SPECIFICATION-GUIDED REINFORCEMENT LEARNING

Kishor Jothimurugan

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2023

Supervisor of Dissertation

Rajeev Alur, Zisman Family Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Sampath Kannan, Henry Salvatori Professor of Computer and Information Science
George J. Pappas, UPS Foundation Professor of Electrical and Systems Engineering
Osbert Bastani, Assistant Professor of Computer and Information Science
Thomas A. Henzinger, Professor at Institute of Science and Technology Austria

SPECIFICATION-GUIDED REINFORCEMENT LEARNING

Dedicated to my parents.

ACKNOWLEDGEMENT

First and foremost, I want to thank my advisor, Rajeev, for being incredibly supportive and encouraging throughout my time at Penn. During my early days as a Ph.D. student, he taught me a lot about how to conduct research as well as effectively present my work. He also gave me the confidence to pursue research in this interdisciplinary area involving formal methods and machine learning despite my background being primarily in formal methods.

Thank you to the many mentors I have had throughout my time as a student. In particular, Osbert Bastani has taught me a lot about reinforcement learning and has always been available to discuss research ideas. I am thankful to Suguman Bansal for providing guidance and helpful feedback regarding talks, statements, CV, etc. I have been fortunate to have had exposure to industry research through multiple internships. I want to thank my internship mentors, Andrew Gacek, Lee Pike, Matthew Andrews, Jeongran Lee, Lorenzo Maggi, Parminder Bhatia, Siddhartha Jain, and Baishakhi Ray. A special thanks to my undergraduate mentors, B. Srivathsan, Madhavan Mukund, and S. Praveen who encouraged me to pursue a research career and to apply to Ph.D. programs.

The research presented in this thesis would not have been possible without my amazing collaborators, including Rajeev, Suguman, Osbert, Steve Hsu, Radoslav Ivanov, and Shaan Vaidya. I also want to thank my collaborators in other projects (not presented in this thesis) from whom I learned a lot, including Yu Chen, Sajeev Khanna, Mateo Perez, Ashutosh Trivedi, and Fabio Somenzi.

I am grateful to family and friends for providing me with the support I needed during my time as a Ph.D. student. I would like to thank my parents, Jothimurugan and Megala, for supporting my decision to move far away from home to pursue graduate studies. Thank you to my uncle, Mahesh Arumugam, and his family for welcoming me with open arms whenever I felt the need to be around family. I am fortunate to have friends who made my life as a Ph.D. student very enjoyable. A special thanks to Raksha, for being my pillar of support for many years. Thank you to Caleb, for not only being a friend but also a mentor, Konstantinos, Filip, Aalok, Bharath, Vishal, and Pradeep.

ABSTRACT

SPECIFICATION-GUIDED REINFORCEMENT LEARNING

Kishor Jothimurugan

Rajeev Alur

Recent advances in Reinforcement Learning (RL) have enabled data-driven controller design for autonomous systems such as robotic arms and self-driving cars. Applying RL to such a system typically involves encoding the objective using a reward function (mapping transitions of the system to real values) and then training a neural network controller (from simulations of the system) to maximize the expected reward. However, many challenges arise when we try to train controllers to perform complex long-horizon tasks—e.g., navigating a car along a complex track with multiple turns. Firstly, it is quite challenging to manually define well-shaped reward functions for such tasks. It is much more natural to use a high-level specification language such as Linear Temporal Logic (LTL) to specify these tasks. Secondly, existing algorithms for learning controllers from logical specifications do not scale well to complex tasks due to a number of reasons including the use of sparse rewards and lack of compositionality. Furthermore, existing algorithms for verifying neural network controllers (trained using RL) cannot be easily applied to verify controllers for complex long-horizon tasks due to large approximation errors.

This thesis proposes novel techniques to overcome these challenges. We show that there are inherent limitations in obtaining theoretical guarantees regarding RL algorithms for learning controllers from temporal specifications. We then preset compositional RL algorithms that achieve state-of-the-art performance in practice by leveraging the structure in the given logical specification. Finally, we show that compositional approaches to learning enable faster verification of learned controllers containing neural network components.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

Reinforcement Learning (RL) has recently been applied to solve challenging robotics control problems, including multi-agent control [94], object manipulation [19], and control from perception [100]. The ability to train a controller without having access to an explicit model of the underlying system makes RL a practical choice for such applications. RL algorithms assume that the underlying system can be represented by a Markov Decision Process (MDP) whose transition probabilities are unknown. Furthermore, they also require a reward function assigning rewards (scalars) to transitions of the MDP. The goal of an RL algorithm is to compute a policy (mapping states of the system to actions) that maximizes the expected aggregate reward over runs of the system generated by the policy. Since the transition probabilities are unknown, the algorithm has to infer a near-optimal policy using samples collected by interacting with the system using an exploration policy—eg., a policy that chooses actions at random.

Recent research has primarily focused on scaling RL to high-dimensional control systems with complex dynamics such as autonomous cars [80], robotic arms [7] and hot-air balloons [25]. In most of these applications, the tasks considered are relatively simple—e.g., reachability and safety. Most RL algorithms, however, do not scale well to complex long-horizon tasks such as navigating a car through a series of turns or controlling a robotic arm to pick and place multiple objects. This is due to a number of drawbacks of existing techniques.

One key shortcoming is that the user must manually encode the task using a real-valued reward function, which can be challenging for several reasons. First, for complex tasks with multiple objectives and constraints, the user must manually devise a single reward function that balances different parts of the task. Second, the state space must often be extended to encode non-Markovian tasks—e.g., adding indicators that keep track of which subtasks have been completed. Third, oftentimes, different reward functions can encode the same task, and the choice of reward function can have a large impact on the convergence of the RL algorithm. Poor reward functions can make it

hard for the RL algorithm to achieve the objective; for instance, it can result in reward hacking [15], where the agent learns to optimize rewards without achieving the objective.

This has motivated many researchers to develop RL algorithms to train policies using formal specifications instead of rewards [8, 49, 29, 39, 67, 66, 158, 61, 156, 82, 102, 76, 28, 145]. Most algorithms follow a common high-level approach which is to (1) translate the specification to an automaton that accepts executions that satisfy the specification, (2) define an MDP that is the product of the MDP being controlled and the specification automaton, (3) associate rewards with the transitions of the product MDP so that the expected aggregate reward (roughly) captures probability of acceptance by the automaton, and (4) apply an off-the-shelf RL algorithm to synthesize a near-optimal policy. Some direct approaches without the use of rewards have also been proposed [49]; however, they are not applicable in settings with continuous state spaces.

Another major drawback of existing algorithms is that the number of samples required to train a policy increases rapidly with increase in the size of the specification. This is due to a few factors including the use of sparse rewards—i.e., the agent is given a non-zero reward only upon either completing the full task or reaching an undesirable state. Furthermore, in continuous state systems, most approaches train a single neural network policy for the entire task in an end-to-end fashion. In principle, one could leverage the compositional structure in the specification to decompose a long-horizon task into multiple short-horizon ones as is done in hierarchical RL [116, 115].

Another important shortcoming of existing algorithms for learning from formal specifications is that they lack strong theoretical guarantees regarding convergence to an optimal policy. Traditionally, RL algorithms for maximizing expected (discounted) reward either have a Probably Approximately Correct (PAC) guarantee or converge to an optimal policy in the limit almost surely. Such guarantees for logical specifications are only provided under the assumption that some additional information about the transition probabilities (e.g., the smallest non-zero probability) is known to the learner [28, 61, 49]. Additionally, there are subtle but important variations in the guarantees provided by different approaches—e.g., some only consider finite executions with a known time horizon [8] and some provide convergence guarantees only when the optimal policy satisfies the specification almost

surely.

To summarize, some of the main challenges in scaling RL techniques to complex long-horizon tasks are as follows.

- *Unsuitability of transition-based rewards as specifications.* Most state-of-the-art RL methods use reward signals as the only way to specify the desired outcome. For a complex task, it is often preferable to use a logical specification such as a Linear Temporal Logic (LTL) formula to describe the task.

- *Sparse rewards.* The performance of RL algorithms depend greatly on the rewards provided and it is, in general, impractical to train policies to perform complex tasks using sparse rewards. However, most existing algorithms for training policies from logical specifications generate sparse rewards.

- *Lack of compositionality.* Existing approaches train monolithic policies without utilizing the structure in the specification during training. This leads to high sample complexity as well as poor generalization of the learnt policy to new tasks.

- *Lack of a common theoretical framework.* Different approaches for learning from logical specifications have different kinds of guarantees and it is difficult to compare and contrast them. For example, multiple translations from logical specifications to rewards have been proposed; however, there is no formal notion of a reduction in the RL setting defining which translations are meaningful.

- *Scalability of verification.* Recent work [80] has shown that it is possible to verify the safety of neural network controllers in closed loop systems for short horizons. However, such techniques do not scale to long horizons rendering applications of RL to safety-critical systems impractical.

In this thesis, we propose new techniques to overcome the above-mentioned challenges.

1.1. Contributions

This study is primarily focused on *using logical specifications to specify RL tasks and learning policies to perform such tasks without requiring additional user-defined reward functions.* We first study theoretical limitations of RL from logical specifications, formalizing two notions of reductions between RL task specifications along the way. Then, we design a simple specification language, called SPECTRL, for specifying robotics tasks. We present compositional algorithms for learning policies from SPECTRL specifications, both in single- and multi-agent scenarios. We show that our algorithms can be used to train policies to perform complex tasks using much fewer samples than existing approaches. Finally, we demonstrate that introducing logical compositionality in RL tasks helps with formally verifying the safety of neural network controllers. The main contributions of this thesis are outlined below.

- *Theoretical considerations in the infinite-horizon setting.* We show that optimality preserving translations of LTL specifications to discounted rewards do not exist. We define a general notion of a *sampling-based* reduction which provides a unified framework to understand existing work on RL from LTL specifications. We also show that there does not exist a probably-approximately correct (PAC) RL algorithm for LTL specifications; in particular, for safety and reachability specifications.

