Control of Dynamical Systems with Temporal Logic Specifications

Thesis by

Eric M. Wolff

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

2014

(Defended May 12, 2014)

To Mom, Dad, Ryan, Shawn, and Erin.

Acknowledgements

I extend my deepest thanks to my committee: Richard Murray, Joel Burdick, Ufuk Topcu, and K. Mani Chandy. They have helped and inspired me throughout my research. I am especially grateful to have had Richard as my advisor. His boundless energy, cheerful encouragement, and insightful advice helped me develop as a researcher. It has been an honor to work with him. I am also thankful to have worked closely with Ufuk throughout my PhD. His generous support and mentorship helped me tremendously as I started my research. I also thank Joel and Mani for insightful discussions, especially Joel's uncanny ability to always recommend a neat article.

I would also like to thank my friends and colleagues, especially those in the Murray and Burdick groups. It has been fun doing everything with them from discussing subtle points about Lie groups, to imagining the future of robotics, to skiing. I would also like to thank Laura Humphrey for hosting me at AFRL for an amazing internship. Finally, I am thankful for the financial support that I have received from an NSF Graduate Research Fellowship, an NDSEG Fellowship, and the Boeing Corporation.

Abstract

This thesis is motivated by safety-critical applications involving autonomous air, ground, and space vehicles carrying out complex tasks in uncertain and adversarial environments. We use temporal logic as a language to formally specify complex tasks and system properties. Temporal logic specifications generalize the classical notions of stability and reachability that are studied in the control and hybrid systems communities. Given a system model and a formal task specification, the goal is to automatically synthesize a control policy for the system that ensures that the system satisfies the specification. This thesis presents novel control policy synthesis algorithms for optimal and robust control of dynamical systems with temporal logic specifications. Furthermore, it introduces algorithms that are efficient and extend to high-dimensional dynamical systems.

The first contribution of this thesis is the generalization of a classical linear temporal logic (LTL) control synthesis approach to optimal and robust control. We show how we can extend automata-based synthesis techniques for discrete abstractions of dynamical systems to create optimal and robust controllers that are guaranteed to satisfy an LTL specification. Such optimal and robust controllers can be computed at little extra computational cost compared to computing a feasible controller.

The second contribution of this thesis addresses the scalability of control synthesis with LTL specifications. A major limitation of the standard automaton-based approach for control with LTL specifications is that the automaton might be doubly-exponential in the size of the LTL specification. We introduce a fragment of LTL for which one can compute feasible control policies in time polynomial in the size of the system and specification. Additionally, we show how to compute optimal control policies for a variety of cost functions, and identify interesting cases when this can be done in polynomial time. These techniques are particularly relevant for online control, as one can guarantee that a feasible solution can be found quickly, and then iteratively improve on the quality as time permits.

The final contribution of this thesis is a set of algorithms for computing feasible trajectories for high-dimensional, nonlinear systems with LTL specifications. These algorithms avoid a potentially computationally-expensive process of computing a discrete abstraction, and instead compute directly on the system's continuous state space. The first method uses an automaton representing the specification to directly encode a series of constrained-reachability subproblems, which can be solved in a modular fashion by using standard techniques. The second method encodes an LTL formula as mixed-integer linear programming constraints on the dynamical system. We demonstrate these approaches with numerical experiments on temporal logic motion planning problems with high-dimensional (10+ states) continuous systems.

Contents

Chapter 1

Introduction

The responsibilities we give to robots, autonomous vehicles, and other cyberphysical systems outpace our ability to reason about the correctness of their behavior. While we may tolerate unanticipated behavior and crashes from our personal computers and cell phones, the incorrect operation of autonomous vehicles could lead to significant loss of life and property. The increasingly tight integration of computation and control in complex systems (e.g., self-driving cars, unmanned aerial vehicles, human-robot collaborative teams, and embedded medical devices) often creates non-trivial failure modes. Thus, there is a growing need for formal methods to specify, design, and verify desired system behavior.

Due to the consequences of unexpected behaviors in many cyberphysical systems, it is important to concisely and unambiguously specify the desired system behavior. Additionally, it is desirable to automatically synthesize a controller that provably implements this behavior. However, current methods for the specification, design, and verification of such hybrid (discrete and continuous state) dynamical systems are *ad hoc*, and may lead to unexpected failures. This thesis draws on ideas from optimization, hybrid systems, and model checking to develop formal methods that guarantee the correct behavior of such systems.

This thesis focuses on the specification and design of controllers that guarantee correct and efficient behaviors of robots, autonomous vehicles, and other cyberphysical systems. These systems need to accomplish complex tasks, e.g., follow the rules of the road, perform surveillance in a dynamic environment, help a human assemble a part, or regulate a physiological process. Temporal logic is an expressive language that can be used to specify these types of complex tasks and properties. These tasks generalize classical point-to-point motion planning. Temporal logic is promising for specifying the combined digital and physical behavior of autonomous systems, in part due to its widespread use in software verification. Many of the results in this thesis will focus on using linear temporal logic (LTL) as a task-specification language.

Formal methods for the specification and verification of software have enjoyed enormous success in both academia and industry. However, formal synthesis of high-performance controllers for hybrid systems carrying out complex tasks in uncertain and adversarial environments remains an open problem. This lack of progress for hybrid systems compared to discrete systems (e.g., software) is largely due to the interaction of the non-convex state constraints arising from the specifications with the continuous dynamics of the system. Uncertainties in the system model and the desire for optimal solutions further complicate the issue. Major problems include the computation of robust controllers, the scalable and optimal synthesis of discrete supervisory controllers, and the computation of controllers (both feasible and optimal) for high-dimensional, nonlinear systems. We have developed new techniques that help overcome these problems. The contributions of this thesis include techniques for optimal and robust control, expressive task specification languages that are also computationally efficient, and algorithms that scale to dynamical systems with more than ten continuous states.

1.1 Thesis Overview and Main Contributions

This thesis presents algorithms for the specification and design of controllers that guarantee correct and efficient behaviors of robots, autonomous vehicles, and other cyberphysical systems. This section gives an overview of the major results of this thesis, focusing on optimal control, robust control, with an emphasis on computational efficient algorithms for high-dimensional dynamical systems.

Optimal and Robust Control

Our first contributions are the extension of a classical control synthesis approach for systems with LTL specifications to optimal control in Chapter 3, and robust control in Chapter 4. These algorithms extend automata-based synthesis techniques for discrete abstractions of dynamical systems to include notions of optimality and robustness. A weighted average cost function is used as a measure of optimality for the optimal control chapter, as it is a natural generalization of an average cost, and it gives the designer more flexibility. An uncertain Markov decision process system model is used in the robust control chapter, as it allows one to model uncertain system dynamics in a natural manner. Surprisingly, synthesizing a control policy that is optimal or robust incurs little computational expense compared to simply computing a feasible controller. The work in these chapters is based on the author's work in [101] and [100], respectively.

Efficient and Optimal Reactive Control

The second contribution of this thesis is the identification of a task specification language that is both expressive and computationally efficient for reactive controller synthesis. A controller that guarantees the correct operation of a system in the presence of a non-deterministic or stochastic environment is said to be *reactive*. Reactive controller synthesis is known to be intractable for LTL specifications [83]. The following questions are answered in Chapter 5:

- *Is there a class of specifications for which one can efficiently compute reactive controllers?*
- *How can one compute optimal controllers?*

Standard automaton-based approaches for reactive controller synthesis scale poorly with problem size, due to the need to construct a deterministic Rabin automaton for the LTL specification [9]. We introduce a class of specifications for which controllers can be synthesized in time linear, instead of doubly-exponential, in the length of the specification. This class is also expressive, and includes specifications such as safe navigation, response to the environment, stability, and surveillance. Specifically, one can compute reactive control policies for non-deterministic transition systems and Markov decision processes in time polynomial in the size of the system and specification. These algorithms are computationally efficient in theory and practice.

In this same setting, we also show how to compute optimal control policies for a variety of relevant cost functions. We identify several important cases when optimal policies (for average and minimax cost functions) can be computed in time polynomial in the size of the system and specification. These algorithms are particularly relevant for online control, as one can guarantee that a feasible solution can be found quickly and then iteratively improve on the quality as time permits. The work in this chapter is based on the author's work in [105] and [102].

Control of High-Dimensional and Nonlinear Systems

The final contribution of this thesis is a pair of methods for computing trajectories for high-dimensional, nonlinear systems with LTL specifications. Standard approaches are based on the process of computing a *discrete abstraction*, which is computationally expensive for high-dimensional and nonlinear systems. Typically, the computation of a discrete abstraction is only feasible for a system with fewer than six continuous dimensions, due to the curse-of-dimensionality associated with such techniques. However, many important systems require more than ten continuous states (e.g., quadrotors and simple aircraft models). The following question is answered in Chapter 5: *Can one compute controllers without the expensive computation of a discrete abstraction?*

The first method uses the automaton representing a specification to guide the computation of a feasible controller for discrete-time nonlinear systems. This method automatically decomposes reasoning about a complex task into a sequence of simpler *constrained reachability* problems. Thus, one can create controllers for any system for which solutions to constrained reachability problems can be computed, e.g., using tools from robotic motion planning and model predictive control. This approach avoids the expensive computation of a discrete abstraction, and is easy to parallelize. Using this method lets us design controllers for nonlinear and high-dimensional (more than ten continuous state) systems with temporal logic specifications, something that was previously not possible. This method is presented in Chapter 6, and is based on the author's work in [106].

The second method is based on encoding temporal logic specifications as mixed-integer linear constraints on a dynamical system. This generalizes previous Boolean satisfiability encodings of temporal logic specifications for finite, discrete systems. This approach directly encodes a linear temporal logic specification as mixed-integer linear constraints on the continuous system variables. Numerical experiments show that this approach scales to previously intractable temporal logic planning problems involving quadrotor and aircraft models. This work is covered in Chapter 7, and is based on the author's work in [99, 103].

Chapter 2

Background

This chapter begins with a brief overview of standard methods for control policy synthesis for systems with temporal logic. We defer the details of related work to the introduction of each chapter. Then, we present basics on linear temporal logic (LTL), a powerful task-specification language that will be used frequently throughout the thesis. This chapter concludes with a brief overview of relevant graph theory.

2.1 Formal Verification and Synthesis

Formal methods are rigorous mathematical techniques specifying and verifying hardware and software systems [9]. A widely used formal method is called *model checking*. Model checking takes as input a formal mathematical description of a system and a specification. Model checking techniques efficiently explore the entire state space of the system, and determine whether any behavior of the system can violate the specification. Any violation is returned as a counterexample—a trace of the system's behavior that explicitly shows how the system fails to satisfy the specification. The theory of model checking for discrete systems is well developed, both in theory [9, 27, 74, 84, 95] and in practice [49, 25, 63].

A widely-used specification language is LTL [84]. LTL allows one to reason about how system properties change over time, and thus specify a wide variety of tasks, such as safety (always avoid B), response (if A, then B), persistence (eventually always stay in A), and recurrence (repeatedly visit A). We will use LTL throughout this thesis

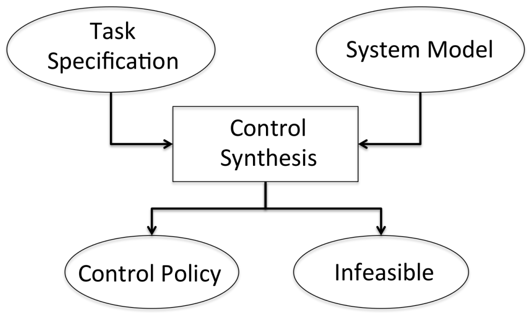


Figure 2.1: High-level view of the control synthesis algorithms presented in this thesis.

due to its prevalence in the formal methods literature, and its ability to express a wide range of useful properties for robotics and cyberphysical systems applications.

Instead of verifying that a pre-built control policy restricts a system's behaviors so that it satisfies a given specification, we will automatically compute such a satisfying control policy (if it exists). Figure 2.1 outlines our approach. Broadly speaking, the different chapters in this thesis consider different combinations of "Task Specifications" and "System Models," and the results are appropriate "Control Synthesis" algorithms.

Formal methods have recently been used in the robotics and control communities to specify desired behaviors for robots and hybrid systems (see Figure 2.2). A common approach is to abstract the original continuous system as a finite discrete system, such as a (non-deterministic) transition system or a Markov decision process (MDP). Sampling-based motion planning techniques can be used for nonlinear systems to create a deterministic transition system that approximates the system, for which a satisfying control policy can be computed [15, 53, 81]. A framework for abstracting a linear system as a discrete transition system, and then constructing a control policy that guarantees that the original system satisfies an LTL specification, is presented in Kloetzer and Belta [61]. Reactive control policies are synthesized for linear systems in the presence of a non-deterministic environment in Kress-Gazit et al. [62], and a receding horizon framework is used in Wongpiromsarn et al. [108] to handle the resulting blow-up in system size. Finally, control policies are created for Markov

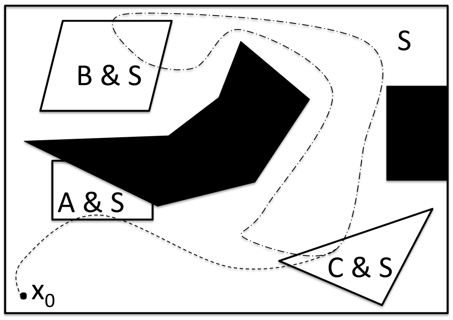


Figure 2.2: Sketch of a system trajectory that satisfies the temporal logic specification $\varphi = \diamond A \wedge \square\diamond B \wedge \square\diamond C \wedge \square S$. Informally, this LTL specification enforces the system to visit A, visit B and C repeatedly, and always remain in the safe region S.

decision processes that represent robots with noisy actuators for both linear temporal logic [34] and probabilistic computational tree logic [66].

While these approaches all generate feasible control policies that satisfy a temporal logic specification, only limited notions of optimality [56, 90] or robustness [72, 93] can be imposed. Furthermore, the complexity of synthesizing a control policy that satisfies an LTL formula is doubly-exponential in the formula length for both non-deterministic and probabilistic systems [30, 83]. Additionally, the creation of a discrete abstraction is typically computationally expensive for high-dimensional continuous systems, due to the curse-of-dimensionality. These limitations are addressed in this thesis.

2.2 Notation

An *atomic proposition* is a statement that is either *True* or *False*. A *propositional formula* is composed of only atomic propositions and propositional connectives, i.e., $\wedge$ (and), $\vee$ (or), and $\neg$ (not). The cardinality of a set X is denoted by $|X|$. Throughout, (in)equality is component-wise for vectors and matrices. Finally, we write $\mathbf{1}$ for a vector of ones of appropriate dimension.

2.3 Linear Temporal Logic

We use LTL to concisely and unambiguously specify the desired system behavior. LTL allows one to generalize the safety and stability properties typically used in controls, hybrid systems, and robotics. We only touch on key aspects of LTL, and defer the reader to [9] for a comprehensive treatment.

Syntax: LTL is built from (a) a set of atomic propositions AP, (b) Boolean operators: $\wedge$ (and), $\vee$ (or), and $\neg$ (not), and (c) temporal operators: $\bigcirc$ (next) and $\mathcal{U}$ (until). An LTL formula is defined by the following grammar in Backus-Naur Form:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \, \mathcal{U} \, \varphi_2,$$

where $p \in AP$ is an atomic proposition.

The standard Boolean operators $\implies$ (implies) and $\iff$ (equivalent) can be defined as $p \implies q = \neg p \vee q$ and $p \iff q = (p \implies q) \wedge (q \implies p)$, respectively. Commonly-used abbreviations for LTL formulas are the derived temporal operators $\diamond\psi = \mathit{True} \, \mathcal{U} \, \psi$ (eventually), $\square\psi = \neg \diamond \neg\psi$ (always), $\square \diamond \psi$ (always eventually), and $\diamond \square \psi$ (eventually always).

Semantics: To define the semantics of LTL, we must first define an abstract system model. Let $\mathcal{T}$ be a system with a (possibly infinite) set of states S. A *labeling function* $L : S \to 2^{AP}$ maps each state to a set of atomic propositions from AP that are *True*. A *run* of $\mathcal{T}$ is an infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots$, where $s_i \in S$ for $i = 0, 1, \ldots$. Let $\sigma_i = s_i s_{i+1} s_{i+2} \ldots$ denote the run σ from position i. A run σ induces a *word* the word $L(\sigma) = L(s_0)L(s_1)L(s_2)\ldots$. The semantics of LTL are defined inductively as follows:

$\sigma_i \vDash p$ if and only if $p \in L(s_i)$

$\sigma_i \vDash \neg\varphi$ if and only if $\sigma_i \nvDash \varphi$

$\sigma_i \vDash \varphi_1 \vee \varphi_2$ if and only if $\sigma_i \vDash \varphi_1 \vee \sigma_i \vDash \varphi_2$

$\sigma_i \vDash \varphi_1 \wedge \varphi_2$ if and only if $\sigma_i \vDash \varphi_1 \wedge \sigma_i \vDash \varphi_2$

$\sigma_i \vDash \bigcirc\varphi$ if and only if $\sigma_{i+1} \vDash \varphi$

$\sigma_i \vDash \varphi_1 \,\mathcal{U}\, \varphi_2$ if and only if $\exists j \geq i$ s.t. $\sigma_j \vDash \varphi_2$ and $\sigma_n \vDash \varphi_1 \forall i \leq n < j$

Informally, the notation $\bigcirc\varphi$ means that φ is true at the next step, $\square\varphi$ means that φ is always true, $\diamond\varphi$ means that φ is eventually true, $\square\diamond\varphi$ means that φ is true infinitely often, and $\diamond\square\varphi$ means that φ is eventually always true [9]. More formally, $\square\varphi$ holds at position i if and only if φ holds at every position in σ starting at position i, $\diamond\varphi$ holds at position i if and only if φ holds at some position $j \geq i$ in σ, $\square\diamond\varphi$ holds at position i if and only if φ holds at infinitely many positions $j \geq i$ in σ, and $\diamond\square\varphi$ holds at position i if and only if there exists some position $j \geq i$ such that φ holds at every position in σ starting at position j.

Definition 2.1. A run $\sigma = s_0 s_1 s_2 \ldots$ *satisfies* φ, denoted by $\sigma \vDash \varphi$, if $\sigma_0 \vDash \varphi$.

A *propositional formula* ψ is composed of only atomic propositions and propositional connectives. We denote the set of states where ψ holds by $[\![\psi]\!]$.

An Automaton Perspective

LTL is an ω-regular language, which is a regular language extended to allow infinite repetition (denoted by ω). Non-deterministic Büchi automata (hereafter called Büchi automata) accept the class of languages equivalent to ω-regular languages. Since LTL is a subset of ω-regular languages, any LTL formula φ can be automatically translated into a corresponding Büchi automaton $\mathcal{A}_\varphi$ [9]. Figure 2.3 shows an example of a Büchi automaton.

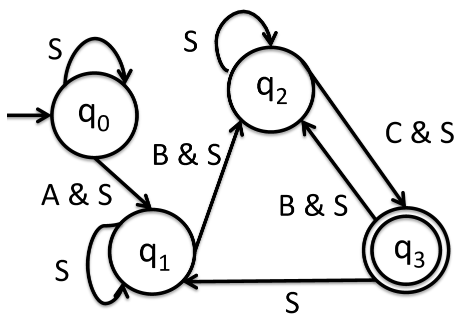


Figure 2.3: A (simplified) Büchi automaton corresponding to the LTL formula $\varphi = \Diamond A \wedge \Box\Diamond B \wedge \Box\Diamond C \wedge \Box S$. Informally, the system must visit A, repeatedly visit B and C, and always remain in S. Here $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{A, B, C, S\}$, $Q_0 = \{q_0\}$, $F = \{q_3\}$, and transitions are represented by labeled arrows.

Definition 2.2. A Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ consisting of (i) a finite set of states Q, (ii) a finite alphabet Σ, (iii) a transition relation $\delta \subseteq Q \times \Sigma \times Q$, (iv) a set of initial states $Q_0 \subseteq Q$, (v) and a set of accepting states $F \subseteq Q$.

Let Σ^ω be the set of infinite words over Σ. A word $L(\sigma) = \Sigma_0\Sigma_1\Sigma_2 \ldots \in \Sigma^\omega$ induces an infinite sequence $q_0q_1q_2\ldots$ of states in $\mathcal{A}$ such that $q_0 \in Q_0$ and $(q_i, \Sigma_i, q_{i+1}) \in \delta$ for $i \geq 0$. Run $q_0q_1q_2\ldots$ is *accepting (accepted)* if $q_i \in F$ for infinitely many indices $i \in \mathbb{N}$ appearing in the run.

Intuitively, a run is accepted by a Büchi automaton if an accepting state, i.e., a state in F, is visited infinitely often.

The length of an LTL formula φ is the number of symbols, and is denoted by $|\varphi|$. A corresponding Büchi automaton $\mathcal{A}_\varphi$ has size $2^{O(|\varphi|)}$ in the worst-case, but this behavior is rarely encountered in practice.

We use the definition of an accepting run in a Büchi automaton and the fact that every LTL formula φ can be represented by an equivalent Büchi automaton $\mathcal{A}_\varphi$ to define satisfaction of an LTL formula φ.

Definition 2.3. Let $\mathcal{A}_\varphi$ be a Büchi automaton corresponding to the LTL formula φ. A run $\sigma = s_0s_1s_2\ldots$ of $\mathcal{T}$ *satisfies* φ, denoted by $\sigma \vDash \varphi$, if the word $L(\sigma)$ is accepted by $\mathcal{A}_\varphi$.

For non-deterministic or stochastic systems, it is necessary to represent an LTL formula as an automaton with deterministic transitions. A deterministic Rabin automaton is therefore used instead of a Büchi automaton. Any LTL formula φ can be automatically translated into a deterministic Rabin automaton of size at most $2^{2^{O(|\varphi|)}}$ [9].

Definition 2.4. A *deterministic Rabin automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the initial state, and accepting state pairs $F \subseteq 2^Q \times 2^Q$.

Let Σ^ω be the set of infinite words over Σ. A word $L(\sigma) = \Sigma_0\Sigma_1\Sigma_2 \ldots \in \Sigma^\omega$ denotes an infinite sequence $q_0q_1q_2\ldots$ of states in $\mathcal{A}$ such that $q_{i+1} = \delta(q_i, \Sigma_i)$ for $i \geq 0$. The run $q_0q_1q_2\ldots$ is accepting if there exists a pair $(J, K) \in F$ and an $n \geq 0$, such that for all $m \geq n$ we have $q_m \notin J$, and there exist infinitely many indices k such that $q_k \in K$.

Intuitively, a run is accepted by a deterministic Rabin automaton if the set of states J is visited finitely often, and the set of states K is visited infinitely often.

2.4 Graph Theory

This section lists basic definitions for graphs that will be used throughout this thesis. Let $G = (V, E)$ be a directed graph (digraph) with $|V|$ vertices and $|E|$ edges. Let $e = (u, v) \in E$ denote an edge from vertex u to vertex v. A *walk* is a finite edge sequence $e_0, e_1, \ldots, e_p$, and a *cycle* is a walk in which the initial vertex is equal to the final vertex. A *path* is a walk with no repeated vertices, and a *simple cycle* is a path in which the initial vertex is equal to the final vertex.

A digraph $G = (V, E)$ is *strongly connected* if there exists a path between each pair of vertices $s, t \in V$. A digraph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. The subgraph of G restricted to states $V' \subseteq V$ is denoted by $G|_{V'}$. A digraph $G' \subseteq G$ is a *strongly connected component* if it is a maximal strongly connected subgraph of G.

Chapter 3

Optimal Control with Weighted Average Costs and Temporal Logic Specifications

This chapter concerns optimal control for a system subject to temporal logic constraints. We minimize a weighted average cost function that generalizes the commonly used average cost function from discrete-time optimal control. Dynamic programming algorithms are used to construct an optimal trajectory for the system that minimizes the cost function while satisfying a temporal logic specification. Constructing an optimal trajectory takes only polynomially more time than constructing a feasible trajectory. We demonstrate our methods on simulations of autonomous driving and robotic surveillance tasks. This chapter is based on results from [101].

3.1 Introduction

Often, there are numerous control policies for a system that satisfy a given temporal logic specification, so it is desirable to select one that is optimal with respect to some cost function, e.g., time or fuel consumption. Since temporal logic specifications include properties that must be satisfied over infinite state sequences, it is important that the form of the cost function is also well-defined over infinite sequences. We consider an average cost, which is bounded under certain mild assumptions discussed in Section 3.3. Additionally, it may be desired to give varying weights to different behaviors, i.e., repeatedly visit a set of regions, but visit a high-weight region more often than others. Thus, we minimize a weighted average cost function over system trajectories subject to the constraint that a given temporal logic specification is satisfied. This cost function generalizes the average cost-per-stage cost function commonly studied in discrete-time optimal control [13].

Optimality has been considered in the related area of vehicle routing [94]. Vehicle routing problems generalize the traveling salesman problem, and are thus NP-complete. A different approach to control with LTL specifications converts the controller design problem into a mixed-integer linear program [55, 99, 103]. We discuss such formulations in detail in Chapter 7. Chatterjee et al. [22] create control policies that minimize an average cost function in the presence of an adversary. The approach in Smith et al. [90] is the most closely related to our work. Motivated by surveillance tasks, they minimize the maximum cost between visiting specific regions. Our work is complementary to [90] in that we instead minimize a weighted average cost function.

The main contribution of this chapter is a solution to the problem of creating a system trajectory that minimizes a weighted average cost function subject to temporal logic constraints. We solve this problem for weighted transition systems by searching for system trajectories in the product automaton, a lifted space that contains only behaviors that are valid for the transition system and also satisfy the temporal logic specification. An optimal system trajectory corresponds to a cycle in the product automaton, which is related to the well-studied cost-to-time ratio problem in operations research [31, 32, 48, 57, 58]. We give computationally efficient dynamic programming algorithms for finding the optimal system trajectory. In fact, it takes only polynomially more effort to calculate an optimal solution than is required for a feasible solution, i.e., one that just satisfies the specification.

3.2 Weighted Transition Systems

We use finite transition systems to model the system behavior. In robotics, however, one is usually concerned with continuous systems that may have complex dynamic constraints. This gap is partially bridged by constructive procedures for abstracting relevant classes of continuous systems, including unicycle models, as finite transition systems [11, 10, 47]. Additionally, sampling-based methods, such as rapidly-exploring random trees [67] and probabilistic roadmaps [59], gradually build a finite transition system that approximates a continuous system, and have been studied in this context [53, 81]. Examples of how one can abstract continuous dynamics by discrete transition systems are given in [10, 11, 47, 53, 81].

Definition 3.1. A *weighted (finite) transition system* is a tuple $\mathcal{T} = (S, R, s_0, AP, L, c, w)$ consisting of (i) a finite set of states S, (ii) a transition relation $R \subseteq S \times S$, (iii) an initial state $s_0 \in S$, (iv) a set of atomic propositions AP, (v) a labeling function $L : S \to 2^{AP}$, (vi) a cost function $c : R \to \mathbb{R}$, (vii) and a weight function $w : R \to \mathbb{R}_{\geq 0}$.

We assume that the transition system is non-blocking, so for each state $s \in S$, there exists a state $t \in S$ such that $(s, t) \in R$.

A *run* of the transition system is an infinite sequence of its states, $\sigma = s_0 s_1 s_2 \ldots$ where $s_i \in S$ is the state of the system at index i, and $(s_i, s_{i+1}) \in R$ for $i = 0, 1, \ldots$. A *word* is an infinite sequence of labels $L(\sigma) = L(s_0)L(s_1)L(s_2)\ldots$ where $\sigma = s_0 s_1 s_2 \ldots$ is a run.

Let S_k^+ be the set of all runs up to index k. An *infinite-memory* control policy is denoted by $\pi = (\mu_0, \mu_1, \ldots)$ where $\mu_k : S_k^+ \to R$ maps a partial run $s_0 s_1 \ldots s_k \in S_k^+$ to a new transition. A policy $\pi = (\mu, \mu, \ldots)$ is *finite-memory* if $\mu : S \times M \to R \times M$, where the finite set M is called the memory.

For the deterministic transition system models we consider, the run of a transition system implicitly encodes the control policy. An *infinite-memory* run is a run that can be implemented by an infinite-memory policy. A *finite-memory* run is a run that can be implemented by a finite-memory policy.

When possible, we abuse notation and refer to the cost $c(t)$ of a state $t \in S$, instead of a transition between states. In this case, we enforce that $c(s, t) = c(t)$ for all transitions $(s, t) \in R$, i.e., the cost of the state is mapped to all incoming transitions. Similar notational simplification is used for weights, and should be clear from context.

The cost function c can be viewed concretely as a physical cost of a transition between states, such as time or fuel. This cost can be negative for some transitions, which could, for example, correspond to refueling if the cost is fuel consumption. The weight function w can be viewed as the importance of each transition, which is a flexible design parameter. In the sequel, we will create a run with the minimal weighted average cost. Thus, the designer can give a higher weight to transitions that she thinks are preferable. As an example, consider an autonomous car that is supposed to visit different locations in a city while obeying the rules-of-the-road. In this case, a task specification would encode the locations that should be visited, along with the rules-of-the-road. Costs might be the time required to traverse different roads. Weights might encode preferences, such as visiting certain landmarks. An example scenario is discussed in detail in Section 3.6.

3.3 Problem Statement

In this section, we formally state the main problem of this chapter, and give an overview of our solution approach. Let $\mathcal{T} = (S, R, s_0, AP, L, c, w)$ be a weighted transition system, and φ be an LTL specification defined over AP.

Definition 3.2. Let $\sigma = s_0 s_1 \ldots$ be a run of $\mathcal{T}$ where s_i is the state at the i-th index of σ. The *weighted average cost* of run σ is

$$J(\sigma) := \limsup_{n \to \infty} \frac{\sum_{i=0}^{n} c(s_i, s_{i+1})}{\sum_{i=0}^{n} w(s_i, s_{i+1})}, \tag{3.1}$$

where J maps runs of $\mathcal{T}$ to $\mathbb{R} \cup \infty$.

Since LTL specifications are typically defined over infinite sequences of states, we consider the (weighted) average cost function in (3.1) to ensure that the cost function is bounded. This cost function is well-defined when (i) $c(s_i, s_{i+1}) < \infty$ for all $i \geq 0$, and (ii) there exists a $j \in \mathbb{N}$ such that $w(s_i, s_{i+1}) > 0$ for infinitely many $i \geq j$, which we assume is true for the sequel. Assumption (ii) enforces that a run does not eventually visit only those states with zero weights.

To better understand the weighted average cost function J, consider the case where $w(s,t) = 1$ for all transitions $(s,t) \in R$. Let a cost $c(s,t)$ be arbitrarily fixed for each transition $(s,t) \in R$. Then, $J(\sigma)$ is the average cost per transition between states (or average cost-per-stage). If $w(s,t) = 1$ for states in $s,t \in S' \subset S$ and $w(s,t) = 0$ for states in $S - S'$, then $J(\sigma)$ is the mean time per transition between states in S'.

As an example, consider $\sigma = (s_0 s_1)^\omega$, where $s_0 s_1$ repeats indefinitely. Let $c(s_0, s_1) = 1$, $c(s_1, s_0) = 2$, $w(s_0, s_1) = 1$, and $w(s_1, s_0) = 1$. Then, $J(\sigma) = 1.5$ is the average cost per transition. Now, let $w(s_1, s_0) = 0$. Then, $J(\sigma) = 3$ is the average cost per transition from s_0 to s_1.

The weighted average cost function is more natural than the minimax cost function of Smith et al. [90] in some applications. For example, consider an autonomous vehicle repeatedly picking people up and delivering them to a destination. It takes a certain amount of fuel to travel between discrete states, and each discrete state has a fixed number of people that need to be picked up. A natural problem formulation is to minimize the fuel consumption per person picked up, which is a weighted average cost where fuel is the cost, and the number of people is the weight. The cost function in Smith et al. [90] cannot adequately capture this task.

Definition 3.3. An *optimal satisfying finite-memory run* of $\mathcal{T}$ is a run σ^* such that

$$J(\sigma^*) = \inf \{ J(\sigma) \mid \sigma \text{ is a finite-memory run of } \mathcal{T} \text{ and } \sigma \vDash \varphi \}, \tag{3.2}$$

i.e., run σ^* achieves the infimum in (3.2).

An *optimal satisfying infinite-memory run* is defined similarly for infinite-memory runs of $\mathcal{T}$.

Although we show in Section 3.5.2 that infinite-memory runs are generally necessary to achieve the infimum in equation (3.2), we focus on finite-memory runs, as these are more practical than their infinite-memory counterparts. However, finding an optimal satisfying finite-memory run is potentially ill-posed, as the infimum might not be achieved due to the constraint that the run must also satisfy φ. This happens when it is possible to reduce the cost of a satisfying run by including an arbitrarily long, low weighted average cost subsequence. For instance, consider the run $\sigma = (s_0 s_0 s_1)^\omega$. Let $c(s_0, s_0) = 1$, $c(s_1, s_0) = c(s_0, s_1) = 2$, and the weights equal 1 for each transition. Assume that a specification is satisfied if s_1 is visited infinitely often. Then, $J(\sigma)$ can be reduced by including an arbitrarily large number of self transitions from s_0 to s_0 in σ, even though these do not affect satisfaction of the specification. Intuitively, one should restrict these repetitions to make finding an optimal satisfying finite-memory run well-posed. We will show that one can always compute an ϵ-suboptimal finite-memory run by restricting the length of these repetitions. The details are deferred to Section 3.4, when we will have developed the necessary technical machinery.

Problem 3.1. Given a weighted transition system $\mathcal{T}$ and an LTL specification φ, compute an optimal satisfying finite-memory run σ^* of $\mathcal{T}$ if one exists.

Remark 3.1. For the sake of completeness, we show how to compute optimal satisfying infinite-memory runs in Section 3.5.2. These runs achieve the minimal weighted average cost, but do so by adding arbitrarily long progressions of states that do not change whether or not the specification is satisfied.

3.4 Reformulation of the Problem

We solve Problem 3.1 by first creating a product automaton that represents runs that are allowed by the transition system $\mathcal{T}$, and also satisfy the LTL specification φ. We can limit our search for finite-memory runs, without loss of generality, to runs in the product automaton that are of the form $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$. Here, σ_{pre} is a finite walk, and σ_{suf} is a finite cycle that is repeated infinitely often. Runs with this structure are said to be in *prefix-suffix* form. As the weighted average cost only depends on σ_{suf}, which reduces the problem to searching for a cycle σ_{suf} in the product automaton. This search can be done using dynamic programming techniques. The optimal accepting run $\sigma_{\mathcal{P}}^*$ is then projected back on $\mathcal{T}$ as σ^*, which solves Problem 3.1.

3.4.1 Product Automaton

We use the standard product automaton construction, due to Vardi and Wolper [95], to represent runs that are allowed by the transition system and also satisfy the LTL specification.

Definition 3.4. Let $\mathcal{T} = (S, R, s_0, AP, L, c, w)$ be a weighted transition system, and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ be a Büchi automaton. The *product automaton* $\mathcal{P} = \mathcal{T} \times \mathcal{A}$ is the tuple $\mathcal{P} := (S_\mathcal{P}, \delta_\mathcal{P}, F_\mathcal{P}, s_{\mathcal{P},0}, AP_\mathcal{P}, L_\mathcal{P}, c_\mathcal{P}, w_\mathcal{P})$, consisting of

(i) a finite set of states $S_\mathcal{P} = S \times Q$,

(ii) a transition relation $\delta_\mathcal{P} \subseteq S_\mathcal{P} \times S_\mathcal{P}$, where $((s, q), (s', q')) \in \delta_\mathcal{P}$ if and only if $(s, s') \in R$ and $(q, L(s), q') \in \delta$,

(iii) a set of accepting states $F_\mathcal{P} = S \times F$,

(iv) a set of initial states $S_{\mathcal{P},0}$, with $(s_0, q_0) \in S_{\mathcal{P},0}$ if $q_0 \in Q_0$,

(v) a set of atomic propositions $AP_\mathcal{P} = Q$,

(vi) a labeling function $L_\mathcal{P} : S \times Q \to 2^Q$,

(vii) a cost function $c_\mathcal{P} : \delta_\mathcal{P} \to \mathbb{R}$, where $c_\mathcal{P}((s, q), (s', q')) = c(s, s')$ for all $((s, q), (s', q')) \in \delta_\mathcal{P}$, and

(viii) a weight function $w_\mathcal{P} : \delta_\mathcal{P} \to \mathbb{R}_{\geq 0}$, where $w_\mathcal{P}((s, q), (s', q')) = w(s, s')$ for all $((s, q), (s', q')) \in \delta_\mathcal{P}$.

A run $\sigma_\mathcal{P} = (s_0, q_0)(s_1, q_1)\ldots$ is *accepting* if $(s_i, q_i) \in F_\mathcal{P}$ for infinitely many indices $i \in \mathbb{N}$.

The *projection* of a run $\sigma_\mathcal{P} = (s_0, q_0)(s_1, q_1)\ldots$ in the product automaton $\mathcal{P}$ is the run $\sigma = s_0 s_1 \ldots$ in the transition system $\mathcal{T}$. The projection of a finite-memory run in $\mathcal{P}$ is a finite-memory run in $\mathcal{T}$ [9].

The following proposition relates accepting runs in $\mathcal{T}$ and $\mathcal{P}$, and is due to Vardi and Wolper [95].

