. State machines enumerated a little

We have been discussing about writing state machines as temporal formulas. In some cases, it can be convenient for

²In TLA*, the vertical arrangement of conjunction ($\land$) and disjunction ($\lor$) aids readability [7].
humans to use a data structure that involves both enumeration and formulas. In TuLiP, these are graphs with nodes and edges annotated with formulas. Formulas that describe assignments are quite common, so assignments of values to variables can be described also more directly, with key-value pairs. For uniformity, we will refer to annotations as formulas, which includes the previous as a special case.

This representation of a state machine is called in TuLiP a transition system. It should be noted that state machines go under many names and formalisms [16], so this is just a name for a particular data structure used here. In order to use symbolic algorithms as those described in Section III-B, it is convenient to write a temporal logic formula that describes the entire discrete problem. If part of the problem is defined using transition systems, then those are converted to formulas that appear as subformulas in the aggregate formula of the discrete problem.

It is helpful to think of state machines as predicate-action diagrams [17] that describe what some part of the world is doing, and leave unconstrained all unmentioned variables. This allows us to safely deduce properties that will continue to hold when we add components to the system. It is reflected by the conversion to temporal logic described below.

A transition system can be defined as a tuple $\langle S, E, L_s, L_e \rangle$ where $S$ is a set of states, $E \subseteq S \times S$ a set of edges, and $L_s \in [S \rightarrow \mathbb{B}]$ a function labeling states, and $L_e \in [S \times S \rightarrow \mathbb{B}]$ a function labeling edges. The functions $L_s, L_e$ can change with time.

Let us use Fig. 3 as an example to describe the conversion to an LTL formula. In this example $S \triangleq \{0,1,2\}$. A variable is selected to be used for signifying the current node. Here we chose variable $s$ for that purpose. The possible transitions between nodes are defined by the edges between them, so we can prime $s$ to express them with the action

$$\text{Next}(s, s') \triangleq (s = 0) \lor (s' = 1) \lor (s' = 2) \lor (s' = 0)$$

This means that when $s = 1$ is the current node, then the next node should be $s = 2$. If some nodes have been designated as initial, then $\text{Init}$ includes a corresponding constraint.

Shifting our attention to the annotation of nodes by formulae, those yield a state predicate\(^3\) of the form

$$\text{StateLabels} \triangleq \bigwedge_{j \in S} ((s = j) \Rightarrow L_s[j]),$$

where $L_s$ is defined so that $L_s[j] \equiv \varphi_j$. So, while at node $s = 0$, the value of $L_s[0]$ should be $\text{TRUE}$. The remaining labels are on edges, and these are handled similarly, with selection based on the edge traversed

$$\text{EdgeLabels} \triangleq \bigwedge_{i \in S} \bigwedge_{j \in S} ((s = i) \Rightarrow L_e[(i,j)]),$$

where $L_e$ is defined so that $L_e[(i,j)] \equiv \psi_{i,j}$. So, the transition from node $s = 0$ to $s = 1$ can occur in a step $\langle s, s' \rangle \equiv \text{i,j}$ that satisfies $L_e[(i,j)] \equiv \text{TRUE}$.

The discrete controllers generated by TuLiP are represented as state machines that output the next value $y'$ of discrete controlled variables, after reading as last input the value $z'$ of uncontrolled variables. The history of $z$ and $y$ is “remembered” in the form of the current node. This representation is shown in Fig. 4. If the controller is at node $s_0$, and it reads the input $z'_0$, then it will produce the output $y'_0$ and change its state to $s_1$. If the input had been $z'_1$, then it would produce the output $y'_1$ and change its state to $s_2$.

