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

Ioannis Filippidis[1], Sumanth Dathathri[1], Scott C. Livingston[2], Necmiye Ozay[3], Richard M. Murray[1]

***Abstract*— This tutorial describes TuLiP, the Temporal Logic Planning toolbox, 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, and of the system to be controlled. The system can have discrete state, or be a hybrid dynamical system with a mixed discrete and continuous state space. The desired behavior can be represented with temporal logic and discrete transition systems. The system description can include uncontrollable variables that take discrete or continuous values, and represent disturbances and other environmental factors that affect the dynamics, as well as communication signals that affect controller decisions.**

**A control design problem is solved in phases that involve abstraction, discrete synthesis, and continuous feedback control. Abstraction yields a discrete description of system dynamics in logic. 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. The discrete description resulting from abstraction is then conjoined with the desired logic specification. 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, which guarantees satisfaction of the specification. This discrete strategy, concretized by using continuous controllers, yields a feedback controller for the original hybrid system. The toolbox frontend is written in Python, with backends in C, Python, and Cython.**

**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, discrete transition systems annotated with predicates, 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 solving 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. It can be difficult, or even impossible, to start by writing a description of the final implemented system. Many times, we can still write a description of what we want the implementation to do, but with fewer details. Both this coarser description and the implementation can usually be written in the same language, for example formulae in some suitable logic. If it is so difficult to describe an implementation, how did we *think* about it at a coarser level? What is the difference between the coarse level (formula) that we have in our minds at an early stage of development, and the implementation level formula?

The difference between thinking and implementation arises because we tend to think *declaratively*, in that we quantify things in our statements. We use quantifiers as a convenient means to express lazily what we want, without putting the effort to articulate it in fine detail. Given a coarse description, finding an implementation requires eliminating the quantifiers. Control problems are typically expressed in this way, by asking whether there exists (and if so, to pick) a controller that constrains the system of interest (plant) enough to ensure their joint behavior is as we desire [2]. Quantifiers can make it easier for us to express our thoughts, but they are computationally expensive to remove [3], [4]. Depending on the type of problem, different methods can eliminate quantifiers with varying computational complexity [5, Ch.1], or it might be impossible to eliminate quantifiers.

Recognizing that control synthesis problems are equivalent to quantifier elimination problems, classical aims of control like set invariance and reachability can be generalized using specification languages that originate in software engineering research, such as temporal logic [6]. The capability of expressing sophisticated properties and dependencies is crucial for hybrid systems, which themselves can have rich and subtle behaviors [7], [8]. As research on verification and synthesis for hybrid dynamical systems progresses, there is a practical need for tools that provide both reference implementations of algorithms and idioms for common problems.

TuLiP was introduced in [9], and has since evolved into a collection of tools described in Section III. In this tutorial, we present the approach for hybrid system control design taken in TuLiP, which relies on separating a problem into solving two specialized feedback control problems: a discrete one at a coarse level, and a continuous problem at a finer level. This allows applying different algorithms to the discrete and continuous subproblems. Real-world hybrid systems are typically nonlinear, and quantifier elimination (controller design) for nonlinear continuous dynamics is hard. Even in the case of linear dynamical systems, the logic specifications are usually nonlinear. This motivates choosing

[1]California Institute of Technology, Pasadena, CA, 91125, USA. E-mails: {ifilippi, sdathath, murray}@caltech.edu.

[2]E-mail: slivingston@cds.caltech.edu.

[3]University of Michigan, Ann Arbor, MI, 48109, USA. E-mail: necmiye@umich.edu.

to solve only simple trajectory planning at the continuous level, and delegate the nonlinear, combinatorial part of the problem to discrete synthesis algorithms.

Abstraction is the connecting link between continuous and discrete planning [10]. An abstraction of a continuous system is a system with discrete-valued state. We require two characteristics from an abstraction: that it faithfully represent some of the behavior that the continuous system is capable of, and that it refines some given continuous geometry, which we mention in the logic specification to specify motion between sets that are polytopes. Faithful abstraction ensures that later we can concretize the discrete controllers we design into continuous feedback controllers that implement the same behavioral objective. It also ensures that the implemented controllers indeed satisfy the specification we wrote by mentioning polytopic sets of continuous states. In the presented framework, these take the form of partition refining algorithms that discretize the continuous flow of a piecewise affine dynamical system to a finite directed graph of possible transitions. In addition, the discretization takes into account external disturbances (noise).

We want to specify also desired behavior. A defining quality of a *dynamical* system is that its state changes in time. Thus, we can write a description using a time variable $t$, as is customary in physics and control, and quantify over $t$. This is a simple and explicit approach. Unfortunately, it leads to unreadable descriptions [11]. If few or nobody can read a design, there is little purpose in writing it, because it won't serve its purpose [12]. Temporal logic has proven useful for reasoning about dynamical systems [6], [13]. It trades off explicitness of temporal reference and quantification for readability. Here, we use linear temporal logic based on syntax and principles from the temporal logic of actions, TLA$^+$ [13], an untyped logic that enables precise descriptions founded on axiomatic set theory. TLA$^+$ differs in some aspects from earlier related literature [14], [15], [16].

The paper is organized as follows. Section II introduces the ingredients used to describe problems that we want to solve. These include temporal logic in Section II-A, two ways of writing state machines in Sections II-A.2 and II-B, assume-guarantee specifications in in Section II-A.3, systems with hybrid dynamics in Section II-C, and the two flavors of problem instances that we consider in Section II-D, one discrete, the other continuous. Section III describes how the toolbox code is organized and why. Section IV-A describes how games are formulated and solved, and Section IV-B the symbolic data structures that enable game solving for problem sizes that are practically useful. The abstraction of time and continuous state to obtain discrete representations that allow for discrete synthesis is described in Section IV-C. The approach is demonstrated with two examples in Section V, and concluding remarks are noted in Section VI. Throughout the paper, more details and some side topics are discussed in boxed figure environments.

*Notation:* We use some notation and terminology from TLA$^+$ [13], with some details omitted. In TLA$^+$, a function $f$ with domain $S$ is written as $f \triangleq [x \in S \mapsto e]$ where

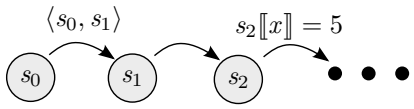



Fig. 1: A behavior $\sigma$ is an infinite sequence of states, each state assigns values to all variables, e.g., state $s_2$ maps $x$ to 5. A step is a pair of consecutive states $\langle s_i, s_{i+1} \rangle$.

$e$ some expression. The notation $f[x]$ denotes the value of function $f$ at $x$. For example, for $f \triangleq [x \in S \mapsto x^2]$ it is $f[3] = 9$. The set of all functions with domain $S$ that take values in $T$ is written $[S \to T]$ [13, p.48]. Let $\mathbb{N}$ ($\mathbb{R}$) denote the set of natural (real) numbers, and $\mathbb{B}$ the Boolean values TRUE, FALSE. Negation, con/disjunction are denoted with $\neg, \wedge, \vee$. 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 $i..j \triangleq \{k \in \mathbb{N} : i \leq k \leq j\}$. A function $f$ with DOMAIN $f = 1..j$, for some $j \in \mathbb{N}$, is a *tuple*, also written as $\langle a, \ldots, w \rangle$.

## 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. In TuLiP, each of these is expressed separately, corresponding to the subsections that follow.