- *Systematic generation of shaped rewards.* Using our simple specification language, SPECTRL, we show that it is possible to automatically generate a well-shaped reward function for a given task as well as for each subtask inferred from the specification.

- *Compositional RL algorithm.* We design an RL algorithm that leverages the structure in the given specification to improve learning. Unlike existing hierarchical methods which only provide recursive optimality guarantees, we justify the overall maximization objective of our algorithm by showing that it is a lower bound on the probability of satisfying the given specification. We empirically demonstrate that our algorithm can be used to train policies to perform complex tasks using fewer samples than existing approaches.

- *Multi-agent RL algorithm in the competitive setting.* We formalize the RL problem in the multi-agent setting where each agent has its own task specification. Here, instead of computing an arbitrary Nash equilibrium, we aim to compute one with high social welfare. We provide a natural extension of our compositional RL algorithm to this setting and show that the learnt policy is guaranteed to be an ε-Nash equilibrium with high probability.

- *Compositional verification.* We demonstrate that decomposing a long-horizon task into multiple short-horizon subtasks enables faster verification. We also provide a synthesis algorithm to automatically generate pre- and post-conditions corresponding to the subtasks. We train and verify a controller for the F1/10^{th} car [80] that works for all tracks of a certain kind.

1.2. Overview

We begin with the necessary background in Chapter 2. In Chapter 3, we first study *specification translations* in the context of RL in which one specification is transformed into another. We show that certain translations from LTL specifications to discounted rewards are impossible. This motivates the need for a more general notion of reductions in the RL setting. We then formalize the notion of a *sampling-based reduction* in which one is also allowed to modify the underlying MDP along with the specification; however, it should be possible to collect samples from the new MDP by interacting with the system corresponding to the original MDP. We then present some existing results in this framework. Finally, we conclude by showing that there does not exist a PAC algorithm [139] for safety and reachability specifications.

In Chapter 4, we introduce SPECTRL, a simple specification language based on a subclass of G-free LTL [137]. We then present an algorithm to generate an automaton model called *task monitor* from a given SPECTRL specification and show that the structure of the task monitor can be used to define a compositional neural network architecture for the control policy. In a SPECTRL specification, each predicate is associated with a quantitative semantics which makes it possible to obtain a well-shaped reward function for each subtask. In the finite-horizon setting, we also provide a method to automatically obtain a reward function for the full task from the task monitor which can be used in conjunction with existing RL algorithms to train policies from SPECTRL specifications.

In Chapter 5, we propose a compositional RL algorithm that interleaves high-level planning with RL for low-level control. We first reduce the problem of learning a policy to maximize the probability of satisfying a SPECTRL specification to an *abstract reachability* problem defined over a Directed Acyclic Graph (DAG); intuitively, vertices and edges of the graph correspond to regions of the state space and simpler sub-tasks, respectively. We then provide an algorithm to solve the abstract reachability problem that incorporates reinforcement learning to learn neural network policies for each edge (sub-task) within a Dijkstra-style planning algorithm to compute a high-level plan in the graph.

In Chapter 6, we extend the algorithm presented in Chapter 5 to the multi-agent setting. Specifically, we consider the competitive setting in which each agent has an individual objective given by a SPECTRL specification. The goal is to compute a joint policy that is in an ε-Nash equilibrium while also achieving high social welfare. We provide an enumerate-and-verify framework in which one first uses heuristics to enumerate candidate policies in decreasing value of social welfare. Then, given a candidate policy, a stochastic verification algorithm is used to check if a joint policy can be constructed such that (1) it is an ε-Nash equilibrium and (2) has the same behaviour as the candidate policy.

In Chapter 7, we show that compositionality helps with verification. Intuitively, policies corresponding to subtasks can be individually verified since the subtasks are completed within a much shorter time horizon than the overall task. In this chapter, we study a more general setting in which the environment adversely selects a sequence of subtasks; this setting is useful when one requires guarantees regarding worst-case performance across multiple tasks (where a task is a sequence of subtasks). We present a synthesis algorithm for synthesizing pre- and post-conditions for the individual subtasks from simulations of the system. We then show that the problem of verifying safety for the entire task duration (possibly infinite) can be decomposed into simpler problems each of which can be solved using existing techniques.

Finally, we discuss some possible directions for future work in Chapter 8. The main content of this dissertation is based on primary references [13, 81, 84, 85, 87].

1.3. Related Work

In this section, we give an overview of recent research in areas broadly related to the topic of this thesis. At the end of each chapter, we also discuss works more closely related to the technical contents of the chapter.

1.3.1. Reinforcement Learning from Temporal Specifications

There has been a lot of recent interest in designing reinforcement learning algorithms for learning policies from temporal specifications. Research in this direction can be classified into two categories depending on whether the specification describes (i) the desired behavior of the system over a fixed finite duration or (ii) the entire infinite-time behavior.

Fixed horizon. Aksaray et al. [8] propose an RL algorithm for a fragment of Signal Temporal Logic (STL) in which an approximate semantics of STL is used to assign rewards to (finite) trajectories. Li et al. [102] use a quantitative semantics of LTL (over finite horizon) to assign continuous (non-sparse) rewards to trajectories of the system; an algorithm to optimize the expected value of this semantics is proposed in [103]. Camacho et al. [32] propose a reduction from deterministic finite automata (DFA) to sparse rewards which are then shaped using a potential-based reward shaping mechanism; such a reduction can be used in conjunction with existing reductions from formal specifications to DFAs.

Infinite horizon. Some recent works [66, 67, 158, 28, 61, 62] propose reductions from LTL specifications to rewards which proceed by first constructing a product of the MDP with a Limit Deterministic Büchi automaton (LDBA) $\mathcal{A}_\varphi$ [137] derived from the LTL formula φ and then generate transition-based rewards in the product MDP. There is work on similar reductions for more complex lexicographic ω-regular objectives [64] as well as in the context of stochastic games [63]. These approaches, however, produce sparse rewards which are ill-suited for training policies to perform complex tasks. Furthermore, they either lack convergence guarantees or only provide weak guarantees—e.g., under the assumption that a lower bound on non-zero transition probabilities of the MDP is known [28, 61, 62]. A recent paper [49] proposes a Probably Approximately Correct

(PAC) algorithm for LTL specifications under the assumption that the structure of the MDP (transitions with non-zero probability) is known; this is an adaptation of the R-MAX [30] algorithm for limit-average rewards.

There has also been work on reward generation and reinforcement learning from STL specifications [22, 90]. These methods either use a quantitative semantics of STL to assign shaped rewards [22] or a model-based approach in conjunction with a planning algorithm such as model predictive control [90]. These approaches lack strong theoretical guarantees and their scalability with respect to complexity in the task specification has not been studied.

Multi-task Learning. In multi-task learning, the goal is to train a policy to perform multiple tasks, preferably without additional task-specific training or fine-tuning. Kuo et al. [96] and Vaezipoor et al. [145] propose frameworks for multi-task learning using LTL specifications; however, such approaches use sparse rewards and require a lot of samples even for relatively simple environments and tasks. In a recent work [88], we propose a compositional approach for multi-task generalization in which subtask policies are trained in a robust way to enable them to be composed sequentially (in any order).

Multi-agent RL. There has also been work on using temporal logic specifications for multi-agent RL [65, 118]; these approaches focus primarily on cooperative scenarios in which there is a common objective that all agents are trying to achieve.

We empirically compare our methods to some of the above-mentioned approaches in our experiments (see Sections 4.5, 5.4 and 6.7).

1.3.2. Policy Synthesis from Temporal Specifications

There has been a lot of work on algorithms for synthesizing an optimal policy for a given MDP and a temporal specification under the assumption that the transition probabilities are known; see [21] for a survey. These techniques, however, can only be applied to finite-state systems and furthermore require an *exact* model of the environment.

There is also work on practical approaches for synthesizing controllers for cyber-physical systems from temporal specifications under the assumption that a model of the system dynamics is available [99, 53, 113, 105, 133, 123, 60]. Lahijanian et al. [99] and Garg and Panagou [53] propose algorithms for robotic motion planning from temporal logic specifications. Moarref and Kress-Gazit [113] propose algorithms for automated synthesis of decentralized controllers to satisfy a common temporal objective in the context of multi-agent systems. In [133], the authors extend STL to incorporate stochastic properties and provide an algorithm to synthesize controllers from such specifications. Quantitative semantics of STL has also been used to encode STL objectives into model predictive control optimization problems for synthesizing controllers for robots [123, 60].

1.3.3. Structured Rewards

There has been research on models for expressing non-Markovian reward functions that enable the user to encode complex tasks. Brafman et al. [29] and De Giacomo et al. [39] consider reward functions which depend on whether or not the current history of the system satisfies certain LTL_f specifications. Icarte et al. [76] propose an automaton-based model called *reward machine* (RM) for high-level task specification and decomposition as well as an RL algorithm (QRM) that exploits this structure. In a later paper [77], they propose variants of QRM including an hierarchical RL algorithm (HRM) to learn policies for tasks specified using RMs. Camacho et al. [32] show that one can generate RMs from temporal specifications but RMs generated this way lead to sparse rewards.

1.3.4. Hierarchical Reinforcement Learning

Several reinforcement learning approaches have been developed where the controller has a hierarchical structure, similar to the ones proposed in this thesis [140, 115, 116]. In these methods, a high-level controller/planner decides on the next high-level action and selects a low-level controller to implement the specific actuator commands. By abstracting away details of the low-level dynamics, the high-level policy can efficiently plan over much longer time horizons. However, research has, for the most part, focused only on simple specifications—e.g., reachability of far away and hard-to-reach goals. See [125] for a recent survey on hierarchical methods in RL.