C. Hybrid dynamical systems

As for descriptions of continuous state systems, TuLiP accepts various types of discrete-time dynamical system models. The simplest model that TuLiP accepts is an affine time-invariant system model of the form:

$$x[t + 1] = Ax[t] + Bu[t] + Ed[t] + K,$$

where $x[t] \in \mathcal{X} \subset \mathbb{R}^n$ is the state, $u[t] \in \mathcal{U} \subset \mathbb{R}^m$ is the input, and $d[t] \in \mathcal{D} \subset \mathbb{R}^p$ is the disturbance at time $t$. The set $\mathcal{X}$ is called the domain of the system. The sets $\mathcal{U}, \mathcal{D}$ are represented as polytopes or unions of polytopes. A polytope $\mathcal{P} \subset \mathbb{R}^n$ is a set defined by linear predicates, that is, $\mathcal{P} = \{ x \mid Hx \leq h \}$. It is also possible to define state dependent input bounds by choosing $\mathcal{U}$ to be a polytopic subset of $\mathbb{R}^{n+m}$.

A more complicated model class is piecewise affine dynamical system models:

$$x[t + 1] = A_i x[t] + B_i u[t] + E_i d[t] + K_i,$$

\(^3\)Assuming that all formulas that annotate nodes contain only unprimed variables.
If \( x[t] \in X_i \), where \( x[t] \in X_i \subset \mathbb{R}^n \) is the state, \( u[t] \in U_i \subset \mathbb{R}^m \) is the input, and \( d[t] \in D_i \subset \mathbb{R}^p \) is the disturbance at time \( t \). The polytopic sets \( X_i \) for \( i = 1, \ldots, k \) form a partition of the domain \( \mathcal{X} \). In order to represent piecewise affine systems, we use a collection of affine systems of the form Eq. (2) with disjoint domains.

It is also possible to describe switched system models with controllable and uncontrollable switches:

\[
 x[t + 1] = A[s[t]]x[t] + B[s[t]]u[t] + E[s[t]]d[t] + K[s[t]],
\]

(4)

where the mode \( s[t] = \langle r[t], e[t] \rangle \) with \( r[t] \in \{1, \ldots, n_r\} \) and \( e[t] \in \{1, \ldots, n_e\} \), being the discrete controllable and uncontrollable inputs that determine the system matrices. Finally, one can also define switched piecewise affine systems where within each mode \( s \), the corresponding system is piecewise affine.

In order to specify properties of continuous-state systems, we introduce continuous propositions that are identified with subsets of the domain \( \mathcal{X} \). Let \( \{X_i\}_{i=1}^k \), \( X_i \subset \mathcal{X} \) be a collection of subsets of interest. For computational tractability, we assume each \( X_i \) is a polytope. Moreover, we assign “names”, \( a_i \equiv (x \in X_i) \), for each of these sets to be used in the LTL formulas. The finite sets of controllable and uncontrollable discrete inputs of switched systems can also be used as atomic propositions in an LTL formula if one wishes to impose assumptions or requirements about how these discrete inputs evolve.

D. Problem instances

Two problems can be solved with TuLiP. The first problem is solving discrete synthesis problems to construct controllers from LTL specifications in the GR(1) fragment. The second problem is to compute controllers for hybrid systems from specifications of piecewise affine dynamics, continuous propositions, and temporal logic specifications. These two problems are defined as follows.

**Problem 1:** [Discrete synthesis] Let \( z (y) \) be variables that represent the environment (system) controls and take discrete values. Let \( \varphi \) be a GR(1) specification defined by the assumption

\[
 \text{Init}_a(z, y) \land \square \text{Next}_a(z, y, z') \land \bigwedge_{j \in [0..N]} \square \diamond R_{e,j}(z, y)
\]

and the guarantee

\[
 \text{Init}_s(z, y) \land \square \text{Next}_s(z, y, z', y') \land \bigwedge_{i \in [0..M]} \square \diamond R_{s,i}(z, y).
\]

Let \( m \) be a memory variable with \( m \in M \) a finite set. The discrete synthesis problem is that of finding a controller function \( f \) for the next value \( y' \) of the controlled variables, such that \( \square (y' = f(z, y, z', m)) \Rightarrow \varphi \) and a memory update function \( f_m \), such that \( \square (m' = f_m((z, y, z', m))) \Rightarrow \square (m \in M) \).

Note that in order to define a discrete synthesis problem in TuLiP, one can express both the system behavior and the desired system behavior using a GR(1) formula and TuLiP internally combines them to form an instance of Problem 1.