### 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 $\sigma \in [\mathbb{N} \to D]$, from the natural numbers $\mathbb{N}$ to some set $D$. Suppose that in the design specification we want to refer to all sequences $\sigma$ such that $x \in R$ all the time, for some desired set $R$. We can quantify over time $t$ to express this collection of sequences as

$$\forall t \in \mathbb{N} : x[t] \in R. \tag{1}$$

A slightly more interesting property we may 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

$$\forall t \in \mathbb{N} : \exists \tau \in \mathbb{N} : (\tau \geq t) \wedge (x[\tau] \in R). \tag{2}$$

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 [17], Amir Pnueli introduced temporal logic [6], as a convenient shorthand that allows for simpler formulae, 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 $\langle s_i, s_{i+1} \rangle$ of

[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.

Fig. 2: *Assume-guarantee properties:*

A precise definition is[2]

$$\varphi \triangleq Init \Rightarrow \wedge\, \square\big((\ominus\boxminus Next_e) \Rightarrow Next_s\big) \\ \wedge\, (\square Next_e) \Rightarrow \vee \bigvee_{j=0}^{m} \diamond\square P_j \\ \vee \bigwedge_{i=0}^{n} \square\diamond R_i \qquad (3)$$

where $P_j \triangleq \neg Recur_{e,j}, R_i \triangleq Recur_{s,i}$, and $Init \triangleq Init_e$. The formula $(\ominus\boxminus Next_e) \Rightarrow Next_s$ requires that the system take a $Next_s$-step, unless the environment has taken earlier some $\neg Next_e$-step. 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 to force a *later* deviation from $Next_e$, a notion called *strict realizability* [19], [20], [21]. So, $Next_e$ can contain constraints that arise from physical modeling and conventions about interface protocols. More details on the "weak previous" and "historically" operators $\ominus, \boxminus$ and this definition can be found in [22]. Fig. 3 discusses complexity considerations.

Fig. 3: *A tractable fragment:*

The formula $\bigvee_{j=0}^{m} \diamond\square P_j \vee \bigwedge_{i=0}^{n} \square\diamond R_i$ in Eq. (3) describes a *liveness* goal, namely to either eventually satisfy some $P_j$ forever ($\diamond\square$), or satisfy repeatedly ($\square\diamond$) all the $R_i$. A *Streett pair* is a formula of the form $\diamond\square \vee \square\diamond$ [24], [25]. The above is a "generalized" pair, due to the $\bigvee_j, \bigwedge_i$ [26]. Generalized Reactivity(1) [25], [26], [21] is the name given to games (Section IV-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 $k$ [18], [27]. "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 $\bigwedge(\diamond\square \vee \square\diamond)$ ($\bigvee(\diamond\square \wedge \square\diamond)$). Thus, they are the conjunctive (disjunctive) normal form (CNF) for temporal formulae expressing liveness [28], in analogy to the propositional CNF and DNF [4, p.10].

consecutive states within a behavior is called a *step*, Fig. 1. A Boolean-valued formula that contains primed variables (for example $x'$) is called an *action*. Given some step, $\langle s_0, s_1\rangle$ and an action, $(x = 1) \wedge (x' = 2)$, 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, Eq. (1) can be written as $\square(x \in R)$ ($\square$ is the "always" operator), and Eq. (2) as $\square\diamond(x \in R)$ ($\diamond$ is the "eventually" operator). Using a variety of temporal operators and nesting them leads to unreadable formulae that defy why we want to use LTL in the first place. So, we will use only a few operators, 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 synthesize controllers for some of the variables, in order to satisfy a given LTL formula. Synthesis from LTL formulae with arbitrarily nested operators can be computationally very hard (exponential time or worse [3], [18]).

*2) Expressing state evolution in temporal logic:* In engineering and physics, systems are usually described by writing some equations that mention time, and an initial condition, as in Section II-C. What these equations and the initial condition define is a *state machine*. A state machine can also be expressed in temporal logic, by describing how the state of a system evolves [13]. 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 $Init \triangleq (c = 0)$ and alternates between 1 and 0, $Next \triangleq (c' = 1 - c)$. Assembling these into an LTL formula, we have $Init \wedge \square Next$. Notice that this formula talks about how *only* variable $c$ changes in a behavior. It leaves other variables free to change, which is useful for composition (see Fig. 7).

*3) Assume-guarantee properties:* 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 formulae $Init_e, Next_e$ and $Next_s$.

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 formulae of the form $\square\diamond Recur$. So, we need to define the predicates $Recur_e$ and $Recur_s$, one for each state machine. To summarize informally, we *require* that the guaranteed property

$$Guarantee \triangleq \square Next_s \wedge \bigwedge_i \square\diamond Recur_{s,i} \qquad (4)$$

is not violated *before* the assumed property

$$Assumption \triangleq Init_e \wedge \square Next_e \wedge \bigwedge_j \square\diamond Recur_{e,j} \qquad (5)$$

is. Thus, if the assumption holds throughout time, so should the commitment. We formalize this notion in Fig. 2.

### B. State machines enumerated a little

We have been discussing about writing state machines as temporal formulae. A state machine can have many states, leading to a long formula that is error-prone for humans to write correctly. Fortunately, these formulae usually have the same intuitive graph-like structure with respect to some

[2]Vertical arrangement of con/disjunction ($\wedge, \vee$) aids readability [23].

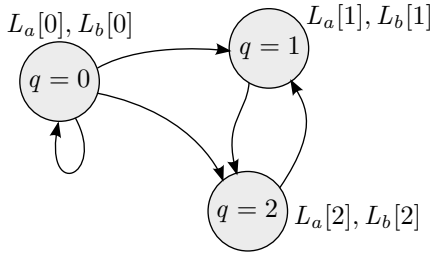



Fig. 4: A state machine defined using a graph.

Fig. 5: *Linguistic relativity:*

It should be noted that state machines go under many names and formalisms [29], so "transition system" is just a name for a particular data structure used here. It is helpful to think of state machines as predicate-action diagrams [30] 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.

variable. The data structure used in TuLiP to help with such definitions is a graph annotated with nodes and edges, and we call it a "transition system" (see also Fig. 5).

A transition system can be defined as a tuple $\langle Q, E, L_a, \ldots, L_w \rangle$ where $Q$ is a set of "states", $E \subseteq Q \times Q$ a set of edges, and $L_a, \ldots, L_w \in [Q \to \mathbb{B}]$ are functions that label states. In order to use symbolic algorithms as those described in Section IV-B, it is convenient to write a temporal logic formula that describes the entire discrete problem. So, we need to convert the transition systems to formulae.

Let us use Fig. 4 as an example to describe the conversion to an LTL formula. In this example $Q \triangleq \{0, 1, 2\}$. A variable is selected to be used for signifying the current node. Here we chose variable $q$ for that purpose. The possible transitions between nodes are defined by the edges $E$ between them, so we can prime $q$ to express $E$ with the action

$$\begin{aligned} Next(q, q') \triangleq \wedge\ & (q = 0) \Rightarrow \big((q' = 1) \vee (q' = 2) \vee (q' = 0)\big) \\ \wedge\ & (q = 1) \Rightarrow (q' = 2) \\ \wedge\ & (q = 2) \Rightarrow (q' = 1). \end{aligned}$$

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

The state labeling functions are used to constrain the values of variables of interest, here variables $a$ and $b$, as

$$StateLabels(q, a, b) \triangleq (a = L_a[q]) \wedge (b = L_b[q]).$$

These result in the temporal formula

$$TS \triangleq \Box\big(Next(q, q') \wedge StateLabels(q, a, b)\big).$$

So, the transition from node $q = 0$ to $q = 1$ can occur in a step $\langle q, q' \rangle = \langle i, j \rangle$ that satisfies also $(a = L_a[i]) \wedge (a' = L_a[j]) \wedge (b = L_b[i]) \wedge (b' = L_b[j])$.