Proposition 3.1. *(Vardi and Wolper [95]) Let $\mathcal{A}_\varphi$ be a Büchi automaton corresponding to the LTL formula φ. For any accepting run $\sigma_\mathcal{P} = (s_0, q_0)(s_1, q_1)\dots$ in the product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\varphi$, its projection $\sigma = s_0 s_1 \dots$ in the transition system $\mathcal{T}$ satisfies φ. Conversely, for any run $\sigma = s_0 s_1 \dots$ in $\mathcal{T}$ that satisfies φ, there exists an accepting run $\sigma_\mathcal{P} = (s_0, q_0)(s_1, q_1)\dots$ in the product automaton.*

Lemma 3.1. *For any accepting run $\sigma_\mathcal{P}$ in $\mathcal{P}$ and its projection σ in $\mathcal{T}$, $J(\sigma_\mathcal{P}) = J(\sigma)$. Conversely, for any σ in $\mathcal{T}$ that satisfies φ, there exists an accepting run $\sigma_\mathcal{P}$ in $\mathcal{P}$ with $J(\sigma_\mathcal{P}) = J(\sigma)$.*

Proof. Consider a run $\sigma_\mathcal{P} = (s_0, q_0)(s_1, q_1)\dots$ in $\mathcal{P}$. By definition, for states $(s_i, q_i), (s_{i+1}, q_{i+1}) \in S_\mathcal{P}$ and $s_i, s_{i+1} \in S_\mathcal{T}$, the cost $c_\mathcal{P}((s_i, q_i), (s_{i+1}, q_{i+1})) = c(s_i, s_{i+1})$, and the weight $w_\mathcal{P}((s_i, q_i), (s_{i+1}, q_{i+1})) = w(s_i, s_{i+1})$ for all $i \geq 0$, so $J(\sigma_\mathcal{P}) = J(\sigma)$. Now, consider a run $\sigma = s_0 s_1 \dots$ in $\mathcal{T}$ that satisfies φ. Proposition 3.1 gives the existence of an accepting run $\sigma_\mathcal{P} = (s_0, q_0)(s_1, q_1)\dots$ in $\mathcal{P}$, and so $J(\sigma_\mathcal{P}) = J(\sigma)$. □

By Lemma 3.1, an accepting run $\sigma^*_\mathcal{P}$ with minimal weighted average cost in the product automaton has a projection in the transition system σ^* that is a satisfying run with minimal weighted average cost.

3.4.2 Prefix-Suffix Form

We show that Problem 3.1 is equivalent to finding a run of the form $\sigma_\mathcal{P} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$, in the product automaton $\mathcal{P}$ that minimizes the weighted average cost function (3.1). We equivalently treat the product automaton as a graph when convenient (see Section 2.4). Our analysis and notation in this section is similar to that of [90]; we optimize a different cost function on the Vardi and Wolper [95] product automaton construction.

Definition 3.5. Let σ_{pre} be a finite walk in $\mathcal{P}$, and σ_{suf} be a finite cycle in $\mathcal{P}$. A run $\sigma_\mathcal{P}$ is in *prefix-suffix* form if it is of the form $\sigma_\mathcal{P} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$.

It is well-known that if there exists an accepting run in $\mathcal{P}$ for an LTL formula φ, then there exists an accepting run in prefix-suffix form for φ [9]. This can be seen since the product automaton $\mathcal{P}$ is finite, but an accepting run is infinite, and visits an accepting state infinitely often. Thus, at least one accepting state must be visited infinitely often, and this can correspond to a repeated cycle including the accepting state. For an accepting run $\sigma_{\mathcal{P}}$, the suffix σ_{suf} is a cycle in the product automaton $\mathcal{P}$ that satisfies the acceptance condition, i.e., it includes an accepting state. The prefix σ_{pre} is a finite run from an initial state $s_{\mathcal{P},0}$ to a state on an accepting cycle.

The following lemma shows that a minimum weighted average cost finite-memory run can be found searching over finite-memory runs of the form $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^{\omega}$.

Lemma 3.2. *For any accepting finite-memory run $\sigma_{\mathcal{P}}$ of $\mathcal{P}$ with cost J, there exists an accepting finite-memory run in prefix-suffix form with cost at most J.*

Proof. Let σ_{gen} be an accepting finite-memory run in $\mathcal{P}$ that is not in prefix-suffix form and has weighted average cost $J(\sigma_{\text{gen}})$. Since σ_{gen} is accepting, it must visit an accepting state $s_{\text{acc}} \in S_{\mathcal{P}}$ infinitely often. Let the finite walk σ_{pre} be from an initial state $s_{\mathcal{P},0}$ to the first visit of s_{acc}. Now, consider the set of walks between successive visits to s_{acc}. Each walk starts and ends at s_{acc} (so it is a cycle), is finite with bounded length, and has a weighted average cost associated with it. For each cycle τ, compute the weighted average cost $J(\tau^{\omega})$. Let σ_{suf} be the finite cycle with the minimum weighted average cost over all τ. Then, $J(\sigma_{\mathcal{P}}) = J(\sigma_{\text{pre}}(\sigma_{\text{suf}})^{\omega}) \leq J(\sigma_{\text{gen}})$. Since σ_{gen} was arbitrary, the claim follows. □

The next proposition shows that the weighted average cost of a run does not depend on any finite prefix of the run.

Proposition 3.2. *Let $\sigma = s_0 s_1 \ldots$ be a run (in $\mathcal{T}$ or $\mathcal{P}$), and $\sigma_k = s_k s_{k+1} \ldots$ be the run σ starting at index $k \in \mathbb{N}$. Then, their weighted average costs are equal, i.e., $J(\sigma) = J(\sigma_k)$.*

Proof. From Definition 3.1, costs and weights depend only on the transition—not the index. Also, from the assumptions that directly follow equation (3.1), transitions with positive weight occur infinitely often. Thus,

$$\begin{aligned} J(\sigma) &:= \limsup_{n\to\infty} \frac{\sum_{i=0}^{n} c(s_i, s_{i+1})}{\sum_{i=0}^{n} w(s_i, s_{i+1})} \\ &= \limsup_{n\to\infty} \frac{\sum_{i=0}^{k-1} c(s_i, s_{i+1}) + \sum_{i=k}^{n} c(s_i, s_{i+1})}{\sum_{i=0}^{k-1} w(s_i, s_{i+1}) + \sum_{i=k}^{n} w(s_i, s_{i+1})} \\ &= \limsup_{n\to\infty} \frac{\sum_{i=k}^{n} c(s_i, s_{i+1})}{\sum_{i=k}^{n} w(s_i, s_{i+1})} = J(\sigma_k). \end{aligned}$$

□

From Proposition 3.2, finite prefixes do not contribute to the weighted average cost function, so $J(\sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega) = J((\sigma_{\text{suf}})^\omega)$. Thus, one can optimize over the suffix σ_{suf}, which corresponds to an accepting cycle in the product automaton. Given an optimal accepting cycle σ^*_{suf}, one then computes a walk from an initial state to σ^*_{suf}.

We now define a weighted average cost function for finite walks in the product automaton that is analogous to (3.1).

Definition 3.6. The *weighted average cost* of a finite walk $\sigma_{\mathcal{P}} = (s_0, q_0)(s_1, q_1)\ldots(s_m, q_m)$ in the product automaton is

$$\tilde{J}(\sigma_{\mathcal{P}}) := \frac{\sum_{i=0}^{m} c_{\mathcal{P}}(s_i, s_{i+1})}{\sum_{i=0}^{m} w_{\mathcal{P}}(s_i, s_{i+1})}, \tag{3.3}$$

with similar assumptions on c and w as for equation (3.1).

Problem 3.2. Let $acc(\mathcal{P})$ be the set of all accepting cycles in the product automaton $\mathcal{P}$ reachable from an initial state. Find a suffix σ^*_{suf} where $\tilde{J}(\sigma^*_{\text{suf}}) = \inf_{\sigma_{\mathcal{P}} \in acc(\mathcal{P})} \tilde{J}(\sigma_{\mathcal{P}})$ if it exists.

Proposition 3.3. *Let $\sigma^*_{\mathcal{P}} = \sigma_{pre}(\sigma^*_{suf})^\omega$ be a solution to Problem 3.2. The projection to the transition system of any optimal accepting run $\sigma^*_{\mathcal{P}}$ is a solution to Problem 3.1.*

Proof. From Lemma 3.2, there exists an accepting run $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$ that minimizes J. From Proposition 3.2 and equation (3.5), $J(\sigma_{\mathcal{P}}) = J((\sigma_{\text{suf}})^\omega) = \tilde{J}(\sigma_{\text{suf}})$. □

We now pause to give a high-level overview of our approach to solving Problem 3.1, using its reformulation as Problem 3.2. The major steps are outlined in Algorithm 1.

First, a Büchi automaton $\mathcal{A}_\varphi$ corresponding to the LTL formula φ is created. Then, we create the product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\varphi$. Reachability analysis on $\mathcal{P}$ determines, in time linear in the size of $\mathcal{P}$, all states that can be reached from an initial state, and thus guarantees existence of a finite prefix σ_{pre} to all remaining states. Next, we compute the strongly connected components (SCCs) of $\mathcal{P}$, since two states can be on the same cycle only if they are in the same strongly connected component. This partitions the original product automaton into sub-graphs, each of which can be searched independently for optimal cycles.

For each strongly connected component of $\mathcal{P}$, we compute the cycle σ_{suf} with the minimum weighted average cost, regardless of whether or not it is accepting (see Section 3.5.2). This is the infimum of the minimum weighted average cost over all accepting cycles. If this cycle is accepting, then the infimum is achieved by a finite-memory run. If not, then the infimum is not achieved by a finite-memory run, and thus, we must further constrain the form of the suffix σ_{suf} to make the optimization well-posed.

A natural choice is finite-memory policies, which correspond to bounding the length of σ_{suf}. We can solve for an optimal accepting σ_{suf} subject to this additional constraint using dynamic programming techniques. The optimal accepting σ_{suf} over all strongly connected components is σ^*_{suf}. Given σ^*_{suf}, we compute a finite walk σ_{pre} from an initial state to any state on σ^*_{suf}. The finite walk σ_{pre} is guaranteed to exist due to the initial reachability computation. The optimal run in the product automaton is then $\sigma^*_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma^*_{\text{suf}})^\omega$. The projection of $\sigma^*_{\mathcal{P}}$ to the transition system as σ^* solves Problem 3.1, given the additional constraint that σ_{suf} has bounded length.

Remark 3.2. In Section 3.5, we treat the product automaton as a graph $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}})$, with the natural bijections between states $S_{\mathcal{P}}$ and vertices $V_{\mathcal{P}}$, and between edges $(u, v) \in E_{\mathcal{P}}$ and transitions in δ_P. We further assume that a reachability computation has been done, so that $G_{\mathcal{P}}$ only includes states reachable from an initial state $s_{\mathcal{P},0}$. We assume that $G_{\mathcal{P}}$ is strongly connected. If not, the strongly connected components of the $\mathcal{P}$ can be found in $O(|V_{\mathcal{P}}| + |E_{\mathcal{P}}|)$ time with Tarjan's algorithm [29]. To compute the optimal cycle for the entire graph, one finds the optimal cycle

Algorithm 1 Overview of Optimal Controller Synthesis

```
Input: Weighted transition system $\mathcal{T}$ and LTL formula $\varphi$
Output: Run $\sigma^*$, a solution to Problem 3.1
 1: Create Büchi automaton $\mathcal{A}_\varphi$
 2: Create product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\varphi$
 3: Compute states in $\mathcal{P}$ reachable from an initial state
 4: Calculate strongly connected components (SCCs) of $\mathcal{P}$
 5: for scc $\in \mathcal{P}$ do
 6:     Let $\sigma^*_{\text{suf}} = \arg\inf \left\{ \tilde{J}(\sigma) \mid \sigma \text{ is cycle in } \mathcal{P} \right\}$
 7:     if $\sigma^*_{\text{suf}}$ is an accepting cycle then
 8:         break {finite-memory run achieves infimum}
 9:     end if
10:     Find best bounded-length accepting $\sigma^*_{\text{suf}}$ over all $s_{\text{acc}} \in$ SCCs (Section 3.5)
11: end for
12: Take optimal $\sigma^*_{\text{suf}}$ over all SCCs
13: Compute finite prefix $\sigma_{\text{pre}}$ from initial state to $\sigma^*_{\text{suf}}$
14: Project run $\sigma^*_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma^*_{\text{suf}})^\omega$ to $\mathcal{T}$ as $\sigma^*$
```

in each strongly connected component, and then selects the optimal over all strongly connected components. We denote each strongly connected component of $G_\mathcal{P}$ by $G = (V, E)$, where $n = |V|$ and $m = |E|$.

3.5 Solution Approach

In this section, we give algorithms for computing optimal finite-memory and infinite-memory runs. We assume that $G = (V, E)$ is a strongly connected component of the product automaton $\mathcal{P}$ and has at least one accepting state. The techniques we adapt were originally developed for the minimum cost-to-time ratio problem [31, 32, 48, 57, 58].

3.5.1 Computing Finite-Memory Runs

We present two related algorithms (that find an optimal accepting cycle σ^*_{suf}) in increasing levels of generality. While the algorithm in Section 3.5.1.2 subsumes the algorithm in Section 3.5.1.1, the latter is more intuitive. It is also more computationally efficient when the weight function is constant.

3.5.1.1 Minimum Mean Cycle

We first investigate the case where $w(e) = 1$ for all $e \in E$, so the total weight of a walk is equivalent to the number of transitions. This is similar to the problem investigated by Karp [57], with the additional constraint that the cycle must be accepting. This additional constraint prevents a direct application of Karp's theorem [57], but our approach is similar. The intuition is that, conditional on the weight of a walk, the minimum cost walk gives the minimum average cost walk.

Let $s \in V$ be an accepting vertex (i.e., accepting state). For every $v \in V$, let $F_k(v)$ be the minimum cost of a walk of length $k \in \mathbb{N}$ from s to v. Thus, $F_k(s)$ is the minimum cost cycle of length k, which we note is accepting by construction. We compute $F_k(v)$ for all $v \in V$ and $k = 1, \ldots, n$ by the recurrence

$$F_k(v) = \min_{(u,v) \in E} \left[F_{k-1}(u) + c(u,v) \right], \tag{3.4}$$

where $F_0(s) = 0$ and $F_0(v) = \infty$ for $v \neq s$.

It follows from equation (3.4) that $F_k(v)$ can be computed for all $v \in V$ in $O(|V||E|)$ operations. To find the minimum mean cycle cost with fewer than M transitions (i.e., bounded-length suffix), simply compute $\min F_k(s)/k$ for all $k = 1, \ldots, M$. If there are multiple cycles with the optimal cost, pick the cycle corresponding to the minimum k.

We repeat the above procedure for each accepting vertex $s \in V$. The minimum mean cycle value is the minimum of these values. We record the optimal vertex s^*, and the corresponding integer k^*. To determine the optimal cycle corresponding to s^* and k^*, we simply determine the corresponding transitions from (3.4) for $F_{k^*}(s^*)$ from vertex s^*. The repeated application of recurrence (3.4) takes $O(n_a|V||E|)$ operations, where n_a is the number of accepting vertices, which is typically significantly smaller than $|V|$.

3.5.1.2 Minimum Cycle Ratio

We now discuss a more general case, which subsumes the discussion in Section 3.5.1.1. This approach is based on that of Hartmann and Orlin [48], who consider the unconstrained case.

Let the possible weights be in the integer set Val = $\{1, \ldots, w_{max}\}$, where w_{max} is a positive integer. Let $E' \subseteq E$, and define weights as

$$w(e) = \begin{cases} x \in \text{Val} & \text{if } e \in E' \\ 0 & \text{if } e \in E - E'. \end{cases}$$

The setup in Section 3.5.1.1 is when $E' = E$ and Val = $\{1\}$.

Let $T_u := \max_{(u,v)\in E} w(u,v)$ for each vertex $u \in V$. Then, $T := \sum_{u\in V} T_u$ is the maximum weight of a path.

Let $s \in V$ be an accepting state. For each $v \in V$, let $G_k(v)$ be the minimum cost walk from s to v that has total weight equal to k. This definition is similar to $F_k(v)$ in Section 3.5.1.1, except that now k is the total weight w of the edges, which is no longer simply the number of edges. Let $G'_k(v)$ be the minimum cost walk from s to v that has total weight equal to k, with the last edge of the walk in E'. Finally, let $d(u,v)$ be the minimum cost of a path from u to v in G consisting solely of edges of $E - E'$. The costs $d(u,v)$ are pre-computed using an all-pairs shortest paths algorithm, which assumes there are no negative-cost cycles in $E - E'$ [29].

The values $G_k(v)$ can be computed for all $v \in V$ and $k = 1, \ldots, T$ by the recurrence:

$$\begin{aligned} G'_k(v) &= \min_{(u,v)\in E'} \left[G_{k-w(u,v)}(u) + c(u,v) \right] \\ G_k(v) &= \min_{u\in V} \left[G'_k(u) + d(u,v) \right] \end{aligned} \quad (3.5)$$

where $G_0(v) = d(s,v)$.

The optimal cycle cost and the corresponding cycle are recovered in a similar manner to that described in Section 3.5.1.1, and are accepting by construction. The recurrence in (3.5) requires $O(n_a T |V|^2)$ operations, where n_a is the number of accepting vertices. This algorithm runs in pseudo-polynomial time, as T is an integer, and so its binary description length is $O(\log(T))$ [29]. The recurrence for G_k can be computed more efficiently if the edge costs are assumed to be non-negative. This gives the overall complexity of the recurrence as $O(n_a T(|E| + |V|\log|V|))$ time [48].

Remark 3.3. Consider the special case where weights are restricted to be 0 or 1. Then, the total weight T is $O(|V|)$, and the above algorithm has polynomial time complexity $O(n_a|V|^3)$ or $O(n_a(|V||E| + |V|^2\log|V|)$ if edge costs are assumed to be non-negative.

Remark 3.4. Although finite prefixes do not affect the cost (cf. Proposition 3.2), it may be desired to create a "good" finite prefix. The techniques described in Section 3.5.1.2 (and similarly Section 3.5.1.1) can be adapted to create these finite prefixes. After computing the optimal accepting cycle C^*, one can compute the values $G_k(v)$, and corresponding walk defined with respect to the initial state s_0 for all states $v \in C^*$.

3.5.2 Computing Infinite-Memory Runs

Infinite-memory runs achieve the minimum weighted average cost J^*. However, their practical use is limited, as they achieve J^* by increasingly visiting states that do not affect whether or not the specification is satisfied. This is unlikely the designer's intent, so we only briefly discuss these runs for the sake of completeness. A related discussion on infinite-memory runs, but in the adversarial environment context, appears in [22].

Let σ_{opt} be the (possibly non-accepting) cycle, with the minimum weighted average cost $J(\sigma_{\text{opt}}^\omega)$ over all cycles in G. Clearly, the restriction that a cycle is accepting can only increase the weighted average cost. Let σ_{acc} be a cycle that contains both an accepting state and a state in σ_{opt}. Let $\sigma_{\text{acc},i}$ denote the ith state in σ_{acc}. For symbols α and β, let $(\alpha\beta^k)^\omega$ for $k = 1, 2, \ldots$ denote the sequence $\alpha\beta\alpha\beta\beta\alpha\beta\beta\beta\ldots$.

Proposition 3.4. *Let $\sigma_{\mathcal{P}} = (\sigma_{acc}\sigma_{opt}^k)^\omega$, where $k = 1, 2, \ldots$. Then, $\sigma_{\mathcal{P}}$ is accepting, and achieves the minimum weighted average cost* (3.1).

Proof. Run $\sigma_{\mathcal{P}}$ is accepting because it repeats σ_{acc} infinitely often. Let $\alpha_c = \sum_{i=0}^{p} c(\sigma_{\text{acc},i}, \sigma_{\text{acc},i+1})$ and $\alpha_w = \sum_{i=0}^{p} w(\sigma_{\text{acc},i}, \sigma_{\text{acc},i+1})$, where integer p is the length of σ_{acc}. Define β_c and β_w similarly for σ_{opt}. Then,

$$\begin{aligned} J(\sigma) &:= \limsup_{n\to\infty} \frac{\sum_{k=1}^{n}(\alpha_c + k\beta_c)}{\sum_{k=1}^{n}(\alpha_w + k\beta_w)} \\ &= \limsup_{n\to\infty} \frac{n\alpha_c + \beta_c \sum_{k=1}^{n} k}{n\alpha_w + \beta_w \sum_{k=1}^{n} k} \\ &= \limsup_{n\to\infty} \frac{\beta_c}{\beta_w} = J((\sigma_{\text{opt}})^{\omega}). \end{aligned}$$

□

A direct application of a minimum cost-to-time ratio algorithm, e.g., [48], can be used to compute σ_{opt}, since there is no constraint specifying that it must include an accepting state. Also, given σ_{opt}, σ_{acc} always exists, as there is an accepting state in the same strongly connected component as σ_{opt} by construction.

The next proposition shows that finite-memory runs can be arbitrarily close to the optimal weighted average cost J^*.

Proposition 3.5. *Given any $\epsilon > 0$, a finite-memory run $\sigma_{\mathcal{P}}$ exists with $J((\sigma_{\mathcal{P}})^{\omega}) < J((\sigma_{opt})^{\omega}) + \epsilon = J^* + \epsilon$.*

Proof. Construct a finite-memory run of the form $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^{\omega}$, where σ_{suf} has fixed length. In particular, let $\sigma_{\text{suf}} = \sigma_{\text{acc}}(\sigma_{\text{opt}})^{M}$ for a large (fixed) integer M. By picking M large enough, the error between $J((\sigma_{\text{suf}})^{\omega})$ and $J((\sigma_{\text{opt}})^{\omega})$ can be made arbitrarily small. □

Thus, finite-memory runs can approximate the performance of infinite-memory runs arbitrarily closely. This allows a designer to tradeoff between runs with low weighted average cost and runs with short lengths.

3.5.3 Complexity

We now discuss the complexity of the entire procedure, i.e., Algorithm 1. The number of states and transitions in the transition system is $n_{\mathcal{T}}$ and $m_{\mathcal{T}}$, respectively. The ω-

regular specification is given by a Büchi automaton $\mathcal{A}_\varphi$. In practice, an LTL formula φ will typically be used to automatically generate a corresponding Büchi automaton $\mathcal{A}_\varphi$. The product automaton has $n_\mathcal{P} = n_\mathcal{T} \times |\mathcal{A}_\varphi|$ states and $m_\mathcal{P}$ edges. For finite-memory runs, the dynamic programming algorithms described in Section 3.5.1 take $O(n_a n_\mathcal{P} m_\mathcal{P})$ and $O(n_a T(m_\mathcal{P} + n_\mathcal{P} \log n_\mathcal{P}))$ operations, assuming non-negative edge weights for the latter bound. Here, n_a is the number of accepting states in the product automaton. Usually, n_a is significantly smaller than $n_\mathcal{P}$. For infinite-memory runs, there is no accepting state constraint for the cycles, so standard techniques [48, 57] can be used that take $O(n_\mathcal{P} m_\mathcal{P})$ and $O(T(m_\mathcal{P} + n_\mathcal{P} \log n_\mathcal{P}))$ operations, again assuming non-negative edge weights for the latter bound. The algorithms in Section 3.5 are easily parallelizable, both between strongly connected components of $\mathcal{P}$ and for each accepting state.

3.6 Examples

The following examples demonstrate the techniques developed in Section 3.5 in the context of autonomous driving and surveillance. Each cell in Figures 3.1 and 3.2 corresponds to a state, and each state has transitions to its four neighbors. Costs and weights are specified over states, as discussed in Section 3.2. Tasks are specified formally by LTL formulas, and informally in English.

The first example is motivated by autonomous driving. The weighted transition system represents an abstracted car that can transition between neighboring cells in the grid (Figure 3.1). The car's task is to repeatedly visit the states labeled a, b, and c, while always avoiding states labeled x. Formally, $\varphi = \Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c \wedge \Box\neg x$. Costs are defined over states to reward driving in the proper lane (the outer boundary), and penalize leaving it. Weights are zero for all states except states labeled a, b, and c, which each have weight of one.

The second example, Figure 3.2, is motivated by surveillance. The robot's task is to repeatedly visit states labeled either a, b, c or d, e, f. States labeled x should always be avoided. Formally, $\varphi = ((\Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c) \vee (\Box\Diamond d \wedge \Box\Diamond e \wedge \Box\Diamond f)) \wedge \Box\neg x$.

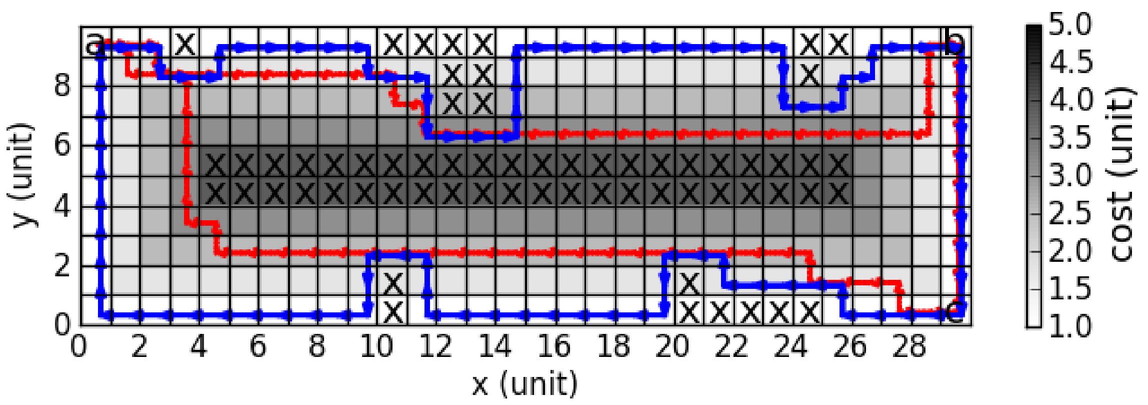


Figure 3.1: Driving task, with optimal run (blue) and feasible run (red).

Costs vary with the state, as described in Figure 3.2, and describe the time to navigate different terrain. The weight is zero at each state, except states a and f, where the weight is one.

Numerical results are in Table 3.1. Computation times for optimal and feasible runs are given by t_{opt} and t_{feas}, respectively. The number of states and transitions are listed for the transition system $\mathcal{T}$, the Büchi automaton $\mathcal{A}_\varphi$, the entire product automaton $\mathcal{P}$ and the reachable portion of the product automaton $\mathcal{P}_{\text{reach}}$. Also listed are the number of strongly connected components (SCC), the number of accepting states, and the costs for optimal J_{opt} and feasible J_{feas} runs. All computations were done using Python on a Linux desktop with a dual-core processor and 2 GB of memory. The feasible satisfying runs were generated with depth-first search. The optimal satisfying runs were generated with the algorithm from Section 3.5.1.2. Since it was possible to decrease the weighted average cost by increasing the length of the cycle (i.e., the infimum was not achieved by a finite-memory satisfying run), we used the shortest cycle such that $J(\sigma_{\text{opt}}) < \infty$. Thus, the optimal values $J(\sigma_{\text{opt}})$ are conservative. The improvement of the optimal runs over a feasible run is evident from Figures 3.1 and 3.2. In Figure 3.1, the optimal run immediately heads back to its lane to reduce costs, while the feasible run does not. In Figure 3.2, the optimal run avoids visiting high-cost regions.

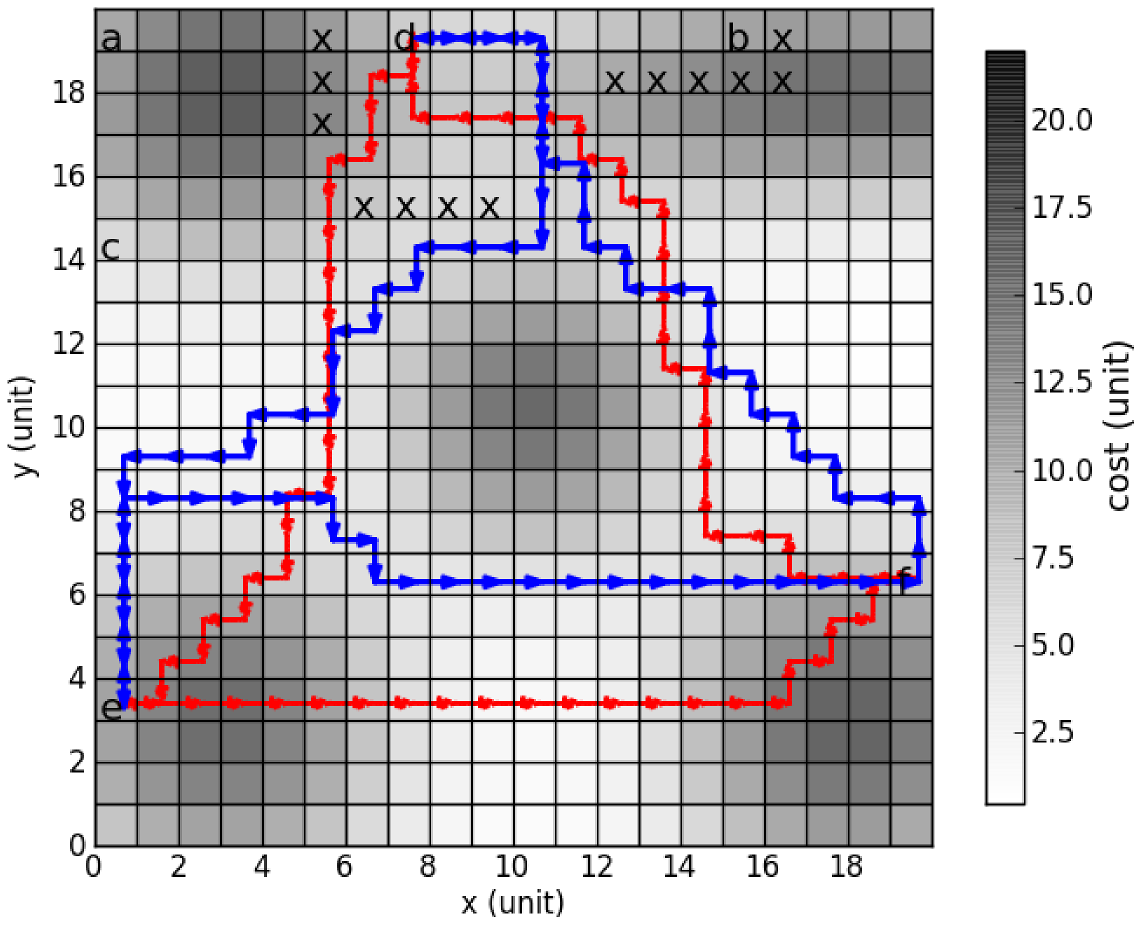


Figure 3.2: Surveillance task, with optimal run (blue) and feasible run (red).

Table 3.1: Numerical Results for Examples

Example	$\mathcal{T}$ (states/trans.)	$\mathcal{A}_\varphi$	$\mathcal{P}$	$\mathcal{P}_{\text{reach}}$	SCC (#)
Driving	300 / 1120	4 / 13	1200 / 3516	709 / 2396	1
Surveillance	400 / 1520	9 / 34	3600 / 14917	2355 / 8835	2

Example	acc. states (#)	J_{opt} (units)	J_{feas} (units)	t_{opt} (sec)	t_{feas} (sec)
Driving	1	49.3	71.3	2.49	0.68
Surveillance	2	340.9	566.3	21.9	1.94

3.7 Conclusions

This chapter presented algorithms for computing optimal runs of a weighted transition system that minimized a weighted average cost function subject to ω-regular language constraints. These constraints include the well-studied linear temporal logic as a subset. Optimal system runs correspond to cycles in a lifted product space, which includes behaviors that are valid for the system and also satisfy the temporal logic specification. Dynamic programming techniques were used to solve for an optimal cycle in this product space.

Chapter 4

Robust Control of Uncertain Markov Decision Processes with Temporal Logic Specifications

This chapter describes a method for designing a robust control policy for an uncertain system subject to temporal logic specifications. The system is modeled as a finite Markov decision process (MDP) whose transition probabilities are not exactly known, but are known to belong to a given uncertainty set. A robust control policy is generated for the MDP that maximizes the worst-case probability of satisfying the specification over all transition probabilities in this uncertainty set. To this end, we use a procedure from probabilistic model checking to combine the system model with an automaton representing the specification. This new MDP is then transformed into an equivalent form that satisfies assumptions for stochastic shortest path dynamic programming. A robust version of dynamic programming solves for a ϵ-suboptimal robust control policy, with time complexity $O(\log 1/\epsilon)$ times that for the non-robust case. This chapter is based on results from [100].

4.1 Introduction

As autonomous systems often operate in uncertain environments, it is important that the system performance is robust to environmental disturbances. Furthermore, system models are only approximations of reality, which makes robustness to modeling errors desirable.

We model the system as a Markov decision process (MDP). MDPs provide a general framework for modeling non-determinism and probabilistic behaviors that are present in many real-world systems. MDPs are also amenable to formal verification techniques for temporal logic properties [9], which can be alternatively used to create control policies. These techniques generate a control policy for the MDP that maximizes the probability of satisfying a given LTL specification. However, these techniques assume that the state transition probabilities of the MDP are known exactly, which is often unrealistic. We relax this assumption by allowing the transition probabilities of the MDP to lie in uncertainty sets. We generate a control policy that maximizes the worst-case probability of a run of the system satisfying a given LTL specification over all admissible transition probabilities in the uncertainty set.

Considering uncertainty in the system model is important to capture unmodeled dynamics and parametric uncertainty, as real systems are only approximated by mathematical models. Additionally, while we consider discrete-state systems here, we are motivated by controlling continuous stochastic systems so that they satisfy temporal logic specifications. Constructive techniques for finite, discrete abstractions of continuous stochastic systems exist (see [2, 7]), but exact abstraction is generally difficult. Even if exact finite-state abstraction techniques are available for a dynamical system model, the resulting MDP abstraction will only represent the real system to the extent that the dynamical system model does. Moreover, if abstraction techniques approximate the dynamical system model, the MDP abstraction will be a further approximation of the real system.

Robustness of control policies for MDPs with respect to uncertain transition probabilities has been studied in the contexts of operations research, formal verification, and hybrid systems. Our approach most closely follows that of Nilim and El Ghaoui [79], who consider general uncertainty models and discounted rewards, but not temporal logic specifications. Related work includes [8, 44, 87]. Formal verification of temporal logic specifications is well-developed for MDPs with exact transition matrices [9, 33] and standard software tools exist [63]. Work in verification of uncertain MDPs primarily considers simple interval uncertainty models for the transition probabilities [20, 23, 109], which we include in our work as a special case. Recent work in hybrid systems that creates control policies for stochastic systems [34, 65] does not consider robustness. Robustness of non-probabilistic discrete-state systems to disturbances is explored in [72].

The main contribution of this chapter is an algorithm for creating an optimal robust control policy π^* that maximizes the worst-case probability of satisfying an LTL specification for a system represented as a finite labeled MDP with transition matrices in an uncertainty set $\mathcal{P}$. The uncertainty set $\mathcal{P}$ can be non-convex, and includes interval uncertainty sets as a special case. This freedom allows more statistically accurate and less conservative results than can be achieved using interval uncertainty sets.

4.2 Uncertain Markov Decision Processes

We use uncertain Markov decision processes (uncertain MDPs) as the system model.

Definition 4.1. A *labeled finite MDP* $\mathcal{M}$ is the tuple $\mathcal{M} = (S, A, P, s_0, AP, L)$, where S is a finite set of states, A is a finite set of actions, $P : S \times A \times S \to [0, 1]$ is the transition probability function, s_0 is the initial state, AP is a finite set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function. Let $A(s)$ denote the set of available actions at state s. Let $\sum_{s' \in S} P(s, a, s') = 1$ if $a \in A(s)$ and $P(s, a, s') = 0$ otherwise.

We assume, for notational convenience, that the available actions $A(s)$ are the same for every $s \in S$. We use P^a_{ij} as shorthand for the transition probability from state i to state j when using action a. We call $P^a \in \mathbb{R}^{n \times n}$ a *transition matrix*, where the (i, j)-th entry of P^a is P^a_{ij}. Where it is clear from context, we refer to the row vector P^a_i as p.

Definition 4.2. A *control policy* for an MDP $\mathcal{M}$ is a sequence $\pi = \{\mu_0, \mu_1, \ldots\}$, where $\mu_k : S \to A$ such that $\mu_k(s) \in A(s)$ for state $s \in S$ and $k = 0, 1, \ldots$. A control policy is *stationary* if $\pi = \{\mu, \mu, \ldots\}$. Let Π be the set of all control policies, and Π_s be the set of all stationary control policies.

A *run* of the MDP is an infinite sequence of its states, $\sigma = s_0 s_1 s_2 \ldots$ where $s_i \in S$ is the state of the system at index i, and $P(s_i, a, s_{i+1}) > 0$ for some $a \in A(s_i)$. A run is induced by a control policy.

Uncertainty Model

To model uncertainty in the system model, we specify uncertainty sets for the transition matrices.

Definition 4.3. Let the transition matrix uncertainty set be defined as $\mathcal{P}$ where every $P \in \mathcal{P}$ is a transition matrix. An *uncertain labeled finite MDP* $\mathcal{M} = (S, A, \mathcal{P}, s_0, AP, L)$ is a family of labeled finite MDPs such that for every $P \in \mathcal{P}$, $\mathcal{M}' = (S, A, P, s_0, AP, L)$ is an MDP.

Let $\mathcal{P}_s^a$ be the uncertainty set corresponding to state $s \in S$ and action $a \in A(s)$.

Definition 4.4. An *environment policy* for an (uncertain) MDP $\mathcal{M}$ is a sequence $\tau = \{\nu_0, \nu_1, \ldots\}$, where $\nu_k : S \times A \to P$ such that $\nu_k(s, a) \in \mathcal{P}_s^a$ for state $s \in S$, action $a \in A(s)$ and $k = 0, 1, \ldots$. An environment policy is *stationary* if $\tau = \{\nu, \nu, \ldots\}$. Let $\mathcal{T}$ be the set of all environment policies, and $\mathcal{T}_s$ be the set of all stationary environment policies.

A *run* of the uncertain MDP is an infinite sequence of its states, $\sigma = s_0 s_1 s_2 \ldots$ where $s_i \in S$ is the state of the system at index i, and $P(s_i, a, s_{i+1}) > 0$ for some $a \in A(s_i)$ and $P \in \mathcal{P}_{s_i}^a$. A run is induced by an environment policy and a control policy.

We associate a reward with each state-action pair in $\mathcal{M}$ through the function $r(s, a) : S \times A \to \mathbb{R}$. This (possibly negative) reward is incurred at each stage k over the horizon of length N, where the total expected reward is

$$V^{\pi\tau}(s) := \lim_{N \to \infty} \mathbb{E}_{\pi\tau} \left[\sum_{k=0}^{N-1} r(s_k, \mu_k(s_k)) \mid s_0 = s \right],$$

and the expectation $\mathbb{E}_{\pi\tau}$ depends on both the control and environment policies.

The optimal worst-case total expected reward starting from state $s \in S$ is

$$V^*(s) := \max_{\pi\in\Pi} \min_{\tau\in\mathcal{T}} V^{\pi\tau}(s). \tag{4.1}$$

4.3 Problem Statement

We now provide a formal statement of the main problem of this chapter, and an overview of our approach.