**Problem 2:** [Synthesis for hybrid systems] Given continuous dynamics \( x[t + 1] = Ax[t] + Bu[t] + Ed[t] + K \) and sets \( \mathcal{X}, D, U \), polytopes \( \{X_i\}_{i=1}^k \) as continuous propositions, and a GR(1) formula \( \varphi \) over \( z, y \) and \( \{X_i\}_{i=1}^k \), the continuous synthesis problem is to find a continuous controller for \( u \) as a function of the continuous state \( x \) and discrete environment variables \( z \), and a discrete controller for \( y' \) as a function of \( z, y, z' \), \( \{X_i\}_{i=1}^k \).

The continuous problem can be solved by constructing and solving a suitable discrete problem first, and then using its solution to solve the hybrid system. One can pose a similar hybrid systems synthesis problem for other classes mentioned in Section II-C.

III. WORKING PRINCIPLES

A. Solving games

Any design problem can be summarized as “We want to construct a system \( f \), such that for every reasonable behavior of the world...”. In mathematics, this can be expressed as

\[
 \text{Pick } f : \forall x : \varphi(f, x),
\]

which can be solved only if the formula \( \exists f : \forall x : \varphi(f, x) \) is true. This is the Skolemized form of the synthesis problem [19], and is similar with how control problems are usually phrased by engineers.

The nesting of different quantifiers \( \exists \) constitutes alternation, imagined as a game [20]. The alternation can be equivalently written in a step-by-step form, where \( f \) actsuates on the system, then the dynamics \( x \) evolves, and this repeats forever. This second description is more amenable to computation, and leads to algorithms that solve games and construct winning strategies, as we discuss below.

The example shown in Fig. 5 is a game where we want to find how to move, in order to successfully reach the goal \( \text{Goal} \). At each disk, we choose the next node, and the environment \( \langle \rangle \) chooses from boxes. Lets first find from where we can win. Precisely, from which nodes we can end in \( \text{Goal} \), after taking 0 or more steps? Clearly, any nodes inside \( \text{Goal} \) are winning. Also, all nodes from where we can reach \( \text{Goal} \) within one step are winning too. But, when can we take a step to reach \( \text{Goal} \) from a node?

Suppose we are at node 1. We can step across the edge \( \langle 1, 4 \rangle \) to reach \( \text{Goal} \), so 1 is such a node. At node 2, the environment moves, but any edge it chooses to traverse leads to \( G \). So, there is a way to ensure that, from nodes 1 and 2, any possible behavior will reach \( G \) in 1 step. Unlike nodes 1 and 2, from node 3 the environment moves, and it can avoid \( \text{Goal} \) by remaining at node 3. The reasoning we just described can be written as

\[
 \text{CPre}(u) \equiv \bigvee \text{IsADisk}(u) \\
\land \exists v : v \in G \land \text{IsAnEdge}(u, v) \\
\lor \bigvee \text{IsABox}(u) \\
\land \neg (\exists v : v \notin G \land \text{IsAnEdge}(u, v))
\]

4Pick requires that we find an \( f \) that has the property \( \forall x : \varphi(f, x) \) [18]. Note that Pick should not be confused with Choose [5], Hilbert’s \( \epsilon \) operator, because the latter does not require that the property be satisfiable.
This is called the controllable predecessor operation, and the game just described a reachability game [21], [10]. By iterating the equation \( X_{k+1} = CPre(X_k) \lor X_k \lor G \) starting with \( X_0 = \text{false} \), we can compute all the states that can reach \( G \) in \( N \geq 0 \) steps, as the fixpoint \( X_{k+1} = X_k \) obtained for some finite \( k \). The result is a least fixpoint, denoted by \( \mu X : CPre(X) \lor G \).

Dually, we might want to solve a safety game, finding the states from where we can remain forever within \( G \). This is obtained by iterating the equation \( X_{k+1} = CPre(k) \land G \), starting with \( X_0 = \text{true} \). The resulting greatest fixpoint is written \( \nu X : CPre(k) \land G \). The solution of a GR(1) game is obtained by finding the solution to a triply nested fixpoint, where the innermost and outermost iterations compute greatest fixpoints, and the middle iteration a least fixpoint.