### 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 of the form:

$$x[t+1] = Ax[t] + Bu[t] + Ed[t] + K, \tag{6}$$

or, in temporal logic

$$AffineTimeInv \triangleq \Box(x' = Ax + Bu + Ed + K)$$

where we require that, at any time $t > 0$, the continuous state $x[t] \in \mathcal{X} \subset \mathbb{R}^n$, and at any time $t \geq 0$ the control input $u[t] \in \mathcal{U} \subset \mathbb{R}^m$, and assume that the disturbance $d[t] \in \mathcal{D} \subset \mathbb{R}^p$, and the initial state $x[0] \in \mathcal{X}_{Init}$. The set $\mathcal{X}$ is called the domain of the system. The sets $\mathcal{X}$, $\mathcal{U}$, $\mathcal{D}$ are represented as polytopes or unions of polytopes. A *polytope* $\mathcal{P}$ is a set defined by linear predicates, that is,

$$\mathcal{P} = \{x \in \mathbb{R}^n : Hx \leq h\} \subset \mathbb{R}^n \tag{7}$$

We can also define state-dependent input bounds, by choosing $\mathcal{U}$ to be a polytopic subset of $\mathbb{R}^{n+m}$. A more general class of models are piecewise affine dynamical systems:

$$x[t+1] = A_i x[t] + B_i u[t] + E_i d[t] + K_i, \tag{8}$$

if $x[t] \in \mathcal{X}_i$, where, for $t > 0$, we require that the state $x[t] \in \mathcal{X}_i \subset \mathbb{R}^n$, and at any time $t \geq 0$, the control input $u[t] \in \mathcal{U}_i \subset \mathbb{R}^m$, and assume that the disturbance $d[t] \in \mathcal{D}_i \subset \mathbb{R}^p$, and the initial state $x[0] \in \mathcal{X}_{Init}$. The polytopic sets $\mathcal{X}_i$ for $i \in 1..k$ 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. (6) with disjoint domains.

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

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

where the mode $s[t] = \langle r[t], e[t] \rangle$ with $r[t] \in 1..n_r$, $e[t] \in 1..n_e$, being the discrete controllable and uncontrollable inputs that determine the system matrices $A, B, E, K$. We can also define switched piecewise affine systems where, for 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 $\{\mathcal{X}_i\}_{i=1}^k$, $\mathcal{X}_i \subset \mathcal{X}$ be a collection of subsets of interest. For computational tractability, we assume that each $\mathcal{X}_i$ is a polytope. Moreover, we assign "names", $a_i \triangleq (x \in \mathcal{X}_i)$, for each of these sets to be used in the LTL formulae. 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. Synthesis problems

Two problems can be solved with TuLiP:

1) discrete synthesis that constructs a controller that realizes an LTL specification in the GR(1) fragment, and
2) computing controllers for hybrid systems from specifications of piecewise affine dynamics, continuous propositions, and GR(1) formulae.

Fig. 6: *Moore or Mealy strategies?*

A Moore strategy $f$ picks the next value $y'$ without knowing $x'$, the environment's evolution in that same step [33]. In contrast, $x'$ is known to a Mealy strategy before it picks $y'$ [34]. Designing Moore machines as the components of a system is less error-prone, and leads to simpler composition and simulation. Mealy machines can lead to harder composition, due to the possibility of cyclic dependencies of function values and arguments [35].

These two problems are defined as follows.

*Problem 1:* [Discrete synthesis] Let $z$ ($y$) be variables that the environment (system) controls and take discrete values. Let $\varphi$ be a specification defined by an assumption of the form of Eq. (5), and a guarantee of the form of Eq. (4), as detailed in Fig. 2. Let $m$ be a *memory* variable, to be used by the controller. Assume that $m$ does not appear in $\varphi$.

The discrete synthesis problem with finite memory is that of finding: a controller function $f$ for the next value $y'$ of the controlled variables, a memory update function $g$, and an initial memory value $m_0$, such that $f$ and $g$ realize $\varphi$, i.e., $\mathit{Realization} \Rightarrow \varphi$, where

$$\begin{aligned} \mathit{Realization} \triangleq & \wedge\, m = m_0 \\ & \wedge\, \square \wedge\, y' = f[\langle m, y, z\rangle] \\ & \quad \wedge\, m' = g[\langle m, y, z\rangle], \end{aligned} \tag{9}$$

and variables $m, y, z$ take finitely many values [13, p.341]

$$\mathit{IsFiniteSet}(\text{DOMAIN } f) \wedge \mathit{IsFiniteSet}(\text{DOMAIN } g). \tag{10}$$

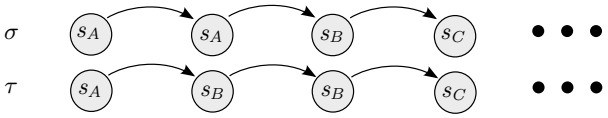
This definition is simplified for introductory purposes. A formal definition can be found in [31], which is based on [32]. Formalization avoids unintended controllers as solutions, for example a function with empty domain. For compatibility with existing literature on synthesis [21], the above definition is stutter sensitive (see Fig. 7), thus not in TLA$^+$. Problem 1 can be defined in TuLiP with a combination of temporal logic formulae Section II-A.3 and transition system data structures Section II-B, which are internally converted to formulae too, in order to assemble $\varphi$. TuLiP can synthesize both Moore and Mealy strategies (see Fig. 6).

*Problem 2:* [Synthesis for hybrid systems] Given continuous dynamics $x[t+1] = Ax[t] + Bu[t] + Ed[t] + K$ and sets $\mathcal{X}, \mathcal{D}, \mathcal{U}$, polytopes $\{\mathcal{X}_i\}_{i=1}^k$ as continuous propositions, and a temporal logic formula $\varphi$ over $z, y$ and $\{\mathcal{X}_i\}_{i=1}^k$, 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 $z, y, \{\mathcal{X}_i\}_{i=1}^k$.

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

Fig. 7: *Stutter-invariant properties:*

A temporal logic property $\varphi$ is *invariant under stuttering* if, however we repeat or remove repetitions of states from a behavior $\sigma$ that satisfies $\varphi$, the resulting behavior $\tau$, too, satisfies $\varphi$ [13, p.17].




In this paper, we did not insist on stutter-invariance, mostly for compatibility with existing literature on games and hybrid systems. Nonetheless, stutter-invariance is the cornerstone for writing hierarchical specifications [12], and their composition [19], because it allows for simple time refinement [36]. Proving system refinement becomes simply proving logical implication ($\Rightarrow$) between formulae at different levels [37].

## III. Software architecture

We discuss the software architecture from the user and developer viewpoints. For the user, we give an overview of entry and exit points, and how different modules correspond to sections in this paper. For the developer, we describe how and why the toolbox has evolved to its current structure. The architecture reported here should be understood as a recommendation, not as a frozen programming interface.

TuLiP can be thought of as a compiler that takes as input a problem description, and returns a controller, or raises an error if a controller cannot be found by the methods implemented. As discussed in Section II, there are several different ways to describe a problem, which can be combined. These include temporal logic formulae, graph-like data structures to represent discrete behaviors (transition systems), and linear difference equations over polytopic domains. Formulae are more naturally represented as strings, graphs as dictionaries of dictionaries (using the Python package `networkx` [38]), linear continuous dynamical systems with matrices, and polytopes in H-representation, as the matrices $H, h$ of Eq. (7), together with methods for operating on them. These can be thought of as a frontend.