1.3.5. Safe Reinforcement Learning

Formal specifications are often used to express safety properties that a system *must* have—e.g., an autonomous car should never collide with another car. There has been a lot of research on deveoloping RL algorithms to train policies that satisfy such safety properties in addition to maximizing rewards. For instance, researchers have combined reinforcement learning and safe control via shielding, where a safe controller overrides the neural network (NN) controller when a control action is deemed unsafe [51, 149, 159, 101]. Such an architecture can be naturally augmented with a hierarchical controller, so as to enable performing more complex control tasks [51]. There has also been work on safe exploration [114, 5, 26] to enforce safety during training; however, these techniques typically only scale to finite or low-dimensional state spaces. A recent review on safe RL methods can be found in [59]; see [52] for a slightly dated but comprehensive survey on this topic.

1.3.6. Interpretable Reinforcement Learning

There has been some recent work on using formal methods to learn and verify interpretable policies. One technique involves synthesizing a symbolic policy to imitate the NN policy trained using RL—e.g., a program in a domain specific language [147] or a decision tree [24]. Verma et al. [148] propose an algorithm that uses a form of mirror descent to train (interpretable) programmatic policies. A (broad) survey on interpretable reinforcement learning can be found in [56]. The scalability of such approaches to complex long-horizon tasks, however, remains a challenge.

1.3.7. Neural Network Verification

Existing research on NN verification can be broadly classified into two categories depending on whether the focus is on verifying input-output properties of NNs or on verifying properties of closed-loop systems with NN components.

Functional properties. Many techniques for verifying general input-output (IO) properties of NNs have been proposed including approaches based on satisfiability modulo theories (SMT) [43, 91, 75], abstract interpretation [55] and mixed-integer linear programming (MILP) [41]. A key focus has been on verifying a special IO property called adversarial robustness [142, 57, 23]—e.g., by casting

the problem into a semi-definite program (SDP) [130], a relaxed linear program [155, 151], or a reachability problem [150, 16]. Alternatively, abstraction techniques based on computing Lipschitz constant bounds [47] have been developed.

Closed-loop systems. More closely related to this thesis is the research on verifying NN-based controllers in closed-loop systems. The first class of approaches [124, 117] employ assume-guarantee reasoning such that if the NN component satisfies a given IO property (as verified using an NN verification tool [91]), then the closed-loop system is safe as well. The challenge with these methods is that in general, it is challenging to reduce a closed-loop property to an IO property for the NN (essentially, this problem is equivalent to synthesizing a loop invariant). Thus, these methods only apply when such a reduction is available.

Alternatively, researchers have developed methods to directly reason about the closed-loop system by adapting control & hybrid system reachability methods. In particular, several techniques have been developed to analyze closed-loop systems with NN controllers [79, 74, 42, 141, 144]. These works combine ideas from NN verification with standard hybrid automaton verification tools [36, 95]—e.g., by transforming the NN into an equivalent hybrid system [79], approximating it with a polynomial with error bounds [42], or using other set representations such as star sets [144]. A key drawback of these techniques is that they do not scale well to long horizons.

CHAPTER 2

Background

In this chapter, we introduce some basic concepts in reinforcement learning necessary for understanding the rest of the thesis. We begin with the definition of a Markov Decision Process (MDP) which models an interactive stochastic environment. We then define various kinds of specifications and the corresponding optimization objectives. Finally, we define what a learning algorithm is and what kinds of convergence guarantees are desired.

2.1. Markov Decision Processes

A finite Markov Decision Process (MDP) is a tuple $\mathcal{M} = (S, A, s_0, P)$, where S is a finite set of states, s_0 is the initial state,[1] A is a finite set of actions, and $P : S \times A \times S \to [0, 1]$ is the transition probability function, with $\sum_{s' \in S} P(s, a, s') = 1$ for all $s \in S$ and $a \in A$.

An *infinite run (or trajectory)* $\zeta \in (S \times A)^\omega$ is a sequence $\zeta = s_0 a_0 s_1 a_1 \ldots$, where $s_i \in S$ and $a_i \in A$ for all $i \in \mathbb{N}$. Similarly, a *finite run* $\zeta \in (S \times A)^* \times S$ is a finite sequence $\zeta = s_0 a_0 s_1 a_1 \ldots a_{t-1} s_t$. For any run ζ of length at least j and any $i \leq j$, we let $\zeta_{i:j}$ denote the subsequence $s_i a_i s_{i+1} a_{i+1} \ldots a_{j-1} s_j$. We use $\mathcal{Z}(S, A) = (S \times A)^\omega$ and $\mathcal{Z}_f(S, A) = (S \times A)^* \times S$ to denote the set of infinite and finite runs, respectively. We omit (S, A) and simply use $\mathcal{Z}$ and $\mathcal{Z}_f$ when it is clear from context.

Let $\mathcal{D}(A) = \{\Delta : A \to [0, 1] \mid \sum_{a \in A} \Delta(a) = 1\}$ denote the set of all distributions over actions. A policy $\pi : \mathcal{Z}_f(S, A) \to \mathcal{D}(A)$ maps a finite run $\zeta \in \mathcal{Z}_f(S, A)$ to a distribution $\pi(\zeta)$ over actions. We denote by $\Pi(S, A)$ the set of all such policies. A policy π is *positional* if $\pi(\zeta) = \pi(\zeta')$ for all $\zeta, \zeta' \in \mathcal{Z}_f(S, A)$ with $\texttt{last}(\zeta) = \texttt{last}(\zeta')$ where $\texttt{last}(\zeta)$ denotes the last state in the run ζ. For a positional policy π, we have $\pi(s) = \pi(\zeta)$ for all ζ with $\texttt{last}(\zeta) = s$ and the policy is simply a function $\pi : S \to \mathcal{D}(A)$. A policy π is *deterministic* if, for all finite runs $\zeta \in \mathcal{Z}_f(S, A)$, there is an action $a \in A$ with $\pi(\zeta)(a) = 1$. For a deterministic policy π, we use $\pi(\zeta)$ to also denote the action a that satisfies $\pi(\zeta)(a) = 1$ in which case $\pi : \mathcal{Z}_f(S, A) \to A$ maps finite runs to actions.

[1] A distribution η over initial states can be modeled by adding a new state s_0 from which taking any action leads to a state sampled from η.

Given a finite run $\zeta = s_0 a_0 \dots a_{t-1} s_t$, the *cylinder* of ζ, denoted by $\texttt{Cyl}(\zeta)$, is the set of all infinite runs starting with prefix ζ. Given an MDP $\mathcal{M}$ and a policy $\pi \in \Pi(S, A)$, we define the probability of the cylinder set by $\mathcal{D}_\pi^{\mathcal{M}}(\texttt{Cyl}(\zeta)) = \prod_{i=0}^{t-1} \pi(\zeta_{0:i})(a_i) P(s_i, a_i, s_{i+1})$. It is known that $\mathcal{D}_\pi^{\mathcal{M}}$ can be uniquely extended to a probability measure over the σ-algebra generated by all cylinder sets. We omit the superscript $\mathcal{M}$ when it is clear from context.

Continuous state systems. We can model continuous state systems by letting $S \subseteq \mathbb{R}^n$ and denoting by $P(s, a, s')$ the probability density function corresponding to the distribution over next states s' when action a is taken in state s—i.e., we have $P : S \times A \times S \to \mathbb{R}_{\geq 0}$ and $\int_S P(s, a, s') ds' = 1$ for all $s \in S$ and $a \in A$. Optionally, we can also have an infinite set of actions A.

Simulator. In reinforcement learning, the standard assumption is that the set of states S, the set of actions A, and the initial state s_0 are known but the transition probability function P is unknown. The learning algorithm has access to a simulator $\mathcal{S}$ which can be used to sample runs of the system $\zeta \sim \mathcal{D}_\pi^{\mathcal{M}}$ using any policy π. The simulator can also be the real system, such as a robot, that $\mathcal{M}$ represents. Internally, the simulator stores the current state of the MDP which is denoted by $\mathcal{S}.\texttt{state}$. It makes the following functions available to the learning algorithm.

$\mathcal{S}.\texttt{reset()}$: This function sets $\mathcal{S}.\texttt{state}$ to the initial state s_0.

$\mathcal{S}.\texttt{step}(a)$: Given as input an action a, this function samples a state $s' \in S$ according to the transition probability function P—e.g., if $\mathcal{M}$ is finite, the probability that a state s' is sampled is $P(s, a, s')$ where $s = \mathcal{S}.\texttt{state}$. It then updates $\mathcal{S}.\texttt{state}$ to the newly sampled state s' and returns s'.

2.2. Task Specification

In this section, we present many different ways in which one can specify the learning objective. We define a *reinforcement learning task* to be a pair $(\mathcal{M}, \varphi)$ where $\mathcal{M}$ is an MDP and φ is a specification for $\mathcal{M}$. In general, a specification φ for $\mathcal{M} = (S, A, s_0, P)$ defines a function $J_\varphi^{\mathcal{M}} : \Pi(S, A) \to \mathbb{R}$ and the reinforcement learning objective is to compute a policy π that maximizes $J_\varphi^{\mathcal{M}}(\pi)$. Let $\mathcal{J}^*(\mathcal{M}, \varphi) = \sup_\pi J_\varphi^{\mathcal{M}}(\pi)$ denote the maximum value of $J_\varphi^{\mathcal{M}}$. We let $\Pi_{\text{opt}}(\mathcal{M}, \varphi)$ denote the set of

all optimal policies in $\mathcal{M}$ w.r.t. φ—i.e., $\Pi_{\text{opt}}(\mathcal{M}, \varphi) = \{\pi \mid J_\varphi^{\mathcal{M}}(\pi) = \mathcal{J}^*(\mathcal{M}, \varphi)\}$. In many cases, it is sufficient to compute an ε-optimal policy $\tilde{\pi}$ with $J_\varphi^{\mathcal{M}}(\tilde{\pi}) \geq \mathcal{J}^*(\mathcal{M}, \varphi) - \varepsilon$; we let $\Pi_{\text{opt}}^{\varepsilon}(\mathcal{M}, \varphi)$ denote the set of all ε-optimal policies in $\mathcal{M}$ w.r.t. φ.