**B. Symbolic algorithms with binary decision diagrams**

a) Symbolic vs enumerative methods: From Section III-A, we see that solving games involves reasoning about states. This requires a representation of states. A simple and “tangible” representation is to enumerate each state within computer memory. For example, store 2 floating-point numbers for the coordinates of a two-dimensional dynamical system, while integrating its trajectory. In general, the number of states is exponential in the number of variables. So, enumerated representations become impractical.

In model checking, enumeration can still be viable, and sometimes behave better than symbolic methods, because there is only one kind of quantification, typically universal quantification for proving validity. For problems with uniform quantification, if the quantification is universal, then we can always negate them to solve a problem with existential quantification.

Unlike model checking, game solving involves alternating quantification. Therefore, negation doesn’t eliminate universal quantification any more. Enumeration in the presence of universal quantification can lead to problems, because of its exhaustive character.

b) Binary decision diagrams: A symbolic data structure represents in memory sets of states, not individual states. This enables symbolic methods to scale well, though the worst-case complexity remains exponential.

The algorithms that we describe here use reduced ordered binary decision diagrams (ROBDDs) [22] as symbolic representation of state sets. For brevity, we will introduce ROBDDs with a small example, shown in Fig. 6.

Suppose that we want to represent some subset of the set \( D = \{0, 1\} \times \{0, 1\} \) with an ROBDD. We will use the variables \( x, y \) to name elements in this set, so that \( (x, y) \in D \).

An ROBDD is a directed acyclic graph, with nodes arranged in layers called levels. The levels are indexed and ordered, with level 0 at the top and level \( n \) at the bottom. Each level above \( n \) is associated to a variable. In this example, \( n = 2 \) and level 0 corresponds to variable \( x \), level 1 to \( y \). The bottom node represents the values “true” and “false”. Each other node \( u \) has exactly two successors: \( v \) (low, dashed edge) and \( w \) (high, solid edge). The node named “x-4” is node \( u = 4 \), at level 0. Level 0 is associated to variable \( x \), so to node 4 too.

A node at level \( j \) describes a set of assignments to the variables of levels \( j \). For example, node 4 describes the assignments \( \{x, y\} \in \{\{0, 1\}, \{1, 0\}, \{1, 1\}\} = \{p \in D : p[0] \lor p[1] \} \). Given the assignment \( (x, y) = (0, 1) \), we decide whether it belongs to the set represented by node 4 as follows. Start at the mentioned node, 4. If \( x \) is true, follow the “low” successor (dashed edge to 3), otherwise the “high” successor (solid edge to 1). If the edge is marked with “-1” (“complemented”), then negate the final answer. Edges point only from higher to lower levels. In the assignment \( (0, 1) \), \( x \) is 0, so we follow the dashed edge to node 3. Variable \( y \) is 1, so we follow the solid edge from node 3 to node 1, obtaining the answer “true”. So, the assignment \( (x, y) = (0, 1) \) is in the ROBDD with node 4 as root. In contrast, \( (x, y) = (0, 0) \) is not, because it leads to node 1 via the complemented edge (dashed) from node 3 to 1. So, the answer is “false” (negated “true”).

BDDs are not a panacea, they, too, suffer from exponential worst-case complexity. Nevertheless, BDDs do enable solving very large problems that would otherwise be unmanageable. The size (number of nodes) of BDDs is what dominates the efficiency of a symbolic approach. Given \( m \) nodes, there are \( m^2 \) directed edges possible, so at most \( 2^m \) different directed graphs we can construct. Given \( n \) variables assigned values from \( \{0, 1\} \), there are \( 2^n \) possible

![Fig. 5: The meaning of controllable predecessors.](image1)

![Fig. 6: x\lor y.](image2)

![Fig. 7: A good variable order places selectors above data.](image3)
sets of assignments. Thus, some set of assignments can be represented only with a BDD of size \( m = O(2^n) \). This means that there are sets that have exponential size, whatever variable order we choose. In fact, this is the case for almost all sets [23, Thm.7]. Even worse, multiplication (thus division and modulo too) is an operation that cannot be represented efficiently with BDDs [24].