Definition 4.5. Let $\mathcal{M}$ be an uncertain MDP with initial state s_0 and atomic propositions AP. Let φ be an LTL formula over AP. Then, $\mathbb{P}^{\pi\tau}(s_0 \vDash \varphi)$ is the *expected satisfaction probability* of φ by $\mathcal{M}$, under control policy π and environment policy τ.

Definition 4.6. An *optimal robust control policy* π^* for uncertain MDP $\mathcal{M}$ is

$$\pi^* = \arg\max_{\pi\in\Pi} \min_{\tau\in\mathcal{T}} \mathbb{P}^{\pi\tau}(s_0 \vDash \varphi).$$

Problem 4.1. Given an uncertain labeled finite MDP $\mathcal{M}$ and an LTL formula φ over AP, create an optimal robust control policy π^*.

We solve Problem 4.1 by first creating the product MDP $\mathcal{M}_p$, which contains only valid system trajectories that also satisfy the LTL specification. We modify $\mathcal{M}_p$ so that all policies for it are proper (i.e., a terminal zero-reward state will be reached), and thus, it satisfies stochastic shortest path assumptions. Maximizing the worst-case probability of satisfying the specification is equivalent to creating a control policy that maximizes the worst-case probability of reaching a certain set of states in $\mathcal{M}_p$. We solve for this policy using robust dynamic programming. Finally, we map the robust control policy back to $\mathcal{M}$.

4.4 The Product MDP

In this section, we create a product MDP $\mathcal{M}_p$ that contains behaviors that satisfy both the system MDP $\mathcal{M}$ and the LTL specification φ. We transform $\mathcal{M}_p$ into an equivalent form $\mathcal{M}_{ssp}$, where all stationary control policies are proper, in preparation for computing optimal robust control policy in Section 4.6.

4.4.1 Forming the Product MDP

The product MDP $\mathcal{M}_p$ restricts behaviors to those that satisfy both the system transitions and a deterministic Rabin automaton $\mathcal{A}_\varphi$ that represents the LTL formula φ.

Definition 4.7. For a labeled finite MDP $\mathcal{M} = (S, A, P, s_0, AP, L)$ and a deterministic Rabin automaton $\mathcal{A}_\varphi = (Q, 2^{AP}, \delta, q_0, F)$, the *product MDP* is given by $\mathcal{M}_p = (S_p, A, P_p, s_{0p}, Q, L_p)$ with $S_p = S \times Q$,

- $P_p((s,q),\alpha,(s',q')) = \begin{cases} P(s,\alpha,s') & \text{if } q' = \delta(q, L(s')) \\ 0 & \text{otherwise,} \end{cases}$
- $s_{0p} = (s_0, q)$ such that $q = \delta(q_0, L(s_0))$,
- and $L_p((s,q)) = \{q\}$.

The accepting product state pairs $F_p = \{(J_1^p, K_1^p), \ldots, (J_k^p, K_k^p)\}$ are lifted directly from F. Formally, for every $(J_i, K_i) \in F$, state $(s,q) \in S_p$ is in J_i^p if $q \in J_i$, and $(s,q) \in K_i^p$ if $q \in K_i$.

There is a one-to-one correspondence between the paths on $\mathcal{M}_p$ and $\mathcal{M}$, which induces a one-to-one correspondence for policies on $\mathcal{M}_p$ and $\mathcal{M}$. Given a policy $\pi^p = \{\mu_0^p, \mu_1^p, \ldots\}$ on $\mathcal{M}_p$, one can induce a policy $\pi = \{\mu_0, \mu_1, \ldots\}$ on $\mathcal{M}$ by setting $\mu_i(s_i) = \mu_i^p((s_i, q_i))$ for every stage $i = 0, 1, \ldots$. This policy always exists, since $\mathcal{M}_p$ and $\mathcal{M}$ have the same action set A. If π^p is stationary, then π is finite-memory [9].

4.4.2 Reachability in the Product MDP

We now show how to use the product MDP $\mathcal{M}_p$ to determine a robust control policy that maximizes the worst-case probability that a given LTL specification is satisfied. Given a control and environment policy, the probability of satisfying an LTL formula is equivalent to the probability of reaching an *accepting maximal end component* [9]. We call this probability the *reachability probability.* Informally, accepting maximal end components are sets of states that the system can remain in forever and where the acceptance condition of the deterministic Rabin automaton is satisfied, and thus, the corresponding LTL formula is satisfied. The following definitions follow [9].

Definition 4.8. A *sub-MDP* of an MDP is a pair of states and action sets (C, D) where: (1) $C \subseteq S$ is non-empty and the map $D : C \to 2^A$ is a function such that $D(s) \subseteq A(s)$ is non-empty for all states $s \in C$, and (2) $s \in C$ and $a \in D(s)$ implies $Post(s, a) = \{t \in S | P^a_{st} > 0\} \subseteq C$.

Definition 4.9. An *end component* is a sub-MDP (C, D) such that the digraph $G_{(C,D)}$ induced by (C, D) is strongly connected.

An end component (C, D) is *maximal* if there is no end component (C', D') such that $(C, D) \neq (C', D')$ and $C \subseteq C'$ and $D(s) \subseteq D'(s)$ for all $s \in C$. Furthermore, (C, D) is *accepting* for the deterministic Rabin automaton $\mathcal{A}$ if for some $(J, K) \in F$, $J \cap C = \varnothing$ and $K \cap C \neq \varnothing$.

Given the accepting maximal end components of $\mathcal{M}_p$, one can determine a control policy that maximizes the worst-case probability of reaching an accepting maximal end component from the initial state. Without considering transition probability uncertainty, a non-robust policy can be computed using either linear or dynamic programming methods [9].

From the preceding discussion, it is clear that the LTL formula satisfaction probability depends on the connectivity of the product MDP $\mathcal{M}_p$. Thus, we will require that the uncertainty sets for the transition matrices of the system MDP $\mathcal{M}$ do not change this connectivity. Let F^a denote the nominal transition matrix for action a.

Assumption 4.1. $F_{ij}^a = 0$ if and only if $P_{ij}^a = 0$ for all $P^a \in \mathcal{P}^a$ and for all $i, j \in S$.

Assumption 4.1 says that if a nominal transition is zero (non-zero) if and only if it is zero (non-zero) for all transition matrices in the uncertainty set.

4.4.3 Stochastic Shortest Path Form of Product MDP

We now transform the product MDP $\mathcal{M}_p$ into an equivalent form $\mathcal{M}_{ssp}$ where all stationary control policies μ are proper, as will be needed for Lemma 4.2 in Section 4.5. Note that $\mathcal{M}_p$ and $\mathcal{M}_{ssp}$ are equivalent only in terms of the probability of reaching an accepting maximal end component—both the states and the transition probabilities may change.

Let MDP $\mathcal{M}$ have a finite set of states $S = \{1, 2, \ldots, n, t\}$ and actions $a \in A(s)$ for all $s \in S$. Let t be a special terminal state, which is absorbing ($P_{tt}^a = 1$) and incurs zero reward ($r(t, a) = 0$) for all $a \in A(t)$ and all $P \in \mathcal{P}_t^a$ [13].

Definition 4.10. A stationary control policy μ is *proper* if, under that policy, there is positive probability that the terminal state will be reached after at most n transitions, regardless of the initial state and transition matrices; that is, if

$$\rho_{\mu\tau} := \max_{s=1,\ldots,n} \max_{\tau \in \mathcal{T}} \mathbb{P}_{\mu\tau}(s_n \neq t | s_0 = s) < 1. \tag{4.2}$$

In the remainder of this section, we use the simplified notation $\mathcal{M}_p = (S, A, P)$ to describe the states, actions, and transition matrices of the product MDP. We refer to a state (s, q) of $\mathcal{M}_p$ as s when clear from context.

Partition the states S of $\mathcal{M}_p$ into three disjoint sets: B, S_0, and S_r. Let set B be the union of all accepting maximal end components in $\mathcal{M}_p$. By definition, every state $s \in B$ has reachability probability of 1. Let S_0 be the set of states that have zero probability of reaching B. Set S_0 can be computed efficiently by graph algorithms [9]. Finally, let set $S_r = S - (B \cup S_0)$ contain states not in an accepting maximal end component, but with non-zero maximum reachability probability. It is easy to see that B, S_0, and S_r form a partition of S.

In Algorithm 2, we augment S with a terminal state t that is absorbing and incurs zero reward. Algorithm 2 does not change the probability of reaching an accepting maximal end component for any state $s \in S$ under any choice of control and environment policies.

```
Algorithm 2 Appending the terminal state
Require: $\mathcal{M}_p = (S, A, P)$ and $S_r, S_0, B$
  $S := S \cup \{t\}$ and $A(t) := \{u\}$ and $r(t, u) := 0$;
  $A(s) := A(s) \cup \{u\}$ and $P^u_{st} := 1$ for all $s \in B \cup S_0$.
```

We eliminate the k maximal end components in S_r and replace them with new states $\hat{s}_i$ for $i = 1, \dots, k$ in Algorithm 3. This procedure is from Section 3.3 of [33], where it is proven (Theorem 3.8 in de Alfaro [33]) that the reachability probability is unchanged by this procedure. The intuition behind this result is that one can move between any two states r and s in a maximal end component in S_r with probability one.

```
Algorithm 3 End component elimination (de Alfaro [33])
Require: MDP $\mathcal{M}_p = (S, A, P)$ and $S_r, S_0, B$
Ensure: MDP $\mathcal{M}_{ssp} = (\hat{S}, \hat{A}, \hat{P})$
  $\{(C_1, D_1), \dots, (C_k, D_k)\}$ max end components in $S_r$
  $\hat{S}_0 := S_0$ and $\hat{B} := B$;
  $\hat{S} := S \cup \{\hat{s}_1, \dots, \hat{s}_k\} - \cup_{i=1}^k C_i$;
  $\hat{S}_r := S_r \cup \{\hat{s}_1, \dots, \hat{s}_k\} - \cup_{i=1}^k C_i$;
  $\hat{A}(s) := \{(s, a) \mid a \in A(s)\}$ for $s \in S - \cup_{i=1}^k C_i$;
  $\hat{A}(\hat{s}_i) := \{(s, a) \mid s \in C_i \wedge a \in A(s) - D(s)\}$ for $1 \le i \le k$;
  For $s \in \hat{S}, t \in S - \cup_{i=1}^k C_i$ and $(u, a) \in \hat{A}(s)$, $\hat{P}^{(u,a)}_{st} := P^a_{ut}$ and $\hat{P}^{(u,a)}_{s\hat{s}_i} := \sum_{t \in C_i} P^a_{ut}$.
```

After applying Algorithms 2 and 3, we call the resulting MDP $\mathcal{M}_{ssp}$. Note that $\hat{S}_r$, $\hat{B}$, and $\hat{S}_0$ form a disjoint partition of $\hat{S}$. All stationary control policies for $\mathcal{M}_{ssp}$ are proper, i.e., they will almost surely reach the terminal state t.

Theorem 4.1. *All stationary control policies for $\mathcal{M}_{ssp}$ are proper.*

Proof. Suppose instead that there exists a stationary control policy μ such that the system starting in state $s_0 \in \hat{S}_r$ has zero probability of having reached the terminal state t after n stages. This implies that under μ, there is zero probability of reaching any state $s \in \hat{B} \cup \hat{S}_0$ from $s_0 \in \hat{S}_r$. Then, under policy μ, there exists a set $U \subseteq \hat{S}_r$ such that if state $s_k \in U$ for some finite integer k, then $s_k \in U$ for all k. Let $U' \subseteq U$ be the largest set where each state is visited infinitely often. Set U' is an end component in $\hat{S}_r$, which is a contradiction. Note that one only needs to consider $s_0 \in \hat{S}_r$, as all $s \in \hat{B} \cup \hat{S}_0$ deterministically transition to t. $\square$

MDP $\mathcal{M}_{ssp}$ is equivalent in terms of reachability probabilities to the original product MDP $\mathcal{M}_p$, and all stationary control policies are proper.

4.5 Robust Dynamic Programming

We now prove results on robust dynamic programming that will be used in Section 4.6 to compute an optimal robust control policy. The results in this section apply to uncertain MDPs that satisfy Assumption 4.1. A specific reward function will be used in Section 4.6 so that the probability of the system satisfying an LTL formula from a given state is equal to the value function at that state.

4.5.1 Dynamic Programming

For technical reasons, we require control policies to be proper, i.e., they almost surely reach the terminal state t for all transition matrices in the uncertainty set (see Section 4.4.3).

Assumption 4.2. All stationary control policies are proper.

Remark 4.1. This assumption implies that the terminal state will eventually be reached under any stationary policy. This assumption allows us to make statements regarding convergence rates. While this assumption is usually a rather strong condition, it is not restrictive for Problem 4.1 (see Theorem 4.1).

In preparation for the main result of this section, we give the following classical theorem [78].

Theorem 4.2 (Contraction Mapping Theorem). *Let (M, d) be a complete metric space, and let $f : M \to M$ be a contraction, i.e., there is a real number β, $0 \leq \beta < 1$, such that $d(f(x), f(y)) \leq \beta d(x, y)$ for all x and y in M. Then, there exists a unique point x^* in M such that $f(x^*) = x^*$. Additionally, if x is any point in M, then* $\lim_{k\to\infty} f^k(x) = x^*$, *where f^k is the composition of f with itself k times.*

We now define mappings that play an important role in the rest of this section. The value $V(s)$ is the total expected reward starting at state $s \in S$. The shorthand V represents the value function for all $s \in S\backslash t$, and can be considered a vector in $\mathbb{R}^n$. Since the reward is zero at the terminal state t, we do not include it. The T and $T_{\mu\nu}$ operators are mappings from $\mathbb{R}^n$ to $\mathbb{R}^n$. For each state $s \in S\backslash t$, define the s-th component of TV and $T_{\mu\nu}V$ respectively as

$$(TV)(s) := \max_{a \in A(s)} [r(s,a) + \min_{p \in \mathcal{P}_s^a} p^T V], \tag{4.3}$$
$$(T_{\mu\nu}V)(s) := r(s, \mu(s)) + \nu(s, \mu(s))^T V. \tag{4.4}$$

In the following two lemmas, we show that these mappings are monotonic and contractive. We prove these for (4.3); the proofs for (4.4) follow by limiting the actions and transition probabilities at each state s to $\mu(s)$ and $\nu(s, \mu(s))$ respectively. T^k is the composition of T with itself k times.

Lemma 4.1 (Monotonicity). *For any vectors $u, v \in \mathbb{R}^n$, such that $u \leq v$, we have that $T^k u \leq T^k v$ for $k = 1, 2, \ldots$.*

Proof. Immediate from equation (4.3) since $\mathcal{P}_s^a$ is in the probability simplex. □

Definition 4.11. The *weighted maximum norm* $\| \cdot \|_w$ of a vector $u \in \mathbb{R}^n$ is defined by $\| u \|_w = \max_{i=1,\ldots,n} |u(i)|/w(i)$, where vector $w \in \mathbb{R}^n$ and $w > 0$.

Lemma 4.2 (Contraction). *If all stationary control policies are proper, then there exists a vector $w > 0$ and a scalar $\gamma \in [0, 1)$ such that $\| Tu - Tv \|_w \leq \gamma \| u - v \|_w$ for all $u, v \in \mathbb{R}^n$.*

Proof. The proof of Lemma 4.2 closely follows that in Bertsekas [13], (Vol. II, Section 2.4) where the environment policy is fixed. More specifically, the proof in Bertsekas [13] is modified to allow minimization over environment policies. This modification holds due to Assumptions 4.1 and 4.2.

First, partition the state space $S = \{1, 2, \dots, n, t\}$. Let $S_1 = \{t\}$ and for $k = 2, 3, \dots$, define

$$S_k = \{i | i \notin S_1 \cup \cdots \cup S_{k-1} \text{ and } \min_{a \in A(i)} \max_{j \in S_1 \cup \cdots \cup S_{k-1}} \min_{P \in \mathcal{P}_i^a} P_{ij}^a > 0\}.$$

Set S_k is the set of states that have a positive probability of reaching state t in $k-1$, but no fewer, steps. Let m be the largest integer such that S_m is nonempty. Because of Assumptions 4.1 and 4.2, $\cup_{k=1}^m S_k = S$.

Choose a vector $w > 0$ so that T is a contraction with respect to $\| \cdot \|_w$. Take the ith component w_i to be the same for states i in the same set S_k. Choose the components w_i of the vector w by $w_i = y_k$ if $i \in S_k$, where $y_1, \dots, y_m$ are appropriately chosen scalars satisfying $1 = y_1 < y_2 < \cdots < y_m$. We will show how to choose these values shortly.

Let

$$\epsilon := \min_{k=2,\dots,m} \min_{a \in A} \min_{i \in S_k} \min_{P_i^a \in \mathcal{P}_i^a} \sum_{j \in S_1 \cup \cdots \cup S_{k-1}} P_{ij}^a.$$

The value ϵ is the minimum probability of transitioning from a state in set S_k to a state in a set that is closer to state t, i.e., a state in sets $S_1, \dots, S_{k-1}$. Note that $0 < \epsilon \leq 1$. We will choose $y_2, \dots, y_m$ so that for some $\beta < 1$, $\frac{y_m}{y_k}(1 - \epsilon) + \frac{y_{k-1}}{y_k}\epsilon \leq \beta < 1$ for $k = 2, \dots, m$ and later show that such a choice exists.

For all vectors $u, v \in \mathbb{R}^n$, select a control policy $\mu \in \Pi_s$ such that $T_\mu v = Tv$, where

T_μ indicates that only the system policy is fixed. Then, for all i,

$$\begin{aligned}(Tv)(i) - (Tu)(i) &= (T_\mu v)(i) - (Tu)(i)\\ &\le (T_\mu v)(i) - (T_\mu u)(i)\\ &= \sum_{j=1}^{n} V_{ij}^{\mu(i)} v(j) - U_{ij}^{\mu(i)} u(j)\\ &\le \sum_{j=1}^{n} P_{ij}^{\mu(i)} (v(j) - u(j)),\end{aligned}$$

where $U_i^{\mu(i)} = \arg\min_{p \in \mathcal{P}_i^{\mu(i)}} p^T u$ and $V_i^{\mu(i)} = \arg\min_{p \in \mathcal{P}_i^{\mu(i)}} p^T v$, and $P_{ij}^{\mu(i)} := \arg\max\{U_{ij}^{\mu(i)}(v(j) - u(j)), V_{ij}^{\mu(i)}(v(j) - u(j))\}$ over $U_{ij}^{\mu(i)}$ and $V_{ij}^{\mu(i)}$ for each j. The notation $U_{ij}^{\mu(i)}$ denotes the jth component of vector $U_i^{\mu(i)}$. Likewise, $V_{ij}^{\mu(i)}$ denotes the jth component of vector $V_i^{\mu(i)}$.

Let $k(j)$ denote that state j belongs to the set S_k. Then, for any constant c, $\| v - u \|_w \le c \implies v(j) - u(j) \le c y_{k(j)}, j = 1, \ldots, n$, by definition of the weighted max norm. Thus for all i,

$$\begin{aligned}\frac{(Tv)(i) - (Tu)(i)}{c y_{k(i)}} &\le \frac{1}{y_{k(i)}} \sum_{j=1}^{n} P_{ij}^{\mu(i)} y_{k(j)}\\ &\le \frac{y_{k(i)-1}}{y_{k(i)}} \sum_{j \in S_1 \cup \cdots \cup S_{k(i)-1}} P_{ij}^{\mu(i)} + \frac{y_m}{y_{k(i)}} \sum_{j \in S_{k(i)} \cup \cdots \cup S_m} P_{ij}^{\mu(i)}\\ &= \left(\frac{y_{k(i)-1}}{y_{k(i)}} - \frac{y_m}{y_{k(i)}} \right) \sum_{j \in S_1 \cup \cdots \cup S_{k(i)-1}} P_{ij}^{\mu(i)} + \frac{y_m}{y_{k(i)}}\\ &\le \left(\frac{y_{k(i)-1}}{y_{k(i)}} - \frac{y_m}{y_{k(i)}} \right) \epsilon + \frac{y_m}{y_{k(i)}}\\ &= \frac{y_m}{y_{k(i)}} (1 - \epsilon) + \frac{y_{k(i)-1}}{y_{k(i)}}\\ &\le \beta,\end{aligned}$$

where the last inequality is due to the earlier choice of y values. Then, $\frac{(Tv)(i) - (Tu)(i)}{w_i} \le c\beta, \quad i = 1, \ldots, n$, which taking the max over $i = 1, 2, \ldots, n$, gives $\| Tv - Tu \|_w \le c\beta$ for all $u, v \in \mathbb{R}^n$ with $\| u - v \|_w \le c$. Thus, T is a contraction under the $\| \cdot \|_w$ norm.

We now show that scalars $y_1, y_2, \ldots, y_m$ exist such that the above assumptions hold. Let $y_0 = 0, y_1 = 1$, and suppose that $y_1, y_2, \ldots, y_k$ have been chosen. If $\epsilon = 1$, we choose $y_{k+1} = y_k + 1$. If $\epsilon < 1$, we choose y_{k+1} to be $y_{k+1} = \frac{1}{2}(y_k + M_k)$ where $M_k = \min_{1 \le i \le k} \left\{ y_i + \frac{\epsilon}{1-\epsilon}(y_i - y_{i-1}) \right\}$.

Since $M_{k+1} = \min \left\{ M_k,\ y_{k+1} + \frac{\epsilon}{1-\epsilon}(y_{k+1} - y_k) \right\}$, by induction we have that for all k, $y_k < y_{k+1} < M_{k+1}$, and thus, one can construct the required sequence. □

We now prove the main result of this section. We remind the reader that the function $V^* : S \to \mathbb{R}$ (equivalently a vector in $\mathbb{R}^n$), defined in equation (4.1), is the optimal worst-case total expected reward starting from state $s \in S$.

Theorem 4.3 (Robust Dynamic Programming)**.** *Under the assumption that all stationary control policies μ are proper for a finite MDP $\mathcal{M}$ with transition matrices in the uncertainty set $\mathcal{P}^a$ for $a \in A$, the following statements hold:*

(a) The optimal worst-case value function V^* is the unique fixed-point of T,

$$V^* = TV^*. \tag{4.5}$$

(b) The optimal worst-case value function V^* is given by,

$$V^* = \lim_{k \to \infty} T^k V, \tag{4.6}$$

for all $V \in \mathbb{R}^n$. This limit is unique.

(c) A stationary control policy μ and a stationary environment policy ν are optimal if and only if

$$T_{\mu\nu} V^* = TV^*. \tag{4.7}$$

Proof. Parts (a) and (b) follow immediately from Theorem 4.2 and Lemma 4.2.

Part (c): First, assume that $T_{\mu\nu}V^* = TV^*$. Then, $T_{\mu\nu}V^* = TV^* = V^*$ from equation (4.5) and $V^{\mu\nu} = V^*$ from the uniqueness of the fixed-point. Thus, μ and ν are optimal policies. Now, assume that μ and ν are optimal policies so that $V^{\mu\nu} = V^*$. Then, $T_{\mu\nu}V^* = T_{\mu\nu}V^{\mu\nu} = V^{\mu\nu} = V^*$. □

Corollary 4.1. *Given the optimal worst-case value function V^*, the optimal control actions a^* satisfy*

$$a^*(s) \in \arg\max_{a \in A(s)} [r(s,a) + \min_{p \in \mathcal{P}_s^a} p^T V^*], \quad s \in S. \tag{4.8}$$

and, with some abuse of notation, the optimal transition vectors (for the environment) are

$$P_s^{*a} \in \arg\min_{p \in \mathcal{P}_s^a} p^T V^*, \quad s \in S, a \in A(s). \tag{4.9}$$

Proof. Follows from Part (c) in Theorem 4.3 and equation (4.3). □

To recap, we showed that T is monotone, and a contraction with respect to a weighted max norm. This lets us prove in Theorem 4.3 that T has a unique fixed-point that can be found by an iterative procedure (i.e., *value iteration*). We gave conditions on the optimality of stationary policies, and showed how to determine optimal actions for the system and the environment.

4.5.2 Uncertainty Set Representations

Referring back to the operator T defined in equation (4.3), we see that it is composed of two nested optimization problems—the outer maximization problem for the system, and the inner minimization problem for the environment. To be clear, the environment optimization problem for a given state $s \in S$ and control action $a \in A(s)$ refers to $\min_{p \in \mathcal{P}_s^a} p^T V$.

The tractability of the environment optimization problem depends on the structure of the uncertainty set $\mathcal{P}_s^a$. In the remainder of this section, we investigate interval and likelihood uncertainty sets, as these are both statistically meaningful and computationally efficient. Due to lack of space, we do not discuss maximum a priori, entropy, scenario, or ellipsoidal uncertainty models, even though these are included in this framework. The reader should refer to Nilim and El Ghaoui for details [79].

We assume that the uncertainty sets of the MDP factor by state and action for the environmental optimization [79].

Assumption 4.3. $\mathcal{P}^a$ can be factored as the Cartesian product of its rows, so its rows are uncorrelated. Formally, for every $a \in A$, $\mathcal{P}^a = \mathcal{P}_1^a \times \ldots \times \mathcal{P}_n^a$ where each $\mathcal{P}_i^a$ is a subset of the probability simplex in $\mathbb{R}^n$.

Interval Models

A common description of uncertainty for transition matrices corresponding to action $a \in A$ is by intervals $\mathcal{P}^a = \{P^a \mid \underline{P}^a \leq P^a \leq \overline{P}^a,\ P^a\mathbf{1} = 1\}$, where $\underline{P}^a$ and $\overline{P}^a$ are nonnegative matrices $\underline{P}^a \leq \overline{P}^a$. This representation is motivated by statistical estimates of confidence intervals on the individual components of the transition matrix [69]. The environmental optimization problem can be solved in $O(n\log n)$ time using a bisection method [79].

Likelihood Models

The likelihood uncertainty model is motivated by determining the transition probabilities between states through experiments. We denote the experimentally measured transition probability matrix corresponding to action a by F^a and the optimal log-likelihood by $\beta_{\max}$.

Uncertainty in the transition matrix for each action $a \in A$ is described by the *likelihood region* [69]

$$\mathcal{P}^a = \{P^a \in \mathbb{R}^{n\times n} | P^a \geq 0, P^a\mathbf{1} = \mathbf{1}, \sum_{i,j} F_{ij}^a \log P_{ij}^a \geq \beta^a\}, \tag{4.10}$$

where $\beta^a < \beta_{\max}^a$ and can be estimated for a desired confidence level by using a large sample Gaussian approximation [79]. As described in Assumption 4.1, we enforce that $F_{ij}^a = 0$ if and only if $P_{ij}^a = 0$ for all $i, j \in S$ and all $a \in A$.

Since the likelihood region in (4.10) does not satisfy Assumption 4.3, it must be projected onto each row of the transition matrix. Even with the approximation, likelihood regions are more accurate representations than intervals, which are further approximations of the likelihood region. A bisection algorithm can approximate the environment optimization problem to within an accuracy δ in $O(\log(V_{\max}/\delta))$ time, where $V_{\max}$ is the maximum value of the value function [79].

4.6 Computing an Optimal Control Policy

We now find a robust control policy that maximizes the probability of satisfying φ over all transitions in an uncertainty set. We use robust value iteration as described in Section 4.5 on the transformed product MDP $\mathcal{M}_{ssp}$ created in Section 4.4. Finally, we project this control policy to a policy for $\mathcal{M}$.

The dynamic programming approach is formulated in terms of total expected reward maximization, we define the total expected reward as the reachability probability, which is equivalent to the probability of satisfying the LTL formula. Thus, for all $a \in \hat{A}$, the appropriate rewards are $r(s,a) = 1$ for all $s \in \hat{B}$, and $r(s,a) = 0$ for all $s \in \hat{S}_0$. For the remaining states, $s \in \hat{S}_r$, we initialize the rewards arbitrarily in $[0,1]$, and compute the optimal worst-case value function using the iteration presented in Theorem 4.3. The resulting value function V^*_{ssp} gives the satisfaction probability for each state in $\mathcal{M}_{ssp}$.

The value function V^*_p for $\mathcal{M}_p$ is determined from V^*_{ssp}. For $s_p \in S_p$, determine the corresponding state $s_{ssp} \in \hat{S}$, and let $V^*_p(s_p) = V^*_{ssp}(s_{ssp})$. This mapping is surjective, as there is at least one s_p for each s_{ssp}.

Given the optimal worst-case value function V^*_p for the original product MDP $\mathcal{M}_p$, the optimal actions $a^*(s) \in A(s)$ for each $s \in S_r$ can be computed. We do not consider actions for states in $S_0 \cup B$ at this time. However, one cannot simply use the approach for selecting actions given by (4.8), because not all stationary control policies on $\mathcal{M}_p$ are proper. For states in a maximal end component in S_r, there may be multiple actions that satisfy (4.8). Arbitrarily selecting actions can lead to situations where the stationary control policy stays in the maximal end component forever, and thus never satisfies the specification. We avoid this situation by only selecting an action if it is both optimal, i.e., satisfies (4.8), and it has a non-zero probability of transitioning to a state that is not in a maximal end component in S_r. Algorithm 4 selects the action with the highest probability of transitioning to a state not in a maximal end component in S_r.

Algorithm 4 Product MDP Control Policy

```
Require: $V_p^* \in \mathbb{R}^n$, $\mathcal{M}_p = (S, A, P)$, $S_r$, $S_0$, $B$
Ensure: Robust control policy $\mu$
  $visited := S_0 \cup B$;
  $possAct(s) := \{a \in A(s) | (T_a V_p^*)(s) = V_p^*(s)\}$;
  for $s \in S_r$ do
    if $|possAct(s)| = 1$ then
      $\mu(s) := possAct(s)$
      $visited := visited \cup \{s\}$;
    end if
  end for
  while $visited \neq S$ do
    for $s \in S_r \backslash visited$ do
      $maxLeaveProb := 0$;
      $leaveProb := \max_{a \in possAct(s)} \sum_{t \in visited} P_{st}^a$;
      if $leaveProb > maxLeaveProb$ then
        $optAct := a$
        $optState := s$;
      end if
    end for
    $\mu(s) := optAct$
    $visited := visited \cup \{optState\}$;
  end while
```

Theorem 4.4. *Algorithm 4 returns a robust control policy μ that satisfies $V_p^{\mu\nu} = V_p^*$ for the worst-case environmental policy ν.*

Proof. For each state $s \in S_r$, only actions $a \in A(s)$ that satisfy $(T_a V_p^*)(s) = V_p^*(s)$ need to be considered, as all other actions cannot be optimal. We call these *possible actions*. Every state has at least one possible action by construction. A state $s \in visited$ if a possible action has been selected for it that also has a positive probability of leaving S_r. Thus, states in *visited* are not in an end component in S_r. Initialize $visited = S_0 \cup B$. For every state with only one possible action, select that action and add the state to *visited*. For states with multiple possible actions, only select an action if it has a non-zero probability of reaching *visited*, and thus, leaving S_r. It is always possible to choose an action in this manner from the definition of S_r. Select actions this way until *visited* $= S$, and return the corresponding policy μ. By construction, μ satisfies $T_{\mu\nu}V_p^* = V_p^*$, and is proper. □

The optimal control policy for satisfying the LTL specification φ consists of two parts: a stationary deterministic policy for reaching an accepting maximal end component, and a finite-memory deterministic policy for staying there. The former policy is given by Algorithm 4, and is denoted by μ_{reach}. The latter policy is a finite-memory policy π_B that selects actions in a round-robin fashion to ensure that the system stays inside the accepting maximal end component forever and satisfies φ [9]. The overall optimal policy is $\pi_p^* = \mu_{reach}$ if $s \notin B$, and $\pi_p^* = \pi_B$ if $s \in B$. We induce an optimal policy π^* on $\mathcal{M}$ from π_p^*, as described in Section 4.4.1.

Complexity

The (worst-case) size of the deterministic Rabin automaton $\mathcal{A}_\varphi$ is doubly-exponential in the length of the LTL formula φ [30]. Experimental work has shown that deterministic Rabin automaton sizes are often exponential or lower for many common types of LTL formulae [60] . Also, there are fragments of LTL, which include all safety and guarantee properties, that generate a deterministic Rabin automaton whose size is singly-exponential in the length of the formula [3].

The size of the product MDP $\mathcal{M}_p$ is equal to the size of $\mathcal{M}$ times the size of $\mathcal{A}_\varphi$. $\mathcal{M}_p$ has n states and m transitions. Maximal end components can be found in $O(n^2)$ time. Since T is a contraction, an ϵ-suboptimal control policy can be computed in $O(n^2 m \log(1/\epsilon))$ time without uncertainty sets [13] and $O(n^2 m \log(1/\epsilon)^2)$ when using likelihood transition uncertainty sets. Thus, the computational cost for incorporating robustness is $O(\log(1/\epsilon))$ times the non-robust case.

4.7 Example

We demonstrate our robust control approach on a discrete-time point-mass robot model. The system model is $x_{k+1} = x_k + (u_k + d_k)\Delta t$, with state $x \in \mathcal{X} \subset \mathbb{R}^2$, control

$u \in \mathcal{U} \subset \mathbb{R}^2$, disturbance $d \in \mathcal{D} \subset \mathbb{R}^2$, and time interval $\Delta t = t_{k+1} - t_k$ for $k = 0, 1, \ldots$. The disturbance $d \sim \mathcal{N}(0, \Sigma)$ where $\Sigma = \text{diag}(0.225^2, 0.225^2)$ and d has support on $\mathcal{D}$. The control set $\mathcal{U} = [-0.3, 0.3]^2$ and the disturbance set $\mathcal{D} = [-0.225, 0.225]^2$.

The task for the robot is to sequentially visit three regions of interest, while always remaining safe. Once the robot has visited the regions, it should return to the start and remain there. The atomic propositions are $\{\text{home}, \text{unsafe}, R1, R2, R3\}$. The LTL formula for this task is $\varphi = \text{home} \wedge \Diamond\Box\text{home} \wedge \Box\neg\text{unsafe} \wedge \Diamond(R1 \wedge \Diamond(R2 \wedge \Diamond R3))$.

We used Monte Carlo simulation (75 samples) to create a finite MDP abstraction $\mathcal{M}$ of our system model, where each state of $\mathcal{M}$ is a square region $[0, 1]^2$. The actions at each state include transitioning to one of four neighbors (up, left, down, right) or not moving (which guarantees that the robot remains in its current state). Due to symmetry of the regions and robot, we only calculated transitions for one region. The transition probabilities corresponding to action "up" are (up, left, down, right, self, error) = (0.8133, 0.0267, 0.000, 0.0133, 0.120, 0.0267), with other actions defined similarly. The error state is entered if the robot accidentally transitions across multiple states. We used large sample approximations to estimate β^a in (4.10) for each uncertainty level [79].

All computations were run on an 2.4 GHz dual-core desktop with 2 GB of RAM. The deterministic Rabin automaton representing φ has 8 states. The product MDP has 437 states and 8671 transitions. It took 1.8 seconds to compute $\mathcal{M}_{ssp}$. It took 5.7, 4.7, and 0.47 seconds to generate an ϵ-suboptimal control policy with likelihood, interval, and no (nominal) uncertainty sets. In all cases, $\epsilon = 10^{-6}$.

We calculated the worst-case satisfaction probabilities for the optimal nominal and robust (likelihood and interval) policies by letting the environment pick transition matrices given a fixed control policy. Transitions were assumed to be exact if they did not lead to an "unsafe" cell. The robust likelihood policy outperformed the nominal and interval, as shown in Figure 4.2. Figure 4.1 shows sample trajectories of the robot using the robust likelihood and nominal control policies.

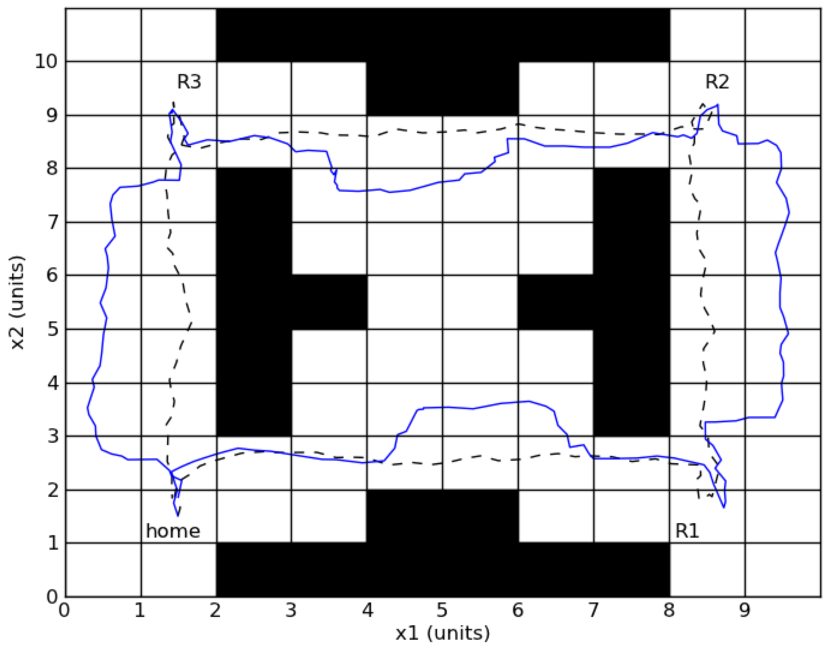


Figure 4.1: Sample trajectories of the robot using the nominal (dotted black) and robust likelihood (solid blue) control policies. Cells that are "unsafe" are in black.

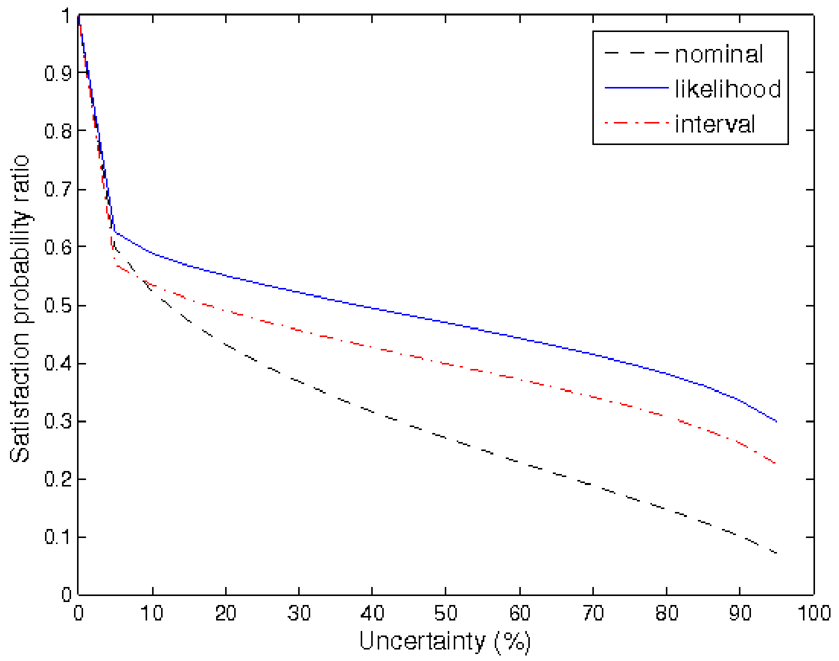


Figure 4.2: Ratio of worst-case satisfaction probability of φ for nominal and robust (likelihood and interval) control policies to the satisfaction probability for the nominal control policy with no uncertainty. Larger ratios are better.