This affinity to data structures is reflected in how the TuLiP package is organized. In Python, there are scripts, modules, and packages. Scripts and modules are single files, whereas packages comprise of modules and subpackages. The subpackages and modules of the `tulip` package [39], shown in Fig. 8, correspond to the aforementioned ways of describing a problem instance. Four organizational entities comprise the frontend:

- `tulip.spec`: a module for parsing, representing, manipulating, and translating formulae (Section II-A),
- `tulip.transys`: a subpackage for representing, translating to logic, and plotting graphs that serve as transition systems (Section II-B),
- `tulip.hybrid`: a module to represent piecewise

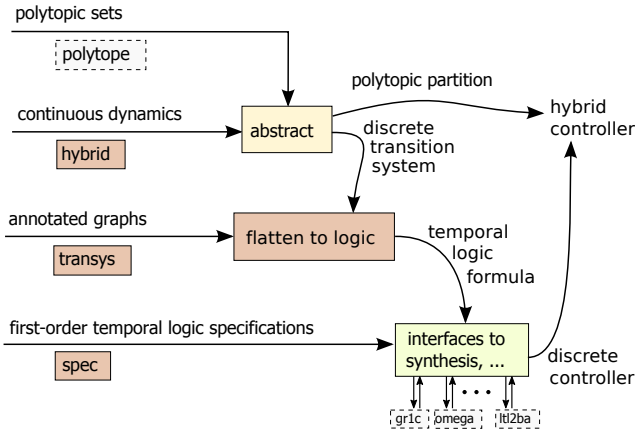



Fig. 8: Architecture of the temporal logic planning toolbox.

affine continuous dynamical systems with control inputs and disturbances (Section II-C),

- `polytope`: a package for representing and operating on polytopes [40], both convex and non-convex, using linear programming in the background. Initially based on v2 of the multi-parametric toolbox (MPT) [41].

The above pieces provide the means for representing and processing input to TuLiP. They do not include algorithms that solve the control design problem. These algorithms form the core of problem solving. Some of these algorithms have been implemented within TuLiP, and others separately:

- `tulip.abstract`: discretization of continuous dynamics (Section IV-C),
- `tulip.interfaces` and `tulip.synth`: interfaces to solvers of discrete games (specs in GR(1) or LTL),
- `gr1c`: solver for GR(1) games [42] (Section IV-A), implemented in C using the BDD library CUDD [43],
- `omega`: Python package that implements solving GR(1) games, compiling first-order logic over bounded integers [44], [45] (Section IV-A), and interfaces to BDDs using CUDD via `dd` [46] (Section IV-B). Structured on principles and experience from `tulip`.

Organizationally, we have found that as algorithms are used more and their APIs solidify, they grow into separate, reusable packages. TuLiP went through many phases during its development, and it is still changing. We have found that making installation as easy as possible is crucial to encourage people to try new software. TuLiP can run entirely in Python. Packages in pure Python are easy to install, with no need to compile hardware-specific code and the associated complications. However, they can be slower for solving demanding problems, so an implementation in C or another performance-oriented language can be well motivated.

We have approached this problem with a dual implementation: a simpler, more readable, and typically slower pure Python solving layer (both BDD operations to solve games, as well as linear programming), and a faster implementation in C, using `glpk` [47] for linear programming, and CUDD for BDDs [43]. For educational, algorithm prototyping, and software evaluation purposes, the Python implementation

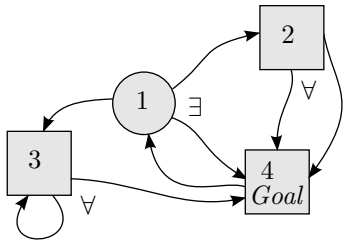



Fig. 9: The meaning of controllable predecessors.

suffices. For large, industrial-size problems, the C implementation can be used. In many cases, the same code is running, just with a different library underneath. Other helpful observations were Alan Perlis' advice to structure data late and prefer functions [48], and to follow the design philosophy of PEP20 [49]. Simple is better than complex.

## IV. 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…". More formally, this can be expressed as PICK $f : \forall x : \varphi(f, x)$, which can be solved only if the formula $\exists f : \forall x : \varphi(f, x)$ is true [50]. This is the *Skolemized* form of the synthesis problem [3] [5, p.18], and is similar with how control problems are usually phrased by engineers.

The nesting of different quantifiers $\exists\forall$ constitutes *alternation*, imagined as a game [51], [52]. Alternation can equivalently be written in a step-by-step form, where $f$ actuates on the system, then the dynamics $x$ evolves, and this repeats forever. This iterative 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. 9 is a game where we want to find how to move, in order to successfully reach the $Goal$. Let $Nodes$ be the set of nodes in this game. At each disk, we $(\exists)$ pick the next node, and the environment $(\forall)$ picks from boxes. Let us first find from *where* we can win. Precisely, from which nodes can we end in $Goal$, after taking 0 or more steps? Clearly, any nodes inside $Goal$ are winning. Also, all nodes from where we can reach $Goal$ within one step are winning too. But, when can we take a step to reach $Goal$ from a node?

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

$$\begin{aligned} &CPre_i(u, Goal) \triangleq \\ &\quad \vee \wedge IsADisk(u) \\ &\qquad \wedge \exists v \in Nodes : v \in Goal \wedge IsAnEdge(u, v) \\ &\quad \vee \wedge IsABox(u) \\ &\qquad \wedge \forall v \in Nodes : v \in Goal \vee \neg IsAnEdge(u, v). \end{aligned} \tag{11}$$

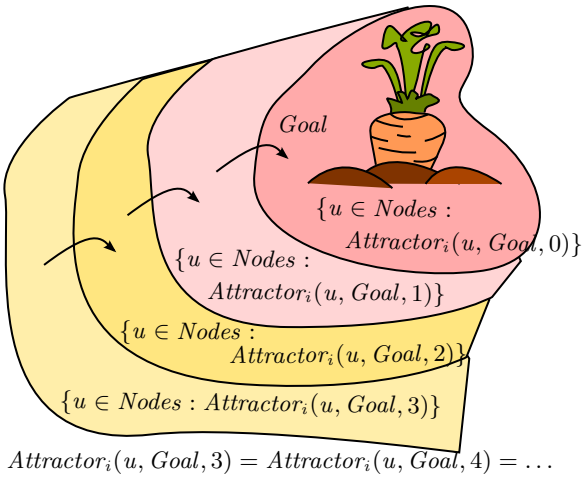



Fig. 10: An attractor is the answer to a reachability game.

This is called the *controllable predecessor* operation, and the game just described a *reachability game* [24]. The subscript $i$ signifies our player. By iterating Eq. (11), we can compute whether $Goal$ can be reached from some node $u$ in at most $k$ steps (where $k \geq 0$) [53]

$$\begin{aligned} &Attractor_i(u, Goal, k) \triangleq \text{ IF } k = 0 \\ &\quad \text{THEN } u \in Goal \\ &\quad \text{ELSE} \vee Attractor_i(u, Goal, k-1) \\ &\qquad \vee CPre_i(u, \ \{w \in Nodes : \\ &\qquad\qquad Attractor_i(w, Goal, k-1)\}). \end{aligned} \tag{12}$$

For $k = 0$, the $Attractor_i(u, Goal, 0)$ is $u \in Goal$, meaning that $u$ is already in the $Goal$. For $k > 0$, the $Attractor_i(u, Goal, k)$ is defined recursively: from $u$, either we can reach $Goal$ in at most $k-1$ steps (the disjunct $Attractor_i(u, Goal, k-1)$), or we can reach some node that can reach $Goal$ in at most $k-1$ steps (disjunct $CPre_i(u, ...)$).

When writing a game solver, Eq. (12) is used iteratively to compute sets of nodes, starting with $X_0 \triangleq \{u \in Nodes : Attractor_i(u, Goal, 0)\}$. Then, we compute $X_k \triangleq \{u \in Nodes : Attractor_i(u, Goal, k)\}$ for $k = 1, 2, \ldots$. Under certain conditions that are typically satisfied by the games under study [5], it can be proved that this iteration reaches a *fixpoint* $X_{k+1} = X_k$ for some *finite* $k \in \mathbb{N}$. In this particular iteration, the result is a *least* fixpoint, shown in Fig. 10, and is the *attractor* of the $Goal$ nodes.