In addition, the same set can be represented with different variable (level) orders. By reordering the levels, we can represent some sets with exponentially smaller BDDs. To understand what effect the variable order has, it helps to think of control (or selector) and data variables [25, 1, Sec.7.2.3, 7.4.1, 7.5.3]. A simple example is a bit array \( a[k] \) with 8 elements. To represent any bit array of this size, we need 3 bits \( k_0, k_1, k_2 \) to encode \( k \) and 8 bits \( a_0...a_7 \) for the element values. In this example, the index bits \( k_i \) are control variables, and the element values are data variables. If \( k_i \) appear above \( a_i \) in the variable order, then the BDD has a small size, as shown in Fig. 7a. Reversing this order causes a blowup in size, as in Fig. 7b. The reason is that, as we go from top to bottom, if we first encounter \( k_i \), then it “tells” us which \( a_i \) bit to read next. In this sense, index \( k \) “selects” the data \( a_i \) to read. In contrast, if we encounter \( a_i \) first, then we have to read all\( a_i \) the array, so that when we finally reach \( k \), we can recall the appropriate value of bit \( a_k \).

Operations on BDDs rely on memoization to avoid recomputing a result that is encountered multiple times during traversal [26]. A BDD manager takes care of keeping reference counts for each BDD node, collecting garbage that is not referenced any more, and invoking reordering of levels based on some suitable heuristic. A reordering heuristic that works well is Rudell’s sifting [27]. In practice, a BDD package operates on BDDs with thousands to millions of nodes within seconds to minutes.

TuLiP interfaces to both a Python and a C implementation of BDDs. The package \texttt{dd} [28], [29] contains a pure Python BDD implementation, as well as Cython [30] bindings to the C library \texttt{CUDD} [31], with the same API for both. This allows developing an algorithm in Python, where debugging is simpler and modifications faster, and choosing whether to compile \texttt{CUDD} only at deployment, without changing the implemented algorithm. This makes it much easier for students to install the Python packages on their own computers in the context of an introductory course. The performance difference of implementing user code in Python calling Cython bindings, instead of C, is negligible, because realistic problems runtime is dominated by the BDD operations. In addition, the synthesizer \texttt{GR1C} [32] provides a pure C implementation that uses \texttt{CUDD} directly.

### C. Abstraction and low-level controllers

The main synthesis routine in TuLiP solves Problem 1 or its variants with finite transition systems as inputs. In order to synthesize controllers for a hybrid system using TuLiP, we construct a finite transition system representation of the hybrid system. We call this process abstraction. There are several abstraction techniques developed in the past decade [33], [34], [35], [36] and some of them are being integrated in TuLiP. Next, we briefly explain the main abstraction algorithm implemented in TuLiP, proposed in [37], [38] and how the transitions in the abstract transition system can be implemented by a low-level controller that picks the inputs of the hybrid system.

For the abstraction to be useful, it needs to preserve certain properties. Recall that we have identified a collection of continuous propositions associated with the subsets \( \{X_i\}_{i=1}^n \) of interest in the domain \( X \) of the hybrid system. The properties we want the abstraction to preserve are those related to these propositions.

Let us introduce a labeling function \( L_X \in [X \to \mathbb{B}] \) for the hybrid system such that \( L_X \triangleq [x \in X \mapsto \bigwedge (x \in X_i)] \). Let \( a_i \triangleq (x \in X_i) \) be “names” associated with these propositions.