4.8 Conclusions

This chapter presented a method for creating robust control policies for finite MDPs with temporal logic specifications. Robustness is useful when the model is not exact or comes from a finite abstraction of a continuous system. Designing an ϵ-suboptimal robust control policy increases the time complexity only by a factor of $O(\log(1/\epsilon))$ compared to the non-robust policy for statistically relevant uncertainty sets.

Potential future work involves extending these results to other temporal logics, such as probabilistic computational tree logic (PCTL). It would also be useful to weaken the assumptions on the transition matrix uncertainty sets to allow for correlation. Finally, this framework appears useful for a unified analysis of adversarial and stochastic environments.

Chapter 5

Efficient and Optimal Reactive Controller Synthesis for a Fragment of Temporal Logic

This chapter develops a framework for control policy synthesis for both non-deterministic transition systems and Markov decision processes that are subject to temporal logic task specifications. We introduce a fragment of linear temporal logic that can be used to specify common motion planning tasks such as safe navigation, response to the environment, persistent coverage, and surveillance. By restricting specifications to this fragment, we avoid a potentially doubly-exponential automaton construction. We compute feasible control policies in time polynomial in the size of the system and specification. We also compute optimal control policies for average, minimax (bottleneck), and average cost-per-task-cycle cost functions. We highlight several interesting cases when optimal policies can be computed in time polynomial in the size of the system and specification. Additionally, we make connections between computing optimal control policies for an average cost-per-task-cycle cost function and the generalized traveling salesman problem. We give simulation results for representative motion planning tasks, and compare to generalized reactivity(1). This chapter is based on results from [105, 102].

5.1 Introduction

Due to the high computational complexity of control synthesis for non-deterministic systems with LTL [83], there has been much interest in determining fragments of LTL that are computationally efficient to reason about. Fragments of LTL that have exponential complexity for control policy synthesis were analyzed in Alur and La Torre [3]. In the context of timed automata, certain fragments of LTL have been used for efficient control policy synthesis [73]. The generalized reactivity(1), i.e., GR(1), fragment can express many tasks, and control policies can be synthesized in time polynomial in the size of the system [19]. This fragment is extended to generalized Rabin(1), which is the largest such fragment of specifications for which control policy synthesis can be done efficiently [36].

The main contribution of this chapter is an expressive fragment of LTL for efficient control policy synthesis for non-deterministic transition systems and Markov decision processes. A unified approach for control policy synthesis is presented that covers representative tasks and modeling frameworks. The algorithms used are simple, and do not require detailed understanding of automata theory. The fragment used is effectively a Rabin acceptance condition, which allows us to compute directly on the system.

5.2 Preliminaries

5.2.1 System Model

We use non-deterministic finite transition systems and MDPs to model the system behavior. Relevant definitions for MDPs will be introduced in Section 5.8.

Definition 5.1. A *non-deterministic (finite) transition system* (NTS) is a tuple $\mathcal{T} = (S, A, R, s_0, AP, L, c)$ consisting of a finite set of states S, a finite set of actions A, a transition function $R : S \times A \to 2^S$, an initial state $s_0 \in S$, a set of atomic propositions AP, a labeling function $L : S \to 2^{AP}$, and a non-negative cost function $c : S \times A \times S \to \mathbb{R}$.

Let $A(s)$ denote the set of available actions at state s. Denote the parents of the states in the set $S' \subseteq S$ by $\mathit{Parents}(S') := \{s \in S \mid \exists a \in A(s) \text{ and } R(s,a) \cap S' \neq \varnothing\}$. The set $\mathit{Parents}(S')$ includes all states in S that can (possibly) reach S' in a single transition. We assume that the transition system is non-blocking, i.e., $|R(s,a)| \geq 1$ for each state $s \in S$ and action $a \in A(s)$. A *deterministic transition system* (DTS) is a non-deterministic transition system where $|R(s,a)| = 1$ for each state $s \in S$ and action $a \in A(s)$.

A *memoryless control policy* for a non-deterministic transition system $\mathcal{T}$ is a map $\mu : S \to A$, where $\mu(s) \in A(s)$ for state $s \in S$. A *finite-memory control policy* is a map $\mu : S \times M \to A \times M$ where the finite set M is called the memory, and $\mu(s,m) \in A(s) \times M$ for state $s \in S$ and mode $m \in M$. An *infinite-memory* control policy is a map $\mu : S^+ \to A$, where S^+ is a finite sequence of states ending in state s and $\mu(s) \in A(s)$.

Given a state $s \in S$ and action $a \in A(s)$, there may be multiple possible successor states in the set $R(s,a)$, i.e., $|R(s,a)| > 1$. A single successor state $t \in R(s,a)$ is non-deterministically selected. We interpret this selection (or action) as an uncontrolled, adversarial environment resolving the non-determinism. A different interpretation of the environment will be given for MDPs in Section 5.8.

A *run* $\sigma = s_0 s_1 s_2 \ldots$ of $\mathcal{T}$ is an infinite sequence of its states, where $s_i \in S$ is the state of the system at index i and for each $i = 0, 1, \ldots$, there exists $a \in A(s_i)$ such that $s_{i+1} \in R(s_i, a)$. A *word* is an infinite sequence of labels $L(\sigma) = L(s_0)L(s_1)L(s_2)\ldots$ where $\sigma = s_0 s_1 s_2 \ldots$ is a run. The set of runs of $\mathcal{T}$ with initial state $s \in S$ induced by a control policy μ is denoted by $\mathcal{T}^{\mu}(s)$.

Graph Theory

We will often consider a non-deterministic transition system as a graph with the natural bijection between the states and transitions of the transition system and the vertices and edges of the graph. Let $G = (S, R)$ be a directed graph (digraph) with vertices S and edges R. There is an edge e from vertex s to vertex t if and only if $t \in R(s,a)$ for some $a \in A(s)$. A *walk* w is a finite edge sequence $w = e_0 e_1 \ldots e_p$.

Denote the set of all nodes visited along walk w by $Vis(w)$. A digraph $G = (S, R)$ is *strongly connected* if there exists a path between each pair of vertices $s, t \in S$, no matter how the non-determinism is resolved.

5.2.2 A Fragment of Temporal Logic

We use a fragment of temporal logic to specify tasks such as safe navigation, immediate response to the environment, persistent coverage, and surveillance. For a propositional formula φ, the notation $\bigcirc\varphi$ means that φ is true at the next step, $\square\varphi$ means that φ is always true, $\diamond\varphi$ means that φ is eventually true, $\square\diamond\varphi$ means that φ is true infinitely often, and $\diamond\square\varphi$ means that φ is eventually always true [9].

Syntax

We consider formulas of the form

$$\varphi = \varphi_{\text{safe}} \wedge \varphi_{\text{resp}} \wedge \varphi_{\text{resp}}^{\text{ss}} \wedge \varphi_{\text{per}} \wedge \varphi_{\text{task}}, \tag{5.1}$$

where

$$\begin{aligned}
\varphi_{\text{safe}} &:= \square\psi_s, \\
\varphi_{\text{resp}} &:= \bigwedge_{j\in I_r} \square(\psi_{r,j} \implies \bigcirc\phi_{r,j}), \\
\varphi_{\text{resp}}^{\text{ss}} &:= \bigwedge_{j\in I_{sr}} \diamond\square(\psi_{sr,j} \implies \bigcirc\phi_{sr,j}), \\
\varphi_{\text{per}} &:= \diamond\square\psi_p, \\
\varphi_{\text{task}} &:= \bigwedge_{j\in I_t} \square\diamond\psi_{t,j}.
\end{aligned}$$

Note that $\square\psi_s = \square\bigwedge_{j\in I_s}\psi_{s,j} = \bigwedge_{j\in I_s}\square\psi_{s,j}$ and $\diamond\square\psi_p = \diamond\square\bigwedge_{j\in I_p}\psi_{p,j} = \bigwedge_{j\in I_p}\diamond\square\psi_{p,j}$. In the above definitions, I_s, I_r, I_{sr}, I_p, and I_t are finite index sets, and $\psi_{i,j}$ and $\phi_{i,j}$ are propositional formulas for any i and j.

We refer to each $\psi_{t,j}$ in φ_{task} as a recurrent *task*.

Remark 5.1. Guarantee and obligation, i.e., $\diamond p$ and $\square(p \implies \diamond q)$ respectively (where p and q are propositional formulas), are not included in equation (5.1). We show how to include these specifications in Section 5.11.1. It is also natural to consider specifications that are disjunctions of formulas of the form (5.1). We give conditions for this extension in Section 5.11.2.

Remark 5.2. The fragment in formula (5.1) is clearly a strict subset of LTL. This fragment is incomparable to other commonly-used temporal logics, such as computational tree logic (CTL and PCTL), and GR(1). The fragment that we consider allows persistence ($\diamond\square$) to be specified, which cannot be specified in either CTL or GR(1). However, it cannot express existential path quantification as in CTL, or allow disjunctions of formulas as in GR(1) [9, 19]. The fragment is part of the generalized Rabin(1) logic [36] and the μ-calculus of alternation depth two [38].

Remark 5.3. The results in this chapter easily extend to include a fixed order for some or all of the tasks in φ_{task}, as well as ordered tasks with different state constraints between the tasks.

Semantics

We use set operations between a run σ of $\mathcal{T} = (S, A, R, s_0, AP, L, c)$ and subsets of S where particular propositional formulas hold to define satisfaction of a temporal logic formula [46]. We denote the set of states where propositional formula ψ holds by $[\![\psi]\!]$. A run σ *satisfies* the temporal logic formula φ, denoted by $\sigma \vDash \varphi$, if and only if certain set operations hold.

Let σ be a run of the system $\mathcal{T}$, $\mathit{Inf}(\sigma)$ denote the set of states visited infinitely often in σ, and $\mathit{Vis}(\sigma)$ denote the set of states visited at least once in σ. Given propositional formulas ψ and ϕ, we relate satisfaction of a temporal logic formula of the form (5.1) with set operations as follows:

- $\sigma \vDash \square\psi$ if and only if $\mathit{Vis}(\sigma) \subseteq [\![\psi]\!]$,
- $\sigma \vDash \diamond\square\psi$ if and only if $\mathit{Inf}(\sigma) \subseteq [\![\psi]\!]$,

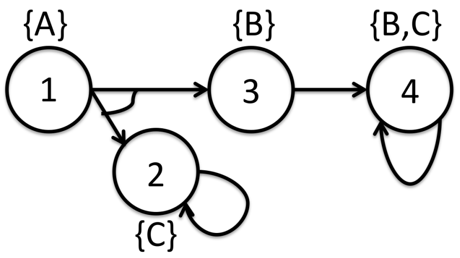


Figure 5.1: Example of a non-deterministic transition system

- $\sigma \vDash \square \diamond \psi$ if and only if $\mathit{Inf}(\sigma) \cap [[\psi]] \neq \emptyset$,
- $\sigma \vDash \square(\psi \implies \bigcirc\phi)$ if and only if $s_i \notin [[\psi]]$ or $s_{i+1} \in [[\phi]]$ for all i,
- $\sigma \vDash \diamond \square (\psi \implies \bigcirc\phi)$ if and only if there exists an index j such that $s_i \notin [[\psi]]$ or $s_{i+1} \in [[\phi]]$ for all $i \geq j$.

A run σ *satisfies* a conjunction of temporal logic formulas $\varphi = \bigwedge_{i=1}^{m} \varphi_i$ if and only if the set operations for each temporal logic formula φ_i holds.

The set $\mathcal{T}^{\mu}(s)$ might include many possible runs because of the non-determinism. A system $\mathcal{T}$ under control policy μ *satisfies* the formula φ at state $s \in S$, denoted $\mathcal{T}^{\mu}(s) \vDash \varphi$, if and only if $\sigma \vDash \varphi$ for all $\sigma \in \mathcal{T}^{\mu}(s)$. Given a system $\mathcal{T}$, state $s \in S$ is *winning* (for the system over the environment) for φ if there exists a control policy μ such that $\mathcal{T}^{\mu}(s) \vDash \varphi$. Let $W \subseteq S$ denote the set of winning states.

An example illustrating the acceptance conditions is given in Figure 5.1. The non-deterministic transition system $\mathcal{T}$ has states $S = \{s_1, s_2, s_3, s_4\}$; labels $L(s_1) = \{a\}$, $L(s_2) = \{c\}$, $L(s_3) = \{b\}$, $L(s_4) = \{b, c\}$; a single action a_0; and transitions $R(s_1, a_0) = \{s_2, s_3\}$, $R(s_2, a_0) = \{s_2\}$, $R(s_3, a_0) = \{s_4\}$, $R(s_4, a_0) = \{s_4\}$. From the acceptance conditions, it follows that $W = \{s_2, s_4\}$ for formula $\square(a \vee c)$, $W = \{s_2, s_3, s_4\}$ for formula $\square(a \implies \bigcirc b)$, $W = \{s_1, s_2, s_3, s_4\}$ for formula $\diamond \square (a \implies \bigcirc b)$, $W = \{s_1, s_2, s_3, s_4\}$ for formula $\square \diamond c$, and $W = \{s_3, s_4\}$ for formula $\diamond \square b$. State s_4 is winning for all of the above formulas.

5.3 Problem Statement

We now formally state the two main problems of this chapter, and give an overview of our solution approach.

Problem 5.1. Given a non-deterministic transition system $\mathcal{T}$ and a temporal logic formula φ of the form (5.1), determine whether there exists a control policy μ such that $\mathcal{T}^{\mu}(s_0) \models \varphi$. Return the control policy μ if it exists.

We introduce a generic cost function J to distinguish among solutions to Problem 5.1. Let J map a set of runs $\mathcal{T}^{\mu}(s_0)$, and the corresponding control policy μ to $\mathbb{R} \cup \infty$.

Problem 5.2. Given a non-deterministic transition system $\mathcal{T}$ and a temporal logic formula φ of the form (5.1), determine whether there exists an optimal control policy μ^* such that $\mathcal{T}^{\mu^*}(s_0) \models \varphi$ and $J(\mathcal{T}^{\mu^*}(s_0)) \leq J(\mathcal{T}^{\mu}(s_0))$ for all feasible μ. Return the control policy μ^* if it exists.

We begin by defining the value function in Section 5.4, which is a key component of all later algorithms. Then, we detail a sound (but not complete) algorithm for computing feasible control policies (i.e., solve Problem 5.1) in Section 5.6. Then, we introduce average cost-per-task-cycle, minimax (bottleneck), and average cost functions in Sections 5.7.2, 5.7.3, and 5.7.4, respectively. We discuss sound procedures for computing optimal control policies for these cost functions in Section 5.7. We solve analogous problems for MDPs in Section 5.8.

Remark 5.4. The restriction to the fragment in (5.1) is critical for tractability. Problem 5.1 is intractable in for the full LTL, as determining if there exists a control policy takes time doubly-exponential in the length of φ [83].

5.4 The Value Function and Reachability

We now introduce standard dynamic programming notions [13], as applied to non-deterministic systems. We consider the case where the controller selects an action, and then the environment selects the next state. Our results easily extend to the case, used in GR(1) [19], where the environment first resolves the non-determinism (selects an action), and then the controller selects its action.

We define *controlled reachability* in a non-deterministic transition system $\mathcal{T}$ with a value function. Let $B \subseteq S$ be a set of states that the controller wants the system to reach. Let the *controlled value function* for system $\mathcal{T}$ and target set B be a map $V^c_{B,\mathcal{T}} : S \to \mathbb{R} \cup \infty$, whose value $V^c_{B,\mathcal{T}}(s)$ at state $s \in S$ is the minimum (over all possible control policies) cost needed to reach the set B, under the worst-case resolution of the non-determinism. If the value $V^c_{B,\mathcal{T}}(s) = \infty$, then the non-determinism can prevent the system from reaching set B from state $s \in S$. For example, consider the system in Figure 5.1 with unit cost on edges and $B = \{s_4\}$. Then, $V^c_B(s_1) = \infty$, $V^c_B(s_2) = \infty$, $V^c_B(s_3) = 1$, and $V^c_B(s_4) = 0$. The system cannot guarantee reaching set B from states s_1 or s_2.

The value function satisfies the optimality condition

$$V^c_{B,\mathcal{T}}(s) = \min_{a \in A(s)} \max_{t \in R(s,a)} V^c_{B,\mathcal{T}}(t) + c(s,a,t),$$

for all $s \in S$.

An optimal control policy μ_B for reaching the set B is memoryless [13], and can be computed at each state $s \in S$ as

$$\mu_B(s) = \arg \min_{a \in A(s)} \max_{t \in R(s,a)} V^c_{B,\mathcal{T}}(t) + c(s,a,t).$$

If multiple actions achieve the minimum, select an action in this set with a minimal number of transitions to reach B.

We use the value function to define the *controllable predecessor* set, $CPre^\infty_{\mathcal{T}}(B)$, for a given system $\mathcal{T}$ with target set $B \subseteq S$. Let $CPre^\infty_{\mathcal{T}}(B) := \{s \in S \mid V^c_{B,\mathcal{T}}(s) < \infty\}$ be the set of all states that can reach a state in B for any resolution of the non-determinism.

We define *forced reachability* similarly. Let the *forced value function* for system $\mathcal{T}$ and target set B be a map $V^f_{B,\mathcal{T}} : S \to \mathbb{R} \cup \infty$, whose value $V^f_{B,\mathcal{T}}(s)$ at state $s \in S$

is the maximum (over all possible control policies) cost of reaching the set B. The forced value function satisfies the optimality condition

$$V^f_{B,\mathcal{T}}(s) = \max_{a\in A(s)} \max_{t\in R(s,a)} V^f_{B,\mathcal{T}}(t) + c(s,a,t).$$

For a given system $\mathcal{T}$ with target set $B \subseteq S$, the *forced predecessor* set $FPre^{\infty}_{\mathcal{T}}(B) := \{s \in S \mid V^f_{B,\mathcal{T}}(s) < \infty\}$, is the set of all states from which no control policy can avoid reaching a state in B.

Remark 5.5. It is not necessary to compute the exact value function at each state when computing the predecessor sets; it is only necessary to compute whether it is finite or infinite.

Computing the Value Function

Algorithm 5 computes the controlled value function as defined previously. This algorithm is a minor extension of Dijkstra's algorithm [29] to non-deterministic transition systems. Similar reasoning applies to the forced value function. Set Q is a priority queue with standard operations ExtractMin (extracts an element with minimal value) and DecreaseKey (updates an element's value).

Algorithm 5 Value function (controlled)

```
Require: NTS $\mathcal{T}$, set $B \subseteq S$
Ensure: The controlled value function $V^c_{B,\mathcal{T}}$ (herein $V$)
  $V(s) \leftarrow \infty$ for all $s \in S - B$
  $V(s) \leftarrow 0$ for all $s \in B$
  $Q \leftarrow S$
  while $Q \neq \emptyset$ do
    $u \leftarrow$ ExtractMin($Q$)
    if $V(u) = \infty$ then
      return $V$
    for $s \in Parents(u) \cap Q$ do
      $tmp \leftarrow \min_{a\in A(s)} \max_{t\in R(s,a)} V(t) + c(s,a,t)$
      if $tmp < V(s)$ then
        $V(s) \leftarrow tmp$
        DecreaseKey($Q, s, V(s)$)
  return $V$
```

Theorem 5.1. *Algorithm 5 computes the controlled value function for all states in $\mathcal{T}$ in $O(|S|log|S| + |R|)$ time.*

Proof. The proof follows that of Dijkstra's algorithm (see Ch. 24.3 in [29]), modified to account for non-determinism. Let $V^*(s)$ denote the optimal cost of reaching set B from state $s \in S$, and $V(s)$ denote the current upper bound. We show that $V(u) = V^*(u)$ whenever a state u is added to $S-Q$, i.e., $u \leftarrow \textsc{ExtractMin}(Q)$. This is trivially true initially. To establish a contradiction, assume state u is the first state added to $S - Q$, with $V(u) > V^*(u)$. Thus, $V^*(u) < \infty$, and there exists a policy to reach B from u. Consider such a policy that reaches a state $y \in Q$ that can reach subset $X \subseteq S-Q$ in a single step, i.e., there exists $a \in A(y), R(y,a) \subseteq X$. Such a state y exists, since B is reachable from u. Since all states in X have optimal costs, $V(y) = V^*(y)$. The non-negativity of edge weights then implies that $V(y) = V^*(y) \le V^*(u) \le V(u)$. The contradiction is that $V(u) \le V(y)$. The algorithm runs in $O(|S|\log|S| + |R|)$ time, since ExtractMin and DecreaseKey are called at most $|S|$ and $|R|$ times, respectively. □

5.5 Feasible Control Policies for Deterministic Transition Systems

We first create control policies for deterministic transition systems. We will compute the winning set $W \subseteq S$ for each specification separately, and then combine them in Algorithm 6. Recall that $\mathcal{T}$ is originally non-blocking.

First, remove all actions from $\mathcal{T}$ that do not satisfy the next-step response specification $\varphi_{\text{resp}} = \bigwedge_{j \in I_r} \Box(\psi_{r,j} \implies \bigcirc \phi_{r,j})$. For each $j \in I_r$, remove an action $a \in A(s)$ from a state $s \in S$ if $s \in [[\psi_{r,j}]]$ and $R(s,a) \notin [[\phi_{r,j}]]$. Let $B \subseteq S$ contain all states that are blocking (due to the removal of an action). Create the subgraph $\mathcal{T}_{\text{resp}} := \mathcal{T}|_{S - FPre^{\infty}_{\mathcal{T}}(B)}$.

Proposition 5.1. *A state is in $\mathcal{T}_{resp}$ if and only if it is winning for φ_{resp}.*

Proof. An action is removed from $\mathcal{T}$ if and only if it directly violates the acceptance condition for φ_{resp}. All blocking states, i.e., those in $B \subseteq S$, must use an action that was removed. Thus, the set $FPre^{\infty}_{\mathcal{T}}(B)$ contains all and only states that violate the acceptance condition for φ_{resp}. $\mathcal{T}_{\text{resp}}$ is non-blocking, so any run of the system satisfies φ_{resp}. □

Next, remove the states that violate the safety specification $\varphi_{\text{safe}} = \Box\psi_s$ by creating the subgraph $\mathcal{T}_{\text{safe}} := \mathcal{T}|_{S - FPre^{\infty}_{\mathcal{T}}(S - [\![\psi_s]\!])}$.

Proposition 5.2. *A state is in $\mathcal{T}_{safe}$ if and only if it is winning for φ_{safe}.*

Proof. The acceptance condition for φ_{safe} is $Vis(\sigma) \subseteq [\![\psi_s]\!]$. The set $FPre^{\infty}_{\mathcal{T}}(S - [\![\psi_s]\!]))$ contains a state if and only if it either is not in $[\![\psi_s]\!]$ or cannot avoid visiting a state not in $[\![\psi_s]\!]$. $\mathcal{T}_{\text{safe}}$ is non-blocking, so any run of the system satisfies φ_{safe}. □

Incorporate the persistence specification $\varphi_{\text{per}} = \Diamond \Box \psi_p$ by creating the subgraph $\mathcal{T}_{\text{per}} := \mathcal{T}|_{S - FPre^{\infty}_{\mathcal{T}}(S - [\![\psi_p]\!])}$. The winning set is $CPre^{\infty}_{\mathcal{T}}(S_{\text{per}})$, where S_{per} is the set of states in $\mathcal{T}_{\text{per}}$.

Proposition 5.3. *A state is in $\mathcal{T}_{per}$ if it is winning for φ_{per}.*

Proof. As in Proposition 5.2, but with acceptance condition $Inf(\sigma) \subseteq [\![\psi_p]\!]$. □

We now compute the winning set for the recurrence specification $\varphi_{\text{task}} = \bigwedge_{j \in I_t} \Box \Diamond \psi_{t,j}$ by computing the sets of states that can be visited infinitely often.

Proposition 5.4. *Let σ be a run of $\mathcal{T}$. If states $s, t \in Inf(\sigma)$, then they must be in the same strongly connected component.*

Proof. By definition of $Inf(\sigma)$, states s and t are visited infinitely often. Thus, there must exist a walk starting at s and ending at t, and vice versa. Thus, s and t are in the same strongly connected component. □

The strongly connected components of $\mathcal{T}$ can be computed in $O(|S| + |R|)$ time using Tarjan's algorithm [29]. Let $SCC(\mathcal{T}_{\text{per}})$ be the set of all strongly connected components of $\mathcal{T}_{\text{per}}$ that have at least one transition between states in the component. A strongly connected component $C \in SCC(\mathcal{T}_{\text{per}})$ is *accepting* if $C \cap [\![\psi_{t,j}]\!] \neq \varnothing$ for all $j \in I_t$. Let $\mathcal{A}$ be the set of all accepting strongly connected components, and

$S_{\mathcal{A}} := \{s \in S \mid s \in C \text{ for some } C \in \mathcal{A}\}$. Every state in an accepting strongly connected component is in the winning set $W := CPre^{\infty}_{\mathcal{T}}(S_{\mathcal{A}})$.

Proposition 5.5. *A state is in $S_{\mathcal{A}}$ if it is winning for φ_{task}.*

Proof. The relevant acceptance condition is $\mathit{Inf}(\sigma) \cap [[\psi_{t,j}]] \neq \varnothing$ for all $j \in I_t$. By definition, every state in $C \in \mathcal{A}$ can be visited infinitely often. Since $C \cap [[\psi_{t,j}]] \neq \varnothing$ for all $j \in I_t$, the result follows. □

We now give an overview of our approach for control policy synthesis for deterministic transition systems in Algorithm 6. Optionally, remove all states from $\mathcal{T}$ that cannot be reached from the initial state s_0 in $O(|S| + |R|)$ time using breadth-first search from s_0 [29]. Compute the set of states W that are winning for φ (lines 1-5). If the initial state $s_0 \notin W$, then no control policy exists (lines 6-8). If $s_0 \in W$, compute a walk on $\mathcal{T}_{\text{safe}}$ from s_0 to a state $t \in C$ for some accepting strongly connected component $C \in \mathcal{A}$ and where $t \in \bigcup_{j \in I_t} [[\psi_{t,j}]]$ (lines 9-10). Compute a walk σ_{suf} starting and ending at state t such that $\mathit{Vis}(\sigma_{\text{suf}}) \cap [[\psi_{t,j}]] \neq \varnothing$ for all $j \in I_t$ and $\mathit{Vis}(\sigma_{\text{suf}}) \subseteq C$ (line 11). The control policy is implicit in the (deterministic) run $\sigma = \sigma_{\text{pre}}(\sigma_{\text{suf}})^{\omega}$, where ω denotes infinite repetition. The total complexity of the algorithm is $O(|I_r|(|S| + |R|))$ to check feasibility and $O((|I_r| + |I_t|)(|S| + |R|))$ to compute a control policy, where the extra term is for computing σ_{suf}. Note that any policy that visits every state in C infinitely often (e.g., randomized or round-robin) could be used to avoid computing σ_{suf}.

5.6 Feasible Control Policies for Non-Deterministic Systems

We now give a sound, but not complete, solution to Problem 5.1 by creating feasible control policies for non-deterministic transition systems that must satisfy a temporal logic formula of the form (5.1). Algorithm 8 summarizes the main results.

Effectively, the formulas aside from φ_{task} restrict states that can be visited, or transitions that can be taken. Safety, φ_{safe}, limits certain states from ever being

Algorithm 6 Feasible Controller Synthesis for Deterministic Transition Systems

```
Require: Deterministic transition system $\mathcal{T}$ with $s_0 \in S$, formula $\varphi$ of the form (5.1)
Ensure: Run $\sigma$
 1: Compute $\mathcal{T}_{\text{act}}$
 2: $\mathcal{T}_{\text{safe}} \leftarrow \mathcal{T}_{\text{act}}|_{S - FPre^{\infty}_{\mathcal{T}_{\text{act}}}(S - [[\psi_s]])}$
 3: $\mathcal{T}_{\text{per}} \leftarrow \mathcal{T}_{\text{safe}}|_{S - FPre^{\infty}_{\mathcal{T}_{\text{safe}}}(S - [[\psi_p]])}$
 4: $\mathcal{A} := \{C \in \text{ SCC } (\mathcal{T}_{\text{per}}) \mid C \cap [[\psi_{t,j}]] \neq \varnothing \; \forall j \in I_t\}$
 5: $S_{\mathcal{A}} := \{s \in S \mid s \in C$ for some $C \in \mathcal{A}\}$
 6: if $s_0 \notin W := CPre^{\infty}_{\mathcal{T}_{\text{safe}}}(S_{\mathcal{A}})$ then
 7:     return "no satisfying control policy exists"
 8: end if
 9: Pick state $t \in C$ for some $C \in \mathcal{A}$ and $t \in \bigcap_{j \in I_t} [[\psi_{t,j}]]$
10: Compute walk $\sigma_{\text{pre}}$ from $s_0$ to $t$ s.t. $Vis(\sigma_{\text{pre}}) \subseteq W$
11: Compute walk $\sigma_{\text{suf}}$ from $t$ to $t$, s.t. $Vis(\sigma_{\text{suf}}) \subseteq C \subseteq W$ and $Vis(\sigma_{\text{suf}}) \cap [[\psi_{t,j}]] \neq \varnothing \; \forall j \in I_t$
12: return $\sigma = \sigma_{\text{pre}}(\sigma_{\text{suf}})^{\omega}$
```

visited, next-step response, φ_{resp}, limits certain transitions from ever being taken, persistence, φ_{per}, limits certain states from being visited infinitely often, and steady-state, next-step response, $\varphi^{\text{ss}}_{\text{resp}}$, limits certain transitions from being taken infinitely often. The remaining formula is recurrence, φ_{task}, which constrains certain states to be visited infinitely often. One creates the subgraph that enforces all constraints except for φ_{task}, and then computes a finite-memory control policy that repeatedly visits all φ_{task} constraints. Then, one relaxes the constraints that only have to be satisfied over infinite time, and computes all states that can reach the states that are part of this policy. A feasible control policy exists for all and only the states in this set. Details are in Algorithm 8.

We now discuss control policy construction for non-deterministic transition systems, which subsumes the development in Section 5.5. Our approach here differs primarily in the form of the control policy and how to determine the set of states that satisfy the persistence and recurrence formulas.

We address formulas for φ_{act} and φ_{safe} in a similar manner to that in Section 5.5, because both the set $FPre^{\infty}$ and the subgraph operation are already defined for non-deterministic transition systems.

Next, consider the recurrence specification $\varphi_{\text{task}} = \bigwedge_{j \in I_t} \Box \Diamond \psi_{t,j}$. An approach similar to the strongly connected component decomposition in Section 5.5 could be used, but it is less efficient to compute due to the non-determinism. Büchi (and the more general parity) acceptance conditions have been extensively studied [19, 46].

The key insight is that the ordering of tasks does not affect feasibility, which is not the case for optimality (see Section 5.7).

Proposition 5.6. *Algorithm 7 computes a subset of the winning set for φ_{task}.*

Proof. To satisfy the acceptance condition $\mathit{Inf}(\sigma) \cap [[\psi_{t,j}]] \neq \varnothing$ for all $j \in I_t$, there must exist non-empty sets $F_j \subseteq [[\psi_{t,j}]]$ such that $F_i \subseteq \mathit{CPre}^{\infty}_{\mathcal{T}}(F_j)$ holds for all $i, j \in I_t$, i.e., all tasks are completed infinitely often. Algorithm 7 selects an arbitrary order on the tasks, e.g., $F_{i+1} \subseteq \mathit{CPre}^{\infty}_{\mathcal{T}}(F_i)$ for all $i = 1, 2, \ldots, |I_t| - 1$ and $F_1 \subseteq \mathit{CPre}^{\infty}_{\mathcal{T}}(F_{|I_t|})$, without loss of generality, since tasks are completed infinitely often and the graph does not change. Starting with sets $F_j := [[\psi_{t,j}]]$ for all $j \in I_t$, all and only the states in each F_j that do not satisfy the constraints are iteratively removed. At each iteration, at least one state in F_1 is removed, or the algorithm terminates. At termination, each F_j is the largest subset of $[[\psi_{t,j}]]$ from which φ_{task} can be satisfied. □

The outer while loop runs at most $|F_1|$ iterations. During each iteration, $\mathit{CPre}^{\infty}_{\mathcal{T}}$ is computed $|I_t|$ times, which dominates the time required to compute the set intersections (when using a hash table). Thus, the total complexity of Algorithm 7 is $O(|I_t| F_{\min}(|S| + |R|))$, where $F_{\min} = \min_{j \in I_t} |F_j|$.

Feasible control policy synthesis is detailed in Algorithm 8. Compute the set W of states that are winning for φ (lines 1-7). If the initial state $s_0 \notin W$, then no control policy exists. If $s_0 \in W$, compute the memoryless control policy $\mu_{S_{\mathcal{A}}}$, which reaches the set $S_{\mathcal{A}}$. Use $\mu_{S_{\mathcal{A}}}$ until a state in $S_{\mathcal{A}}$ is reached. Then, compute the memoryless control policies μ_j induced from $V^c_{F_j, \mathcal{T}_{\text{safe}}}$ for all $j \in I_t$ (line 11, also see Algorithm 7). The finite-memory control policy μ is defined as follows by switching between memoryless policies depending on the current task. Let $j \in I_t$ denote the current task set F_j. The system uses μ_j until a state in F_j is visited. Then, the system updates its task to $k = (j + 1, \mod |I_t|) + 1$ and uses control policy μ_k until a state in F_k is visited, and

Algorithm 7 BUCHI ($\mathcal{T}$, $\{[[\psi_{t,j}]]$ for $j \in I_t\}$)

```
Require: Non-deterministic transition system $\mathcal{T}$, $F_j := [[\psi_{t,j}]] \subseteq S$ for $j \in I_t$
Ensure: Winning set $W \subseteq S$
 1: while $True$ do
 2:   for $i = 1, 2, 3, \ldots, |I_t| - 1$ do
 3:     $F_{i+1} \leftarrow F_{i+1} \cap CPre^{\infty}_{\mathcal{T}}(F_i)$
 4:     if $F_{i+1} = \varnothing$ then
 5:       return $W \leftarrow \varnothing$, $F_j \leftarrow \varnothing$ for all $j \in I_t$
 6:     end if
 7:   end for
 8:   if $F_1 \subseteq CPre^{\infty}_{\mathcal{T}}(F_{|I_t|})$ then
 9:     return $W \leftarrow CPre^{\infty}_{\mathcal{T}}(F_{|I_t|})$, $F_j$ for all $j \in I_t$
10:   end if
11:   $F_1 \leftarrow F_1 \cap CPre^{\infty}_{\mathcal{T}}(F_{|I_t|})$
12: end while
```

so on. The total complexity of the algorithm is $O((|I_r| + |I_{sr}| + |I_t||F_{\min}|)(|S| + |R|))$, which is polynomial in the size of the system and specification.

Algorithm 8 Feasible Controller Synthesis for Non-Deterministic Transition Systems

```
Require: Non-deterministic transition system $\mathcal{T}$ and formula $\varphi$
Ensure: Control policy $\mu$
 1: Compute $\mathcal{T}_{\text{resp}}$ on $\mathcal{T}$
 2: $\mathcal{T}_{\text{safe}} \leftarrow \mathcal{T}_{\text{resp}}|_{S - FPre^{\infty}(S - [[\psi_s]])}$
 3: $\mathcal{T}_{\text{per}} \leftarrow \mathcal{T}_{\text{safe}}|_{S - FPre^{\infty}(S - [[\psi_p]])}$
 4: Compute $\mathcal{T}^{\text{ss}}_{\text{resp}}$ on $\mathcal{T}_{\text{per}}$
 5: $\Psi := \{[[\psi_{t,j}]]$ for all $j \in I_t\}$
 6: $S_{\mathcal{A}}, F := \{F_1, \ldots, F_{|I_t|}\} \leftarrow$ BUCHI$(\mathcal{T}^{\text{ss}}_{\text{resp}}, \Psi)$
 7: $W := CPre^{\infty}_{\mathcal{T}_{\text{safe}}}(S_{\mathcal{A}})$
 8: if $s_0 \notin W$ then
 9:   return "no control policy exists"
10: end if
11: $\mu_{S_{\mathcal{A}}} \leftarrow$ control policy induced by $V^c_{S_{\mathcal{A}}, \mathcal{T}_{\text{safe}}}$
12: $\mu_j \leftarrow$ control policy induced by $V^c_{F_j, \mathcal{T}_{\text{safe}}}$ for all $j \in I_t$
13: return Control policies $\mu_{S_{\mathcal{A}}}$ and $\mu_j$ for all $j \in I_t$
```

5.7 Optimal Control Policies for Non-Deterministic Transition Systems

We now give a sound, but not complete, solution to Problem 5.2 by computing control policies for non-deterministic transitions systems that satisfy a temporal logic formula of the form (5.1) and also minimize a cost function. We consider average cost-per-task-cycle, minimax, and average cost functions. The last two admit polynomial time solutions for deterministic and special cases of non-deterministic transition systems. However, we begin with the average cost-per-task-cycle cost function, as it is quite natural in applications.

The values these cost functions take are independent of any finite sequence of states, as they depend only on the long-term behavior of the system. Thus, we optimize the infinite behavior of the system, which corresponds to completing tasks specified by φ_{task} on a subgraph of $\mathcal{T}$ as constructed in Section 5.6. We assume that we are given $\mathcal{T}_{\text{resp}}^{\text{ss}}$ (denoted hereafter by $\mathcal{T}_{\text{inf}}$) and the task sets $F_j \subseteq S$ returned by Algorithm 7 (see Algorithm 8). Note that $\mathcal{T}_{\text{inf}}$ is the largest subgraph of $\mathcal{T}$ where all constraints from φ_{safe}, φ_{resp}, φ_{per}, and $\varphi_{\text{resp}}^{\text{ss}}$ hold. Each F_j is the largest set of states for the jth task that are part of a feasible control policy. The problem is now to compute a feasible (winning) control policy that also minimizes the relevant cost function.