Dually, we might want to solve a *safety* game, finding the states from where we can remain forever within a set of nodes $G$, which is computed similarly by slightly differently. 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 (see also Fig. 11).

### B. Symbolic algorithms using binary decision diagrams

*a) Symbolic vs enumerative methods:* From Section IV-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.

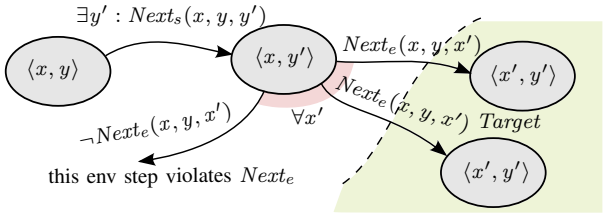
In model checking [54], enumeration can be viable, because there is only one kind of quantification, typically universal quantification for proving validity. For problems with only universal quantification, we can always negate them to solve a problem with existential quantification. In contrast, game solving involves alternating quantification (Section IV-A), so negation doesn't eliminate universal quantification any more. Enumeration in the presence of universal quantification can lead to problems, because of its exhaustive character.

[3]Writing primed values as arguments to a predicate is incorrect. We do it only here, to emphasize what variables appear in each predicate.

Fig. 11: *Assume-guarantee Controllable predecessors:* When we program a game solver, we need to write the controllable predecessor $CPre_i$ of Eq. (11) for the specification that we are given in the form of an assumption, Eq. (5), and a guarantee, Eq. (4) (that describes an "open system" [20]). For the way that Fig. 2 combines an assumption with a guarantee, $CPre_s$ ($s$ stands for "system") becomes[3]

$$\begin{aligned} &CPre_s(\langle x, y\rangle, Target) \triangleq \exists y' : \forall x' : \\ &\quad \wedge Next_s(x, y, y') \\ &\quad \wedge \big(Next_e(x, y, x') \Rightarrow \langle x', y'\rangle \in Target\big), \end{aligned} \tag{13}$$

where the system controls variable $y$ and the environment $x$. The order that quantifiers are nested means that the system picks $y'$ without knowing the exact value of $x'$. Also, it can violate $Next_s$ only in some step *after* $Next_e$ is violated. This corresponds to ⦵⊟ [22] from Fig. 2.




First, our system picks $y'$, and then all $x'$ should either violate $Next_e$, or lead to the desired $Target$.

Notice that the $Next_e$ above mentions the values $x, y$ and $x'$. It does not mention the next system values $y'$. This means that we make no assumption as to whether the environment can react or not to $y'$, when it picks $x'$. In other words, we do not rely in the assumption $Next_e$ on whether the environment will comprise Moore or Mealy strategies (Fig. 6). From the perspective of $CPre_s$, the environment could have been a Moore strategy, it just doesn't matter, because $Next_e$ is *independent* of $y'$. If, instead, we mentioned $y'$ within $Next_e$, then we would have to ensure that such a specification faithfully models the physical problem that we are solving.

*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) [55] as symbolic representation of state sets. For brevity, we will introduce ROBDDs with a small example, shown in Fig. 12.

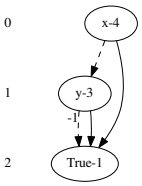


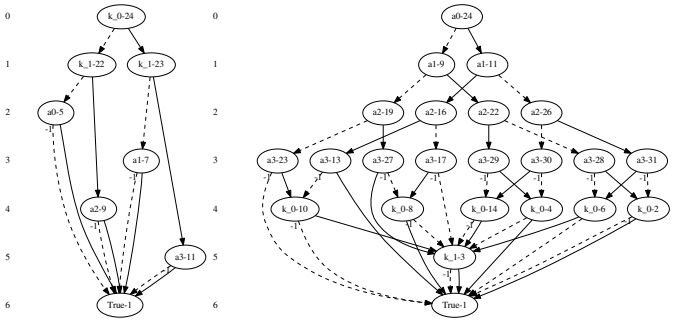
Fig. 12: $x \vee y$.

Suppose that we want to represent some subset of the set $D \triangleq \{0,1\} \times \{0,1\}$ with an ROBDD. We will use the variables $x, y$ to name elements in this set, so that $\langle x, y\rangle \in D$. 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 variabe. 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 $j..(n-1)$. For example, node 4 describes the assignments $\langle x, y\rangle \in \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\} = \{p \in D : p[0] \vee p[1]\}$. Given the assignment $\langle x, y\rangle = \langle 0,1\rangle$, 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 $\langle 0,1\rangle$, $x$ 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 $\langle x, y\rangle = \langle 0,1\rangle$ *is* in the ROBDD with node 4 as root. In contrast, $\langle x, y\rangle = \langle 0,0\rangle$ is *not*, because it leads to node 1 via the complemented edge (dashed) from node 3 to 1, with answer 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 $m(m-1)$ directed edges possible, so at most $2^{m(m-1)}$ different directed graphs we can construct. Given $n$ variables assigned values from $\{0,1\}$, there are $2^{2^n}$ possible *sets* of assignments. Thus, some set of assignments can be represented only with a BDD of size $m = \mathcal{O}(\sqrt{2^n})$. This means that there are sets that have exponential BDD size, whatever variable order we choose. In fact, this is the case for *almost all* sets [56, Thm.7]. Even worse, multiplication (thus division and modulo too) is an operation that cannot be represented efficiently with BDDs [57].

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 *selector* and *data* variables [58, 1, Sec.7.2.3, 7.4.1, 7.5.3]. A simple example is a bit-valued 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. 13a. Reversing this order causes a blowup in size, as in Fig. 13b. The reason is that, as we go from top to bottom, if we first encounter $k$, 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* the array, so that when we finally reach $k$, we can recall the appropriate value of bit $a_k$.



(a) Good order. (b) Bad variable order.

Fig. 13: A good variable order places selectors above data.

Operations on BDDs rely on memoization to avoid recomputing a result that is encountered multiple times during traversal [59]. 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 [60]. 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 `dd` [44], [46] contains a pure Python BDD implementation, as well as Cython [61] bindings to the C library CUDD [43], 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 CUDD only at deployment, without changing the implemented algorithm.

To write a game solver that uses BDDs, we need to initialize a BDD manager, define the variables of interest, and then call Boolean operators to create bottom-up the BDDs that represent the actions $Next_e$ and $Next_s$, and the other predicates in Eqs. (4) and (5), from the parsed Boolean formulae. After this preprocessing is done, we can again call Boolean operators with BDDs as arguments, to iteratively compute Eq. (11) and thus solve a reachability game, as described in Section IV-A.

### 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 [62], [63], [64], [65] and some of them are being integrated in TuLiP. Next, we briefly explain the main abstraction algorithm implemented in TuLiP, proposed in [14] 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 $\{\mathcal{X}_i\}_{i=1}^k$ of interest in the domain $\mathcal{X}$ of the hybrid system. The properties we want the abstraction to preserve are those related to these propositions.

Let us introduce labeling functions $L_{X,i} \in [\mathcal{X} \to \mathbb{B}]$ for the hybrid system such that $L_{X,i} \triangleq [x \in \mathcal{X} \mapsto (x \in \mathcal{X}_i)]$. Let $a_i$ be variable names (propositions) that will be set equal to $(x \in \mathcal{X}_i)$ below. The abstraction algorithm aims to find an abstraction function $\alpha \in [\mathcal{X} \to Q]$ and a set $E \subset Q \times Q$ of transitions such that

(i) $\alpha$ is proposition preserving, that is, for any $q \in Q$ and for any $x \in \alpha^{-1}[q]$, the labels match, i.e., for all variables $a_i$, it is $L_{a_i}[q] = L_{X,i}[x]$,

(ii) $\langle q, q' \rangle \in E$ only if, for all $x_1 \in \alpha^{-1}[q]$, there exists an input sequence $\langle u_1, \ldots, u_N \rangle \in \mathcal{U}^N$ such that for all $\langle d_1, \ldots, d_N \rangle \in \mathcal{D}^N$, $x_{N+1} \in \alpha^{-1}[q']$, and $x_i \in \alpha^{-1}[q]$ for $i \in 1..N$.