The abstraction algorithm aims to find an abstraction function \( \alpha \in [X \to \bar{S}] \) and a set \( E \subset S \times S \) of transitions such that

\[
\begin{align*}
(i) & \quad \alpha \text{ is proposition preserving, that is, for any } s \in S \text{ and for any } x \in \alpha^{-1}[s], \quad L_s[s] = L_X[x]; \\
(ii) & \quad (s, s') \in E \text{ only if for all } x_1 \in \alpha^{-1}[s] \text{ and for all } d_1, \ldots, d_N \in D^N, \exists u, \ldots, u_N \in U^N \text{ such that } x_{N+1} \in \alpha^{-1}[s'], \text{ and } x_i \in \alpha^{-1}[u] \text{ for } i = 1, \ldots, N.
\end{align*}
\]

Here, the horizon length \( N \) is a user-defined parameter that synchronizes the steps the abstract transition system takes with \( N \) discrete time steps the hybrid system takes.

Roughly speaking, the abstraction function partitions the state space, where each part is associated with a discrete state of the transition system. And, the transitions of the transition system are constructed in a way that for each transition, there is a continuous control input sequence that can mimic that transition on the hybrid system. The continuous control input sequences can be computed as needed at run-time by solving finite time constrained reachability problems [37], [39]. A TuLiP controller for a hybrid system has a two-layered hierarchical structure. We call the software module computing continuous control inputs by solving the reachability problem the continuous controller or the low-level controller. Whereas, the software module resulting from the game solving is called the discrete controller or the high-level controller.

Some variants of the condition (ii) are also implemented in TuLiP. The version defined above is called open-loop as the control inputs \( u_1, \ldots, u_{i+N-1} \in U^N \) are computed once \( x_i \) is observed. There is also a closed-loop variant, where, we require:

\[
(ii)^* \quad (s, s') \in E \text{ only if for all } x_1 \in \alpha^{-1}[s] \text{ and for all } d_1, \text{ there exists } u_1 \in U, \ldots, d_N, \text{ there exists } u_N \in U \text{ such that } x_{N+1} \in \alpha^{-1}[s'], \text{ and } x_i \in \alpha^{-1}[u] \text{ for } i = 1, \ldots, N.
\]

Here, the continuous controller is allowed to measure the state \( x \) at each discrete time step as opposed to the open-loop case where the state is measured only every other \( N \) discrete time steps [40]. Other variants of condition
(ii) include relaxing the synchronization assumption and allowing transitions from one part to the other to take up to \( N \) steps instead of exactly \( N \) steps.

Finally, let us allude to the computation of the abstraction function (or equivalently, the partition it induces) and the transitions. The main idea behind the computation of the partition is the bisimulation algorithm [41], [33]. This algorithm starts with the coarsest possible partition that satisfies (i), and incrementally creates new parts by splitting the existing parts based on reachability relations. A part \( P_i \) in the partition is said to be reachable from another part \( P_j \) if for all \( x_1 \in P_j \), there is a control input sequence as in condition (ii) (or its variants) such that \( x_N \in P_i \). Let \( P_i \) be reachable from a subset \( P'_i \) of \( P_j \). Then, the part \( P_j \) is split into two parts: \( P'_j \) and \( P_j \setminus P'_j \). The algorithm stops if no more parts can be split or the parts in the partition become sufficiently small [38]. The transitions in the end are also inferred from the reachability relations. Whether a part \( P_i \) is reachable from another part \( P_j \) can be verified using polytopic operations such as lifting and projection [39].

IV. EXAMPLES

A. Simple autonomous vehicle

This section describes a simple example where a controller is synthesized using TuLiP for a representative hybrid system with piecewise affine dynamics. This demonstration is loosely inspired by the problem of logic-planning based on faults in the sensor system for ‘Alice’, the autonomous vehicle developed by Caltech for the 2004–2007 DARPA Grand Challenge [42]. The vehicle has an operating range of \( \mathcal{X} = [0, 20] \) miles/hr. The vehicle is assumed to have three driving modes - Slow/Stop, Moderate and Fast. Slow/Stop corresponds to the speed range 0-10 miles/hr, Moderate corresponds to 10-15 miles/hr and Fast to 15-20 miles/hr.