2.2.1. Rewards

The most common kind of specifications used in reinforcement learning is reward functions that map transitions in $\mathcal{M}$ to real values. We first define the more general *reward machines* and then define standard transition-based reward functions as a special case.

Reward Machines. Reward Machines [76] extend simple transition-based reward functions to history-dependent ones by using an automaton model. Formally, a reward machine for an MDP $\mathcal{M} = (S, A, s_0, P)$ is a tuple $\mathcal{R} = (U, u_0, \delta_u, \delta_r)$, where U is a finite set of states, u_0 is the initial state, $\delta_u : U \times S \to U$ is the state transition function, and $\delta_r : U \to [S \times A \times S \to \mathbb{R}]$ is the reward function. Given an infinite run $\zeta = s_0 a_0 s_1 a_1 \ldots$, we can construct an infinite sequence of reward machine states $\rho_{\mathcal{R}}(\zeta) = u_0 u_1, \ldots$ defined by $u_{i+1} = \delta_u(u_i, s_{i+1})$. Then, we can assign either a discounted-sum or a limit-average reward to ζ:

- *Discounted Sum.* Given a discount factor $\gamma \in (0, 1)$, the full specification is $\varphi = (\mathcal{R}, \gamma)$ and we have
$$\mathcal{R}_\gamma(\zeta) = \sum_{i=0}^{\infty} \gamma^i \delta_r(u_i)(s_i, a_i, s_{i+1}).$$
Though less standard, one can use different discount factors in different states of $\mathcal{M}$, in which case we have $\gamma : S \to (0, 1)$ and
$$\mathcal{R}_\gamma(\zeta) = \sum_{i=0}^{\infty} \Big(\prod_{j=0}^{i-1} \gamma(s_j) \Big) \delta_r(u_i)(s_i, a_i, s_{i+1}).$$
The value of a policy π is $J_\varphi^{\mathcal{M}}(\pi) = \mathbb{E}_{\zeta \sim \mathcal{D}_\pi^{\mathcal{M}}}[\mathcal{R}_\gamma(\zeta)]$.

- *Limit Average.* The specification is just a reward machine $\varphi = \mathcal{R}$. The t-step average reward of the run ζ is
$$\mathcal{R}_{\text{avg}}^t(\zeta) = \frac{1}{t} \sum_{i=0}^{t-1} \delta_r(u_i)(s_i, a_i, s_{i+1}).$$

The value of a policy π is $J_\varphi^{\mathcal{M}}(\pi) = \liminf_{t\to\infty} \mathbb{E}_{\zeta\sim\mathcal{D}_\pi^{\mathcal{M}}}[\mathcal{R}_{\text{avg}}^t(\zeta)]$.

A standard *transition-based reward function* R is simply a reward machine $\mathcal{R}$ with a single state u_0; in this case, we use $R(s,a,s')$ to denote $\delta_r(u_0)(s,a,s')$.

2.2.2. Abstract Specifications

The above specifications are defined w.r.t. a given set of states S and actions A, and can only be interpreted over MDPs with the same state and action spaces. In this section, we look at *abstract specifications*, which are defined independently of S and A. To achieve this, a common assumption is that there is a fixed set of propositions $\mathcal{P}$, and the simulator provides access to a labeling function $L : S \to 2^{\mathcal{P}}$ denoting which propositions are true in any given state. Given a run $\zeta = s_0 a_0 s_1 a_1 \ldots$, we let $L(\zeta)$ denote the corresponding sequence of labels $L(\zeta) = L(s_0)L(s_1)\ldots$. A *labeled MDP* is a tuple $\mathcal{M} = (S, A, s_0, P, L)$. When clear from context, we use MDPs to mean labeled MDPs.

Abstract Reward Machines. Reward machines can be adapted to the abstract setting quite naturally. An *abstract reward machine* (ARM) is similar to a reward machine except δ_u and δ_r are independent of S and A—i.e., $\delta_u : U \times 2^{\mathcal{P}} \to U$ and $\delta_r : U \to [2^{\mathcal{P}} \to \mathbb{R}]$. Given current ARM state u_i and next MDP state s_{i+1}, the next ARM state is given by $u_{i+1} = \delta_u(u_i, L(s_{i+1}))$, and the reward is given by $\delta_r(u_i)(L(s_{i+1}))$.

Languages. Formal languages can be used to specify qualitative properties about runs of the system. A language specification $\varphi = \mathcal{L} \subseteq (2^{\mathcal{P}})^\omega$ is a set of "desirable" sequences of labels. The value of a policy π is the probability of generating a sequence in $\mathcal{L}$—i.e.,

$$J_\varphi^{\mathcal{M}}(\pi) = \mathcal{D}_\pi^{\mathcal{M}}\big(\{\zeta \in \mathcal{Z}(S,A) \mid L(\zeta) \in \mathcal{L}\}\big).$$

Some common ways to define languages are as follows.

- *Reachability.* Given an accepting set of propositions $X \in 2^{\mathcal{P}}$, we have

$$\mathcal{L}_{\texttt{reach}}(X) = \{w \in (2^{\mathcal{P}})^\omega \mid \exists i.\ w_i \cap X \neq \emptyset\}.$$

- *Safety.* Given a safe set of propositions $X \in 2^{\mathcal{P}}$, we have

$$\mathcal{L}_{\texttt{safe}}(X) = \{w \in (2^{\mathcal{P}})^{\omega} \mid \forall i.\ w_i \subseteq X\}.$$

- *Linear Temporal Logic (LTL).* Linear Temporal Logic [127] over propositions $\mathcal{P}$ is defined by the grammar

$$\varphi := b \in \mathcal{P} \mid \varphi \vee \varphi \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi\ \mathcal{U}\ \varphi$$

where $\bigcirc$ denotes the "Next" operator and $\mathcal{U}$ denotes the "Until" operator. Given an LTL formula φ and an infinite word $w \in (2^{\mathcal{P}})^{\omega}$, we use $w \models \varphi$ to denote w satisfies φ and this relation is defined inductively as follows.

$w \models b$	iff $b \in w_0$
$w \models \varphi_1 \vee \varphi_2$	iff $w \models \varphi_1$ or $w \models \varphi_2$
$w \models \neg\varphi$	iff $w \not\models \varphi$
$w \models \bigcirc\varphi$	iff $w_{1:\infty} \models \varphi$
$w \models \varphi_1\ \mathcal{U}\ \varphi_2$	iff $\exists\ i, w_{i:\infty} \models \varphi_2$ and $\forall\ 0 \leq j < i, w_{j:\infty} \models \varphi_1$

where $b \in \mathcal{P}$ and $w_{i:\infty}$ denotes the suffix of w starting at position i. We use $\Diamond$ and $\Box$ to denote the derived "Eventually" and "Always" operators, respectively—i.e., $\Diamond\varphi = \texttt{true}\ \mathcal{U}\ \varphi$ and $\Box\varphi = \neg\Diamond\neg\varphi$. Given an LTL specification φ over propositions $\mathcal{P}$, we have $\mathcal{L}_{\text{LTL}}(\varphi) = \{w \in (2^{\mathcal{P}})^{\omega} \mid w \models \varphi\}$. When clear from context, we abuse notation to denote the language specification $\mathcal{L}_{\text{LTL}}(\varphi)$ simply using φ.

2.3. Reinforcement Learning Algorithms

A learning algorithm $\mathcal{A}$ is an iterative process that in each iteration (i) either resets the simulator or takes a step in $\mathcal{M}$ from the current state of the simulator, and (ii) outputs its current estimate of an optimal policy π. A learning algorithm $\mathcal{A}$ induces a random sequence of output policies $\{\pi_n\}_{n=1}^{\infty}$ where π_n is the policy output in the n^{th} iteration.

In the finite MDP setting, one can obtain different kinds of convergence guarantees for learning algorithms. First, we can have algorithms that converge in the limit almost surely.

Definition 2.1. *A learning algorithm $\mathcal{A}$ is said to converge in the limit for a class of specifications $\mathcal{C}$ if, for any RL task $(\mathcal{M}, \varphi)$ with $\varphi \in \mathcal{C}$,*

$$J_\varphi^{\mathcal{M}}(\pi_n) \to \mathcal{J}^*(\mathcal{M}, \varphi) \text{ as } n \to \infty \quad \text{almost surely.}$$

Q-learning [152] is an example of a learning algorithm for finite MDPs that converges in the limit for discounted-sum rewards. There are variants of Q-learning for limit-average rewards [4] which have been shown to converge in the limit under some additional assumptions on the MDP $\mathcal{M}$. Another kind of learning algorithms is the class of *Probably Approximately Correct* (PAC-MDP) [89] algorithms which is defined as follows.

Definition 2.2. *A learning algorithm $\mathcal{A}$ is said to be PAC-MDP for a class of specifications $\mathcal{C}$ if, there is a function h such that for any $p > 0$, $\varepsilon > 0$, and any RL task $(\mathcal{M}, \varphi)$ with $\mathcal{M} = (S, A, s_0, P)$ and $\varphi \in \mathcal{C}$, taking $N = h(|S|, |A|, |\varphi|, \frac{1}{p}, \frac{1}{\varepsilon})$, with probability at least $1 - p$, we have*

$$\left| \left\{ n \mid \pi_n \notin \Pi_{opt}^{\varepsilon}(\mathcal{M}, \varphi) \right\} \right| \leq N.$$

We say a PAC-MDP algorithm is *efficient* if the *sample complexity* function h is polynomial in $|S|, |A|, \frac{1}{p}$ and $\frac{1}{\varepsilon}$. There are many efficient PAC-MDP algorithms for discounted rewards [92, 139].