5.7.1 The Task Graph

Since only the recurrent tasks in φ_{task} on $\mathcal{T}_{\text{inf}}$ will matter for optimization, we construct a new graph that encodes the cost of moving between all tasks. A *task graph* $G' = (V', E')$ encodes the cost of optimal control policies between all tasks in φ_{task} (see Figure 5.2). Let V' be partitioned as $V' = \bigcup_{j \in I_t} V'_j$, where $V'_i \cap V'_j = \emptyset$ for all $i \neq j$. Let $F_j \subseteq S$ denote the set of states that correspond to the jth task in φ_{task}, as returned from Algorithm 7. Create a state $v \in V'_j$ for each of the $2^{|F_j|} - 1$ non-empty subsets of F_j that are reachable from the initial state. Define the map $\tau : V' \to 2^S$ from each

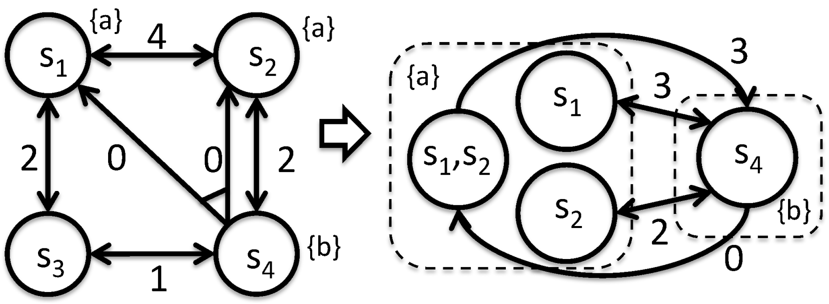


Figure 5.2: A non-deterministic transition system and its task graph (right).

state in V' to subsets of states in S. For each state $v \in V'$, compute the controlled value function $V^c_{\tau(v),\mathcal{T}_{\text{inf}}}$ on $\mathcal{T}_{\text{inf}}$. For all states $u \in V'_i$ and $v \in V'_j$ where $i \neq j$, define an edge $e_{uv} \in E'$. Assign a cost to edge e_{uv} as $c_{uv} := \max_{s \in \tau(u)} V^c_{\tau(v),\mathcal{T}_{\text{inf}}}(s)$. The cost c_{uv} is the maximum worst-case cost of reaching a state $t \in \tau(v)$ from a state $s \in \tau(u)$, when using an optimal control policy.

It is necessary to consider all subsets of states, as the cost of reaching each subset may differ due to the non-determinism. For deterministic systems, one can simply create a state in V'_j for each state in F_j. This is because the cost of all subsets of F_j can be determined by the costs to reach the individual states in F_j.

It may be costly to compute the task graph in its entirety. By incrementally constructing the task graph, one can tradeoff between computation time and conservatism. For example, one can create a task graph with $|I_t|$ states, where each state corresponds to the set F_j. This gives a control policy that leads to an upper bound on the cost of an optimal policy. Additionally, by defining edges in the task graph as the minimum worst-case cost $\min_{s \in \tau(u)} V^c_{\tau(v),\mathcal{T}_{\text{inf}}}(s)$ between tasks, one can compute a lower bound on the cost of an optimal policy. One can use the current control policy and improve performance in an anytime manner by adding more states to the subgraph corresponding to subsets of each F_j.

Algorithm 9 Optimal Controller Synthesis for Non-Deterministic Transition Systems

```
Require: NTS $\mathcal{T}$, formula $\varphi$, cost function $J$
Ensure: Optimal control policy $\mu^*$
1: Compute $\mathcal{T}_{\text{resp}}^{\text{ss}}$, $S_{\mathcal{A}}$, and $F_j$ for all $j \in I_t$ (see Alg. 8)
2: Compute $F_j^* \subseteq F_j$ for all $j \in I_t$ and optimal task order
3: $\mu_{F^*}^* \leftarrow$ control policy from $V_{F^*,\mathcal{T}_{\text{safe}}}^c$ where $F^* = \cup_{j \in I_t} F_j^*$
4: $\mu_j^* \leftarrow$ control policy from $V_{F_j^*,\mathcal{T}_{\text{safe}}}^c$ for all $j \in I_t$
5: return $\mu_{F^*}^*$, $\mu_j^*$ for all $j \in I_t$ and optimal task order
```

5.7.2 Average Cost-Per-Task-Cycle

Recall that for $\varphi_{\text{task}} = \bigwedge_{j \in I_t} \square \diamondsuit \psi_{t,j}$, the propositional formula $\psi_{t,j}$ is the jth *task*. A run σ of system $\mathcal{T}$ *completes* the jth task at time t if and only if $\sigma_t \in [\![\psi_{t,j}]\!]$. A *task cycle* is a sequence of states that completes each task at least once, i.e., it intersects $[\![\psi_{t,j}]\!]$ for each $j = 1, \ldots, m$ at least once. Similarly to [24], we minimize the average cost-per-task-cycle, or equivalently the maximum cost of a task cycle in the limit. For a deterministic system, this corresponds to finding a cycle of minimal cost that completes every task.

We define the cost function over a run σ. Let σ be a run of $\mathcal{T}$ under control policy μ, $\mu(\sigma)$ be the corresponding control input sequence, and $I_{TC}(t) = 1$ indicate that the system completes a task cycle at time t and $I_{TC}(t) = 0$ otherwise. The *average cost per task cycle* of run σ is

$$J'_{TC}(\sigma, \mu(\sigma)) := \limsup_{n \to \infty} \frac{\sum_{t=0}^{n} c(s_t, \mu(s_t), s_{t+1})}{\sum_{t=0}^{n} I_{TC}(t)},$$

which maps runs and control inputs of $\mathcal{T}$ to $\mathbb{R} \cup \infty$. This map is well-defined when (i) $c(\sigma_t, \mu(\sigma_t), \sigma_{t+1})$ is bounded for all $t \geq 0$, and (ii) there exists a $t' \in \mathbb{N}$ such that $I_{TC}(t) = 1$ for infinitely many $t \geq t'$. We assume that (i) is true in the sequel, and note that (ii) holds for every run that satisfies a formula φ with at least one task.

We define the average per-task-cycle cost function

$$J_{TC}(\mathcal{T}^{\mu}(s)) := \max_{\sigma \in \mathcal{T}^{\mu}(s)} J'_{TC}(\sigma, \mu(\sigma)) \qquad (5.2)$$

over the set of runs of system $\mathcal{T}$ starting from initial state s under control policy μ. The cost function (5.2) does not depend on any finite behavior of the system, intuitively because any short-term costs are averaged out in the limit.

We next show that Problem 5.2 with cost function J_{TC} is at least as hard as the NP-hard generalized traveling salesman problem [80].

Generalized traveling salesman problem [80]: Let $G = (V, E)$ be a digraph with vertices V, edges E, and a non-negative cost c_{ij} on each edge $(i, j) \in E$. Set V is the disjoint union of p vertex sets, i.e., $V = V_1 \cup \ldots \cup V_p$, where $V_i \cap V_j = \varnothing$ for all $i \neq j$. There are no edges between states in the same vertex set. The *generalized traveling salesman problem*, GTSP $= \langle (V, E), c\rangle$, is to find a minimum cost cycle that includes a single state from each V_i for all $i = 1, \ldots, p$.

Theorem 5.2. *Any instance of the generalized traveling salesman problem can be reduced (in polynomial time) to an equivalent instance of Problem 5.2 with the cost function J_{TC}.*

Proof. The proof is by construction. Given an instance of the GTSP $\langle (V, E), c\rangle$, we solve Problem 5.2 on a deterministic transition system $\mathcal{T} = (S, A, R, s_0, AP, L, c)$ and formula φ. Let $S = V \cup \{s_0\}$. Define the transitions as $R(u, a_v) = v$, with action $a_v \in A(u)$, and costs $c(u, a_v, v) = c_{uv}$ for each edge $e_{uv} \in E$. Label all states in vertex set V_i with atomic proposition L_i and let $\varphi = \bigwedge_{i \in p} \Box \Diamond L_i$. Finally, add transitions from s_0 to every other state $s \in S$. Although Problem 5.2 does not require that each task is only completed once per cycle, an optimal solution always exists, which completes each task once per cycle. $\square$

Recall that I_t is the index set of all recurrent tasks. We can fix an arbitrary task ordering, denoted $I_t = 1, \ldots, |I_t|$ (with some abuse of notation), without loss of generality for feasible control policies. However, the order that we visit tasks matters for optimal control policies. Additionally, we can select this task order ahead of time, or update it during execution. We now (potentially conservatively) assume that we will select the task order ahead of time. This assumption is not necessary in Sections 5.7.3 or 5.7.4.

We will optimize the task order over all permutations of fixed task orders. This optimization is a generalized traveling salesman problem on the task graph. While this is an NP-hard problem, practical methods exist for computing exact and approximate solutions [80]. Once the optimal ordering of tasks is computed, the finite-memory control policy switches between these tasks in a similar manner described in Section 5.6.

5.7.3 Minimax (Bottleneck) Costs

We now consider a minimax (bottleneck) cost function, which minimizes the maximum accumulated cost between completion of tasks. The notation loosely follows [90], which considers a generalization of this cost function for deterministic transition systems with LTL. Let $\mathbb{T}_{\text{task}}(\sigma, i)$ be the accumulated cost at the ith completion of a task in φ_{task} along a run σ. The *minimax* cost of run σ is

$$J'_{\text{bot}}(\sigma, \mu(\sigma)) := \limsup_{i \to \infty}(\mathbb{T}_{\text{task}}(i+1) - \mathbb{T}_{\text{task}}(i)), \tag{5.3}$$

which maps runs and control inputs of $\mathcal{T}$ to $\mathbb{R} \cup \infty$.

Define the worst-case minimax cost function as

$$J_{\text{bot}}(\mathcal{T}^{\mu}(s)) := \max_{\sigma \in \mathcal{T}^{\mu}(s)} J'_{\text{bot}}(\sigma, \mu(\sigma)) \tag{5.4}$$

over the set of runs of system $\mathcal{T}$ starting from initial state s under control policy μ.

We now solve Problem 5.2 for the cost function J_{bot}. First, compute the task graph as in Section 5.7. The edges in the task graph correspond to the maximum cost accumulated between completion of tasks, assuming that the system uses an optimal strategy. Thus, a minimal value of J_{bot} can be found by minimizing the maximum edge in the task graph, subject to the constraint that a vertex corresponding to each task can be reached. Select an estimate of J_{bot}, and remove all edges in the task graph that are greater than this value. If there exists a strongly connected component of the task graph that contains a state corresponding to each task and is reachable from the initial state, then we have an upper bound on J_{bot}. If not, we have a lower bound. This observation leads to a simple procedure where one selects an estimate of J_{bot} as the median of edge costs that satisfy the previously computed bounds, removes all edges with costs greater than this estimate, determines if the subgraph is strongly connected (with respect to the tasks), and then updates the bounds. Each iteration requires the computation of strongly connected components and the median of edge costs, which can be done in linear time [29]. It is easy to see that this procedure terminates with the correct value of J_{bot} in $O(\log|E'|)$ iterations. Thus, the total complexity is $O(\log|E'|(|V'| + |E'|))$, giving a polynomial time algorithm for deterministic transition systems and non-deterministic transition systems with a single state per task, i.e., $|F_j| = 1$ for all $j \in I_t$.

5.7.4 Average Costs

The *average* cost of run σ is

$$J'_{\text{avg}}(\sigma, \mu(\sigma)) := \limsup_{n\to\infty} \frac{\sum_{t=0}^{n} c(s_t, \mu(s_t), s_{t+1})}{n},$$

which maps runs and control inputs of $\mathcal{T}$ to $\mathbb{R} \cup \infty$.

We now define the worst-case average cost function,

$$J_{\text{avg}}(\mathcal{T}^{\mu}(s)) := \sup_{\sigma \in \mathcal{T}^{\mu}(s_0)} J'_{\text{avg}}(\sigma, \mu(\sigma)) \tag{5.5}$$

over the set of runs of system $\mathcal{T}$ starting from initial state s under control policy μ. Note that this cost function corresponds to J_{TC} when $\varphi_{\text{task}} = \Box \Diamond \mathit{True}$, but without additional tasks.

For non-deterministic transition systems, Problem 5.2 reduces to solving a mean-payoff parity game on $\mathcal{T}_{\text{inf}}$ [22]. An optimal control policy will typically require infinite memory, as opposed to the finite-memory control policies that we have considered. Such a policy alternates between a feasible control policy and an unconstrained minimum cost control policy, spending an increasing amount of time using the unconstrained minimum cost control policy. Given the subgraph $\mathcal{T}_{\text{inf}}$ and the feasible task sets $F_j \subseteq S$ (see Algorithm 8), one can compute an optimal control policy using the results of Chatterjee et al. [22]. For deterministic systems, extensions to a more general weighted average cost function can be found in Chapter 3.

5.8 A Note on Markov Decision Processes

We now consider the Markov decision process (MDP) model. MDPs provide a general framework for modeling non-determinism (e.g., system actions) and probabilistic (e.g., environment actions) behaviors that are present in many real-world systems. We interpret the environment differently than in previous sections; it acts probabilistically through a transition probability, function instead of non-deterministically. We sketch an approach for control policy synthesis for formulas of the form $\varphi = \varphi_{\text{safe}} \wedge \varphi_{\text{per}} \wedge \varphi_{\text{rec}}$ using techniques from probabilistic model checking [9]. We recall definitions from Section 4.2 for convenience.

Definition 5.2. A (finite) *labeled MDP* $\mathcal{M}$ is the tuple $\mathcal{M} = (S, A, P, s_0, AP, L)$, consisting of a finite set of states S, a finite set of actions A, a transition probability function $P : S \times A \times S \to [0,1]$, an initial state s_0, a finite set of atomic propositions AP, and a labeling function $L : S \to 2^{AP}$. Let $A(s)$ denote the set of available actions at state s. Let $\sum_{s' \in S} P(s,a,s') = 1$ if $a \in A(s)$, and $P(s,a,s') = 0$ otherwise. We assume, for notational convenience, that the available actions $A(s)$ are the same for every $s \in S$.

A *run* of the MDP is an infinite sequence of its states, $\sigma = s_0 s_1 s_2 \dots$ where $s_i \in S$ is the state of the system at index i, and $P(s_i, a, s_{i+1}) > 0$ for some $a \in A(s_i)$. The set of runs of $\mathcal{M}$ with initial state s induced by a control policy μ (as defined in Section 5.2.1) is denoted by $\mathcal{M}^{\mu}(s)$. There is a probability measure over the runs in $\mathcal{M}^{\mu}(s)$ [9].

Given a run of $\mathcal{M}$, the syntax and semantics of LTL are identical to a non-deterministic system. However, satisfaction for an MDP $\mathcal{M}$ under a control policy μ

is now defined probabilistically [9]. Let $\mathbb{P}(M^{\mu}(s) \vDash \varphi)$ denote the *expected satisfaction probability* of LTL formula φ by $\mathcal{M}^{\mu}(s)$.

Problem 5.3. Given an MDP $\mathcal{M}$ with initial state s_0 and an LTL formula φ, compute the control policy $\mu^* = \arg\max_{\mu} \mathbb{P}(\mathcal{M}^{\mu}(s_0) \vDash \varphi)$, over all possible finite-memory, deterministic policies.

The value function at a state now has the interpretation as the maximum probability of the system satisfying the specification from that state. Let $B \subseteq S$ be a set from which the system can satisfy the specification almost surely. The value $V_{B,\mathcal{M}}(s)$ of a state $s \in S$ is the probability that the MDP $\mathcal{M}$ will reach set $B \subseteq S$ when using an optimal control policy starting from state $s \in S$.

First compute the winning set $W \subseteq S$ for the LTL formula $\varphi = \varphi_{\text{safe}} \wedge \varphi_{\text{per}} \wedge \varphi_{\text{rec}}$. The probability of satisfying φ is equivalent to the probability of reaching an *accepting maximal end component* [9]. Informally, accepting maximal end components are sets of states that the system can remain in forever, and where the acceptance condition of φ is satisfied almost surely. These sets can be computed in $O(|S||R|)$ time using graph search [9]. The winning set $W \subseteq S$ is the union of all states that are in some accepting maximal end component.

5.8.1 Reachability

Given the winning set $W \subseteq S$, where the system can satisfy the specification φ almost surely, it remains compute a control policy to reach W from the initial state s_0. Let $\mathcal{M}_{\text{safe}}$ be the sub-MDP (see [9]) where all states satisfy φ_{safe}. The set $S_1 = CPre^{\infty}(W)$ contains all states that can satisfy φ almost surely. Let S_r be the set of states that have positive probability of reaching W, which can be computed by graph search [9]. The remaining states $S_0 = S - (S_1 \cup S_r)$ cannot reach W, and thus have zero probability of satisfying φ. Initialize $V_B^c(s) = 1$ for all $s \in S_1$, $V_B^c(s) = 0$ for all $s \in S_0$, and $V_B^c(s) \in (0,1)$ for all $s \in S_r$. It remains to compute the value function, i.e. the maximum probability of satisfying the specification, for each state in S_r. This computation boils down to a standard reachability problem that can be solved by linear programming or value iteration [9, 13].

5.8.2 Control Policy

The control policy for maximizing the probability of satisfying the LTL formula φ consists of two parts: a memoryless deterministic policy for reaching an accepting maximal end component, and a finite-memory deterministic policy for staying there. The former policy is computed from $V^c_{B,\mathcal{M}}$ and denoted μ_{reach}. The latter policy is a finite-memory policy μ_B that selects actions to ensure that the system stays inside the accepting maximal end component forever, and satisfies φ by visiting every state infinitely often [9]. The control policy μ^* is $\mu^* = \mu_{\text{reach}}$ if $s \notin B$ and $\mu^* = \mu_B$ if $s \in B$.

5.8.3 Optimal Control

A similar task graph construction to Section 5.7.1 can be used to compute optimal control policies. A task graph is computed for each accepting maximal end component. Then, one can maximize the probability of reaching the union of all states in accepting maximal end components that meet a user-specified cost threshold.

5.9 Complexity

We summarize our complexity results for feasible control policy synthesis, and compare with LTL and GR(1) [19]. We assume that set membership is determined in constant time with a hash function [29], and that the transition system $\mathcal{T}$ is represented as a game graph [46]. We denote the length of a temporal logic formula by $|\varphi|$. Let $|\varphi| = |I_r| + |I_{sr}| + |I_t|$ for the fragment in (5.1), $|\varphi| = mn$ for a GR(1) formula with m assumptions and n guarantees, and $|\varphi|$ be the formula length for LTL [9]. Recall that $F_{\min} = \min_{j \in I_t} |[[\psi_{t,j}]]|$. For typical motion planning specifications, $F_{\min} \ll |S|$ and $|\varphi|$ is small. We use the non-symbolic complexity results for GR(1) in Bloem et al. [19]. Results are summarized in Table 5.1.

Table 5.1: Complexity of feasible policy synthesis

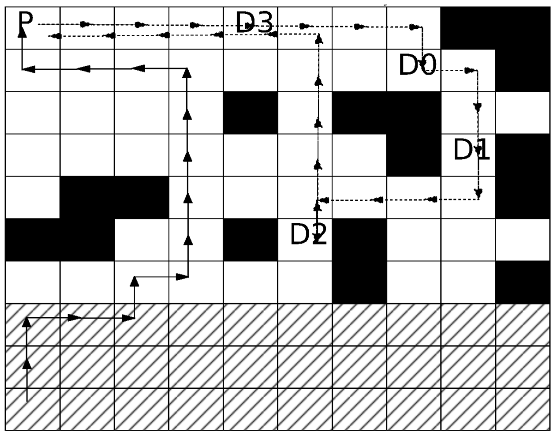
Language	DTS	NTS	MDP
Frag. in (5.1)	$O(\lvert\varphi\rvert(\lvert S\rvert + \lvert R\rvert))$	$O(\lvert\varphi\rvert F_{\min}(\lvert S\rvert + \lvert R\rvert))$	$O(\mathtt{LP}(\lvert\mathcal{T}\rvert))$
GR(1)	$O(\lvert\varphi\rvert\lvert S\rvert\lvert R\rvert)$	$O(\lvert\varphi\rvert\lvert S\rvert\lvert R\rvert)$	N/A
LTL	$O(2^{(\lvert\varphi\rvert)}(\lvert S\rvert + \lvert R\rvert))$	$O(2^{2^{(\lvert\varphi\rvert)}}(\lvert S\rvert + \lvert R\rvert))$	$O(\mathtt{LP}(\lvert\mathcal{T}\rvert)2^{2^{(\lvert\varphi\rvert)}})$

We now summarize the complexity of optimal control policy synthesis. The task graph $G' = (V', E')$ has $O(\sum_{i\in I_t} 2^{|F_i|}-1)$ states, and can be computed in $O((\sum_{i\in I_t} 2^{|F_i|}-1)(|S|\log|S|+|R|))$ time. Computing an optimal control policy for J_{TC} requires solving an NP-hard generalized traveling salesman problem on G'. Computing an optimal control policy for J_{bot} requires $O(\log|E'|(|V'|+|E'|)$ time. An optimal control policy for J_{avg} can be computed in pseudo-polynomial time [22]. For deterministic systems, the task graph has $O(\sum_{i\in I_t} |F_i|)$ states, and can be computed in $O((\sum_{i\in I_t} |F_i|)(|S|\log|S| + |R|))$ time. An optimal control policy for J_{avg} can be computed in $O(|S||R|)$ time. Thus, we can compute optimal control policies for deterministic transition systems with cost functions J_{bot} and J_{avg} in time polynomial in the size of the system and specification. Additionally, for non-deterministic transition systems where $|F_j| = 1$ for all $j \in I_t$, we can compute optimal control policies for J_{bot} in time polynomial in the size of the system and specification.

Remark 5.6. The fragment in (5.1) is not handled well by standard approaches. Using ltl2dstar [60], we created Rabin automaton for formulas of the form φ_{resp}. The computation time and automaton size both increased exponentially with the number of conjunctions in φ_{resp}.

5.10 Examples

The following examples (based on those in Wolff et al. [105]) demonstrate the techniques developed in Sections 5.6 and 5.7 for tasks motivated by robot motion planning in a planar environment (see Figure 5.3). All computations were done in Python on a dual-core Linux desktop with 2 GB of memory. All computation times were av-



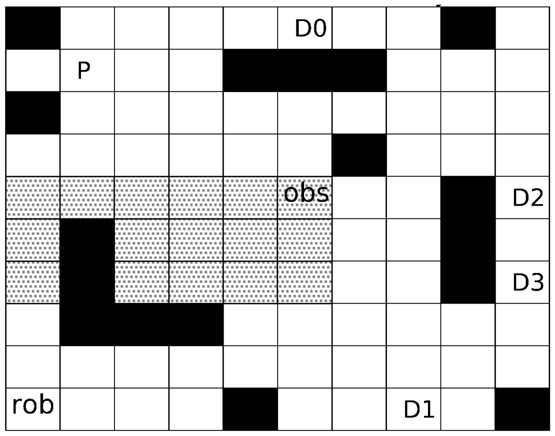


Figure 5.3: Left: Diagram of deterministic setup ($n = 10$). Only white cells are labeled "stockroom." Right: Diagram of non-deterministic setup ($n = 10$). A dynamic obstacle (obs) moves within the shaded region.

eraged over five arbitrarily-generated problem instances, and include construction of the transition system.

5.10.1 Deterministic Transition System

Consider an environment where a robot occupies a single cell at a time, and can choose to either remain in its current cell or move to one of four adjacent cells at each step. We consider square grids with static obstacle densities of 20 percent. The robot's task is to eventually remain in the stockroom while repeatedly visiting a pickup location P and multiple dropoff locations D_0, D_1, D_2, and D_3. The robot must never collide with a static obstacle. The set of atomic propositions is $\{P, D_0, D_1, D_2, D_3, \text{stockroom}, \text{obs}\}$. This task is formalized by $\varphi = \Diamond \Box \, \text{stockroom} \wedge \Box \Diamond P \wedge \bigwedge_{j \in I_t} \Box \Diamond D_j \wedge \Box \neg \text{obs}$. In all following results, D_j holds at a single state in the transition system. Results for optimal control policy synthesis are shown in Figure 5.4 for $n \times n$ grids where $n \in \{200, 300, 400\}$.

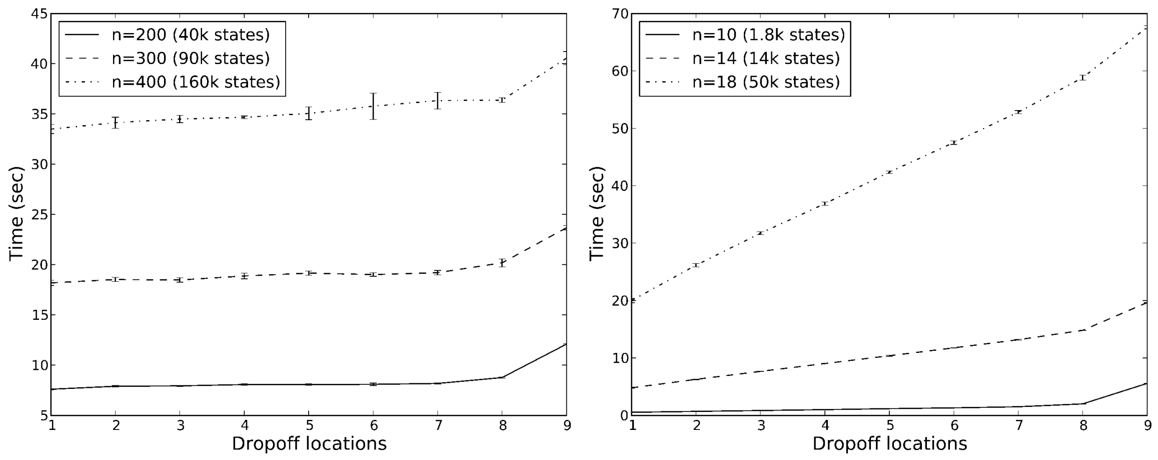


Figure 5.4: Control policy synthesis times for deterministic (left) and non-deterministic (right) grids.

5.10.2 Non-Deterministic Transition System

We now consider a similar setup with a dynamically moving obstacle. The state of the system is the product of the robot's location and the obstacle's location, both of which can move as previously described for the robot. The robot selects an action, and then the obstacle non-deterministically moves. The robot's task is similar to before, and is formalized as $\varphi = \square \diamond P \wedge \bigwedge_{j \in I_t} \square \diamond D_j \wedge \square \neg \text{obs}$. Results for optimal control policy synthesis are shown in Figure 5.4 for $n \times n$ grids where $n \in \{10, 14, 18\}$.

5.11 Extensions

We discuss two natural extensions to the fragment in formula (5.1). The first includes guarantee and obligation properties, and the second includes disjunctions of formulas.

5.11.1 Guarantee and Obligation

While guarantee and obligation, i.e., $\diamond p$ and $\square(p \implies \diamond q)$ respectively (where p and q are propositional formulas), specifications are not explicitly included in (5.1), they can be incorporated by introducing new system variables, which exponentially increases the system size [108]. Even including conjunctions of guarantee formulas is

NP-complete [89].

Another approach is to use the stricter specifications $\square\diamond p$ for guarantee and $\square\neg p \vee \square\diamond q$ for obligation. The $\square\diamond$ formulas are part of the fragment in (5.1), and disjunctions can be included in some cases (see Section 5.11.2). If the transition system is strongly connected, then these stricter formulas are feasible if and only if the original formulas are. Strong connectivity is a natural assumption in many robotics applications. For example, an autonomous car can typically drive around the block to revisit a location.

5.11.2 Disjunctions of Specifications

We now consider an extension to specifications that are disjunctions of formulas of the form (5.1).

For a deterministic transition system, a control policy for a formula given by disjunctions of formulas of the form (5.1) can be computed by independently solving each individual subformula using the algorithms given earlier in this section.

Proposition 5.7. *Let $\varphi = \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$, where φ_i is a formula of the form* (5.1) *for $i = 1, \ldots, n$. Then, there exists a control policy μ such that $\mathcal{T}^\mu(s) \vDash \varphi$ if and only if there exists a control policy μ such that $\mathcal{T}^\mu(s) \vDash \varphi_i$ for some $i = 1, \ldots, n$.*

Proof. Sufficiency is obvious. For necessity, assume that there exists a control policy μ such that $\mathcal{T}^\mu(s)$ satisfies φ. The set $\mathcal{T}^\mu(s)$ contains a single run σ, since $\mathcal{T}$ is deterministic. Thus, σ satisfies φ_i for some $i = 1, \ldots, n$. □

For non-deterministic transition systems, necessity in Proposition 5.7 no longer holds because the non-determinism may be resolved in multiple ways, and thus, independently evaluating each subformula may not work.

Algorithm 10 is a sound, but not complete, procedure for synthesizing a control policy for a non-deterministic transition system with a specification given by disjunctions of formulas of the form (5.1). Arbitrary disjunctions of this form are intractable to solve exactly, as this extension subsumes Rabin games, which are NP-complete [37].

Algorithm 10 computes winning sets $W_i \subseteq S$ for each subformula φ_i and checks whether the initial state can reach their union $\mathcal{W} := \bigcup_{i=1}^{n} W_i$. The control policy μ_{reach} is used until a state $s \in W_i$ is reached for some i, after which μ_i is used.

Algorithm 10 DISJUNCTION

```
Require: NTS $\mathcal{T}$, formula $\varphi_i$, $i = 1, \dots, n$
Ensure: Winning set $W \subseteq S$ and control policy $\mu$
  $W_i \subseteq S$ and $\mu_i \leftarrow$ winning states and control policy for $\varphi_i$
  $\mathcal{W} \leftarrow \bigcup_{i=1}^{n} W_i$
  if $s_0 \notin CPre_{\mathcal{T}}^{\infty}(\mathcal{W})$ then
    return $\mu = \emptyset$
  $\mu_{\text{reach}} \leftarrow$ control policy induced by $V_{\mathcal{W},\mathcal{T}}^{c}$
  return $\mu_{\text{reach}}$ and $\mu_i$ for all $i$
```

5.12 Conclusions

We presented a framework for control policy synthesis for both non-deterministic transition systems and Markov decision processes that are subject to temporal logic task specifications. Our approach for control policy synthesis is straightforward and efficient, both theoretically and according to our preliminary experimental results. Additionally, we presented a framework for optimal control policy synthesis for non-deterministic transition systems with specifications from a fragment of temporal logic. Our approach is simple, and makes explicit connections with dynamic programming through our extensive use of value functions. Additionally, optimal policies can be computed in polynomial time for certain combinations of cost functions and system restrictions.

Future work includes investigating the incremental computation of the value function and exploring the underlying automata structure of this fragment of temporal logic. Detailed experimental analysis would also be useful for better comparing empirical performance to GR(1) and automata-based methods.

Chapter 6

Automaton-Guided Controller Synthesis for Nonlinear Systems with Temporal Logic

This chapter describes a method for the control of discrete-time nonlinear systems subject to temporal logic specifications. Our approach uses a coarse abstraction of the system and an automaton representing the temporal logic specification to guide the search for a feasible trajectory. This decomposes the search for a feasible trajectory into a series of constrained reachability problems. Thus, one can create controllers for any system for which techniques exist to compute (approximate) solutions to constrained reachability problems. Representative techniques include sampling-based methods for motion planning, reachable set computations for linear systems, and graph search for finite discrete systems. Our approach avoids the expensive computation of a discrete abstraction, and its implementation is amenable to parallel computing. We demonstrate our approach with numerical experiments on temporal logic motion planning problems with high-dimensional (more than 10 continuous state) dynamical systems. This chapter is based on results from [106].

6.1 Introduction

Common approaches to temporal logic motion planning construct a finite discrete abstraction of the dynamical system [15, 53, 61]. An abstraction of a system is a partition of the continuous state space into a finite number of abstract states, i.e., sets of system states, with transitions between abstract states that represent possible system behaviors. Finite abstractions are typically expensive to compute, conservative, and not guided by the underlying specification (see [4, 10, 15, 47, 53, 61, 108]).

Instead of blindly doing expensive reachability computations to construct an abstraction of a dynamical system, we use a coarse abstraction of the system and perform constrained reachability checks as needed for the task. We first create an *existential abstraction*, which is a finite abstraction of the system where transitions between abstract states are assumed to exist, but have yet not been verified, e.g., through reachability computations, to exist in the system. We then create a product automaton from the finite-state abstraction, and an automaton representing the underlying specification. The product automaton guides reasoning about complicated temporal logic properties as a sequence of simple temporal properties that can be analyzed using constrained reachability techniques. This sequence of constrained reachability problems is called an abstract plan. However, the system might not be able to follow a given abstract plan, since dynamic constraints were not considered in the existential abstraction. Thus, we check the abstract plan with the continuous dynamics by solving a sequence of constrained reachability problems. If this sequence is infeasible, the product automaton is updated, and a new abstract plan is generated.

This approach lets one take advantage of the significant work in computing constrained reachability relations over continuous state spaces. The related literature includes robotic motion planning [68], optimization-based methods for trajectory generation [14, 77, 86], and PDE-based methods [92]. Exactness in computing constrained reachability is not critical; we will only require a sound technique.

The main contribution of this chapter is a general framework for synthesizing controllers for nonlinear dynamical systems subject to temporal logic specifications. Our approach is independent of the specific techniques used to compute constrained reachability, and is amenable to parallel computation. Despite the framework's generality, it is also computationally powerful, as we show with examples that improve on state-of-the-art techniques for temporal logic motion planning for high-dimensional dynamical systems with more than 10 continuous states.

This work is part of the counterexample-guided abstraction refinement (CEGAR) framework [26, 5, 91]. Here, an abstract model of the system is checked to see if the specification holds. If the check fails, then the abstraction is refined based on a counterexample (e.g. a system trajectory) generated during the check. Our approach differs in that we associate weights with the abstraction, and use them to update a ranking of promising abstract plans.

This approach is also related to that on combining task and motion planning [15, 21, 52, 81, 82, 96, 98]. These approaches first compute an abstract plan, and then use sampling-based motion planning techniques to check whether the plan satisfies the dynamic constraints. This idea is applied to co-safe linear temporal logic specifications in [15, 81, 82]. Our framework is agnostic to the method used to check constrained reachability, and applies to a wider class of specifications.

This work is also related to the specification-guided work in Gol et al. [45], where they compute feedback controllers. We consider more general systems and specifications. Finally, coarse bisimulations of discrete-time piecewise-affine systems based on temporal logic specifications are computed inYordanov et al. [110]. Our approach focuses on controller synthesis, and does not require the exact computation of reachable sets for the system.

6.1.1 Problem Statement

We consider discrete-time nonlinear systems of the form

$$x_{t+1} = f(x_t, u_t), \quad t = 0, 1, \ldots, \tag{6.1}$$

where $x \in \mathcal{X} \subseteq (\mathbb{R}^{n_c} \times \{0,1\}^{n_l})$ are the continuous and binary states, $u \in \mathcal{U} \subseteq (\mathbb{R}^{m_c} \times \{0,1\}^{m_l})$ are the inputs, and $x_0 \in \mathcal{X}$ is the initial state. The system is called the *concrete* system to distinguish it from its abstraction, which will be introduced in Section 6.2.1.

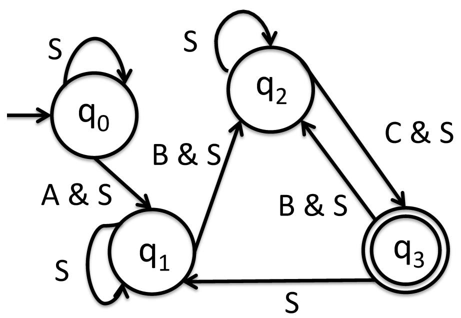


Figure 6.1: A (simplified) Büchi automaton corresponding to the LTL formula $\varphi = \Diamond A \wedge \Box\Diamond B \wedge \Box\Diamond C \wedge \Box S$ (stated without definition). Informally, the system must visit A, repeatedly visit B and C, and always remain in S. Here $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{A, B, C, S\}$, $Q_0 = \{q_0\}$, $F = \{q_3\}$, and transitions are represented by labeled arrows.

Let AP be a finite set of atomic propositions. The *labeling function* $L : \mathcal{X} \to 2^{AP}$ maps the continuous part of each state to the set of atomic propositions that are *True*. The set of states where atomic proposition p holds is denoted by $[[p]]$.

A *run* (*trajectory*) $\mathbf{x} = x_0x_1x_2\ldots$ of system (6.1) is an infinite sequence of its states, where $x_t \in \mathcal{X}$ is the state of the system at index t and for each $t = 0, 1, \ldots$, there exists a control input $u_t \in \mathcal{U}$ such that $x_{t+1} = f(x_t, u_t)$. A *word* is an infinite sequence of labels $L(\mathbf{x}) = L(x_0)L(x_1)L(x_2)\ldots$ where $\mathbf{x} = x_0x_1x_2\ldots$ is a run. Given an initial state x_0 and a control input sequence $\mathbf{u}$, the resulting run $\mathbf{x}(x_0, \mathbf{u})$ is unique.

We now formally state the main problem of this chapter, and give an overview of our solution approach.

Problem 6.1. Given a dynamical system of the form (6.1) with initial state $x_0 \in \mathcal{X}$ and a Büchi automaton $\mathcal{A}$ representing the specification, determine whether there exists a control input sequence $\mathbf{u}$ such that the word $L(\mathbf{x}(x_0, \mathbf{u}))$ is accepted by $\mathcal{A}$. Return the control $\mathbf{u}$ if it exists.

Problem 6.1 is undecidable in general due to the dynamics over a continuous state space [4]. Thus, we consider sound, but not complete, solutions. Our approach is to create an existential finite abstraction $\mathcal{T}$ of the system that does not necessarily check constrained reachability between states in $\mathcal{T}$. Then, we create a product automaton by combining $\mathcal{T}$ with a Büchi automaton $\mathcal{A}$. An accepting run in the product automaton is an abstract plan. However, an abstract plan may be infeasible due to dynamic constraints. We check the corresponding sequence of constrained reachability problems to see if it is dynamically feasible (see Section 6.3). If a trajectory is not found, we update the product automaton and search for a new abstract plan. This process is repeated until a feasible trajectory is found, or no more abstract plans exist.