The vehicle is modeled as having the simple piecewise-affine discrete dynamics of:

- **Slow/Stop**: \( x[t+1] = a_1 x[t] + b_1 u[t] + c_1 d[t], \)
- **Moderate**: \( x[t+1] = a_2 x[t] + b_2 u[t] + c_2 d[t], \)
- **Fast**: \( x[t+1] = a_3 x[t] + b_3 u[t] + c_3 d[t], \)

where for each time \( t \), \( x[t] \) is assumed to take a value in \( \mathcal{X} \), i.e., \( x[t] \in \mathcal{X} \). Similarly, the control input \( u[t] \in \mathcal{U} \) and disturbance \( d[t] \in \mathcal{D} \). Both \( \mathcal{U} \) and \( \mathcal{D} \) are bounded real intervals. The vehicle has multiple sensors (Lidar and Stereo). It assumed that a healthy Stereo camera can make accurate short-range measurements while a functional Lidar can make accurate long-range measurements. Based on the sensor health, the vehicle is required to determine the control action for the vehicle to transition to a drive-mode (appropriate speed) that is safe with regard to the current sensor health. The vehicle has a sensor that measures the ambient lighting and provides feedback on whether the vehicle lights should be turned on or off. The synthesized controller also has to ensure this requirement.

System Specifications:

<table>
<thead>
<tr>
<th>Specification</th>
<th>LTL formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>( \neg \text{lidon} \land \neg \text{steron} )</td>
</tr>
<tr>
<td>d</td>
<td>((\text{fast} \land (\text{steron} \land \neg \text{lidon})) \Rightarrow \bigcirc (\text{fast}))</td>
</tr>
<tr>
<td>e</td>
<td>((\text{slow} \land (\neg \text{steron} \land \neg \text{lidon})) \Rightarrow \bigcirc (\text{slow}))</td>
</tr>
<tr>
<td></td>
<td>((\text{moderate} \land (\text{steron} \land \neg \text{lidon})) \Rightarrow \bigcirc (\text{moderate}))</td>
</tr>
<tr>
<td>g</td>
<td>((\text{lidon} \land \text{steron}) \Rightarrow \bigcirc (\text{fast} \land \neg (\text{lidon} \land \text{steron})))</td>
</tr>
<tr>
<td>h</td>
<td>(\Box (\neg \text{steron} \Rightarrow \bigcirc (\text{init} \land \neg \text{steron})))</td>
</tr>
<tr>
<td>i</td>
<td>(\Box \neg \text{lidon} \land \Box \text{steron})</td>
</tr>
<tr>
<td>j</td>
<td>(\Box \neg \text{lidon} \land \Box \text{steron})</td>
</tr>
<tr>
<td>k</td>
<td>\text{dark} \equiv \text{lights}</td>
</tr>
</tbody>
</table>

**TABLE I: Table of translation of the natural language specifications into LTL**

- a) Continuously valued controls are from \( \mathcal{U} = [-2, 2] \)
- b) Disturbances are from \( \mathcal{D} = [-0.01, 0.01] \)
- c) The vehicle is initially at the Stop/Reboot state
- d) Initially all sensors are off (or are not functional)
- e) The vehicle does not changes driving modes unless necessary
- g) If both long-range and short-range sensing are healthy, the vehicle must drive fast unless it loses sensing
- h) If short-range sensing is not healthy, the vehicle must eventually stop until the sensor comes back on
- i) The vehicle must always eventually drive in moderate/fast modes
- j) Both sensors are always eventually functional
- k) If and only if it is dark, the lights must go on

The inputs that are supplied to TuLiP for solving the synthesis problem are the system dynamics, the above specifications expressed as an LTL formula and the bounds on control inputs and disturbances. The synthesized controller controls the speed of the vehicle and if the lights for the vehicle are on or off. An observation to make is that if specification ‘j’ is not satisfied by the environment, the system cannot satisfy specification ‘i’ because it is not safe to drive when both sensors are faulty (specification ‘h’). The set of specifications can be divided into environmental assumptions and system requirements. For writing the LTL formula, specifications d, j form the environmental assumption part of the formula and specifications e,g,h and i describe the required system behavior.