CHAPTER 3

Theoretical Framework and Hardness Results

In this chapter, we study some theoretical properties of reinforcement learning from logical specifications with an emphasis on Linear Temporal Logic (LTL). In particular, we propose a formal framework for defining transformations among RL tasks. Recall that an RL task consists of an MDP $\mathcal{M}$ together with a specification φ. We define *sampling-based reductions* to formalize the process of transforming one RL task $(\mathcal{M}, \varphi)$ into another $(\bar{\mathcal{M}}, \varphi')$. While the relationship between the transformed model $\bar{\mathcal{M}}$ and the original model $\mathcal{M}$ is inspired by the classical definitions of simulation maps over (probabilistic) transition systems, the main challenge is that the transition probabilities of $\bar{\mathcal{M}}$ cannot be directly defined in terms of the unknown transition probabilities of $\mathcal{M}$. Intuitively, the step-function to sample transitions of $\bar{\mathcal{M}}$ should be definable in terms of the step-function of $\mathcal{M}$ used as a black-box, and our formalization allows this.

The notion of reduction among RL tasks naturally leads to formalization of preservation of optimal policies, convergence, and robustness (that is, near-optimal policies for the new task get mapped to near-optimal policies for the original task). We use this framework to revisit existing results, fill in some gaps, and identify some open problems aimed at improving our theoretical understanding of RL from formal specifications.

Throughout this chapter, we restrict our attention to finite MDPs; however, many concepts and results can be naturally extended to the infinite-state setting as well. The rest of the chapter is organized as follows. In Section 3.1, we show that it is not possible to reduce certain LTL specifications to (discounted-sum) reward machines when the underlying MDP $\mathcal{M}$ is kept fixed. We then define sampling-based reductions and restate existing results using our framework. In Section 3.2, we introduce the notions of robust specifications and robust reductions, and show that robust sampling-based reductions do not exist for transforming safety (as well as reachability) specifications to discounted-sum rewards. Finally, we present our result on non-existence of RL algorithms with PAC guarantees for safety (and reachability) specifications in Section 3.3.

3.1. Reductions

There has been a lot of research on RL algorithms for reward-based specifications. The most common approach for language-based specifications is to transform the given specification into a reward function and then apply an existing RL algorithm that tries to maximize the expected reward. In such cases, it is important to ensure that maximizing the expected reward corresponds to maximizing the probability of satisfying the specification. In this section, we study such reductions and formalize a general notion of *sampling-based reductions* in the RL setting—i.e., the transition probabilities are unknown and only a simulator of $\mathcal{M}$ is available.

3.1.1. Specification Translations

We first consider the simplest form of reductions, which involves translating the given specification into another one. Given a specification φ for MDP $\mathcal{M} = (S, A, s_0, P, L)$ we want to construct another specification φ' such that for any $\pi \in \Pi_{\text{opt}}(\mathcal{M}, \varphi')$, we also have $\pi \in \Pi_{\text{opt}}(\mathcal{M}, \varphi)$. This ensures that φ' can be used to compute a policy that maximizes the objective of φ. Note that since the transition probabilities P are not known, the translation has to be independent of P and furthermore the above *optimality preservation* criterion must hold for all P.

Definition 3.1. *An optimality preserving specification translation is a computable function $\mathcal{F}$ that maps the tuple (S, A, s_0, L, φ) to a specification φ' such that for all transition probability functions P, letting $\mathcal{M} = (S, A, s_0, P, L)$, we have $\Pi_{opt}(\mathcal{M}, \varphi') \subseteq \Pi_{opt}(\mathcal{M}, \varphi)$.*

A first attempt at a reinforcement learning algorithm for language-based specifications is to translate the given specification to a reward machine (either discounted sum or limit average). However there are some limitations to this approach. First, we show that it is not possible to reduce reachability and safety objectives to reward machines with discounted-sum rewards.

Theorem 3.2. *Let $\mathcal{P} = \{b\}$ and $\varphi = \mathcal{L}_{\texttt{reach}}(\{b\})$. There exists S, A, s_0, L such that for any reward machine specification with a discount factor $\varphi' = (\mathcal{R}, \gamma)$, there is a transition probability function P such that for $\mathcal{M} = (S, A, s_0, P, L)$, we have $\Pi_{opt}(\mathcal{M}, \varphi') \not\subseteq \Pi_{opt}(\mathcal{M}, \varphi)$.*

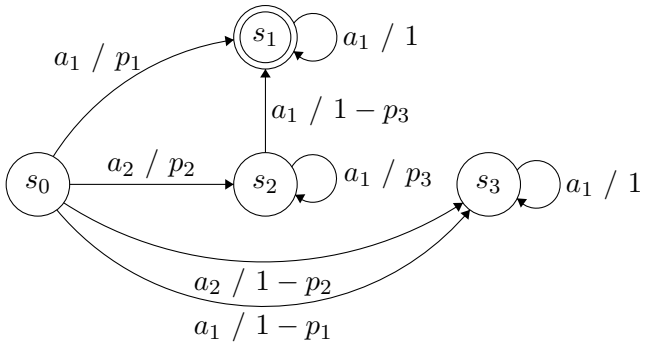


Figure 3.1: Counterexample for reducing reachability to discounted RM.

The main idea behind the proof is that one can make the transition probabilities small enough so that the expected time taken to reach the goal is large while maintaining an optimal probability of 1 for eventually reaching the goal. Using this idea, it is possible to define transition probabilities such that the expected reward w.r.t. an optimal policy is smaller than the expected reward obtained by a suboptimal policy.

Proof. Consider the MDP in Figure 3.1, which has states $S = \{s_0, s_1, s_2, s_3\}$, actions $A = \{a_1, a_2\}$, and labeling function L given by $L(s_1) = \{b\}$ (marked with double circles) and $L(s_0) = L(s_2) = L(s_3) = \emptyset$. Each edge denotes a state transition and is labeled by an action followed by the transition probability; the latter are parameterized by p_1, p_2, and p_3. At states s_1, s_2, and s_3 the only action available is a_1.[2] There are only two deterministic policies π_1 and π_2 in $\mathcal{M}$; π_1 always chooses a_1, whereas π_2 first chooses a_2 followed by a_1 afterwards.

For the sake of contradiction, suppose there is a $\varphi' = (\mathcal{R}, \gamma)$ that preserves optimality w.r.t. φ for all values of p_1, p_2, and p_3. WLOG, we assume that the rewards are normalized—i.e. $\delta_r : U \to [S \times A \times S \to [0, 1]]$. If $p_1 = p_2 = p_3 = 1$, then taking action a_1 in s_0 achieves reach probability of 1, whereas taking action a_2 in s_0 leads to a reach probability of 0. Hence, we must have that $\mathcal{R}_\gamma(s_0 a_1 (s_1 a_1)^\omega) \geq \mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^\omega) + \varepsilon$ for some $\varepsilon > 0$, as otherwise, π_2 maximizes $J^{\mathcal{M}}_{\varphi'}$ but does not maximize $J^{\mathcal{M}}_{\varphi}$.

For any finite run $\zeta \in \mathcal{Z}_f(S, A)$, let $\mathcal{R}_\gamma(\zeta)$ denote the finite discounted-sum reward of ζ. Let t be

[2]This can be modeled by adding an additional dead state that is reached upon taking action a_2 in these states.

such that $\frac{\gamma^t}{1-\gamma} \leq \frac{\varepsilon}{2}$. Then, for any $\zeta \in \mathcal{Z}(S, A)$, we have

$$\begin{aligned}\mathcal{R}_\gamma(s_0 a_1 (s_1 a_1)^\omega) &\geq \mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^\omega) + \varepsilon \\ &\geq \mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^t s_2) + \varepsilon \\ &\geq \mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^t s_2) + \frac{\gamma^t}{1-\gamma} + \frac{\varepsilon}{2}.\end{aligned}$$

Since $\lim_{p_3 \to 1} p_3^t = 1$, there exists $p_3 < 1$ such that $1 - p_3^t \leq \frac{\varepsilon}{8}(1-\gamma)$. Let $p_1 < 1$ be such that $p_1 \cdot \mathcal{R}_\gamma(s_0 a_1 (s_1 a_1)^\omega) \geq \mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^t s_2) + \frac{\gamma^t}{1-\gamma} + \frac{\varepsilon}{4}$ and let $p_2 = 1$. Then, we have

$$\begin{aligned}J_{\varphi'}^{\mathcal{M}}(\pi_1) &\geq p_1 \cdot \mathcal{R}_\gamma(s_0 a_1 (s_1 a_1)^\omega) \\ &\geq \mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^t s_2) + \frac{\gamma^t}{1-\gamma} + \frac{\varepsilon}{4} \\ &\geq p_3^t \cdot \left(\mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^t s_2) + \frac{\gamma^t}{1-\gamma}\right) + \frac{\varepsilon}{4} \\ &\geq p_3^t \cdot \left(\mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^t s_2) + \frac{\gamma^t}{1-\gamma}\right) + (1 - p_3^t) \cdot \left(\frac{1}{1-\gamma}\right) + \frac{\varepsilon}{8} \\ &> J_{\varphi'}^{\mathcal{M}}(\pi_2),\end{aligned}$$

where the last inequality followed from the fact that when using π_2, the system stays in state s_2 for at least t steps with probability p_3^t, and the reward of such trajectories is bounded from above by $\mathcal{R}_\gamma(s_0 a_2 (s_2 a_1)^t s_2) + \frac{\gamma^t}{1-\gamma}$, along with the fact that the reward of any other trajectory is bounded by $\frac{1}{1-\gamma}$. This leads to a contradiction since π_1 maximizes $J_{\varphi'}^{\mathcal{M}}$ but $J_{\varphi}^{\mathcal{M}}(\pi_1) = p_1 < 1 = J_{\varphi}^{\mathcal{M}}(\pi_2)$. □

We do not use the fact that the reward machine is finite state in the proof; therefore, the above result applies to general non-Markovian reward functions of the form $R : \mathcal{Z}_f(S, A) \to [0, 1]$ with γ-discounted reward defined by $R_\gamma(\zeta) = \sum_{i=0}^{\infty} \gamma^i R(\zeta_{0:i})$. The proof can be easily modified to show the result for safety specifications as well.

The main challenge in translating to discounted-sum rewards is the fact that the rewards vanish over time and the overall reward depends primarily on the first few steps. This issue can be partly overcome by using limit-average rewards. In fact, we have the following theorem.