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. If there is an external environment the system interacts with, it is assumed to remain constant in this horizon.

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 [14], [15]. 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*. Fig. 14 demonstrates the partitioning, abstraction, and evolution of the state of the hybrid system with the hierarchical controller.

Some variants of condition (ii) are also implemented in TuLiP. The version defined above is called *open-loop* as the control inputs $u_i$ are computed once $x_t$ is observed. There is also a *closed-loop* variant, where, we require:

(ii)* $\langle q, q' \rangle \in E$ only if for all $x_1 \in \alpha^{-1}[q]$, there exists $u_1 \in \mathcal{U}$, for all $d_1 \in \mathcal{D}$, ..., there exists $u_N \in \mathcal{U}$, for all $d_N \in \mathcal{D}$ such that $x_{N+1} \in \alpha^{-1}[q']$, and $x_i \in \alpha^{-1}[q]$ for $i \in 1..N$.

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 [66]. 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 [54], [62]. 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 $P_j'$ 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 [14]. 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 [15].

## V. 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 planning based on faults in the sensor system for "Alice", the autonomous vehicle developed by Caltech for the 2004–2007 DARPA Grand Challenge [67]. The vehicle velocity 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 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 the form

$$x[t+1] = a_i x[t] + b_i u[t] + c_i d[t],$$

where $i$ denotes the driving mode. The constants $\langle a_i, b_i, c_i \rangle$ have values $\langle 1, 1, 1 \rangle$ for the mode "Slow/stop", $\langle 1.3, 2, 1 \rangle$ for "Moderate", and $\langle 1.6, 2.8, 1 \rangle$ for "Fast". For each time $t > 0 : x[t] \in \mathcal{X}$. 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 is assumed that a healthy Stereo camera can make accurate short-range measurements, while a functional Lidar accurate long-range measurements. Based on sensor health,

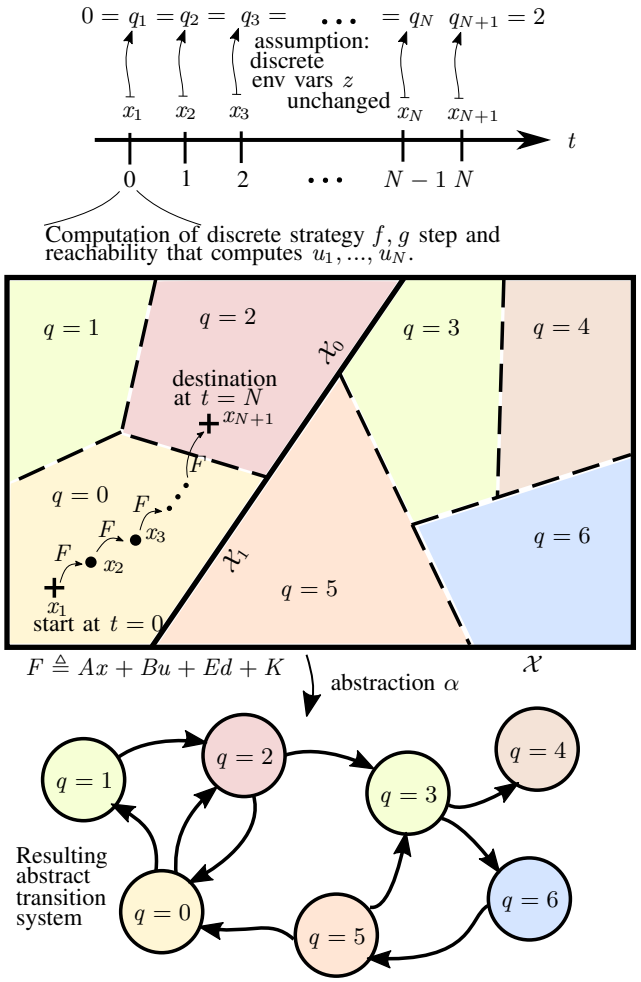



Fig. 14: Abstraction of continuous state $x$ to discrete state $q$, refinement of partition $\mathcal{X}_0, \mathcal{X}_1$, and concretization of synthesized discrete controller using finite-horizon control.

the vehicle is required to determine the control action for the vehicle to transition to a drive-mode (appropriate speed) that is safe with respect to the current sensor health status. 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. The system specification is the following:

1) Continuously-valued control inputss in $\mathcal{U} \triangleq [-2, 2]$
2) Disturbance assumed in $\mathcal{D} \triangleq [-0.01, 0.01]$
3) The vehicle is initially at the Stop/Reboot state
4) Initially all sensors are off (or are not functional)
5) The vehicle does not change driving modes, unless necessary
6) If both long-range and short-range sensing are healthy, the vehicle must drive fast, unless it loses sensing
7) If short-range sensing is not healthy, the vehicle must eventually stop until the sensor comes back on
8) The vehicle must always eventually drive in moderate/fast modes
9) Both sensors are always eventually functional
10) If and only if it is dark, the lights must go on.

| Index | LTL formula of specification |
|---|---|
| 3 | $init$ |
| 4 | $\neg lidon, \neg steron$ |
| 5 | $(fast \wedge (steron \wedge lidon)) \Rightarrow fast'$<br>$(slow \wedge (\neg steron \wedge \neg lidon)) \Rightarrow slow'$<br>$(moderate \wedge (steron \wedge \neg lidon)) \Rightarrow moderate'$ |
| 6 | $\square((lidon \wedge steron) \Rightarrow \diamond(fast\ \mathcal{U} \neg(lidon \wedge steron)))$ |
| 7 | $\square((\neg steron) \Rightarrow \diamond(init\ \mathcal{U} steron))$ |
| 8 | $\square\diamond(fast \vee moderate)$ |
| 9 | $\square\diamond lidon, \square\diamond steron$ |
| 10 | $dark \equiv lights$ |

TABLE I: Specification expressed in LTL.

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 whether the lights for the vehicle are on or off. It is worth observing that, if the assumption no.9 is not satisfied by the environment, then the system cannot satisfy the requirement no. 8, because it is unsafe to drive when both sensors are faulty (requirement no. 7). The set of specifications can be divided into environmental assumptions and system requirements. For writing the LTL formula, the statements no. 4, 9 form the environmental assumption part of the formula, and the statements no. 5, 6, 7 and 8 describe the required system behavior.

*1) LTL Specification:* Let $init$ be a Boolean-valued variable that holds TRUE if the velocity is in the Slow/Stop mode. Similarly define $moderate$ and $fast$ for the Moderate and Fast driving modes. $steron$ and $lidon$ are variables that evaluate to TRUE if the Stereo and Lidar, respectively, are functional, and FALSE otherwise. Table I translates the specifications in listed 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 Sections II-A.3 and II-D. Specifications no. 6 and no. 7 as written above are not in the GR(1) fragment, but can be manipulated by the introduction of an auxillary variable. Consider specification no. 6. An equivalent specification in GR(1) can be obtained by introducing an auxiliary variable $aux$ [25]. The variable $aux$ is initialized to TRUE, and the transition rule governing it is $(aux' \equiv (fast \vee (\neg(lidon \wedge steron)))) \vee (aux \wedge \neg(lidon \wedge steron))$, in conjunction with $\square\diamond aux$. The first time that $lidon \wedge steron$ evaluates to TRUE, if $fast$ evaluates to FALSE, then $aux'$ becomes FALSE. $aux'$ can become TRUE only if either $fast$ becomes TRUE, or $lidon \wedge steron$ becomes FALSE. The specification that was not directly in GR(1) was thus transformed to be expressed in GR(1) syntax, as a combination of a liveness specification and a transition rule. The translated specifications however are not directly equivalent because the specifications above only ensure $\square((lidon \wedge steron) \Rightarrow \diamond fast)$ but specification no. 5 ensures that $(fast\ \mathcal{U} \neg(lidon \wedge steron)))$ is satisfied. Similarly, no. 7 can also be translated to GR(1) with the introduction of an auxillary variable. These specifications are then input to TuLiP as initial, action, and recurrence conditions for the system and the environment.