Remark 6.1. Our general automaton-guided approach may be extended to feedback control of systems with disturbances either by 1) assuming that the disturbances do not change the word, i.e., the sequence of labels, or 2) using an appropriate deterministic automaton.

6.2 The Abstract Model

We now describe an *existential* finite abstraction $\mathcal{T}$. This abstract model over-approximates reachability of the concrete system, i.e., it might produce behaviors that the concrete system cannot execute. This abstract model of the system will late be combined with the Büchi automaton representing the specification.

6.2.1 Existential Abstraction

We use a transition system to represent the existential abstraction of a concrete system.

Definition 6.1. A *deterministic (finite) transition system* is a tuple $\mathcal{T} = (S, R, s_0, AP, L)$ consisting of a finite set of states S, a transition relation $R \subseteq S \times S$, an initial state $s_0 \in S$, a set of atomic propositions AP, and a labeling function $L : S \to 2^{2^{AP}}$.

We use the finite transition system model to define an existential abstraction $\mathcal{T}$ for the concrete system as follows. We partition the concrete system's state space into equivalence classes of states, and associate an abstract state $s \in S$ with each equivalence class. The *concretization map* $C : S \to X \subseteq \mathcal{X}$ maps each abstract state to a subset of the concrete system's state space.

The abstraction $\mathcal{T}$ is existential in the sense that there is an abstract transition $(s,t) \in R$ if there exists a control input such that the system evolves from some concrete state in $C(s)$ to some concrete state in $C(t)$ in finite time. Thus, the existential abstraction $\mathcal{T}$ is an over-approximation of the concrete system in the sense that it contains more behaviors, i.e., a series of transitions might exist for the abstraction that does not exist for the concrete system.

Remark 6.2. A partition is *proposition preserving* if, for every abstract state $s \in S$ and every atomic proposition $p \in AP$, $p \in L(u)$ if and only if $p \in L(v)$ for all concrete states $u, v \in C(s)$ [4]. We do not require that the abstract states used in creating the existential abstraction $\mathcal{T}$ are proposition preserving, which necessitates the non-standard definition of the labeling function.

6.2.2 Product Automaton

We use a slight modification of the product automaton construction [95] to represent runs that are allowed by the transition system and satisfy the specification.

Definition 6.2. Let $\mathcal{T} = (S, R, s_0, AP, L)$ be a transition system, and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ be a Büchi automaton. The *weighted product automaton* $\mathcal{P} = \mathcal{T} \times \mathcal{A}$ is the tuple $\mathcal{P} = (S_\mathcal{P}, \delta_\mathcal{P}, F_\mathcal{P}, s_{\mathcal{P},0}, AP_\mathcal{P}, L_\mathcal{P}, w_\mathcal{P})$, consisting of

(i) a finite set of states $S_\mathcal{P} = S \times Q$,

(ii) a transition relation $\delta_\mathcal{P} \subseteq S_\mathcal{P} \times S_\mathcal{P}$, where $((s,q),(s',q')) \in \delta_\mathcal{P}$ if and only if $(s,s') \in R$ and there exists an $L \in L(s)$ such that $(q, L, q') \in \delta$,

(iii) a set of accepting states $F_\mathcal{P} = S \times F$,

(iv) a set of initial states $S_{\mathcal{P},0}$, with $(s_0, q_0) \in S_{\mathcal{P},0}$ if $q_0 \in Q_0$,

(v) a set of atomic propositions $AP_\mathcal{P} = Q$,

(vi) a labeling function $L_{\mathcal{P}} : S \times Q \to 2^Q$, and

(vii) a non-negative valued weight function $w_{\mathcal{P}} : \delta_{\mathcal{P}} \to \mathbb{R}$.

A run $\sigma_{\mathcal{P}} = (s_0, q_0)(s_1, q_1)\ldots$ is *accepting* if $(s_i, q_i) \in F_{\mathcal{P}}$ for infinitely many indices $i \in \mathbb{N}$. The *projection* of a run $\sigma_{\mathcal{P}} = (s_0, q_0)(s_1, q_1)\ldots$ in the product automaton $\mathcal{P}$ is the run $\sigma = s_0 s_1 \ldots$ in the transition system $\mathcal{T}$.

We will often consider an automaton as a graph with the natural bijection between the states and transitions of the automaton and the vertices and edges of the graph. Let $G = (S, R)$ be a directed graph with vertices S and edges R. There exists an edge e from vertex s to vertex t if and only if $t \in \delta(s, a)$ for some $a \in \Sigma$. A *walk* w is a finite edge sequence $w = e_0 e_1 \ldots e_p$. A *cycle* is a walk where $e_0 = e_p$.

It is well-known that if there exists an accepting run in $\mathcal{P}$, then there exists an accepting run of the form $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^{\omega}$, where σ_{pre} is a finite walk in $\mathcal{P}$, and σ_{suf} is a finite cycle in $\mathcal{P}$ [9]. For an accepting run $\sigma_{\mathcal{P}}$, the suffix σ_{suf} is a cycle in the product automaton $\mathcal{P}$ that satisfies the acceptance condition, i.e., it includes an accepting state. The prefix σ_{pre} is a finite run from an initial state $s_{\mathcal{P},0}$ to a state on an accepting cycle. We call $\sigma_{\mathcal{P}}$ an *abstract plan*.

6.3 Concretizing an Abstract Plan

Given an abstract plan, it is necessary to check whether it is feasible for the concrete system. We first define the constrained reachability problem, and then show how to compose these problems to check the feasibility of an abstract plan.

6.3.1 Set-to-Set Constrained Reachability

We now define the *set-to-set constrained reachability problem*, which is a key component of our solution approach.

Definition 6.3. Consider a concrete system of the form (6.1) with given sets $X_1, X_2 \subseteq \mathcal{X}$, a non-negative integer horizon length N, and a control input sequence u. Set X_2

is *constrained reachable* (under control u) through set X_1, denoted by $X_1 \rightsquigarrow_{X_1} X_2$, if there exist $x_1, \ldots, x_{N-1} \in X_1$, $x_N \in X_2$ such that $x_{t+1} = f(x_t, u_t)$ for $t = 1, \ldots, N-1$.

Constrained reachability problem: Given a system of the form (6.1) and sets $X_1, X_2 \subseteq \mathcal{X}$, find a control input sequence u and a non-negative integer horizon length N such that $X_1 \rightsquigarrow_{X_1} X_2$. Return control u if it exists.

Solving a constrained reachability problem is generally undecidable [4]. However, there exist numerous sound algorithms that compute solutions to constrained reachability problems. Sampling-based algorithms are probabilistically or resolution complete [68]. Optimization-based methods are used for state constrained trajectory generation for nonlinear [14, 77] and linear [86] systems. Computationally expensive PDE-based methods are generally applicable [92]. Finally, for a discrete transition system, computing constrained reachability is simply graph search [9].

We make the standing assumption that there exists an oracle for computing a sound solution to a constrained reachability problem for the system. We denote this method by CstReach(X_1, X_2, N), with constraint set X_1, reach set X_2, and horizon length $N \in \mathbb{N}$. For a given query, CstReach returns Yes or No. Yes returns a control input, and No means that a control input does not exist.

```
**Algorithm 11** CstReach($X_1, X_2, N$)
**Require:** Sets $X_1, X_2 \subseteq \mathcal{X}$, and horizon $N \in \mathbb{N}$
**Ensure:** Yes and control input $u$, No
```

Note that the CstReach oracle is sound, but not complete. If it does not return a solution after a given amount of computation time, nothing about the constrained reachability problem can be inferred: the problem could be infeasible, or feasible but require more computation time.

6.3.2 Concretization of Abstract Plans

The concrete plan is the set of constrained reachability problems corresponding to the transitions along an abstract plan $\sigma = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$. Each transition $((s, q), (s', q')) \in \delta_{\mathcal{P}}$ encodes a constrained reachability problem (see Section 6.3.1) for the concrete system.

We enforce that the system remains in state (s, q) until it eventually reaches state (s', q'). Let $L_1 \in L(s)$ correspond to the set of atomic propositions so that $(q, L_1, q) \in \delta$, and $L_2 \in L(s)$ correspond to the set of atomic propositions so that $(q, L_2, q') \in \delta$. Let $X_1 = [\![L_1]\!]$ if there exists the transition $((s,q),(s,q)) \in \delta_{\mathcal{P}}$ or else $X_1 = \varnothing$, and $X_2 = C(s') \cap [\![L_2]\!]$. Then, the existence of a concrete transition corresponding to the abstract transition $((s,q),(s',q'))$ can be checked by solving $\textsc{CstReach}(X_1, X_2, N)$, for a given horizon length N. These $\textsc{CstReach}$ problems are concatenated along the abstract plan in the obvious manner, with a loop closure constraint for the repeated suffix.

We demonstrate the concatenation of $\textsc{CstReach}$ problems on the example in Figure 6.1. We assume that there is a single abstract state s in the existential abstraction, and thus consider transitions (q, q') instead of $((s,q),(s',q'))$. An abstract plan is given by $q_0(q_1q_2q_3)^\omega$ where $\sigma_{\text{pre}} = q_0$ and $\sigma_{\text{suf}} = q_1q_2q_3$. Let x_k^{ij} denote the kth continuous state along the transition from state q_i to q_j. The sequence of states for this abstract plan is $x_1^{01}, \ldots, x_N^{01}, x_1^{12}, \ldots, x_N^{12}, x_1^{23}, \ldots, x_N^{23}, x_1^{31}, \ldots, x_N^{31}$, where $x_1^{12} = f(x_N^{31}, u)$ for some $u \in \mathcal{U}$ is the loop closure constraint. The corresponding state constraints for the sequence of $\textsc{CstReach}$ problems are $x_1^{01}, \ldots, x_{N-1}^{01} \in [\![S]\!]$, $x_N^{01} \in [\![A \wedge S]\!]$, $x_1^{12}, \ldots, x_{N-1}^{12} \in [\![S]\!]$, $x_N^{12} \in [\![B \wedge S]\!]$, $x_1^{23}, \ldots, x_{N-1}^{23} \in [\![S]\!]$, $x_N^{23} \in [\![C \wedge S]\!]$, and $x_1^{31}, \ldots, x_{N-1}^{31} \in [\![\varnothing]\!]$, $x_N^{31} \in [\![S]\!]$.

6.4 Solution

We outline our solution to Problem 6.1, and discuss tradeoffs regarding the levels of granularity of the existential abstraction. Note that if the specification is given as an LTL formula φ, a corresponding Büchi automaton $\mathcal{A}$ can be automatically computed using standard software [42].

6.4.1 The Solution Algorithm

We now overview our solution approach, as detailed in Algorithm 12. First, create an existential abstraction $\mathcal{T}$ of the concrete system, as described in Section 6.2.1. We discuss tradeoffs on this construction in Section 6.4.2. Then, construct the product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}$.

The problem is now to find an abstract plan in $\mathcal{P}$ that is implementable by the concrete system. Compute a minimal weight abstract plan in $\mathcal{P}$, e.g., using Dijkstra's algorithm. As there are an exponential number of paths in $\mathcal{P}$, it is important to only select the most promising plans. We do this with heuristic weights on transitions in $\mathcal{P}$. The weights $w_{\mathcal{P}}$ represent the likelihood that the corresponding abstract transition in $\mathcal{P}$ corresponds to a concrete transition, i.e., that CstReach returns a feasible control input. For example, these weights could be the expected necessary horizon length for the CstReach problem, or the size of the corresponding constraint sets. Using weights contrasts with most methods in the literature (with notable exceptions [15, 82]), which perform expensive reachability computations ahead of time to ensure that all transitions in the product automaton can be executed by the concrete system.

Given an abstract plan, it must be checked with respect to the system dynamics. Each abstract plan corresponds to a sequence of constrained reachability problems, as detailed in Section 6.3. If the concrete plan if feasible, then a control input is returned. Otherwise, mark the path as infeasible, and update the weights in $\mathcal{P}$. A simple approach is to increase the weight of each edge along the infeasible path by a constant. Additionally, one may compute constrained reachability along a subpath of the infeasible path in an attempt to determine a specific transition that is the cause. There might not be a single transition that invalidates a path, though. Invalidated paths are stored in a set so that they are not repeated. The algorithm then computes another abstract plan until either a feasible control input is found, or every path in $\mathcal{P}$ is checked.

A benefit of this simple approach is that it is easy to parallelize. A central process can search for abstract plans in the product automaton, and then worker processes can check constrained reachability on them. The workers report their results to the central process, which then modifies its search accordingly. There are interesting tradeoffs between searching for accepting plans and checking individual transitions in $\mathcal{P}$, and they are the subject of future work.

Algorithm 12 Automaton-Guided Solution Overview

```
Require: Dynamical system, Büchi aut. $\mathcal{A}$, pathLimit $\in \mathbb{N}$
Ensure: Feasible control input $u$
 1: Compute existential abstraction $\mathcal{T}$ of concrete system
 2: Create product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}$
 3: Assign heuristic weights to transitions of $\mathcal{P}$
 4: checkedPaths = $\varnothing$; paths = 0
 5: while paths < pathLimit do
 6:    paths + = 1
 7:    Compute $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^{\omega}$, the current minimum weight abstract plan not in
       checkedPaths
 8:    Check constrained reachability problem CstReach corresponding to abstract
       plan
 9:    if CstReach returns Yes then
10:       return control input $u$
11:    else
12:       Add plan $\sigma_{\mathcal{P}}$ to checkedPaths
13:       (Optional) Check CstReach of sub-paths $\sigma_{\mathcal{P}}$
14:       Increase weights on transitions along $\sigma_{\mathcal{P}}$
15:    end if
16: end while
```

6.4.2 Tradeoffs

We now discuss some tradeoffs between different levels of granularity in the existential abstraction. Contrary to the counterexample-guided abstraction refinement framework [28], we assume that the number of states in the abstraction is fixed. Instead, we use information from constrained reachability computations to update a ranking over abstract plans.

We will consider varying levels of abstraction in which there is: a single state, a state for each atomic proposition, a state for each polytope in a polytopic partition of the state space, or a state for a given set of discrete points. A natural question is when to use a fine or coarse abstraction. Informally, a coarse abstraction requires solving a small number of large constrained reachability problems, while a fine abstraction requires solving a large number of small constrained reachability problems. Additionally, it may be easier to compose solutions to constrained reachability problems on a fine abstraction than a coarse abstraction, as the initial and final sets are smaller.

Selecting the appropriate level of abstraction is directly related to the difficulty of solving constrained reachability problems of different sizes.

The coarsest abstraction of the system contains only a single state with a self transition. It is labeled with every label that corresponds to a concrete state. Thus, the product automaton is the Büchi automaton. This case is conceptually appealing, as it imposes no discretization of the system, and results in a minimal number of abstract plans that must be checked by constrained reachability. A predicate abstraction is the next coarsest abstraction. This abstraction maps every set of atomic propositions to an abstract state [5]. Thus, the abstraction only depends on the system's labels. A polytopic abstraction assumes that the state space has been partitioned into polytopes. The finest level is when a set of concrete states are abstract states, as in Liu et al. [70] and sampling-based methods [15, 81, 55].

The above discussion can be viewed as a continuum of abstractions that depend on the largest volume of the state space corresponding to an abstract state. Intuitively, it should become easier to concatenate solutions of constrained reachability problems as the size of state space corresponding to each abstract state shrinks. In the limit, when each abstract state maps to a single concrete state, all constrained reachability problems can be concatenated.

This continuum of abstractions leads to a novel way of thinking about abstraction refinement. First consider an abstraction where each abstract state maps to a single concrete state, as in sampling-based motion planning. If a feasible solution cannot be found with this abstraction, one can iteratively expand each abstract state to include a larger set of concrete states until a feasible control input can be found. In the limit, the abstract states would partition the state space. This is in contrast to typical counterexample-guided abstraction refinement approaches [28], since the abstraction becomes coarser instead of finer.

6.5 Complexity

Given an existential abstraction $\mathcal{T}$ with state set S and a Büchi automaton $\mathcal{A}$, the product automaton $\mathcal{P}$ has $O(|S||\mathcal{A}|)$ states. There may be an exponential number of accepting runs (i.e., abstract plans) in $\mathcal{P}$ that must be checked via constrained reachability computations. The complexity of checking a constrained reachability problem depends on the system under consideration.

Proposition 6.1. *Algorithm 12 is complete in the sense that it will return every accepting run in* $\mathcal{P}$.

As there may be an exponential number of accepting runs, completeness is mostly a theoretical curiosity. Our approach depends on having good heuristic weights on the product automaton transitions.

6.6 An Application to Nonlinear Systems in Polygonal Environments

We now discuss an application of our framework to nonlinear systems with atomic propositions that can be represented as the unions of polyhedra. We will solve the constrained reachability problems using mixed-integer linear programming. Mixed-integer linear constraints let one specify that the system is in a certain non-convex region (union of polyhedra) at each time step.

6.6.1 A Mixed-integer Formulation of Constrained Reachability

We assume that each propositional formula ψ is represented by a union of polyhedra. The finite index set I^{ψ} lists the polyhedra where ψ is *True*. The i-th polyhedron is $\{x \in \mathcal{X} \mid H^{\psi_i} x \leq K^{\psi_i}\}$, where $i \in I^{\psi}$. Thus, the set of states where atomic proposition ψ is *True* is given by $[[\psi]] = \{x \in \mathcal{X} \mid H^{\psi_i} x \leq K^{\psi_i} \text{ for some } i \in I^{\psi}\}$. This set is the finite union of polyhedra (finite conjunctions of halfspaces), and it may be non-convex.

For propositional formula ψ and time t, introduce binary variables $z_t^{\psi_i} \in \{0,1\}$ for all $i \in I^{\psi}$. Let M be a vector of sufficiently large constants. The *big-M* formulation

$$\begin{aligned} &H^{\psi_i} x_t \leq K^{\psi_i} + M(1 - z_t^{\psi_i}), \quad \forall i \in I^{\psi} \\ &\sum_{i \in I^{\psi}} z_t^{\psi_i} = 1 \end{aligned}$$

enforces the constraint that $x_t \in [[\psi]]$.

The constrained reachability problem $\textsc{CstReach}(\psi_1, \psi_2, N)$ can then be encoded with the big-M formulation so that $x_t \in [[\psi_1]]$ for $t = 1, \ldots, N-1$ and $x_N \in [[\psi_2]]$. One can specify a fixed horizon length, N, for each set of constrained reachability problems, or can leave the horizon length as a free variable. Additionally, one can decompose the problem by first computing an accepting loop and then computing a prefix that reaches this loop from the initial state, instead of computing both simultaneously. In both cases, the former approach is computationally more efficient, but can miss feasible solutions.

6.6.2 System Dynamics

The mixed-integer constraints in Section 6.6.1 are over a sequence of continuous states; they are independent of the specific system dynamics. Dynamic constraints on the sequence of states can also be enforced by standard transcription methods [14]. However, the resulting optimization problem may then be a mixed-integer *nonlinear* program due to the dynamics. We highlight two useful classes of nonlinear systems where the dynamics can be encoded using mixed-integer *linear* constraints.

Mixed Logical Dynamical Systems

Mixed logical dynamical (MLD) systems have both continuous and discrete-valued states, and allow one to model nonlinearities, logic, and constraints [12]. These systems include constrained linear systems, linear hybrid automata, and piecewise affine systems. An MLD system is of the form

$$\begin{aligned} x_{t+1} &= Ax_t + B_1 u_t + B_2 \delta_t + B_3 z_t \\ \text{subject to} \;\; E_2 \delta_t + E_3 z_t &\leq E_1 u_t + E_4 x_t + E_5, \end{aligned} \tag{6.2}$$

where $t = 0, 1, \ldots$, $x \in \mathcal{X} \subseteq (\mathbb{R}^{n_c} \times \{0,1\}^{n_l})$ are the continuous and binary states, $u \in \mathcal{U} \subseteq (\mathbb{R}^{m_c} \times \{0,1\}^{m_l})$ are the inputs, and $\delta \in \{0,1\}^{r_l}$ and $z \in \mathbb{R}^{r_l}$ are auxiliary binary and continuous variables, respectively. The system matrices A, B_1, B_2, B_3, E_1, E_2, E_3, E_4, and E_5 are of appropriate dimension. We assume that the system is deterministic and well-posed (see Definition 1 in Bemporad and Morari [12]). Thus, for an initial condition x_0 and a control input sequence $\mathbf{u} = u_0 u_1 u_2 \ldots$, there is a unique run $\mathbf{x} = x_0 x_1 x_2 \ldots$ that satisfies the constraints in (6.2).

Differentially Flat Systems

A system is *differentially flat* if there exists a set of outputs such that all states and control inputs can be determined from these outputs without integration. If a system has states $x \in \mathbb{R}^n$ and control inputs $u \in \mathbb{R}^m$, then it is flat if one can find outputs $y \in \mathbb{R}^m$ of the form $y = y(x, u, \dot{u}, \ldots, u^{(p)})$ such that $x = x(y, \dot{y}, \ldots, y^{(q)})$ and $u = u(y, \dot{y}, \ldots, y^{(q)})$. Thus, one can plan trajectories in output space, and then map these to control inputs [68].

Differentially flat systems may be encoded using mixed integer linear constraints in certain cases, e.g., the flat output is constrained by mixed integer linear constraints. This condition holds for relevant classes of robotic systems, including quadrotors and car-like robots. However, control input constraints are typically non-convex in the flat output. Common approaches to satisfy control constraints are to plan a sufficiently smooth trajectory or slow down along a trajectory [76].

6.6.3 Computing Sets of Feasible Initial States

Our framework can be extended to compute a *set* of initial states from which there exists a satisfying control input. This is possible (when all labels are unions of polytopes) by performing a projection on a lifted polytope. The key insight is that a satisfying system trajectory has a corresponding sequence of polytopes. One can construct a lifted polytope in the initial state x_0 and control input u, and then project on x_0 to compute a set of feasible initial conditions. We defer to Section V-B in Wongpiromsarn et al. [108] for details on this construction.

6.7 Examples

We demonstrate our techniques on a variety of motion planning problems. The first example is a chain of integrators parameterized by dimension. Our second example is a quadrotor model that was previously considered in Webb and van den Berg [97]. Our final example is a nonlinear car-like vehicle with drift. All computations were done on a laptop with a 2.4 GHz dual-core processor and 4 GB of memory using CPLEX [1] with YALMIP [71].

The environment and task are motivated by a delivery scenario. All properties should be understood to be with respect to regions in the plane (see Figure 6.2). Let $D1$, $D2$, $D3$, and $D4$ be regions where supplies must be delivered. The robot must stay in the safe region S (in white). Formally, we consider task specifications of the form $\text{specF}(n) = \bigwedge_{i=1}^{n} \Diamond D_i \wedge \Box S$ and $\text{specGF}(n) = \bigwedge_{i=1}^{n} \Box \Diamond D_i \wedge \Box S$ for single and repeated deliveries, respectively.

In the remainder of this section, we consider this temporal logic motion planning problem for different system models. All continuous-time models are discretized using a first-order hold and time-step of 0.5 seconds. We use a fixed horizon $N = 20$ for each constrained reachability problem. These CSTREACH problems are concatenated between two to six times for an abstract path, resulting in between 40 to 120 time steps (see Figure 6.2). At each time step, approximately 8 binary variables are used to

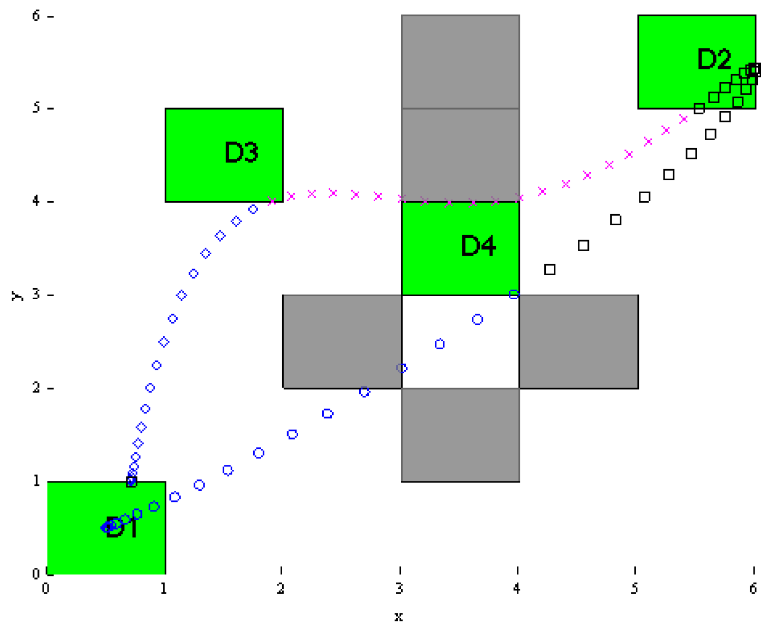


Figure 6.2: Illustration of the environment. The goals are labeled $D1$, $D2$, $D3$, and $D4$. Dark regions are obstacles. A representative trajectory for the quadrotor is shown with the five concatenated CstReach problems, i.e., $([\![S]\!], [\![D4 \wedge S]\!], 20)$, $([\![S]\!], [\![D2 \wedge S]\!], 20)$, $([\![S]\!], [\![D3 \wedge S]\!], 20)$, $([\![S]\!], [\![D1 \wedge S]\!], 20)$, and $([\![S]\!], [\![S]\!], 20)$ in varied colors and shapes.

represent the current label. A computation limit of 60 seconds is enforced for checking reachability of each abstract path, and up to three abstract paths are checked for each trial. Finally, all results are averaged over 20 randomly generated environments.

We use the coarsest possible abstraction of the dynamical system, a single abstract state as described in Section 6.4.2. This abstraction is not proposition-preserving, and effectively means that we directly use the Büchi automaton to guide the constrained reachability problems that we solve.

6.7.1 Chain of Integrators

The first system is a chain of orthogonal integrators in the x and y directions. The k-th derivative of the x and y positions are controlled, i.e., $x^{(k)} = u_x$ and $y^{(k)} = u_y$, subject to the constraints $|u_x| \leq 0.5$ and $|u_y| \leq 0.5$. The state constraints are $|x^{(i)}| \leq 1$ and $|y^{(i)}| \leq 1$ for $i = 1, \ldots, k-1$. Results are given in Figures 6.3, 6.4, and 6.5 under "chain-2," "chain-6," and "chain-10," where "chain-k" is a $2k$-dimensional system

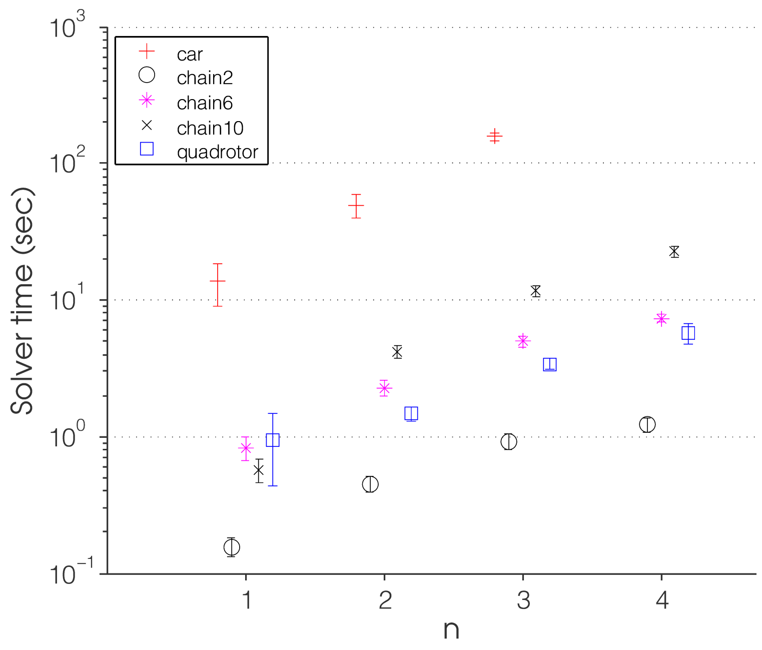


Figure 6.3: Solver time (mean ± standard error) to compute a control input for various system models for specF(n).

where the k-th derivative in both the x and y positions is controlled.

6.7.2 Quadrotor

We now consider the quadrotor model used in Webb and van den Berg [97] for point-to-point motion planning, to which we refer the reader for a complete description of the model. The state $x = (p, v, r, w)$ is 10-dimensional, consisting of position $p \in \mathbb{R}^3$, velocity $v \in \mathbb{R}^3$, orientation $r \in \mathbb{R}^2$, and angular velocity $w \in \mathbb{R}^2$. This model is the linearization of a nonlinear model about hover with the yaw constrained to be zero. The control input $u \in \mathbb{R}^3$ is the total, roll, and pitch thrust. Results are given in Figures 6.3, 6.4, and 6.5 under "quadrotor," and a sample trajectory is shown in Figure 6.2.

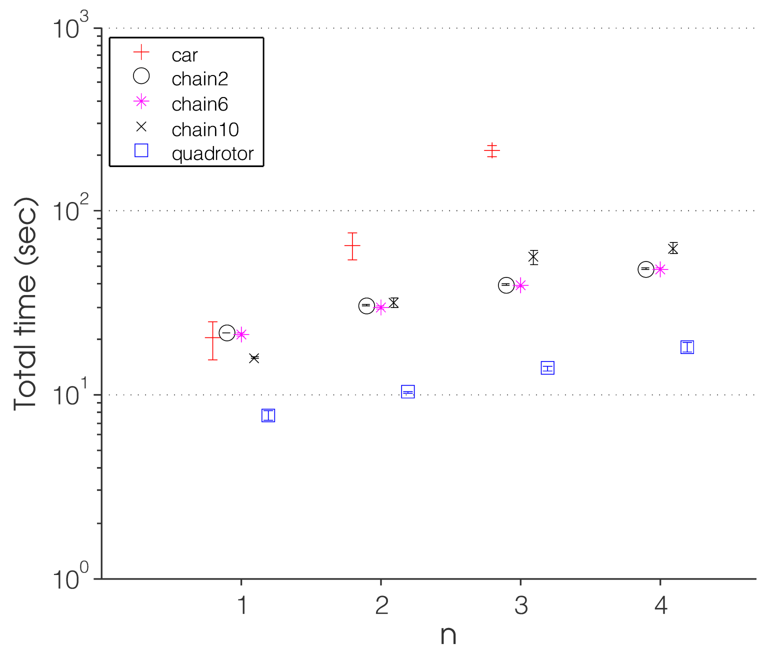


Figure 6.4: Total time (mean ± standard error) to compute a control input for various system models for specF(n).

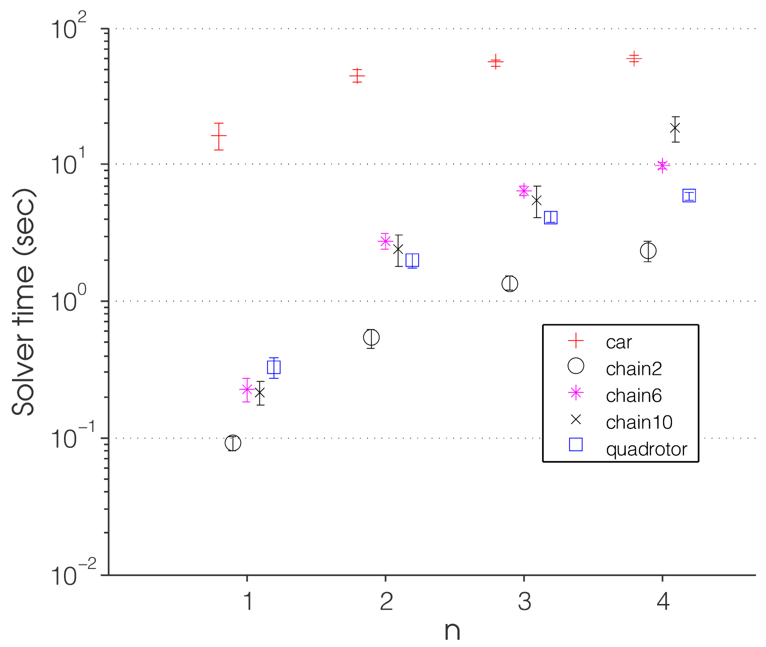


Figure 6.5: Solver time (mean ± standard error) to compute a control input for various system models for specGF(n). Total time is not shown.

6.7.3 Nonlinear Car

Consider a nonlinear car-like vehicle with state $x = (p_x, p_y, \theta)$ and dynamics $\dot{x} = (v\cos(\theta), v\sin(\theta), u)$. The variables p_x, p_y are position (m), and θ is orientation (rad). The vehicle's speed v is fixed at 0.5 (m/s), and its control input is constrained as $|u| \leq 2.5$. We form a hybrid MLD model by linearizing the system about different orientations $\hat{\theta}_i$ for $i = 1, 2, 3$. The dynamics are governed by the closest linearization to the current θ. Results are given in Figures 6.3, 6.4, and 6.5 under "car."

6.7.4 Discussion

We are able to generate satisfying trajectories for 20-dimensional constrained linear systems, which is not possible with finite abstraction approaches such as [61] or [108] or the specification-guided approach of [45]. For the 10-dimensional quadrotor model, feasible solutions are returned in a matter of seconds. The nonlinear car model required additional binary variables to describe the hybrid modes, which led to larger mixed-integer optimization problems, and thus its poor relative performance. Our results appear particularly promising for situations where the environment is dynamically changing, and a finite abstraction must be repeatedly computed.

Typically, few abstract paths needed to be checked to determine a feasible solution. This is because the ordering between visits to the different labeled regions did not usually affect the problem's feasibility. The intuition is that the robot can (almost) move to any state in the safe region S from any other state.

Finally, the total time (e.g., Figure 6.3) is typically an order of magnitude more than the solver time (e.g., Figure 6.4). The main component of the total time is the translation of the YALMIP model to the input for the CPLEX optimizer, which could be avoided by interfacing directly with CPLEX. Thus, we believe that the solver time is more indicative of performance than total time.

6.8 Conclusions

In this chapter, we developed a framework for computing controllers for discrete-time nonlinear systems with temporal logic specifications. Our approach uses a coarse approximation of the system along with the logical specification to guide the computation of constrained reachability problems as needed for the task. Notably, we do not require any discretization of the original system, and our method lends itself to a parallel implementation.

There are multiple directions for future work, including investigating tradeoffs between checking an entire sequence of constrained reachability problems or only a subsequence, choosing the appropriate abstraction level given a system and a specification, and applying PDE-based methods [92] for the computation of the constrained reachability problems

Chapter 7

Optimization-Based Trajectory Generation with Linear Temporal Logic Specifications

In this chapter, we present a mathematical programming-based method for optimal control of discrete-time dynamical systems subject to temporal logic task specifications. We use linear temporal logic (LTL) to specify a wide range of properties and tasks, such as safety, progress, response, surveillance, repeated assembly, and environmental monitoring. Our method directly encodes an LTL formula as mixed-integer linear constraints on the continuous system variables, avoiding the computationally expensive processes of creating a finite abstraction of the system and a Büchi automaton for the specification. In numerical experiments, we solve temporal logic motion planning tasks for high-dimensional (10+ continuous state) dynamical systems. This chapter is based on results from [99, 103].

7.1 Introduction

Standard methods for motion planning with LTL task specifications first create a finite abstraction of the original dynamical system (see [4, 10, 61, 108]). This abstraction can informally be viewed as a labeled graph that represents possible behaviors of the system. Given a finite abstraction of a dynamical system and an LTL specification, controllers can be automatically constructed using an automata-based approach [9, 15, 39, 61]. The main drawbacks of this approach are: 1) it is expensive to compute a finite abstraction, 2) the size of the automaton may be exponential in the length of the specification, and 3) optimality may only be with respect to the discrete abstraction's level of refinement.

Instead of the automata-based approach, we directly encode an LTL formula as mixed-integer linear constraints on the original dynamical system. We enforce that an infinite sequence of system states satisfies the specification by using a finite number of constraints on a trajectory parameterization of bounded length. This is possible by enforcing that the system's trajectory is eventually periodic, i.e., it contains a loop. The loop assumption is motivated by the use of "lassos" in LTL model checking of finite, discrete systems [9]. This direct encoding of the LTL formula avoids the potentially expensive computations of a finite abstraction of the system and a Büchi automaton for the specification.

Our approach applies to any deterministic system model that is amenable to finite-dimensional optimization, as the temporal logic constraints are independent of any particular system dynamics or cost functions. Of particular interest are mixed logical dynamical (MLD) systems [12] and certain differentially flat systems [68], whose dynamics can be encoded with mixed-integer linear constraints. MLD systems include constrained linear systems, linear hybrid automata, and piecewise affine systems. Differentially flat systems include quadrotors [75] and car-like robots (see Section 6.6.2).

Our work extends the bounded model checking paradigm for finite, discrete systems [16] to continuous dynamical systems. In bounded model checking, one searches for counterexamples (i.e., trajectories) of a fixed length by transforming the problem into a Boolean satisfiability (SAT) problem. This approach was extended to hybrid systems in Giorgetti et al. [43] by first computing a finite abstraction of the system, and then using a SAT solver for the resulting discrete problem. Hybrid systems are also considered in [6, 40], where SAT solvers are extended to reason about linear inequalities. In contrast, we consider a larger class of dynamics and use mixed-integer linear programming techniques.

It is well-known that mixed-integer linear programming can be used for reasoning about propositional logic [18, 50], generating state-constrained trajectories [35, 85, 96], and modeling vehicle routing problems [54, 94]. Mixed-integer linear programming has been used for trajectory generation for continuous systems with finite-horizon LTL specifications in Karaman et al. [55] and Kwon and Agha [64]. However, finite-horizon properties are too restrictive to model a large class of interesting robotics problems, including surveillance, repeated assembly, periodic motion, and environmental monitoring. We generalize these results by encoding all LTL properties using mixed-integer linear constraints.

The main contribution of this chapter is a novel method for encoding LTL specifications as mixed-integer linear constraints on a dynamical system. This encoding generalizes previous Boolean satisfiability encodings of LTL formulas for finite, discrete systems. Our mixed-integer encoding works for any LTL formula, as opposed to previous approaches that only consider finite-horizon properties or fragments. Our encoding is also efficient; it requires a number of variables linear in the length of the trajectory, instead of quadratic as in previous approaches [55]. We demonstrate how our mixed-integer programming formulation can be used with off-the-shelf optimization solvers (e.g. CPLEX [1]) to compute both feasible and optimal controllers for high-dimensional (more than 10 continuous state) dynamical systems with temporal logic specifications.