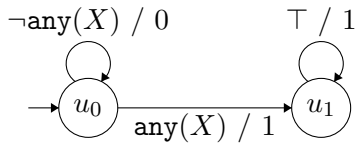


Figure 3.2: ARM for $\varphi = \mathcal{L}_{\texttt{reach}}(X)$.

Theorem 3.3. *There exists an optimality preserving specification translation from reachability and safety specifications to abstract reward machines (with limit-average aggregation).*

Proof. An abstract reward machine for the specification $\varphi = \mathcal{L}_{\texttt{reach}}(X)$ is shown in Figure 3.2. Each transition is labeled by a Boolean formula over $\mathcal{P}$ followed by the reward. We use $\texttt{any}(X)$ to denote $\bigvee_{b \in X} b$. It is easy to see that for any MDP $\mathcal{M}$ and any policy π of $\mathcal{M}$, we have $J^{\mathcal{M}}_{\mathcal{R}}(\pi) = J^{\mathcal{M}}_{\varphi}(\pi)$. An ARM for $\varphi = \mathcal{L}_{\texttt{safe}}(\mathcal{P} \setminus X)$ is obtained by replacing the reward value r by $1 - r$ on all transitions. □

However, we can show that there does not exist an ARM for the specification $\varphi = \mathcal{L}_{\text{LTL}}(\square\lozenge b)$, which requires the proposition b to be true infinitely often. Intuitively, the result follows from the fact that, given any ARM, we can construct an infinite word $w \in (2^{\mathcal{P}})^{\omega}$ in which b holds true rarely but infinitely often such that w achieves a lower limit-average reward than another word w' in which b holds true more frequently.

Theorem 3.4. *Let $\mathcal{P} = \{b\}$ and $\varphi = \mathcal{L}_{\text{LTL}}(\square\lozenge b)$. For any ARM specification $\varphi' = \mathcal{R}$ with limit-average rewards, there exists an MDP $\mathcal{M} = (S, A, s_0, P, L)$ such that $\Pi_{opt}(\mathcal{M}, \varphi') \not\subseteq \Pi_{opt}(\mathcal{M}, \varphi)$.*

Proof. For the sake of contradiction, let $\varphi' = \mathcal{R} = (U, u_0, \delta_u, \delta_r)$ be an ARM that preserves optimality w.r.t φ for all MDPs. The extended state transition function $\delta_u : U \times (2^{\mathcal{P}})^* \to U$ is defined naturally. WLOG, we assume that all states in $\mathcal{R}$ are reachable from the initial state and that the rewards are normalized—i.e., $\delta_r : U \to [2^{\mathcal{P}} \to [0, 1]]$.

A *cycle* in $\mathcal{R}$ is a sequence $C = u_1 \ell_1 u_2 \ell_2 \dots \ell_k u_{k+1}$ where $u_i \in U$, $\ell_i \in 2^{\mathcal{P}}$, $u_{i+1} = \delta_u(u_i, \ell_i)$ for all i, $u_{k+1} = u_1$, and the states $u_1, \dots, u_k$ are distinct. A cycle is *negative* if $\ell_i = \emptyset$ for all i, and *positive* if $\ell_i = \{b\}$ for all i. The average reward of a cycle C is given by $\mathcal{R}_{\text{avg}}(C) = \frac{1}{k} \sum_{i=1}^{k} \delta_r(u_i)(\ell_i)$. For any cycle $C = u_1 \ell_1 \dots \ell_k u_{k+1}$ we can construct a deterministic MDP $\mathcal{M}_C$ with a single action that first

generates a sequence of labels σ such that $\delta_u(u_0, \sigma) = u_1$, and then repeatedly generates the sequence of labels $\ell_1 \dots \ell_k$. The limit average reward of the only policy π in $\mathcal{M}_C$ is $J_{\mathcal{R}}^{\mathcal{M}_C}(\pi) = \mathcal{R}_{\text{avg}}(C)$ since the cycle C repeats indefinitely.

Now, given any positive cycle C_+ and any negative cycle C_-, we claim that $\mathcal{R}_{\text{avg}}(C_+) > \mathcal{R}_{\text{avg}}(C_-)$. To show this, consider an MDP $\mathcal{M}$ with two actions a_1 and a_2 such that taking action a_1 in the initial state s_0 leads to $\mathcal{M}_{C_+}$, and taking action a_2 in s_0 leads to $\mathcal{M}_{C_-}$. The policy π_1 that takes action a_1 in s_0 achieves a satisfaction probability of $J_{\varphi}^{\mathcal{M}}(\pi_1) = 1$, whereas the policy π_2 taking action a_2 in s_0 achieves $J_{\varphi}^{\mathcal{M}}(\pi_2) = 0$. Since $J_{\mathcal{R}}^{\mathcal{M}}(\pi_1) = \mathcal{R}_{\text{avg}}(C_+)$ and $J_{\mathcal{R}}^{\mathcal{M}}(\pi_2) = \mathcal{R}_{\text{avg}}(C_-)$, we must have that $\mathcal{R}_{\text{avg}}(C_+) > \mathcal{R}_{\text{avg}}(C_-)$ to preserve optimality w.r.t. φ. Since there are only finitely many cycles in $\mathcal{R}$, there exists an $\varepsilon > 0$ such that for any positive cycle C_+ and any negative cycle C_- we have $\mathcal{R}_{\text{avg}}(C_+) \geq \mathcal{R}_{\text{avg}}(C_-) + \varepsilon$.

Consider a bottom strongly connected component (SCC) of the graph of $\mathcal{R}$. We show that this component contains a negative cycle $C_- = u_1 \ell_1 \dots \ell_k u_{k+1}$ along with a second cycle $C = u'_1 \ell'_1 \dots \ell'_{k'} u'_{k'+1}$ such that $u'_1 = u_1$ and $\ell'_1 = \{b\}$. To construct C_-, from any state in the bottom SCC of $\mathcal{R}$, we can follow edges labeled $\ell = \emptyset$ until we repeat a state, and let this state be u_1. Then, to construct C, from $u'_1 = u_1$, we can follow the edge labeled $\ell'_1 = \{b\}$ to reach $u'_2 = \delta(u'_1, \ell'_1)$; since we are in the bottom SCC, there exists a path from u'_2 back to u'_1. Now, consider a sequence of the form $C_m = C_-^m C$, where $m \in \mathbb{N}$. We have

$$\begin{aligned}
\mathcal{R}_{\text{avg}}(C_m) &= \frac{mk\mathcal{R}_{\text{avg}}(C_-) + k'\mathcal{R}_{\text{avg}}(C)}{mk + k'} \\
&\leq \frac{mk}{mk + k'}\mathcal{R}_{\text{avg}}(C_-) + \frac{k'}{mk + k'} \\
&\leq \mathcal{R}_{\text{avg}}(C_-) + \frac{k'}{mk + k'}.
\end{aligned}$$

Let m be such that $\frac{k'}{mk+k'} \leq \frac{\varepsilon}{2}$ and C_+ be any positive cycle. Then, we have $\mathcal{R}_{\text{avg}}(C_+) \geq \mathcal{R}_{\text{avg}}(C_m) + \frac{\varepsilon}{2}$; therefore, there exists $p < 1$ such that $p \cdot \mathcal{R}_{\text{avg}}(C_+) \geq \mathcal{R}_{\text{avg}}(C_m) + \frac{\varepsilon}{4}$. Now, we can construct an MDP $\mathcal{M}$ in which (i) taking action a_1 in initial state s_0 leads to $\mathcal{M}_{C_+}$ with probability p, and to a dead state (where b does not hold) with probability $1 - p$, and (ii) taking action a_2 in initial state

s_0 leads to a deterministic single-action component that forces $\mathcal{R}$ to reach u_1 (recall that WLOG, all states in $\mathcal{R}$ are assumed to be reachable from u_0), and then generates the sequence of labels in C_m indefinitely. Let π_1 and π_2 be policies that select a_1 and a_2 in s_0, respectively. Then, we have

$$J_{\mathcal{R}}^{\mathcal{M}}(\pi_1) \geq p \cdot \mathcal{R}_{\text{avg}}(C_+) \geq \mathcal{R}_{\text{avg}}(C_m) + \frac{\varepsilon}{4} = J_{\mathcal{R}}^{\mathcal{M}}(\pi_2) + \frac{\varepsilon}{4}.$$

However $J_{\varphi}^{\mathcal{M}}(\pi_1) = p < 1 = J_{\varphi}^{\mathcal{M}}(\pi_2)$, which is a contradiction. □

Note that the above theorem only claims the non-existence of *abstract* reward machines for the LTL specification $\square\lozenge b$, whereas Theorem 3.2 holds for arbitrary reward machines and history dependent reward functions. Also, we do not rule out the possibility of a specification translation that constructs different ARMs (with limit-average rewards) for the same LTL objective depending on S, A, s_0 and L. This leads to the following natural question.

Open Problem 1. *Does there exist an optimality preserving specification translation from LTL specifications to reward machines with limit-average rewards?*

3.1.2. Sampling-based Reduction

The previous section suggests that keeping the MDP $\mathcal{M}$ fixed might be insufficient for reducing LTL specifications to reward-based ones. In this section, we formalize the notion of a *sampling-based reduction* where we are allowed to modify the MDP $\mathcal{M}$ in a way that makes it possible to simulate the modified MDP $\bar{\mathcal{M}}$ using a simulator for $\mathcal{M}$ without the knowledge of the transition probabilities of $\mathcal{M}$.

Given an RL task $(\mathcal{M}, \varphi)$ we want to construct another RL task $(\bar{\mathcal{M}}, \varphi')$ and a function f that maps policies in $\bar{\mathcal{M}}$ to policies in $\mathcal{M}$ such that for any policy $\bar{\pi} \in \Pi_{\text{opt}}(\bar{\mathcal{M}}, \varphi')$, we have $f(\bar{\pi}) \in \Pi_{\text{opt}}(\mathcal{M}, \varphi)$. Since it should be possible to simulate $\bar{\mathcal{M}}$ without the knowledge of the transition probability function P of $\mathcal{M}$, we impose several constraints on $\bar{\mathcal{M}}$.