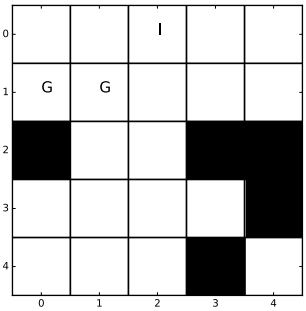



Fig. 15: Example of a gridworld.

*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 continous part of the state-space is abstracted into a finite-state transition system Section IV-C, which is then flattened to temporal logic formulae Section II-B. Discrete synthesis is then performed with the specification formulae above, conjoined with the flat formula for the abstracted dynamics.

### *B. Gridworld with moving obstacle*

In this section, we consider the planar motion planning in an environment of the form shown in Fig. 15, known as a gridworld. The robot begins at cell I and has to visit the two goal cells G infinitely often, and avoid collision with walls. In each time step, the robot can transition to any of its non-diagonally adjacent cells. The black cells signify walls, The obstacle moves according to similar rules, but begins at a different initial location, and has two different goals to visit infinitely often. The robot should visit its goals and ensure that it never collides with the moving obstacle. We synthesize a strategy (controller) for the robot that takes into account all possible moves of the obstacle, expressed as an assumption on its behavior.

The dynamics for this system are specified through LTL specifications. We consider a $5 \times 5$ grid. We begin by specifying the integer values the variables can take. Let $X_r$ be the value the current row of the obstacle and $X_c$ the current column. Similarly, $Y_r$ and $Y_c$ for the robot for which the motion plan is being synthesized. $X_r, X_c, Y_r, Y_c$ can take integer values in the range $0-4$.

We begin by specifying the initial pose for the robot and the obstacle. Let $\langle r_{sys}, c_{sys}\rangle$ be the initial pose of the robot and $\langle r_{obs}, c_{obs}\rangle$ be that for the obstacle. So, the assumed initial condition is $Init \triangleq Y_r = r_{sys} \wedge Y_c = c_{sys} \wedge X_r = r_{obs} \wedge X_c = c_{obs}$. Subsequently, the possible transitions are specified, for the obstacle:

$$\bigwedge_{i=0}^{i=4}\bigwedge_{j=0}^{j=4}\Big(\begin{matrix}\wedge\, X_r = i \\ \wedge\, X_c = j\end{matrix} \Rightarrow \bigvee_{k=r^i_{\min}}^{k=r^i_{\max}} \bigvee_{l=c^j_{\min},|k-i|+|l-j|\leq 1}^{l=c^j_{\max}} \wedge\, X_r = k \wedge\, X_c = l\Big)$$

where $c^j_{\max} = \min(4, j+1), c^j_{\min} = \max(0, j-1), r^i_{\min} = \max(0, i-1)$ and $r^i_{\max} = \min(4, i+1)$. This ensures that the obstacle transitions from a cell to itself, or to any of the non-diagonal adjacent cells. A similar formula is defined for the system as well. Let $Q$ be the set of cells, denoted by an ordered pair of the form $\langle row, column\rangle$. that form the walls in the grid. Transition rules with regard to non-collision for the obstacle with the wall are specified as

$$\bigwedge_{\langle i,j\rangle \in Q} \Box\neg(X_r = i \wedge X_c = j)$$

and form part of the assumption on the environment. Similarly, rules prohibiting collision of the robot with the wall are also specified. Let $G$ be the set of goal cells for the environment that it must visit infinited often. The recurrence goals for the environment's behavior are encoded as

$$\bigwedge_{\langle i,j\rangle \in G} \Box\Diamond(X_r = i \wedge X_c = j).$$

In a similar manner, the recurrence goals corresponding to the system are also specified in temporal logic. 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 ime must not occupy the same cell as the obstacle. This is expressed in temporal logic with the formula

$$\bigwedge_{i=0}^{i=4}\bigwedge_{j=0}^{j=4} \Box\big((X_r = i \wedge X_c = j) \Rightarrow \neg((\,Y_r = i) \wedge (\,Y_c = j))\big)$$

This specification is in the GR(1) fragment, and thus synthesis can be performed with a solver from those available in TuLiP e.g., `gr1c` which accepts as input the GR(1) specification assembled from the above formulae.

## VI. CONCLUSIONS

This tutorial presented the control design approaches that are possible by using the tools in the `tulip` toolbox [9], [15]. Several tools have been developed for the verification and synthesis of systems [68]. Work more relevant to that presented here is [69], [70]. In the future, we plan to integrate into the toolbox algorithms for decomposing properties to create contracts [22], and more expressive logics.

*Acknowledgments:* This work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. NO was supported in part by NSF grants CNS-1446298 and ECCS-1553873, and DARPA grant N66001-14-1-4045.

## REFERENCES

[1] L. Lamport, "Who builds a house without drawing blueprints?" *CACM*, vol. 58, no. 4, pp. 38–41, 2015.
[2] J. C. Willems, "The behavioral approach to open and interconnected systems," *IEEE CSM*, pp. 46–99, Dec 2007.
[3] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL*, 1989, pp. 179–190.
[4] D. Kroening and O. Strichman, *Decision procedures*. Springer, 2008.
[5] K. Schneider, *Verification of reactive systems: formal methods and algorithms*. Springer, 2004.
[6] A. Pnueli, "The temporal logic of programs," in *FOCS*, 1977, pp. 46–57.
[7] M. S. Branicky, "Studies in hybrid systems: Modeling, analysis, and control," Ph.D. dissertation, Massachusetts Institute of Technology, June 1995.
[8] T. A. Henzinger, "The theory of hybrid automata," in *Proc. of the 11th Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 1996, pp. 278–292.

[9] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, "TuLiP: A software toolbox for receding horizon temporal logic planning," in *HSCC*, 2011, pp. 313–314.

[10] S. M. LaValle, *Planning Algorithms*. Cambridge Univ. Press, 2006.

[11] N. Francez and A. Pnueli, "A proof method for cyclic programs," *Acta Informatica*, vol. 9, pp. 133–157, 1978.

[12] L. Lamport, "What good is temporal logic?" in *Information Processing*, vol. 83, 1983, pp. 657–668.

[13] ——, *Specifying Systems*. Addison-Wesley, 2002.

[14] T. Wongpiromsarn, "Formal methods for design and verification of embedded control systems: Application to an autonomous vehicle," Ph.D. dissertation, California Institute of Technology, 2010. [Online]. Available: http://resolver.caltech.edu/CaltechTHESIS:05272010-153304667

[15] P. Nilsson, N. Ozay, U. Topcu, and R. M. Murray, "Temporal logic control of switched affine systems with an application in fuel balancing," in *ACC*, June 2012, pp. 5302–5309.

[16] S. C. Livingston, R. M. Murray, and J. W. Burdick, "Backtracking temporal logic synthesis for uncertain environments," in *ICRA*, 2012, pp. 5163–5170.