1) LTL Specification: Let \( \text{init} \) be a Boolean proposition that holds true if the velocity is in the Slow/Stop mode. Similarly define \( \text{moderate} \) and \( \text{fast} \) for the Moderate and Fast drive modes. \( \text{steron} \) and \( \text{lidon} \) be propositions that evaluate to true if the Stereo and Lidar are functional respectively, and false otherwise. Table I translates the specifications in the list above into LTL.

2) Translation to GR(1): TuLiP accepts these specifications as liveness and transition (safety) rules for the environment and the system as described in section [Add section ref]. Specifications ‘g’ and ‘h’ do not directly fit into the GR(1) syntax, but can be manipulated by the introduction of an auxiliary variable. Consider specification ‘g’, an equivalent specification can be obtained by introducing an auxiliary
variable *aux* that fits into the GR(1) class. The variable
*aux* is initialized to *True* and the transition rule governing
it is \((\bigcirc (\text{aux}) \equiv (\text{fast} \lor (\neg (\text{lidon} \land \text{steron})))) \lor (\text{aux} \land
\neg (\text{lidon} \land \text{steron}))\), in conjunction with \(\bigcirc \text{aux}\). The first time
*lidon* \land *steron* evaluates to true, if *fast* evaluates to *False*,
*aux* is set to *False* and can set to *True* only if *fast* turns to
true or *lidon* \land *steron* turns to *False*. The specification that
was not directly in GR(1) was thus transformed to fit into
the GR(1) syntax as a combination of a liveness specification
and a transition rule. Similarly, ‘h’ can also be transformed
through the introduction of an auxiliary variable. These
specifications are then input into TuLiP as initial, progress
and safety conditions for the system and the environment.

3) Synthesis with TuLiP: The system dynamics, bounds
on the control-input and the disturbance are specified as
inputs to TuLiP. The polytopes are labelled and then the
continuous part of the state-space is first abstracted into a
finite-state transition system as described in Section II. The
synthesis happens in grlc with the abstracted transition
system and the specification as inputs.

B. GridWorld with moving obstacle

The task of planar robot motion planning in the environ-
ment shown (See Figure 8). The robot begins at a specified
location and has to visit the two goal regions infinitely often
and avoid collision with the walls. The robot in one time
step can transition to any of it’s non-diagonally adjacent
cells in the gridworld. The black cells indicate the walls,
cells marked ‘G’ indicate the goal-cells that must be visited
infinitely often and ‘I’ marks the starting pose of the robot.
The moving obstacle moves with similar rules, but begins
at a different initial location and has two different goals to
visit infinitely often. The robot should visit it’s goals and
avoid collision with the walls. The robot in one time
step can transition to any of it’s non-diagonally adjacent
cells marked ‘G’ indicate the goal-cells that must be visited
infinitely often. The robot should visit it’s goals and
avoid collision with the walls. The robot in one time
step can transition to any of it’s non-diagonally adjacent
cells in the grid. Transition rules with regard to non-collision for the
adversary with the wall are specified as

\[ \bigwedge_{(i,j) \in Q} \neg (X_r = i \land X_c = j) \]

and forms a part of the environmental assumption. Similarly,
rules prohibiting collision for the robot with the wall are also
specified.

Let \( G \) be the set of progress cells for the environment
that it must visit infinitely often. The progress goals for the
environment’s behavior are encoded as

\[ \bigwedge_{(i,j) \in G} \Box (X_r = i \land X_c = j) \]

In a similar manner, the progress goals corresponding to the
system are also specified in LTL. As an additional set
of requirements for the system, we add non-collision with
respect to the moving obstacle i.e the system at any instant
of the time must not occupy the same cell as the adversary.
This is done in LTL as

\[ \bigwedge_{i=0}^{i=4} \bigwedge_{j=0}^{j=4} \Box ((X_r = i \land X_c = j) \Rightarrow \neg ((Y_r = i) \land (Y_c = j))) \]
This specification directly fits into the GR(1) class directly and synthesis can be done in TuLiP using a solver, like grlc which accepts the GR(1) specification as the input.

V. CONCLUSIONS

This tutorial presented the control design approaches that are possible by using the tools in the tulip toolbox [43], [39], [32].

Acknowledgments: This work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA.

REFERENCES