Let $\mathcal{M} = (S, A, s_0, P, L)$ and $\bar{\mathcal{M}} = (\bar{S}, \bar{A}, \bar{s}_0, \bar{P}, \bar{L})$. First, it must be the case that $\bar{S}$, $\bar{A}$, $\bar{s}_0$, $\bar{L}$ and f are independent of P. Second, since the simulator of $\bar{\mathcal{M}}$ uses the simulator of $\mathcal{M}$ we can

assume that at any time, the state of the simulator of $\bar{\mathcal{M}}$ includes the state of the simulator of $\mathcal{M}$. Formally, there is a map $\beta : \bar{S} \to S$ such that for any $\bar{s}$, $\beta(\bar{s})$ is the state of $\mathcal{M}$ stored in $\bar{s}$. Since it is only possible to simulate $\mathcal{M}$ starting from s_0 we must have $\beta(\bar{s}_0) = s_0$. Next, when taking a step in $\bar{\mathcal{M}}$, a step in $\mathcal{M}$ may or may not occur, but the probability that a transition is sampled from $\mathcal{M}$ should be independent of P. Given these desired properties, we are ready to define a step-wise sampling-based reduction.

Definition 3.5. *A step-wise sampling-based reduction is a computable function $\mathcal{F}$ that maps the tuple (S, A, s_0, L, φ) to a tuple $(\bar{S}, \bar{A}, \bar{s}_0, \bar{L}, f, \beta, \alpha, q_1, q_2, \varphi')$ where $f : \Pi(\bar{S}, \bar{A}) \to \Pi(S, A)$, $\beta : \bar{S} \to S$, $\alpha : \bar{S} \times \bar{A} \to \mathcal{D}(A)$, $q_1 : \bar{S} \times \bar{A} \times \bar{S} \to [0, 1]$, $q_2 : \bar{S} \times \bar{A} \times A \times \bar{S} \to [0, 1]$ and φ' is a specification such that*

- $\beta(\bar{s}_0) = s_0$,
- $q_1(\bar{s}, \bar{a}, \bar{s}') = 0$ *if* $\beta(\bar{s}) \neq \beta(\bar{s}')$ *and,*
- *for any $\bar{s} \in \bar{S}$, $\bar{a} \in \bar{A}$, $a \in A$, and $s' \in S$ we have*

$$\sum_{\bar{s}' \in \beta^{-1}(s')} q_2(\bar{s}, \bar{a}, a, \bar{s}') = 1 - \sum_{\bar{s}' \in \bar{S}} q_1(\bar{s}, \bar{a}, \bar{s}'). \tag{3.1}$$

For any transition probability function $P : S \times A \times S \to [0, 1]$, the new transition probability function $\bar{P} : \bar{S} \times \bar{A} \times \bar{S} \to [0, 1]$ is defined by

$$\bar{P}(\bar{s}, \bar{a}, \bar{s}') = q_1(\bar{s}, \bar{a}, \bar{s}') + \mathbb{E}_{a \sim \alpha(\bar{s}, \bar{a})}[q_2(\bar{s}, \bar{a}, a, \bar{s}') P(\beta(\bar{s}), a, \beta(\bar{s}'))]. \tag{3.2}$$

In Equation 3.2, $q_1(\bar{s}, \bar{a}, \bar{s}')$ denotes the probability with which $\bar{\mathcal{M}}$ steps to $\bar{s}'$ without sampling a transition from $\mathcal{M}$. In the event that a step in $\mathcal{M}$ does occur, $\alpha(\bar{s}, \bar{a})(a)$ gives the probability of the action a taken in $\mathcal{M}$ and $q_2(\bar{s}, \bar{a}, a, \bar{s}')$ is the (unnormalized) probability with which $\bar{\mathcal{M}}$ transitions to $\bar{s}'$ given that action a in $\mathcal{M}$ caused a transition to $\beta(\bar{s}')$. It is easy to see that, for any P, $\bar{P}$ defined in Equation 3.2 is a valid transition probability function.

Lemma 3.6. *Given a step-wise sampling-based reduction $\mathcal{F}$, for any MDP $\mathcal{M} = (S, A, s_0, P, L)$ and specification φ, the function $\bar{P}$ defined by $\mathcal{F}$ is a valid transition probability function.*

Proof. It is easy to see that $\bar{P}(\bar{s}, \bar{a}, \bar{s}') \geq 0$ for all $\bar{s}, \bar{s}' \in \bar{S}$ and $\bar{a} \in \bar{A}$. Now for any $\bar{s} \in \bar{S}$ and $\bar{a} \in \bar{A}$, letting $\sum_{\bar{s}'} q_1(\bar{s}, \bar{a}, \bar{s}') = p(\bar{s}, \bar{a})$, we have

$$\begin{aligned}
\sum_{\bar{s}' \in \bar{S}} P(\bar{s}, \bar{a}, \bar{s}') &= p(\bar{s}, \bar{a}) + \sum_{\bar{s}' \in \bar{S}} \mathbb{E}_{a \sim \alpha(\bar{s}, \bar{a})}[q_2(\bar{s}, \bar{a}, a, \bar{s}') P(\beta(\bar{s}), a, \beta(\bar{s}'))] \\
&= p(\bar{s}, \bar{a}) + \mathbb{E}_{a \sim \alpha(\bar{s}, \bar{a})}\Big[\sum_{\bar{s}' \in \bar{S}} q_2(\bar{s}, \bar{a}, a, \bar{s}') P(\beta(\bar{s}), a, \beta(\bar{s}')) \Big] \\
&= p(\bar{s}, \bar{a}) + \mathbb{E}_{a \sim \alpha(\bar{s}, \bar{a})}\Big[\sum_{s' \in S} \sum_{\bar{s}' \in \beta^{-1}(s')} q_2(\bar{s}, \bar{a}, a, \bar{s}') P(\beta(\bar{s}), a, \beta(\bar{s}')) \Big] \\
&= p(\bar{s}, \bar{a}) + \mathbb{E}_{a \sim \alpha(\bar{s}, \bar{a})}\Big[\sum_{s' \in S} P(\beta(\bar{s}), a, s') \sum_{\bar{s}' \in \beta^{-1}(s')} q_2(\bar{s}, \bar{a}, a, \bar{s}')) \Big] \\
&= p(\bar{s}, \bar{a}) + \mathbb{E}_{a \sim \alpha(\bar{s}, \bar{a})}\Big[\sum_{s' \in S} P(\beta(\bar{s}), a, s')(1 - p(\bar{s}, \bar{a})) \Big] \\
&= 1,
\end{aligned}$$

where the penultimate step followed from Equation 3.1. □

Example 3.1. *A simple example of a step-wise sampling-based reduction is the product construction used to translate reward machines to regular reward functions [76]. Let $\mathcal{R} = (U, u_0, \delta_u, \delta_r)$. Then, we have $\bar{S} = S \times U$, $\bar{A} = A$, $\bar{s}_0 = (s_0, u_0)$, $\bar{L}(s, u) = L(s)$, $\beta(s, u) = s$, $\alpha(a)(a') = \mathbb{1}(a' = a)$, $q_1 = 0$, and $q_2((s, u), a, a', (s', u')) = \mathbb{1}(u' = \delta_u(u, s'))$. The specification φ' is a reward function given by $R((s, u), a, (s', u')) = \delta_r(u)(s, a, s')$, and $f(\bar{\pi})$ is a policy that keeps track of the reward machine state and acts according to $\bar{\pi}$.*

Given an MDP $\mathcal{M} = (S, A, s_0, P, L)$ and a specification φ, the reduction $\mathcal{F}$ defines a unique triplet $(\bar{\mathcal{M}}, \varphi', f)$ with $\bar{\mathcal{M}} = (\bar{S}, \bar{A}, \bar{s}_0, \bar{P}, \bar{L})$, where $\bar{S}, \bar{A}, \bar{s}_0, \bar{L}, f$ and φ' are obtained by applying $\mathcal{F}$ to (S, A, s_0, L, φ) and $\bar{P}$ is defined by Equation 3.2. We let $\mathcal{F}(\mathcal{M}, \varphi)$ denote the triplet $(\bar{\mathcal{M}}, \varphi', f)$. Given a simulator $\mathcal{S}$ of $\mathcal{M}$, we can construct a simulator $\bar{\mathcal{S}}$ of $\bar{\mathcal{M}}$ as follows.

Algorithm 1 Step function of the simulator $\bar{\mathcal{S}}$ of $\bar{\mathcal{M}}$ given β, α, q_1, q_2 and a simulator $\mathcal{S}$ of $\mathcal{M}$.

function $\bar{\mathcal{S}}$.`step`($\bar{a}$)
  $\bar{s} \leftarrow \bar{\mathcal{S}}$.`state`
  $p \leftarrow \sum_{\bar{s}'} q_1(\bar{s}, \bar{a}, \bar{s}')$
  $x \sim$ `Uniform`$(0, 1)$
  if $x \leq p$ **then**
    $\bar{\mathcal{S}}$.`state` $\leftarrow \bar{s}' \sim \dfrac{q_1(\bar{s}, \bar{a}, \bar{s}')}{p}$
  else
    $a \sim \alpha(\bar{s}, \bar{a})$
    $s' \leftarrow \mathcal{S}$.`step`$(a)$
    $\bar{\mathcal{S}}$.`state` $\leftarrow \bar{s}' \sim \dfrac{q_2(\bar{s}, \bar{a}, a, \bar{s}')\mathbb{1}(\beta(\bar{s}') = s')}{1 - p}$ {Ensures $\beta(\bar{s}') = s'$}
  end if
  return $\bar{\mathcal{S}}$.`state`
end function

$\bar{\mathcal{S}}$.`reset()`: This function internally sets the current state of the MDP to $\bar{s}_0$ and calls the reset function of $\mathcal{M}$.