7.2 Problem Statement

As in Section 6.1.1, we consider discrete-time nonlinear systems of the form

$$x_{t+1} = f(x_t, u_t), \tag{7.1}$$

where $t = 0, 1, \ldots$ are the time indices, $x \in \mathcal{X} \subseteq (\mathbb{R}^{n_c} \times \{0,1\}^{n_l})$ are the continuous and binary states, $u \in \mathcal{U} \subseteq (\mathbb{R}^{m_c} \times \{0,1\}^{m_l})$ are the control inputs, and $x_0 \in \mathcal{X}$ is the initial state.

In this chapter, we will assume that an LTL formula is given in *positive normal* *form.* An LTL formula in *positive normal form* (negation normal form) is defined by the following grammar:

$$\varphi ::= p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \, \mathcal{U} \, \varphi_2 \mid \varphi_1 \, \mathcal{R} \, \varphi_2,$$

where $p \in AP$ is an atomic proposition, and $\mathcal{R}$ (release) is a temporal operator with semantics defined as follows

$$\sigma_i \vDash \varphi_1 \, \mathcal{R} \, \varphi_2 \text{ if and only if } \forall j \geq i : \sigma_j \vDash \varphi_2 \text{ or } \sigma_n \vDash \varphi_1 \exists i \leq n < j.$$

Every LTL formula can be rewritten in positive normal form, where all negations only appear in front of atomic propositions [9]. The following rules transform a given LTL formula into an equivalent LTL formula in positive normal form: $\neg \mathit{True} = \mathit{False}$, $\neg\neg\varphi = \varphi$, $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$, $\neg \bigcirc \varphi = \bigcirc \neg\varphi$, and $\neg(\varphi_1 \, \mathcal{U} \, \varphi_2) = \neg\varphi_1 \, \mathcal{R} \, \neg\varphi_2$. An LTL formula φ of size $|\varphi|$ can always be rewritten as a formula φ' in positive normal form of size $|\varphi'| = O(|\varphi|)$ [9].

We now formally state both a feasibility and an optimization problem, and give an overview of our solution approach. Let φ be an LTL formula defined over AP.

Problem 7.1. Given a system of the form (7.1) and an LTL formula φ, compute a control input sequence $\mathbf{u}$ such that $\mathbf{x}(x_0, \mathbf{u}) \vDash \varphi$.

To distinguish among all trajectories that satisfy Problem 7.1, we introduce a generic cost function $J(\mathbf{x}, \mathbf{u})$ that maps trajectories and control inputs to $\mathbb{R} \cup \infty$.

Problem 7.2. Given a system of the form (7.1) and an LTL formula φ, compute a control input sequence $\mathbf{u}$ such that $\mathbf{x}(x_0, \mathbf{u}) \vDash \varphi$ and $J(\mathbf{x}(x_0, \mathbf{u}), \mathbf{u})$ is minimized.

We now briefly overview our solution approach. We represent the system trajectory as a finite sequence of states. Infinite executions of the system are captured by enforcing that a loop occurs in this sequence, making the trajectory eventually periodic. We then encode an LTL formula as mixed-integer linear constraints on this finite trajectory parameterization in Section 7.3.2. Additionally, both dynamic

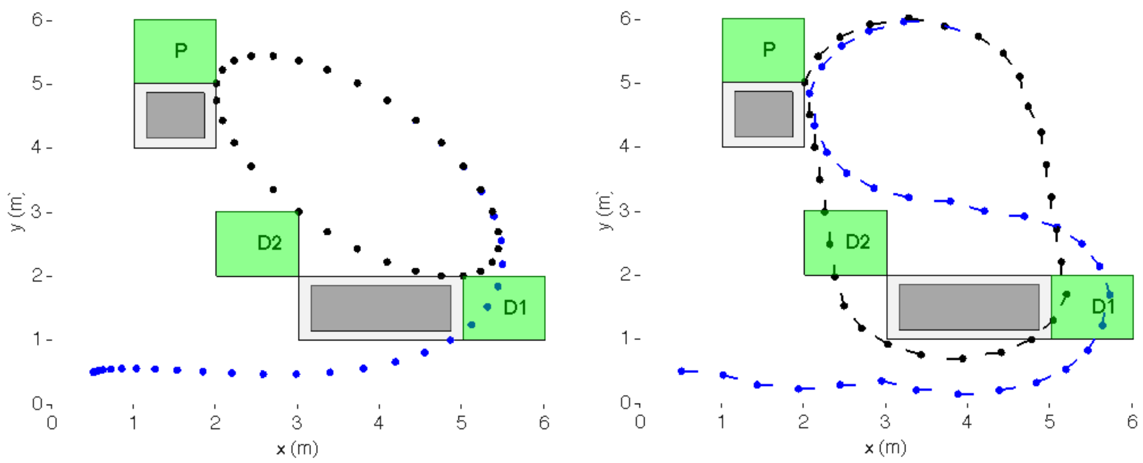


Figure 7.1: Illustration of a problem instance. The task is to repeatedly visit regions P, $D1$, and $D2$, where dark regions are obstacles that must be avoided. Representative trajectories for a quadrotor (left) and nonlinear car (right) are shown with the prefix (blue) and suffix (black).

constraints (see Section 7.3.2) and a cost function can also be included as part of the mixed-integer optimization problem. For mixed logical dynamical (or piecewise affine) and certain differentially flat systems, Problems 7.1 and 7.2 (with linear costs) can thus be solved using a mixed-integer linear program (MILP) solver. While even checking feasibility of a MILP is NP-hard, modern solvers using branch and bound methods routinely solve large problems [1]. We give results on systems with more than 10 continuous states in Section 7.4.

Remark 7.1. We only consider open-loop trajectory generation, which is already a challenging problem due to the LTL specifications and continuous dynamics. Incorporating disturbances during trajectory generation is the subject of future work.

7.3 Solution

This section builds on the work in bounded model checking of finite, discrete systems [16], and extend it to dynamical systems using mixed-integer programming. Our presentation and notation follows that of [17].

We will search for a trajectory of length k that satisfies φ. Although a satisfying trajectory for an LTL formula describes an *infinite* sequence of states, an infinite sequence can be captured by a *finite* trajectory that has a loop. Note that this approach is conservative for systems of the form (7.1) due to both the bounded trajectory length k and the assumption that the trajectory is eventually periodic. This is in contrast to finite systems, where the assumption that the trajectory is eventually periodic is without loss of generality.

Definition 7.1. A run $\mathbf{x}$ is a (k,l)-*loop* if $\mathbf{x} = (x_0 x_1 \dots x_{l-1})(x_l \dots x_k)^\omega$ such that $0 < l \leq k$ and $x_{l-1} = x_k$, where ω denotes infinite repetition.

Definition 7.2. Given a run $\mathbf{x}$ and a bound $k \in \mathbb{N}$, $\mathbf{x} \vDash_k \varphi$ iff $\mathbf{x}$ is a (k,l)-loop for some $0 < l \leq k$ and $\mathbf{x}_0 \vDash \varphi$.

For a bounded trajectory length k, Problems 7.1 and 7.2, with $\mathbf{x} \vDash \varphi$ replaced by $\mathbf{x} \vDash_k \varphi$, can be represented as a finite-dimensional mixed-integer program. We will build a set $[\![M, \varphi, k]\!]$ of mixed-integer constraints that is satisfiable if and only if a trajectory of length k exists for system M that satisfies φ. The satisfiability of these constraints can then be checked with a mixed-integer programming solver. In the following sections, we will describe the construction of the loop constraints, the LTL constraints, and the system constraints. However, we first detail how to relate the continuous state to the set of valid atomic propositions.

Remark 7.2. The results in this chapter can be extended to the bounded semantics for LTL, i.e., the no-loop case, as detailed in Biere et al. [17].

7.3.1 Representing the Labels

We now relate the state of a system to the set of atomic propositions that are *True* at each time instance. We assume that each propositional formula ψ is described by the Boolean combination of a finite number of halfspaces. Our approach here is standard, see e.g., [12, 55].

Halfspace Representation

We now give a necessary and sufficient condition on a state x_t being in a halfspace $H = \{x \in \mathcal{X} \mid h^T x \leq k\}$ at time t. This constraint can be extended to unions of polyhedra using conjunctions and disjunction, as detailed in Section 7.3.1.

First, introduce binary variables $z_t \in \{0,1\}$ for time indices $t = 0,\dots,k$. Then, enforce that $z_t = 1$ if and only if $h^T x_t \leq k$ with the following constraints

$$\begin{aligned} h^T x_t &\leq k + M_t(1 - z_t), \\ h^T x_t &> k - M_t z_t + \epsilon, \end{aligned}$$

where M_t are large positive numbers, and ϵ is a small positive number. Note that $z_t = 1$ if and only if the state is in the halfspace H at time t (with precision ϵ on the boundary).

Boolean Operations

In this section, we encode $\neg$ (not), $\wedge$ (and), and $\vee$ (or) of propositions using mixed-integer linear constraints. We assume that each proposition p has a corresponding variable (binary or continuous) P_t^p which equals 1 if p is *True* at time t, and equals 0 otherwise. We will use new continuous variables $P_t^\psi \in [0,1]$ to represent the resulting propositions. In each case, $P_t^\psi = 1$ if ψ holds at time t, and $P_t^\psi = 0$ otherwise.

The negation of proposition p, i.e., $\psi = \neg p$, is modeled for $t = 0,\dots,k$ as

$$P_t^\psi = 1 - P_t^p.$$

The conjunction of propositions p_i for $i = 1,\dots,m$, i.e., $\psi = \wedge_{i=1}^m p_i$, is modeled for $t = 0,\dots,k$ as

$$\begin{aligned} P_t^\psi &\leq P_t^{p_i}, \quad i = 1,\dots,m, \\ P_t^\psi &\geq 1 - m + \sum_{i=1}^m P_t^{p_i} \end{aligned}$$

The disjunction of propositions p_i for $i = 1, \ldots, m$, i.e., $\psi = \vee_{i=1}^{m} p_i$, is modeled for $t = 0, \ldots, k$ as

$$
\begin{aligned}
P_t^{\psi} &\geq P_t^{p_i}, \quad i = 1, \ldots, m, \\
P_t^{\psi} &\leq \sum_{i=1}^{m} P_t^{p_i}.
\end{aligned}
$$

7.3.2 A Mixed-integer Encoding

We now encode Problem 7.1 as a set $[[M, \varphi, k]]$ of mixed-integer constraints, which includes loop constraints, LTL constraints, and system constraints. Note that while the loop and LTL constraints are always mixed-integer *linear* constraints, the system constraints will depend on the dynamic model used. Problem 7.2 uses the same constraint set $[[M, \varphi, k]]$, with the addition of a cost function defined over the (k, l)-loop.

Loop Constraints

The loop constraints are used to determine where a loop is formed in the system trajectory. We introduce k binary variables $l_1, \ldots, l_k$, which determine where the trajectory loops. These are constrained so that only one loop selector variable is allowed to be *True*, and if l_j is *True*, then $x_{j-1} = x_k$. These constraints are enforced by $\sum_{i=1}^{k} l_i = 1$ and

$$
\begin{aligned}
x_k &\leq x_{j-1} + M_j(1 - l_j), \quad j = 1, \ldots, k, \\
x_k &\geq x_{j-1} - M_j(1 - l_j), \quad j = 1, \ldots, k,
\end{aligned}
$$

where M_j are sufficiently large positive numbers.

LTL Constraints

Given a formula φ, we denote the satisfaction of φ at position i by $[[\varphi]]_i$. The variable $[[\varphi]]_i \in \{0, 1\}$ corresponds to an appropriate set of mixed-integer linear constraints so that $[\![\varphi]\!]_i = 1$ if and only if φ holds at position i. We recursively generate the mixed-integer linear constraints corresponding to $[\![\varphi]\!]_0$ to determine whether or not a formula φ holds in the initial state, i.e., if $[\![\varphi]\!]_0 = 1$. In this section, we use the encoding of the $\mathcal{U}$ and $\mathcal{R}$ temporal operators from [17] for Boolean satisfiability. Our contribution here is linking this satisfiability encoding to the continuous system state through the mixed-integer linear constraints described in Section 7.3.1. This encoding first computes an under-approximation for $\mathcal{U}$ and an over-approximation for $\mathcal{R}$ with the auxiliary encoding $\langle\!\langle\cdot\rangle\!\rangle$. The under-approximation of $\varphi_1\,\mathcal{U}\,\varphi_2$ assumes that φ_2 does not hold in the successor of state x_k. The over-approximation of $\varphi_1\,\mathcal{R}\,\varphi_2$ assumes that φ_2 holds in the successor of state x_k. These approximations are then refined to exact values by $[\![\cdot]\!]$. The encoding is recursively defined over an LTL formula, where there is a case for each logical and temporal connective.

The reader might wonder why an auxiliary encoding is necessary. A seemingly straightforward approach for $\mathcal{U}$ is to use the textbook identity (see [9]) $[\![\psi_1\,\mathcal{U}\,\psi_2]\!]_i = [\![\psi_2]\!]_i \vee ([\![\psi_1]\!]_i \wedge [\![\psi_1\,\mathcal{U}\,\psi_2]\!]_{i+1})$ for $i = 0,\ldots,k$, where index $k+1$ is replaced by an appropriate index to form a loop. However, this approach can lead to circular reasoning. Consider a trajectory consisting of a single state with a self loop, and the LTL formula $\mathit{True}\,\mathcal{U}\,\psi$, i.e., $\diamond\psi$ (eventually). The corresponding encoding is $[\![\mathit{True}\,\mathcal{U}\,\psi]\!]_0 = [\![\psi]\!]_0 \vee [\![\mathit{True}\,\mathcal{U}\,\psi]\!]_0$. This can be trivially satisfied by setting $[\![\mathit{True}\,\mathcal{U}\,\psi]\!]_0$ equal to 1, regardless of whether or not ψ is visited. The auxiliary encoding prevents this circular reasoning, as detailed in Biere et al. [17].

We first define the encoding of propositional formulas as

$$\begin{aligned}
[\![\psi]\!]_i &= P_i^{\psi},\\
[\![\neg\psi]\!]_i &= P_i^{\neg\psi},\\
[\![\psi_1 \wedge \psi_2]\!]_i &= [\![\psi_1]\!]_i \wedge [\![\psi_2]\!]_i,\\
[\![\psi_1 \vee \psi_2]\!]_i &= [\![\psi_1]\!]_i \vee [\![\psi_2]\!]_i,
\end{aligned}$$

for $i = 0,\ldots,k$, where these operations were defined in Section 7.3.1 using mixed-integer linear constraints.

Next, we define the auxiliary encodings of $\mathcal{U}$ and $\mathcal{R}$. The until (release) formulas at k use the auxiliary encoding $\langle\!\langle \psi_1 \,\mathcal{U}\, \psi_2 \rangle\!\rangle_j$ ($\langle\!\langle \psi_1 \,\mathcal{R}\, \psi_2 \rangle\!\rangle_j$) at the index j where the loop is formed, i.e., where l_j holds. The auxiliary encoding of the temporal operators is

$$\begin{aligned}
\langle\!\langle \psi_1 \,\mathcal{U}\, \psi_2 \rangle\!\rangle_i &= [\![\psi_2]\!]_i \vee ([\![\psi_1]\!]_i \wedge \langle\!\langle \psi_1 \,\mathcal{U}\, \psi_2 \rangle\!\rangle_{i+1}),\\
\langle\!\langle \psi_1 \,\mathcal{R}\, \psi_2 \rangle\!\rangle_i &= [\![\psi_2]\!]_i \wedge ([\![\psi_1]\!]_i \vee \langle\!\langle \psi_1 \,\mathcal{R}\, \psi_2 \rangle\!\rangle_{i+1}),
\end{aligned}$$

for $i = 1, \ldots, k-1$, and is

$$\begin{aligned}
\langle\!\langle \psi_1 \,\mathcal{U}\, \psi_2 \rangle\!\rangle_i &= [\![\psi_2]\!]_i,\\
\langle\!\langle \psi_1 \,\mathcal{R}\, \psi_2 \rangle\!\rangle_i &= [\![\psi_2]\!]_i,
\end{aligned}$$

for $i = k$.

Finally, we define the encoding of the temporal operators as

$$\begin{aligned}
[\![\bigcirc\psi]\!]_i &= [\![\psi]\!]_{i+1},\\
[\![\psi_1 \,\mathcal{U}\, \psi_2]\!]_i &= [\![\psi_2]\!]_i \vee ([\![\psi_1]\!]_i \wedge [\![\psi_1 \,\mathcal{U}\, \psi_2]\!]_{i+1}),\\
[\![\psi_1 \,\mathcal{R}\, \psi_2]\!]_i &= [\![\psi_2]\!]_i \wedge ([\![\psi_1]\!]_i \vee [\![\psi_1 \,\mathcal{R}\, \psi_2]\!]_{i+1}),
\end{aligned}$$

for $i = 0, \ldots, k-1$, and as

$$\begin{aligned}
[\![\bigcirc\psi]\!]_i &= \bigvee_{j=1}^{k} (l_j \wedge [\![\psi]\!]_j),\\
[\![\psi_1 \,\mathcal{U}\, \psi_2]\!]_i &= [\![\psi_2]\!]_i \vee ([\![\psi_1]\!]_i \wedge (\bigvee_{j=1}^{k} (l_j \wedge \langle\!\langle \psi_1 \,\mathcal{U}\, \psi_2 \rangle\!\rangle_j))),\\
[\![\psi_1 \,\mathcal{R}\, \psi_2]\!]_i &= [\![\psi_2]\!]_i \wedge ([\![\psi_1]\!]_i \vee (\bigvee_{j=1}^{k} (l_j \wedge \langle\!\langle \psi_1 \,\mathcal{R}\, \psi_2 \rangle\!\rangle_j))),
\end{aligned}$$

for $i = k$.

We also explicitly give the encodings for safety, persistence, and liveness formulas. These formulas frequently appear in specifications, and can be encoded more efficiently than the general approach just described.

A safety formula $\Box\psi$ can be encoded as

$$\begin{aligned}
[\![\Box\psi]\!]_i &= [\![\psi]\!]_i \;\wedge\; [\![\Box\psi]\!]_{i+1}, \quad i = 0,\ldots,k-1\\
[\![\Box\psi]\!]_k &= [\![\psi]\!]_k.
\end{aligned}$$

An auxiliary encoding is not necessary here, as noted in Biere et al. [17].

Due to the loop structure of the trajectory, both persistence and liveness properties either hold at all indices or no indices. We encode a persistence $\Diamond\,\Box\,\psi$ and liveness $\Box\,\Diamond\,\psi$ formulas as

$$\begin{aligned}
[\![\Diamond\,\Box\,\psi]\!] &= \bigvee_{i=1}^{k}\left(l_i \;\wedge\; \bigwedge_{j=i}^{k}[\![\psi]\!]_j\right),\\
[\![\Box\,\Diamond\,\psi]\!] &= \bigvee_{i=1}^{k}\left(l_i \;\wedge\; \bigvee_{j=i}^{k}[\![\psi]\!]_j\right),
\end{aligned}$$

for $i = 0,\ldots,k$. Although the encodings for persistence and liveness appear to require a number of variables that are quadratic in k, one can share subformulas to make this linear in k [16].

System Constraints

The system constraints encode valid trajectories of length k for a system of the form (7.1), i.e., they hold if and only if trajectory $\mathbf{x}(x_0,\mathbf{u})$ satisfies the constraints in equation (7.1) for $t = 0,1,\ldots,k$.

System constraints, e.g., the dynamics in equation (7.1), can be encoded on the sequence of states using standard transcription methods [14]. However, the resulting optimization problem may then be a mixed-integer *nonlinear* program due to the dynamics. A useful class of nonlinear systems where the dynamics can be encoded using mixed-integer *linear* constraints are mixed logical dynamical (MLD) systems and certain differentially flat systems (see Section 6.6.2).

The intersection of the LTL constraints, the system constraints, and the loop constraints gives the full mixed-integer linear encoding of the bounded model checking problem, i.e., $[[M, \varphi, k]]$. Problem 7.1 is solved by checking the feasibility of this set using a MILP solver (assuming the dynamics are mixed logical dynamical). Similarly, Problem 7.2 is solved by optimizing over this set using a MILP solver (assuming the cost function is linear). More general dynamics and cost functions can be included by using an appropriate mixed-integer solver, potentially at the expense of completeness.

7.3.3 Complexity

The main source of complexity of this approach comes from the number of binary variables needed to relate the satisfaction of an atomic proposition to the continuous state of the system. A binary variable is introduced at each time index for each halfspace. If n_h halfspaces are used to represent the atomic propositions used in the LTL formula at each time index, then $O(k \cdot n_h)$ binary variables are introduced, where k is the number of time indices.

Continuous variables are used to represent the propositions introduced during the encoding of the LTL formula. The number of continuous variables used is $O(k \cdot |\varphi|)$, where k is the number of time indices, and $|\varphi|$ is the length of the formula. This linear dependence on k improves on the quadratic dependence on k in [55]. Finally, although the loop constraints introduce k additional binary variables, they are constrained such that only one is active. In summary, our mixed-integer linear encoding of an LTL formula φ requires the use of $O(k \cdot n_h)$ binary variables and $O(k \cdot |\varphi|)$ continuous variables. Note that the complexity of solving a mixed-integer linear program is worst-case exponential in the number of binary variables [1].

We do not give bounds on the number of constraints, as they depend heavily on the details of a given specification, and are not the major factor in the complexity of solving a mixed-integer linear program.

Remark 7.3. Our solution approaches for Problems 7.1 and 7.2 are only complete with respect to a given bound of k. If a solution is not found for a given value of k, there may exist a solution for a larger value. Future work will identify cases when an upper bound on k can be computed a priori, similar to the notion of a completeness

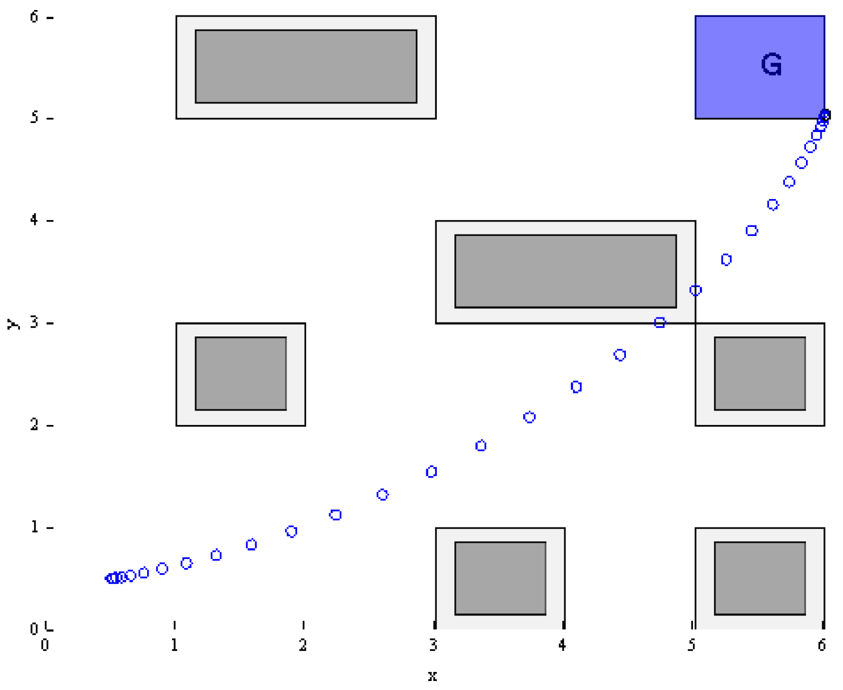


Figure 7.2: Illustration of the environment for the reach-avoid scenario. The goal is labeled G, and dark regions are obstacles. The light regions around the obstacles give a safety margin so that the continuous evolution of the system does not intersect an obstacle, which would otherwise be possible, since constraints are enforced at discrete time steps. A representative trajectory for the “chain-6” model is shown, where we additionally minimized an additive cost function with cost $c(x_t, u_t) = |u_t|$ accrued at each time index.

threshold [17].

7.4 Examples

We consider two LTL motion planning problems for different system models. We demonstrate our techniques on the models from Section 6.7: a chain of integrators model, a quadrotor model, and nonlinear car-like model with v fixed at 1 (m/s). All computations are done on a laptop with a 2.4 GHz dual-core processor and 4 GB of memory using CPLEX [1] through Yalmip [71]. All continuous-time models are discretized with a 0.35 second sample time.

Our first example is a simple reach-avoid motion planning scenario (see Figure 7.2), which we use to directly compare our encoding of the until operator ($\mathcal{U}$) with that

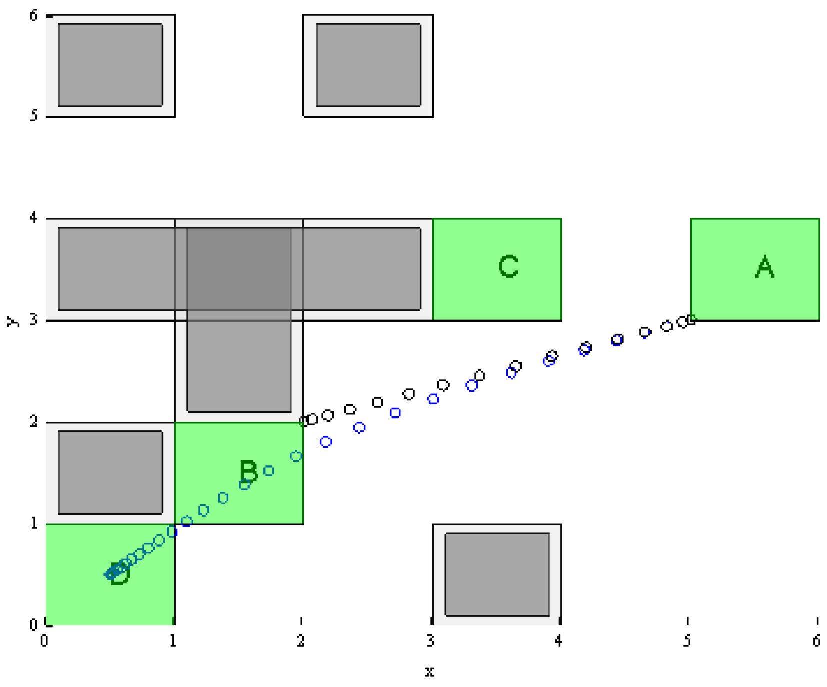


Figure 7.3: Illustration of the environment for the surveillance scenario. The goals are labeled A, B, C, and D. Dark regions are obstacles. A representative trajectory for the quadrotor model is shown, where we additionally minimized an additive cost function with cost $c(x_t, u_t) = |u_t|$ accrued at each time index. The system starts in region D, and repeatedly visits regions A and C.

given in Karaman et al. [55]. The task here is to remain in the safe region S until the goal region G is reached. The corresponding LTL formula is $\varphi = S\,\mathcal{U}\,G$. This formula is a key component for building more complex LTL formulas, and thus performance of this encoding is important. We show that our formulation scales better with respect to the trajectory length k. This is anticipated, as our encoding of $\mathcal{U}$ requires a number of variables linear in k, while the encoding in Karaman et al. [55] requires a number of variables quadratic in k. For this example, all results are averaged over ten randomly generated environments where 75 percent of the area is safe (labeled S), i.e., the remaining regions are obstacles.

The second example is motivated by a surveillance mission. Let A, B, C, and D describe regions of interest in a planar planning space that must be repeatedly visited (see Figure 7.3). The robot must remain in the safe region S (in white) and either visit regions A and B repeatedly, or visit regions C and D repeatedly. Formally, the task specification is $\varphi = \square S \wedge ((\square \diamond A \wedge \square \diamond B) \vee (\square \diamond C \wedge \square \diamond D))$. This formula is recursively parsed and encoded as mixed-integer linear constraints, as described in Section 7.3.2. For this example, all results are averaged over five randomly-generated environments where 85 percent of the area is safe, i.e., the remaining regions are obstacles. The length of the trajectory is $k = 25$, corresponding to 8.75 seconds.

The results for the first example are presented in Figure 7.4, where we used the "chain-2" and "chain-6" models. Our encoding of the key $\mathcal{U}$ temporal operator scaled significantly better than a previous approach, likely due to the fact that our encoding is linear in the trajectory length k, while the previous encoding was quadratic in k.

The results for the surveillance example are presented in Figure 7.5. Due to the periodic nature of these tasks, the approaches presented in Karaman et al. [55] and Kwon and Agha [64] are not applicable. We were able to quickly compute trajectories that satisfied periodic temporal tasks for systems with more than 10 continuous states. The total time was dominated by preprocessing in YALMIP, and we expect that this can be reduced close to the solver time with a dedicated implementation.

We also compared our approach to reachability-based algorithms that compute a finite abstraction [61, 107]. We used the method in Wongpiromsarn et al. [107]

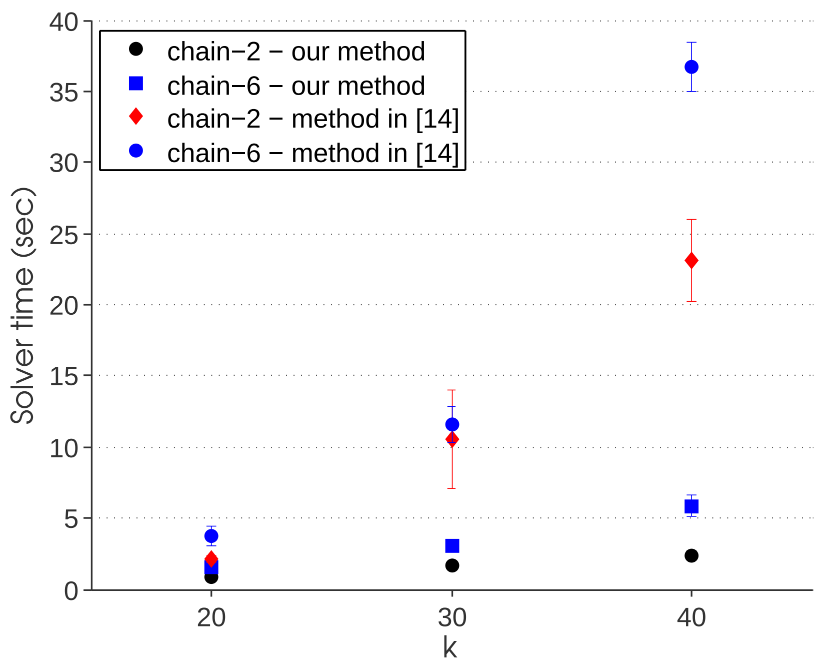


Figure 7.4: Solver time (mean ± standard error) to compute a feasible control input for the reach-avoid example, with our approach and the approach in Karaman et al. [55].

to compute a discrete abstraction for a two dimensional system in 22 seconds, and Kloetzer and Belta [61] report abstracting a four dimensional system in just over a minute. This contrasts with our mixed-integer approach that can routinely find solutions to such problems in seconds, although we do not compute a feedback controller. Our results appear particularly promising for situations where the environment is dynamically changing, and a finite abstraction must be repeatedly computed.

7.5 A Fragment of Temporal Logic

We also develop an improved encoding for a useful library of temporal operators for robotic tasks such as safe navigation, surveillance, persistent coverage, response to the environment, and visiting goals. The improvement is due to the fact that one only needs a single binary variable to represent the system being inside a polytope, instead of a binary variable for each halfspace. In the following definitions, ψ, ϕ,

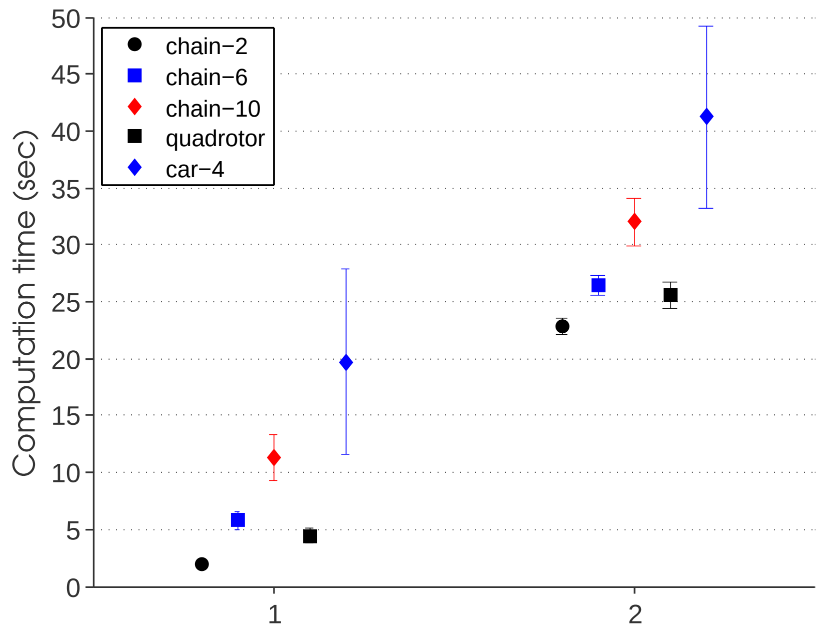


Figure 7.5: Solver (1) and total (2) time (mean ± standard error) to compute a feasible control input for various systems for the surveillance example. Each example used an average of 570 binary variables and 4,800 constraints.

and ψ_j (for a finite number of indices j) are propositional formulas. To simplify the presentation, we split these into three groups: core Φ_{core}, response Φ_{resp}, and fairness Φ_{fair}. We first define the syntax of the temporal operators, and then their semantics.

Syntax:

The core operators, $\Phi_{\text{core}} := \{\varphi_{\text{safe}}, \varphi_{\text{goal}}, \varphi_{\text{per}}, \varphi_{\text{live}}, \varphi_{\text{until}}\}$, specify fundamental properties such as safety, guarantee, persistence, liveness (recurrence), and until. These operators are:

$$\varphi_{\text{safe}} := \Box\psi, \qquad \varphi_{\text{goal}} := \Diamond\psi, \qquad \varphi_{\text{per}} := \Diamond\,\Box\,\psi, \qquad \varphi_{\text{live}} := \Box\,\Diamond\,\psi, \qquad \varphi_{\text{until}} := \psi\,\mathcal{U}\,\phi,$$

where φ_{safe} specifies safety, i.e., a property should invariantly hold, φ_{goal} specifies goal visitation, i.e., a property should eventually hold, φ_{per} specifies persistence, i.e., a property should eventually hold invariantly, and φ_{live} specifies liveness (recurrence), i.e., a property should hold repeatedly, as in surveillance, and φ_{until} specifies until, i.e., a property ψ should hold until another property ϕ holds.

The response operators, $\Phi_{\text{resp}} := \{\varphi^1_{\text{resp}}, \varphi^2_{\text{resp}}, \varphi^3_{\text{resp}}, \varphi^4_{\text{resp}}\}$, specify how the system responds to the environment. These operators are:

$$\varphi^1_{\text{resp}} := \square(\psi \implies \bigcirc\phi), \qquad \varphi^2_{\text{resp}} := \square(\psi \implies \Diamond\phi),$$
$$\varphi^3_{\text{resp}} := \Diamond\,\square\,(\psi \implies \bigcirc\phi), \qquad \varphi^4_{\text{resp}} := \Diamond\,\square\,(\psi \implies \Diamond\phi),$$

where φ^1_{resp} specifies next-step response to the environment, φ^2_{resp} specifies eventual response to the environment, φ^3_{resp} specifies steady-state next-step response to the environment, and φ^4_{resp} specifies steady-state eventual response to the environment.

Finally, the fairness operators, $\Phi_{\text{fair}} := \{\varphi^1_{\text{fair}}, \varphi^2_{\text{fair}}, \varphi^3_{\text{fair}}\}$, allow one to specify conditional tasks. These operators are:

$$\varphi^1_{\text{fair}} := \Diamond\psi \implies \bigwedge_{j=1}^{m} \Diamond\phi_j, \qquad \varphi^2_{\text{fair}} := \Diamond\psi \implies \bigwedge_{j=1}^{m} \square\,\Diamond\,\phi_j,$$
$$\varphi^3_{\text{fair}} := \square\,\Diamond\,\psi \implies \bigwedge_{j=1}^{m} \square\,\Diamond\,\phi_j,$$

where φ^1_{fair} specifies conditional goal visitation, and φ^2_{fair} and φ^3_{fair} specify conditional repeated goal visitation.

The fragment of LTL that we consider is built from the temporal operators defined above, as follows:

$$\varphi ::= \varphi_{\text{core}} \mid \varphi_{\text{resp}} \mid \varphi_{\text{fair}} \mid \varphi_1 \wedge \varphi_2, \tag{7.2}$$

where $\varphi_{\text{core}} \in \Phi_{\text{core}}$, $\varphi_{\text{resp}} \in \Phi_{\text{resp}}$, and $\varphi_{\text{fair}} \in \Phi_{\text{fair}}$.

Remark 7.4. To include disjunctions (e.g., $\varphi_1 \vee \varphi_2$), one can rewrite a formula in disjunctive normal form, where each clause is of the form (7.2). In what follows, each clause can then be considered separately, as the system (7.1) is deterministic.

Semantics:

We use set operations between a trajectory (run) $\mathbf{x} = \mathbf{x}(x_0, \mathbf{u})$ and subsets of $\mathcal{X}$ where particular propositional formulas hold to define satisfaction of a temporal logic formula [9]. We denote the set of states where propositional formula ψ holds by $[[\psi]]$. A run $\mathbf{x}$ *satisfies* the temporal logic formula φ, denoted by $\mathbf{x} \vDash \varphi$, if and only if certain set operations hold. Given propositional formulas ψ and ϕ, we relate satisfaction of (a partial list of) formulas of the form (7.2), with set operations as follows:

- $\mathbf{x} \vDash \Box\psi$ if and only if $x_i \in [[\psi]]$ for all i,
- $\mathbf{x} \vDash \Diamond \Box \psi$ if and only if there exists an index j such that $x_i \in [[\psi]]$ for all $i \geq j$,
- $\mathbf{x} \vDash \Diamond\psi$ if and only if $x_i \in [[\psi]]$ for some i,
- $\mathbf{x} \vDash \Box \Diamond \psi$ if and only if $x_i \in [[\psi]]$ for infinitely many i,
- $\mathbf{x} \vDash \psi \, \mathcal{U} \, \phi$ if and only if there exists an index j such that $x_j \in [[\phi]]$ and $x_i \in [[\psi]]$ for all $i < j$,
- $\mathbf{x} \vDash \Box(\psi \implies \bigcirc\phi)$ if and only if $x_i \notin [[\psi]]$ or $x_{i+1} \in [[\phi]]$ for all i,
- $\mathbf{x} \vDash \Box(\psi \implies \Diamond\phi)$ if and only if $x_i \notin [[\psi]]$ or $x_k \in [[\phi]]$ for some $k \geq i$ for all i,
- $\mathbf{x} \vDash \Diamond \Box (\psi \implies \bigcirc\phi)$ if and only if there exists an index j such that $x_i \notin [[\psi]]$ or $x_{i+1} \in [[\phi]]$ for all $i \geq j$,
- $\mathbf{x} \vDash \Diamond \Box (\psi \implies \Diamond\phi)$ if and only if there exists an index j such that $x_i \notin [[\psi]]$ or $x_k \in [[\phi]]$ for some $k \geq i$ for all $i \geq j$.