[17] L. Lamport, "Temporal logic: The lesser of three evils," Amir Pnueli Memorial Symposium, NYU, May 2010. [Online]. Available: http://www.cs.nyu.edu/acsys/pnueli/

[18] N. Piterman and A. Pnueli, "Faster solutions of Rabin and Streett games," in *LICS*, 2006, pp. 275–284.

[19] M. Abadi and L. Lamport, "Conjoining specifications," *TOPLAS*, vol. 17, no. 3, pp. 507–535, 1995.

[20] ——, "Open systems in TLA," in *PODC*, 1994, pp. 81–90.

[21] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reacive(1) designs," *JCSS*, vol. 78, no. 3, pp. 911–938, 2012.

[22] I. Filippidis and R. M. Murray, "Symbolic construction of GR(1) contracts for systems with full information," in *ACC*, 2016, pp. 782–789.

[23] L. Lamport, "How to write a long formula," *Formal aspects of computing*, vol. 6, pp. 580–584, 1994.

[24] W. Thomas, "On the synthesis of strategies in infinite games," in *STACS*, 1995, pp. 1–13.

[25] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *VMCAI*, 2006, pp. 364–380.

[26] Y. Kesten, N. Piterman, and A. Pnueli, "Bridging the gap between fair simulation and trace inclusion," *Information and Computation*, vol. 200, no. 1, pp. 35–61, 2005.

[27] A. Pnueli and U. Klein, "Synthesis of programs from temporal property specifications," in *MEMOCODE*, 2009, pp. 1–7.

[28] W. Thomas and H. Lescow, "Logical specifications of infinite computations," in *A decade of concurrency reflections and perspectives*, 1993, pp. 583–621.

[29] L. Lamport, "Concurrency, compositionality, and correctness," 2010, ch. Computer Science and State Machines, pp. 60–65.

[30] ——, "TLA in pictures," *TSE*, vol. 21, no. 9, pp. 768–775, 1995.

[31] I. Filippidis and R. M. Murray, "Formalizing synthesis in TLA$^+$," Caltech, Tech. Rep. CaltechCDSTR:2016.004, 2016. [Online]. Available: http://resolver.caltech.edu/CaltechCDSTR:2016.004

[32] L. Lamport, "Miscellany," 21 April 1991, note sent to TLA mailing list. [Online]. Available: http://lamport.org/tla/notes/91-04-21.txt

[33] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata studies*, vol. 34, pp. 129–153, 1956.

[34] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[35] L. de Alfaro, T. Henzinger, and F. Y. Mang, "The control of synchronous systems," in *CONCUR*, 2000, pp. 458–473.

[36] M. Abadi and L. Lamport, "The existence of refinement mappings," *TCS*, vol. 82, no. 2, pp. 253–284, 1991.

[37] L. Lamport, "Composition: A way to make proofs harder," in *Int. Symp. on Compositionality*, ser. COMPOS, 1998, pp. 402–423.

[38] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *SciPy*, Pasadena, CA USA, 2008, pp. 11–15.

[39] "`tulip`: The temporal logic planning toolbox (Python package)," 2010–2016, v1.3.0. [Online]. Available: http://tulip-control.org

[40] "`polytope`: Geometric operations on polytopes of any dimension (Python package)," 2010–2016. [Online]. Available: https://github.com/tulip-control/polytope

[41] M. Herceg, M. Kvasnica, C. Jones, and M. Morari, "Multi-Parametric Toolbox 3.0," in *ECC*, 2013, pp. 502–510. [Online]. Available: http://people.ee.ethz.ch/~mpt/3/

[42] S. C. Livingston, "Incremental control synthesis for robotics in the presence of temporal logic specifications," Ph.D. dissertation, California Institute of Technology, 2016. [Online]. Available: http://resolver.caltech.edu/CaltechTHESIS:12312015-131513787

[43] F. Somenzi, "CUDD: CU Decision Diagram package - v2.5.1," *Colorado University at Boulder*, 2015.

[44] I. Filippidis, R. M. Murray, and G. J. Holzmann, "A multi-paradigm language for reactive synthesis," in *SYNT'15*, ser. EPTCS, vol. 202, 2016, pp. 73–97.

[45] "`omega`: Symbolic algorithms for solving games of infinite duration (Python package)." [Online]. Available: https://github.com/johnyf/omega

[46] "`dd`: Decision diagrams (Python package)." [Online]. Available: https://github.com/johnyf/dd

[47] "GNU Linear programming kit, version 4.60." [Online]. Available: http://www.gnu.org/software/glpk/glpk.html

[48] A. J. Perlis, "Special feature: Epigrams on programming," *ACM SIGPLAN Notices*, vol. 17, no. 9, pp. 7–13, 1982.

[49] T. Peters, "PEP20: The Zen of Python," Python Enhancement Proposal, 2004. [Online]. Available: https://www.python.org/dev/peps/pep-0020/

[50] L. Lamport, "How to write a 21st century proof," *Journal of fixed point theory and applications*, vol. 11, no. 1, pp. 43–63, 2012.

[51] G. L. Peterson and J. H. Reif, "Multiple-person alternation," in *FOCS*, 1979, pp. 348–363.

[52] I. Walukiewicz, "A landscape with games in the background," *LICS*, pp. 356–366, 2004.

[53] W. Thomas, "Solution of Church's problem: A tutorial," *New Perspectives on Games and interaction*, vol. 5, 2008.

[54] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT, 2008.

[55] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.

[56] C. E. Shannon, "The synthesis of two-terminal switching circuits," *Bell System Technical Journal*, vol. 28, no. 1, pp. 59–98, 1949.

[57] R. E. Bryant, "On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication," *TOC*, vol. 40, no. 2, pp. 205–213, 1991.

[58] D. L. Beatty, "A methodology for formal hardware verification, with application to microprocessors," Ph.D. dissertation, CMU, 1993.

[59] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a bdd package," in *DAC*, 1990, pp. 40–45.

[60] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *ICCAD*, 1993, pp. 42–47.

[61] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science and Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[62] P. Tabuada and G. J. Pappas, "Linear time logic control of discrete-time linear systems," *TAC*, vol. 51, no. 12, pp. 1862–1877, 2006.

[63] P. Tabuada, *Verification and control of hybrid systems: a symbolic approach*. Springer, 2009.

[64] M. Kloetzer and C. Belta, "A fully automated framework for control of linear systems from temporal logic specifications," *TAC*, vol. 53, no. 1, pp. 287–297, 2008.

[65] N. Ozay, J. Liu, P. Prabhakar, and R. M. Murray, "Computing augmented finite transition systems to synthesize switching protocols for polynomial switched systems," in *ACC*, 2013, pp. 6237–6244.

[66] F. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for linear and hybrid systems*, 19 Oct 2015.

[67] J. W. Burdick, N. duToit, A. Howard, C. Looman, J. Ma, R. M. Murray, and T. Wongpiromsarn, "Sensing, navigation and reasoning technologies for the DARPA Urban Challenge," *DARPA Urban Challenge Final Report*, 2007.

[68] I. Filippidis and contributors. (2013) List of verification and synthesis tools. [Online]. Available: https://github.com/johnyf/tool_lists/blob/master/verification_synthesis.md

[69] C. Finucane, G. Jing, and H. Kress-Gazit, "LTLMoP: Experimenting with language, temporal logic and robot control," in *IROS*, 2010, pp. 1988–1993. [Online]. Available: https://github.com/VerifiableRobotics/LTLMoP

[70] M. Mazo Jr., A. Davitian, and P. Tabuada, "PESSOA: A tool for embedded controller synthesis," in *CAV*, 2010, pp. 566–569. [Online]. Available: http://www.cyphylab.ee.ucla.edu/pessoa/