A run $\mathbf{x}$ *satisfies* a conjunction of temporal logic formulas $\varphi = \bigwedge_{i=1}^{m} \varphi_i$ if and only if the set operations for each temporal logic formula φ_i holds. The LTL formula φ is *satisfiable* by a system at state $x_0 \in \mathcal{X}$ if and only if there exists a control input sequence $\mathbf{u}$ such that $\mathbf{x}(x_0, \mathbf{u}) \vDash \varphi$.

7.6 A Mixed-Integer Linear Formulation for the Fragment

We will encode the temporal operators as mixed-integer constraints on $\mathbf{x}_{\text{pre}}$ and $\mathbf{x}_{\text{suf}}$. Let $\mathbf{x}_{\text{cat}} := \mathbf{x}_{\text{pre}}\mathbf{x}_{\text{suf}}$ denote the concatenation of $\mathbf{x}_{\text{pre}}$ and $\mathbf{x}_{\text{suf}}$, and assign time indices to $\mathbf{x}_{\text{cat}}$ as $\mathcal{T}_{\text{cat}} := \{0, 1, \ldots, T_s, \ldots, T\}$. Let $\mathcal{T}_{\text{pre}} := \{0, 1, \ldots, T_s - 1\}$ and $\mathcal{T}_{\text{suf}} := \{T_s, \ldots, T\}$, where T_s is the first time instance on the suffix. The infinite repetition of $\mathbf{x}_{\text{suf}}$ is enforced by the constraint $\mathbf{x}_{\text{cat}}(T_s) = f(\mathbf{x}_{\text{cat}}(T), u)$ for some $u \in \mathcal{U}$. We often identify $\mathbf{x}_{\text{pre}}(0)\cdots\sigma_{\text{pre}}(T_{\text{pre}})$ with $\mathbf{x}_{\text{cat}}(0)\cdots\mathbf{x}_{\text{cat}}(T_s-1)$ and $\mathbf{x}_{\text{suf}}(0)\cdots\sigma_{\text{suf}}(T_{\text{suf}})$ with $\mathbf{x}_{\text{cat}}(T_s)\cdots\mathbf{x}_{\text{cat}}(T)$ in the obvious manner.

7.6.1 Relating the Dynamics and Propositions

We now relate the state of a system to the set of atomic propositions that are *True* at each time instance. We assume that each propositional formula ψ is described at time t by the union of a finite number of polytopes, indexed by the finite index set I_t^ψ. Let $[[\psi]](t) := \{x \in \mathcal{X} \mid H_t^{\psi_i} x \leq K_t^{\psi_i} \text{ for some } i \in I_t^\psi\}$ represent the set of states that satisfy propositional formula ψ at time t. We assume that these have been constructed as necessary from the system's original atomic propositions. We note that a proposition preserving partition [4] is not necessary or even desired.

For each propositional formula ψ, introduce binary variables $z_t^{\psi_i} \in \{0,1\}$ for all $i \in I_t^\psi$ and for all $t \in \mathcal{T}$. Let x_t be the state of the system at time t, and M be a vector of sufficiently large constants. The *big-M* formulation

$$\begin{aligned} &H_t^{\psi_i} x_t \leq K_t^{\psi_i} + M(1 - z_t^{\psi_i}), \quad \forall i \in I_t^\psi \\ &\sum_{i \in I_t^\psi} z_t^{\psi_i} = 1 \end{aligned} \tag{7.3}$$

enforces the constraint that $x_t \in [[\psi]](t)$ at time t. Define $P_t^\psi := \sum_{i \in I_t^\psi} z_t^{\psi_i}$. If $P_t^\psi = 1$, then $x_t \in [[\psi]](t)$. If $P_t^\psi = 0$, then nothing can be inferred.

The big-M formulation may give poor continuous relaxations of the binary variables, i.e., $z_t^{\psi_i} \in [0,1]$, which may lead to poor performance during optimization [1]. Such relaxations are frequently used during the solution of mixed-integer linear programs [1]. Thus, we introduce an alternate representation whose continuous relaxation is the convex hull of the original set $[\![\psi]\!](t)$. This formulation is well-known in the optimization community [51], but does not appear in the trajectory generation literature ([35, 85, 96] and references therein). As such, this formulation may be of independent interest for trajectory planning with obstacles.

The *convex hull* formulation

$$\begin{aligned} &H_t^{\psi_i} x_t^i \le K_t^{\psi_i} z_t^{\psi_i}, \quad \forall i \in I_t^{\psi} \\ &\sum_{i \in I_t^{\psi}} z_t^{\psi_i} = 1, \\ &\sum_{i \in I_t^{\psi}} x_t^i = x_t \end{aligned} \tag{7.4}$$

represents the same set as the big-M formulation (7.3). While the convex hull formulation introduces more continuous variables, it gives the tightest linear relaxation of the disjunction of the polytopes, and reduces the need to select the M parameters [51]. Note that we will only use the convex hull formulation (7.4) for safety and persistence formulas (i.e., φ_{safe} and φ_{per}) in Section 7.6.2, as $P_t^{\psi} = 0$ enforces $x = 0$.

Regardless of whether one uses the big-M or convex hull formulation, only one binary variable is needed for each polyhedron (i.e., finite conjunction of halfspaces). This compares favorably with our previous approach (see also Karaman et al. [55]), where a binary variable is introduced for each halfspace. Additionally, the auxiliary continuous variables and mixed-integer constraints previously used are not needed because we use implication. For simple tasks such as $\varphi = \Diamond\psi$, our method can use significantly fewer binary variables than previously needed, depending on the number of halfspaces and polytopes needed to describe $[\![\psi]\!]$.

For every temporal operator described in the following section, the constraints in (7.3) or (7.4) should be understood to be implicitly applied to the corresponding propositional formulas so that $P_t^{\psi} = 1$ implies that the system satisfies ψ at time

t. Also, note that we use different binary variables for each formula—even when representing the same set.

7.6.2 The Mixed-Integer Linear Constraints

In this section, the trajectory parameterization $\mathbf{x}$ has been *a priori* split into a prefix $\mathbf{x}_{\text{pre}}$ and a suffix $\mathbf{x}_{\text{suf}}$. This assumption can be relaxed, so that the size of $\mathbf{x}_{\text{pre}}$ and $\mathbf{x}_{\text{suf}}$ are optimization variables (see [104] for details).

In the following, the correctness of the constraints applied to $\mathbf{x}_{\text{pre}}$ and $\mathbf{x}_{\text{suf}}$ comes directly from the temporal logic semantics given in Section 7.5 and the form of the trajectory $\mathbf{x} = \mathbf{x}_{\text{pre}}(\mathbf{x}_{\text{suf}})^{\omega}$. The most important factors are whether a property can be verified over finite- or infinite-horizons. All infinite-horizon (liveness) properties must be satisfied on the suffix $\mathbf{x}_{\text{suf}}$.

We begin with the fundamental temporal operators Φ_{core}. Safety and persistence require a mixed-integer linear constraint for each time step, while guarantee and liveness only require a single mixed-integer linear constraint.

Safety, $\varphi_{\text{safe}} = \Box\psi$, is satisfied by the constraints

$$\begin{aligned} P_t^{\psi} &= 1, \quad \forall t \in \mathcal{T}_{\text{pre}}, \\ P_t^{\psi} &= 1, \quad \forall t \in \mathcal{T}_{\text{suf}}, \end{aligned}$$

which ensure that the system is always in a $[[\psi]]$ region. Similarly, persistence, $\varphi_{\text{per}} = \Diamond \Box \psi$, is enforced by

$$P_t^{\psi} = 1, \quad \forall t \in \mathcal{T}_{\text{suf}},$$

which ensures that the system eventually remains in a $[[\psi]]$ region.

Guarantee, $\varphi_{\text{goal}} = \Diamond\psi$, is satisfied by the constraints

$$\sum_{t \in \mathcal{T}_{\text{pre}}} P_t^{\psi} + \sum_{t \in \mathcal{T}_{\text{suf}}} P_t^{\psi} = 1,$$

which ensures that the system eventually visits a $[\![\psi]\!]$ region. Similarly, liveness $\varphi_{\text{live}} = \square\diamond\psi$ is enforced by

$$\sum_{t\in\mathcal{T}_{\text{suf}}} P_t^{\psi} = 1,$$

which ensures that the system repeatedly visits a $[\![\psi]\!]$ region.

Until, $\varphi_{\text{until}} = \psi\,\mathcal{U}\,\phi$, is enforced by

$$\begin{aligned} P_0^{\phi} &= s_0, \\ P_t^{\phi} &= s_t - s_{t-1}, \quad t = 1,\dots,T \\ P_t^{\psi} &= 1 - s_t, \quad \forall t\in\mathcal{T}, \end{aligned}$$

where we use auxiliary binary variables $s_t \in \{0,1\}$ for all $t\in\mathcal{T}$ such that $s_t \le s_{t+1}$ for $t = 0,\dots,T-1$ and $s_T = 1$.

Now, consider the response temporal operators Φ_{resp}. For these formulas, the definition of implication is used to convert each inner formula into a disjunction between a property that holds at a state and a property that holds at some point in the future. The response formulas require a mixed-integer linear constraint for each time step.

For next-step response, $\varphi^1_{\text{resp}} = \square(\psi \implies \bigcirc\phi) = \square(\neg\psi \vee \bigcirc\phi)$, the additional constraints are

$$\begin{aligned} &P_t^{\neg\psi} + P_{t+1}^{\phi} = 1, \quad t = 0,\dots,T_s,\dots,T-1, \\ &P_T^{\neg\psi} + P_{T_s}^{\phi} = 1, \end{aligned}$$

Similarly, steady-state next-step response, $\varphi^3_{\text{resp}} = \diamond\square(\psi \implies \bigcirc\phi) = \diamond\square(\neg\psi \vee \bigcirc\phi)$, is satisfied by

$$\begin{aligned} &P_t^{\neg\psi} + P_{t+1}^{\phi} = 1, \quad t = T_s,\dots,T-1, \\ &P_T^{\neg\psi} + P_{T_s}^{\phi} = 1, \end{aligned}$$

Eventual response, $\varphi^2_{\text{resp}} = \Box(\psi \implies \Diamond\phi) = \Box(\neg\psi \vee \Diamond\phi)$, requires the following constraints

$$\begin{aligned} P_t^{\neg\psi} + \sum_{\tau=t}^{T} P_\tau^{\phi} = 1, \quad \forall t \in \mathcal{T}_{\text{pre}}, \\ P_t^{\neg\psi} + \sum_{t\in\mathcal{T}_{\text{suf}}} P_t^{\phi} = 1, \quad \forall t \in \mathcal{T}_{\text{suf}}. \end{aligned}$$

Similarly, for steady-state eventual response, $\varphi^4_{\text{resp}} = \Diamond\Box(\psi \implies \Diamond\phi) = \Diamond\Box(\neg\psi \vee \Diamond\phi)$, the additional constraints are

$$P_t^{\neg\psi} + \sum_{t\in\mathcal{T}_{\text{suf}}} P_t^{\phi} = 1, \quad \forall t \in \mathcal{T}_{\text{suf}}.$$

Now, consider the fairness temporal operators Φ_{fair}. In the following, the definition of implication is used to rewrite the inner formula as disjunction between a single safety (persistence) property and a conjunction of guarantee (liveness) properties. These formulas require a mixed-integer linear constraint for each conjunction in the response, and each time step.

Conditional goal visitation, $\varphi^1_{\text{fair}} = \Diamond\psi \implies \bigwedge_{j=1}^m \Diamond\phi_j = \Box\neg\psi \vee \bigwedge_{j=1}^m \Diamond\phi_j$, is specified by

$$P_t^{\neg\psi} + \sum_{t\in\mathcal{T}} P_t^{\phi_j} = 1, \quad \forall j = 1,\dots,m, \forall t \in \mathcal{T}.$$

Conditional repeated goal visitation, $\varphi^2_{\text{fair}} = \Diamond\psi \implies \bigwedge_{j=1}^m \Box\Diamond\phi_j = \Box\neg\psi \vee \bigwedge_{j=1}^m \Box\Diamond\phi_j$, is enforced as

$$P_t^{\neg\psi} + \sum_{t\in\mathcal{T}_{\text{suf}}} P_t^{\phi_j} = 1, \quad \forall j = 1,\dots,m, \forall t \in \mathcal{T}.$$

Similarly, $\varphi^3_{\text{fair}} = \Box\Diamond\psi \implies \bigwedge_{j=1}^m \Box\Diamond\phi_j = \Diamond\Box\neg\psi \vee \bigwedge_{j=1}^m \Box\Diamond\phi_j$, is represented by

$$P_t^{\neg\psi} + \sum_{t\in\mathcal{T}_{\text{suf}}} P_t^{\phi_j} = 1, \quad \forall j = 1,\dots,m, \ \forall t \in \mathcal{T}_{\text{suf}}.$$

We have encoded the temporal logic specifications on the system variables using mixed-integer linear constraints. Note that the equality constraints on the binary variables dramatically reduce search space. In Section 7.3.2, we discuss adding dynamics to further constrain the possible behaviors of the system.

7.7 More Examples

We demonstrate our techniques on a variety of motion planning problems. The first example is a chain of integrators parameterized by dimension. Our second example is a quadrotor model from [97]. Our final example is a nonlinear car-like vehicle with drift. All computations were done on a laptop with a 2.4 GHz dual-core processor and 4 GB of memory using CPLEX [1] with YALMIP [71].

The environment and task is motivated by a pickup-and-delivery scenario. All properties should be understood to be with respect to regions in the plane (see Figure 7.1). Let P be a region where supplies can be picked up, and $D1$ and $D2$ be regions where supplies must be delivered. The robot must remain in the safe region S (in white). Formally, the task specification is $\varphi = \square S \wedge \square\diamond P \wedge \square\diamond D1 \wedge \square\diamond D2$. Additionally, we minimize an additive cost function where $c(x_t, u_t) = |u_t|$ penalizes the control input.

In the remainder of this section, we consider this temporal logic motion planning problem for different system models. We use the simultaneous (sim.) approach described in Section 7.6.2, and also a sequential (seq.) approach from [104] that first computes the suffix and then the prefix. A trajectory of length 60 (split evenly between the prefix and suffix) is used in all cases, and all results are averaged over 20 randomly-generated environments. The simultaneous approach uses between 300 and 469 binary variables, with a mean of 394. Finally, all continuous-time models are discretized with a 0.5 second sample time.

We use the quadrotor model previously defined in (6.7.2). Also, we use the fact that the quadrotor is differentially flat [76] to generate trajectories for the nonlinear model (with fixed yaw). We parameterize the flat output $p \in \mathbb{R}^3$ with eight piecewise polynomials of degree three, and then optimize over their coefficients to compute a

		Feasible soln. (sec)		Num. solved	
Model	Dim.	Sim.	Seq.	Sim.	Seq.
chain-2	4	1.10 ± .09	0.64 ± .06	20	20
chain-6	12	4.70 ± .48	2.23 ± .15	20	20
chain-10	20	9.38 ± 1.6	3.74 ± .29	20	19
quadrotor	10	4.20 ± .66	1.80 ± .15	20	20
quadrotor-flat	10	2.26 ± .36	1.99 ± 1.0	20	20
car-3	3	43.9 ± .77	10.7 ± 2.0	4	20
car-4	3	42.4 ± 1.7	18.7 ± 3.1	2	18
car-flat	3	15.8 ± 3.8	14.0 ± 4.4	12	14

Table 7.1: Time until a feasible solution was found (mean ± standard error) and number of problems (out of 20) solved in 45 seconds using the big-M formulation (7.3) with M = 10.

		Feasible soln. (sec)		Num. solved	
Model	Dim.	Sim.	Seq.	Sim.	Seq.
chain-2	4	1.94 ± .23	0.94 ± .11	20	20
chain-6	12	12.4 ± 2.7	2.89 ± .32	20	20
chain-10	20	16.9 ± 3.0	7.28 ± 1.2	17	15
quadrotor	10	18.9 ± 3.8	2.80 ± .35	16	20
car-3	3	37.3 ± 3.1	13.3 ± 1.6	8	20

Table 7.2: Time until a feasible solution was found (mean ± standard error) and number of problems (out of 20) solved in 45 seconds using the convex hull formulation (7.4).

smooth trajectory. Afterwards, we check that the trajectory does not violate the control input constraints. Results are given in Table 7.1 under "quadrotor-flat."

We use the nonlinear car-like model introduced in (6.7.3), but with v fixed at 1 (m/s). Additionally, we use the flat output $(x, y) \in \mathbb{R}^2$ to generate trajectories for the nonlinear car-like model in a similar manner as for the quadrotor model. Results are given in Table 7.1 under "car-flat."

We compare to the finite-horizon mixed-integer formulation given in Karaman et al. [55]. Consider the task $\varphi = \diamond\psi$, where $[[\psi]]$ is a convex polytope defined by m halfspaces. Our method uses one binary variable at each time step, while their approach uses m. Additionally, while we encode eventually ($\diamond$) using a single constraint, their approach uses a number of constraints quadratic in the the trajectory length.

Finally, the convex hull formulation performed poorly in our examples. There is an empirical tradeoff between having tighter continuous relaxations and the number of continuous variables in the formulation. We hypothesize that the convex hull formulation will be most useful in cases when 1) the number of binary variables is large, or 2) the cost function is minimized near the boundary of the region.

7.8 Conclusions

In this chapter, we presented a mixed-integer programming-based method for optimal control of nonlinear systems subject to linear temporal logic task specifications. We directly encoded an LTL formula as mixed-integer linear constraints on continuous system variables, which avoided the computationally expensive processes of creating a finite abstraction of the system and creating a Büchi automaton for the specification. We solved LTL motion planning tasks for dynamical systems with more than 10 continuous states, and showed that our encoding of the until operator theoretically and empirically improves on previous work.

Possible future work includes incorporating reactive environments and receding horizon control approaches. It would also be useful to directly encode metric and past operators, exploit incremental computation for varying bounds k, and perform a more detailed comparison of mixed-integer linear programming solvers with SMT solvers [41].

Chapter 8

Conclusions and Future Work

8.1 Summary

This thesis presented novel algorithms for the specification and design of controllers that guarantee correct and efficient behaviors of robots, autonomous vehicles, and other cyberphysical systems. Temporal logic was used as a language to formally specify complex tasks and system properties. We developed control synthesis algorithms for optimal control and robust control of dynamical systems with linear temporal logic (LTL) specifications. These algorithms are efficient and some extend to high-dimensional dynamical systems.

The first contribution of this thesis was the extension of a classical control synthesis approach for systems with LTL specifications to optimal and robust control, respectively. We showed how to extend automata-based synthesis techniques for discrete abstractions of dynamical systems to include notions of optimality or robustness. Importantly, optimal or robust controllers can be synthesized at little extra computational cost compared to synthesizing a feasible controller.

The second contribution of this thesis addressed the scalability of control synthesis with linear temporal logic (LTL) specifications. We introduced a fragment of LTL for which one can compute feasible control policies in time polynomial in the size of the system and specification, as opposed to doubly-exponential in the size of the specification for the full LTL. Additionally, we showed how to compute optimal control policies for a variety of cost functions, and identify interesting cases in which this can be done in polynomial time. These methods are particularly relevant in an online control setting, as one is guaranteed that a feasible solution will be found in a predictable amount of time.

The final contribution of this thesis was the development of algorithms for computing feasible trajectories for high-dimensional, nonlinear systems with LTL specifications. These algorithms avoid a potentially computationally expensive process of computing a discrete abstraction, and instead compute directly on the system's continuous state space. We first showed how an automaton representing the specification can be used to directly encode a series of constrained-reachability subproblems, which can be solved with standard methods. We also gave a complete encoding of LTL as mixed-integer linear programming constraints. We demonstrated these approaches with numerical experiments on temporal logic motion planning problems with high-dimensional dynamical systems with more than 10 continuous states.

8.2 Future Work

There are many directions for future research towards developing the theoretical foundations and computational tools for the systematic specification, design, and verification of complex embedded systems. A fundamental challenge is the development of a scalable framework for design and verification that mirrors the way that we build complex systems: component-by-component, with interfaces that help decouple subsystems. Another promising area is the use of temporal logic as a language for human-robot collaboration. There are also opportunities in bridging the gap between how industry currently designs systems (simulate and test) and a formal, model-based approach. Finally, the experimental application of these techniques to aerospace and robotics systems has the potential to make a large impact.

Decoupling Task and Motion Planning

A major obstacle to scalable controller synthesis for robots and cyberphysical systems is the coupling between the constraints imposed by the task and those imposed by the dynamics. Identifying conditions when these constraints can be decoupled will be critical for solving large, real-world problems. While the creation of a *discrete abstraction* theoretically lets one decouple these constraints, abstractions are computationally expensive to compute many high-dimensional problems.

Our work on automaton-guided control design (see Chapter 6) is a preliminary step towards decoupling these constraints. By using control-theoretic notions such as small-time local controllability, one can begin to give conditions under which a sequence of tasks (corresponding to regions of the state space) is dynamically feasible. Additionally, one can potentially use the automaton to guide the construction of a library of appropriate motion primitives to complete the task.

Human-Robot Teaming

A benefit of using linear temporal logic to specify tasks is that it is similar to human language—at least compared to other common logics. There is great potential in the use of formal methods to reduce mistakes and increase the supervisory capability of human operators overseeing groups of robots and unmanned vehicles. It also seems promising to explore the use of temporal logic as a task-specification language for human-robot collaborative assembly tasks.

Learning Specifications from Simulations

The process of formalizing a set of logical requirements can be a significant undertaking for a complex system. On the other hand, it is relatively easy to say whether or not a given system simulation is good or bad, and use machine learning techniques to train classifiers. The combination of formal specifications and learning-based classifiers appears to be a promising new approach for the specification and design of large-scale dynamical systems. Initial work in this area (e.g., Seshia [88]) as already started to show the power of combining machine learning with formal methods.

Bibliography

[1] *User's Manual for CPLEX V12.4. IBM, 2011.* URL `http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r4/topic/ilog.odms.studio.help/pdf/usrcplex.pdf`.

[2] A. Abate, A. D'Innocenzo, and M. D. Di Benedetto. Approximate abstractions of stochastic hybrid systems. *IEEE Trans. on Automatic Control*, 56:2688–2694, 2011.

[3] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Logic*, 5(1):1–25, 2004.

[4] Rajeev Alur, Thomas A. Henzinger, Gerardo Lafferriere, and George J. Pappas. Discrete abstractions of hybrid systems. *Proc. IEEE*, 88(7):971–984, 2000.

[5] Rajeev Alur, Thao Dang, and Franjo Ivancic. Counterexample-guided predicate abstraction of hybrid systems. In *Proc. of TACAS*, 2003.

[6] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying industrial hybrid systems with MathSAT. *Electronic Notes in Theoretical Computer Science*, 119:17–32, 2005.

[7] S. Azuma and G. J. Pappas. Discrete abstraction of stochastic nonlinear systems: A bisimulation function approach. In *Proc. of American Control Conference*, 2010.

[8] J. Andrew Bagnell, Andrew Y. Ng, and Jeff G. Schneider. Solving uncertain Markov decision processes. Technical report, Carnegie Mellon University, 2001.

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

[10] C. Belta and L. C. G. J. M. Habets. Controlling of a class of nonlinear systems on rectangles. *IEEE Trans. on Automatic Control*, 51:1749–1759, 2006.

[11] C. Belta, V. Isler, and G. J. Pappas. Discrete abstractions for robot motion planning and control in polygonal environments. *IEEE Trans. on Robotics*, 21: 864–874, 2005.

[12] A. Bemporad and M. Morari. Control of systems integrating logic, dynamics, and constraints. *Automatica*, 35:407–427, 1999.

[13] D. P. Bertsekas. *Dynamic Programming and Optimal Control (Vol. I and II).* Athena Scientific, 2001.

[14] John T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming, 2nd edition.* SIAM, 2000.

[15] A. Bhatia, M. R. Maly, L. E. Kavraki, and M. Y. Vardi. Motion planning with complex goals. *IEEE Robotics and Automation Magazine*, 18:55–64, 2011.

[16] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, 1999.

[17] Armin Biere, Kejo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2:1–64, 2006.

[18] C. E. Blair, R. G. Jeroslow, and J. K. Lowe. Some results and experiments in programming techniques for propositional logic. *Computers and Operations Research*, 13:633–645, 1986.

[19] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) designs. *J. of Computer and System Sciences*, 78:911–938, 2012.

[20] O. Buffet. Reachability analysis for uncertain ssps. In *Proc. of the IEEE Int. Conf. on Tools with Artificial Intelligence*, 2005.

[21] Stephane Cambon, Rachid Alami, and Fabien Gravot. A hybrid approach to intricate motion, manipulation and task planning. *Int. J. of Robotics Research*, 28:104–126, 2009.

[22] K. Chatterjee, T. A. Henzinger, and M. Jurdzinski. Mean-payoff parity games. In *Annual Symposium on Logic in Computer Science (LICS)*, 2005.

[23] Krishnendu Chatterjee, Koushik Sen, and Tom Henzinger. Model-checking omega-regular properties of interval Markov chains. In *Foundations of Software Science and Computation Structure (FoSSaCS)*, pages 302–317, 2008.

[24] Y. Chen, J. Tumova, and C. Belta. LTL robot motion control based on automata learning of environmental dynamics. In *Proc. of Int. Conf. on Robotics and Automation*, 2012.

[25] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *CAV*, number 1633 in Lecture Notes in Computer Science, pages 495–499, 1999.

[26] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*. Springer, 2000.

[27] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[28] Edmund M. Clarke, Ansgar Fehnker, Bruce H. Krogh, Joel Ouaknine, Olaf Stursberg, and Michael Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. of Foundations of Computer Science*, 14:583–604, 2003.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms: 2nd ed.* MIT Press, 2001.

[30] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *J. of the ACM*, 42:857–907, 1995.

[31] G. B. Dantzig, W. O. Blattner, and M. R. Rao. Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem. In P. Rosenstiehl, editor, *Theory of Graphs*, pages 77–84. Dunod, Paris and Gordon and Breach, New York, 1967.

[32] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17:889–899, 1998.

[33] Luca de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.

[34] Xu Chu Ding, Stephen L. Smith, Calin Belta, and Daniela Rus. LTL control in uncertain environments with probabilistic satisfaction guarantees. In *Proc. of 18th IFAC World Congress*, 2011.

[35] M. G. Earl and R. D'Andrea. Iterative MILP methods for vehicle-control problems. *IEEE Trans. on Robotics*, 21:1158–1167, 2005.

[36] R. Ehlers. Generalized Rabin(1) synthesis with applications to robust system synthesis. In *NASA Formal Methods*. Springer, 2011.

[37] E. Emerson and C. Jutla. The complexity of tree automata and logic of programs. In *Proc. of FOCS*, 1988.

[38] E. Allen Emerson. Handbook of theoretical computer science (vol. B). In Jan van Leeuwen, editor, *Temporal and Modal Logic*, chapter Temporal and modal logic, pages 995–1072. MIT Press, 1990.

[39] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas. Temporal logic motion planning for dynamic robots. *Automatica*, 45:343–352, 2009.

[40] M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *Electronic Notes in Theoretical Computer Science*, 133:119–137, 2005.

[41] Martin Fränzle and Christian Herde. HySAT: an efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30:179–198, 2007.

[42] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proc. of CAV*, 2001.

[43] Nicolo Giorgetti, G. J. Pappas, and A. Bemporad. Bounded model checking of hybrid dynamical systems. In *Proc. of IEEE Conf. on Decision and Control*, 2005.

[44] Robert Givan, Sonia Leach, and Thomas Dean. Bounded-parameter Markov decision processes. *Artificial Intelligence*, 122:71–109, 2000.

[45] Ebru Aydin Gol, Mircea Lazar, and Calin Belta. Language-guided controller synthesis for discrete-time linear systems. In *Proc. of Hybrid Systems: Computation and Control*, 2012.

[46] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research.* Springer-Verlag New York, Inc., 2002.

[47] L. Habets, P. J. Collins, and J. H. van Schuppen. Reachability and control synthesis for piecewise-affine hybrid systems on simplices. *IEEE Trans. on Automatic Control*, 51:938–948, 2006.

[48] M. Hartmann and J. B. Orlin. Finding mimimum cost to time ratio cycles with small integral transit times. *Networks*, 23:567–574, 1993.

[49] Gerard Holzmann. *Spin Model Checker, The Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[50] J. N. Hooker and C. Fedjki. Branch-and-cut solution of inference problems in propositional logic. *Annals of Mathematics and Artificial Intelligence*, 1: 123–139, 1990.

[51] Robert G. Jeroslow. Representability in mixed integer programming, I: Characterization results. *Discrete Applied Mathematics*, 17:223–243, 1987.

[52] Leslie P. Kaelbling and Tomas Lozano-Perez. Hierarchical task and motion planning in the now. In *Proc. of IEEE Int. Conf. on Robotics and Automaton*, 2011.

[53] S. Karaman and E. Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *Proc. of IEEE Conf. on Decision and Control*, 2009.

[54] S. Karaman and E. Frazzoli. Linear temporal logic vehicle routing with applications to multi-UAV mission planning. *Int. J. of Robust and Nonlinear Control*, 21:1372–1395, 2011.

[55] S. Karaman, R. G. Sanfelice, and E. Frazzoli. Optimal control of mixed logical dynamical systems with linear temporal logic specifications. In *Proc. of IEEE Conf. on Decision and Control*, pages 2117–2122, 2008.

[56] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning with deterministic μ-calculus specifications. In *Proc. of American Control Conf.*, 2012.

[57] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[58] R. M. Karp and J. B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete and Applied Mathematics*, 3:37–45, 1981.

[59] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. Autom.*, 12:566–580, 1996.

[60] J. Klein and C. Baier. Experiments with deterministic omega-automata for formulas of linear temporal logic. *Theoretical Computer Science*, 363:182–195, 2006.

[61] M. Kloetzer and C. Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Trans. on Automatic Control*, 53(1):287–297, 2008.

[62] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Temporal logic-based reactive mission and motion planning. *IEEE Trans. on Robotics*, 25:1370–1381, 2009.

[63] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In *Proc. of 23rd Int. Conf. on Computer Aided Verification*, 2011.

[64] Y. Kwon and G. Agha. LTLC: Linear temporal logic for control. In *Proc. of HSCC*, pages 316–329, 2008.

[65] M. Lahijanian, S. B. Andersson, and C. Belta. Control of Markov decision processes from PCTL specifications. In *Proceedings of the American Control Conference*, 2011.

[66] M. Lahijanian, S. B. Andersson, and C. Belta. Temporal logic motion planning and control with probabilistic satisfaction guarantees. *IEEE Trans. on Robotics*, 28:396–409, 2012.

[67] S. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *Int. J. of Robotics Research*, 20:378–400, 2001.

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

[69] E. L. Lehmann and Joseph P. Romano. *Testing Statistical Hypotheses*. Springer, 2005.

[70] J. Liu, U. Topcu, N. Ozay, and R. M. Murray. Synthesis of reactive control protocols for differentially flat systems. In *Proc. of IEEE Conf. on Decision and Control*, 2012.

[71] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proc. of the CACSD Conference*, Taipei, Taiwan, 2004. Software available at http://control.ee.ethz.ch/~joloef/yalmip.php.

[72] Rupak Majumdar, Elaine Render, and Paulo Tabuada. Robust discrete synthesis against unspecified disturbances. In *Proc. Hybrid Systems: Computation and Control*, pages 211–220, 2011.

[73] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS 95*, volume 900, pages 229–242. Springer, 1995.

[74] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992. ISBN 0-387-97664-7.

[75] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *Proc. of Int. Conf. on Robotics and Automaton*, 2011.

[76] Daniel Mellinger, Alex Kushleyev, and Vijay Kumar. Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams. In *Proc. of Int. Conf. on Robotics and Automation*, 2012.

[77] Mark B. Milam, Ryan Franz, J. E. Hauser, and Richard M. Murray. Receding horizon control of vectored thrust flight experiment. *IEEE Proc., Control Theory Appl.*, 152:340–348, 2005.

[78] Arch W. Naylor and George R. Sell. *Linear Operator Theory in Engineering and Science*. Springer-Verlag, 1982.

[79] Arnab Nilim and Laurent El Ghaoui. Robust control of Markov decision processes with uncertain transition matrices. *Operations Research*, 53:780–798, 2005.

[80] Charles E. Noon and James C. Bean. An efficient transformation of the generalized traveling salesman problem. *INFOR*, 31:39–44, 1993.

[81] E. Plaku, L. E. Kavraki, and M. Y. Vardi. Motion planning with dynamics by a synergistic combination of layers of planning. *IEEE Trans. on Robotics*, 26: 469–482, 2010.

[82] Erion Plaku and Gregory D. Hager. Sampling-based motion and symbolic action planning with geometric and differential constraints. In *Proc. of IEEE Int. Conf. on Robotics and Automaton*, 2010.

[83] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symp. on Princp. of Prog. Lang.*, pages 179–190, 1989.

[84] Amir Pnueli. The temporal logic of programs. In *Proc. of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[85] A. Richards and J. P. How. Aircraft trajectory planning with collision avoidance using mixed integer linear programming. In *American Control Conference*, 2002.

[86] A. G. Richards, T. Schouwenaars, J. P. How, and E. Feron. Spacecraft trajectory planning with avoidance constraints using mixed-integer linear programming. *AIAA J. of Guidance, Control, and Dynamics*, 25:755–764, 2002.

[87] Jay K. Satia and Roy E. Lave Jr. Markovian decision processes with uncertain transition probabilities. *Operations Research*, 21(3):pp. 728–740, 1973.

[88] Sanjit A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *Proceedings of the Design Automation Conference (DAC)*, pages 356–365, 2012.

[89] A. Sistla and E. Clarke. The complexity of propositional linear temporal logics. *J. of the ACM*, 32:733–749, 1985.

[90] S. L. Smith, J. Tumova, C. Belta, and D. Rus. Optimal path planning for surveillance with temporal-logic constraints. *Int. J. of Robotics Research*, 30: 1695–1708, 2011.

[91] Olaf Stursberg. Synthesis of supervisory controllers for hybrid systems using abstraction refinement. In *Proc. of the 16th IFAC World Congress*, 2005.

[92] Claire J. Tomlin, Ian M. Mitchell, Alexandre M. Bayen, and Meeko Oishi. Computational techniques for the verification of hybrid systems. *Proc. IEEE*, 91:986–1001, 2003.

[93] U. Topcu, N. Ozay, and J. Liu. On synthesizing robust discrete controllers under modeling uncertainty. In *International Conference on Hybrid Systems: Computation and Control*, 2012.

[94] P. Toth and D. Vigo, editors. *The Vehicle Routing Problem.* Philadelphia, PA: SIAM, 2001.

[95] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Logic in Computer Science*, pages 322–331, 1986.

[96] M. P. Vitus, V. Pradeep, J. Hoffmann, S. L. Waslander, and C. J. Tomlin. Tunnel-MILP: path planning with sequential convex polytopes. In *Proc. of AIAA Guidance, Navigation, and Control Conference*, 2008.

[97] Dustin J. Webb and Jur van den Berg. Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics. In *Proc. of IEEE Int. Conf. on Robotics and Automation*, 2013.

[98] Jason Wolfe, Bhaskara Marthi, and Stuart Russell. Combined task and motion planning for mobile manipulation. In *Proc. of ICAPS*, 2010.

[99] E. M. Wolff and R. M. Murray. Optimal control of nonlinear systems with temporal logic specifications. In *Proc. of Int. Symposium on Robotics Research*, 2013.

[100] E. M. Wolff, U. Topcu, and R. M. Murray. Robust control of uncertain Markov decision processes with temporal logic specifications. In *Proc. of IEEE Conference on Decision and Control*, 2012.

[101] E. M. Wolff, U. Topcu, and R. M. Murray. Optimal control with weighted average costs and temporal logic specifications. In *Proc. of Robotics: Science and Systems*, 2012.

[102] E. M. Wolff, U. Topcu, and R. M. Murray. Optimal control of non-deterministic systems for a computationally efficient fragment of temporal logic. In *Proc. of IEEE Conf. on Decision and Control*, 2013.

[103] E. M. Wolff, U. Topcu, and R. M. Murray. Optimization-based control of nonlinear systems with linear temporal logic specifications. In *Proc. of Int. Conf. on Robotics and Automation*, 2014.

[104] Eric M. Wolff and Richard M. Murray. Optimal control of mixed logical dynamical systems with long-term temporal logic specifications. Technical report, California Institute of Technology, 2013. URL `http://resolver.caltech.edu/CaltechCDSTR:2013.001`.

[105] Eric M. Wolff, Ufuk Topcu, and Richard M. Murray. Efficient reactive controller synthesis for a fragment of linear temporal logic. In *Proc. of Int. Conf. on Robotics and Automation*, 2013.

[106] Eric M. Wolff, Ufuk Topcu, and Richard M. Murray. Automaton-guided controller synthesis for nonlinear systems with temporal logic. In *Proc. of Int. Conf. on Intelligent Robots and Systems*, 2013.

[107] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray. TuLiP: A software toolbox for receding horizon temporal logic planning. In *Proc. of*

Int. Conf. on Hybrid Systems: Computation and Control, 2011. http://tulip-control.sf.net.

[108] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon temporal logic planning. *IEEE Trans. on Automatic Control*, 2012.

[109] Di Wu and Xenofon Koutsoukos. Reachability analysis of uncertain systems using bounded-parameter Markov decision processes. *Artificial Intelligence*, 172:945–954, 2008.

[110] Boyan Yordanov, Jana Tumova, Ivana Cerna, Jiri Barnat, and Calin Belta. Formal analysis of piecewise affine systems through formula-guided refinement. *Automatica*, 49:261–266, 2013.