$\bar{\mathcal{S}}$.`step`($\bar{a}$): This function is outlined in Algorithm 1. We use $\bar{s}' \sim \Delta(\bar{s}')$ to denote that $\bar{s}'$ is sampled from the distribution defined by Δ. It takes a step without calling $\mathcal{S}$.`step` with probability p. Otherwise, it samples an action a according to $\alpha(\bar{s}, \bar{a})$, calls $\mathcal{S}$.`step`(a) to get next state s' of $\mathcal{M}$ and then samples an $\bar{s}'$ satisfying $\beta(\bar{s}') = s'$ based on q_2. Equation 3.1 ensures that $\frac{q_2}{1-p}$ defines a valid distribution over $\beta^{-1}(s')$.

We call the reduction step-wise since at most one transition of $\mathcal{M}$ can occur during a transition of $\bar{\mathcal{M}}$. Under this assumption, we justify the general form of $\bar{P}$. Let $\bar{s}$ and $\bar{a}$ be fixed. Let $X_{\bar{S}}$ be a random variable denoting the next state in $\bar{\mathcal{M}}$ and X_A be a random variable denoting the action taking in $\mathcal{M}$ (it takes a dummy value $\perp \notin A$ when no step in $\mathcal{M}$ is taken). Then, for any $\bar{s}' \in \bar{S}$, we have

$$\Pr[X_{\bar{S}} = \bar{s}'] = \Pr[X_{\bar{S}} = \bar{s}' \wedge X_A = \perp] + \sum_{a \in A} \Pr[X_{\bar{S}} = \bar{s}' \wedge X_A = a].$$

Now, we have

$$\begin{aligned}
&\Pr[X_{\bar{S}} = \bar{s}' \wedge X_A = a] \\
&= \Pr[X_A = a] \Pr[X_{\bar{S}} = \bar{s}' \mid X_A = a] \\
&= \Pr[X_A = a] \Pr[\beta(X_{\bar{S}}) = \beta(\bar{s}') \mid X_A = a] \Pr[X_{\bar{S}} = \bar{s}' \mid X_A = a, \beta(X_{\bar{S}}) = \beta(\bar{s}')] \\
&= P(\beta(\bar{s}), a, \beta(\bar{s}')) \cdot \Pr[X_A = a] \Pr[X_{\bar{S}} = \bar{s}' \mid X_A = a, \beta(X_{\bar{S}}) = \beta(\bar{s}')].
\end{aligned}$$

Taking $q_1(\bar{s}, \bar{a}, \bar{s}') = \Pr[X_{\bar{S}} = \bar{s}' \wedge X_A = \bot]$, $\alpha(\bar{s}, \bar{a})(a) = \Pr[X_A = a] / \Pr[X_A \neq \bot]$, and $q_2(\bar{s}, \bar{a}, a, \bar{s}') = \Pr[X_{\bar{S}} = \bar{s}' \mid X_A = a, \beta(X_{\bar{S}}) = \beta(\bar{s}')] \cdot \Pr[X_A \neq \bot]$, we obtain the form of $\bar{P}$ in Definition 3.5. Note that Equation 3.1 holds since both sides evaluate to $\Pr[X_A \neq \bot]$.

To be precise, it is also possible to reset the MDP $\mathcal{M}$ to s_0 in the middle of a run of $\bar{\mathcal{M}}$. This can be modeled by taking $\alpha(\bar{s}, \bar{a})$ to be a distribution over $A \times \{0, 1\}$, where $(a, 0)$ represents taking action a in the current state $\beta(\bar{s})$ and $(a, 1)$ represents taking action a in s_0 after a reset. We would also have $q_2 : \bar{S} \times \bar{A} \times A \times \{0, 1\} \times \bar{S} \to [0, 1]$ and furthermore $q_1(\bar{s}, \bar{a}, \bar{s}')$ can be nonzero if $\beta(\bar{s}') = s_0$. For simplicity, we use Definition 3.5 without considering resets in $\mathcal{M}$ during a step of $\bar{\mathcal{M}}$. However, the discussions in the rest of the chapter apply to the general case as well. Now we define the *optimality preservation* criterion for sampling-based reductions.

Definition 3.7. *A step-wise sampling-based reduction $\mathcal{F}$ is optimality preserving if for any RL task $(\mathcal{M}, \varphi)$ letting $(\bar{\mathcal{M}}, \varphi', f) = \mathcal{F}(\mathcal{M}, \varphi)$ we have the property $f(\Pi_{opt}(\bar{\mathcal{M}}, \varphi')) \subseteq \Pi_{opt}(\mathcal{M}, \varphi)$ where $f(\Pi) = \{f(\pi) \mid \pi \in \Pi\}$ for a set of policies Π.*

It is easy to see that the reduction in Example 3.1 is optimality preserving for both discounted-sum and limit-average rewards since $J^{\bar{\mathcal{M}}}_{\varphi'}(\bar{\pi}) = J^{\mathcal{M}}_{\varphi}(f(\bar{\pi}))$ for any policy $\bar{\pi} \in \Pi(\bar{S}, \bar{A})$. Another interesting observation is that we can reduce discounted-sum rewards with multiple discount factors $\gamma : S \to]0, 1[$ to the usual case with a single discount factor.

Theorem 3.8. *There is an optimality preserving step-wise sampling-based reduction $\mathcal{F}$ such that for any $\mathcal{M} = (S, A, s_0, P)$ and $\varphi = (R, \gamma)$, where $R : S \times A \times S \to \mathbb{R}$ and $\gamma : S \to (0, 1)$, we have*

$f(\mathcal{M}, \varphi) = (\bar{\mathcal{M}}, \varphi', f)$, *where* $\varphi' = (R', \gamma')$, *with* $R' : \bar{S} \times \bar{A} \times \bar{S} \to \mathbb{R}$ *and* $\gamma' \in (0, 1)$.

Proof. Let $\bar{S} = S \sqcup \{s_\perp\}$, where $s_\perp$ is a new sink state, $\bar{A} = A$, and $\bar{s}_0 = s_0$. We set $\gamma' = \gamma_{\max} = \max_{s \in S} \gamma(s)$, and define R' by $R'(s, a, s') = \frac{\gamma_{\max}}{\gamma(s)} R(s, a, s')$ if $s, s' \in S$ and 0 otherwise. We define $\bar{P}(s_\perp, a, s_\perp) = 1$ for all $a \in A$. For any $s \in S$, we have $\bar{P}(s, a, s') = \frac{\gamma(s)}{\gamma_{\max}} P(s, a, s')$ if $s' \in S$ and $\bar{P}(s, a, s_\perp) = 1 - \frac{\gamma(s)}{\gamma_{\max}}$. Intuitively, $\bar{\mathcal{M}}$ transitions to the sink state $s_\perp$ with probability $1 - \frac{\gamma(s)}{\gamma_{\max}}$ from any state s on taking any action a which has the effect of reducing the discount factor from $\gamma_{\max}$ to $\gamma(s)$ in state s since all future rewards are 0 after transitioning to $s_\perp$. Although we explicitly defined $\bar{P}$, note that it has the general form of Equation 3.2 and can be sampled from without knowing P. Now, we take $\varphi' = (R', \gamma')$, and $f(\bar{\pi})$ to be $\bar{\pi}$ restricted to $\mathcal{Z}_f(S, A)$. It is easy to see that for any $\bar{\pi} \in \Pi(\bar{S}, A)$, we have $J^{\bar{\mathcal{M}}}_{\varphi'}(\bar{\pi}) = J^{\mathcal{M}}_{\varphi}(f(\bar{\pi}))$; therefore, this reduction preserves optimality. □

3.1.3. Reductions from Temporal Logic Specifications

A number of strategies have been recently proposed for learning policies from temporal specifications by reducing them to reward-based specifications. For instance, Hasanbeig et al. [66, 67] propose a reduction from LTL specifications to discounted rewards which proceeds by first constructing a product of the MDP $\mathcal{M}$ with a Limit Deterministic Büchi automaton (LDBA) $\mathcal{A}_\varphi$ derived from the LTL formula φ and then generates transition-based rewards in the product MDP. The strategy is to assign a fixed positive reward of r when an accepting state in $\mathcal{A}_\varphi$ is reached and 0 otherwise. As shown in [61], this strategy does not always preserve optimality if the discount factor γ is required to be strictly less that one. Similar approaches are proposed in [158, 32], though they do not provide optimality preservation guarantees.

A recent paper [61] presents a step-wise sampling-based reduction from LTL specifications to limit-average rewards. It first constructs an LDBA $\mathcal{A}_\varphi$ from the LTL formula φ and then considers a product $\mathcal{M} \otimes \mathcal{A}_\varphi$ of the MDP $\mathcal{M}$ with $\mathcal{A}_\varphi$ in which the nondeterminism of $\mathcal{A}_\varphi$ is handled by adding additional actions that represent the choice of possible transitions in $\mathcal{A}_\varphi$ that can be taken. Now, the reduced MDP $\bar{\mathcal{M}}$ is obtained by adding an additional sink state $\bar{s}_\perp$ with the property that whenever an accepting state of $\mathcal{A}_\varphi$ is reached in $\bar{\mathcal{M}}$, there is a $(1 - \lambda)$ probability of transitioning to $\bar{s}_\perp$ during

the next transition in $\bar{\mathcal{M}}$. They show that for a large enough value of λ, any policy maximizing the probability of reaching $\bar{s}_\perp$ in $\bar{\mathcal{M}}$ can be used to construct a policy that maximizes $J^{\mathcal{M}}_{\mathcal{L}_{\text{LTL}}(\varphi)}$. As shown before, this reachability property in $\bar{\mathcal{M}}$ can be translated to limit-average rewards. The main drawback of this approach is that the lower bound on λ for preserving optimality depends on the transition probability function P; hence, it is not possible to correctly pick the value of λ without the knowledge of P. A heuristic used in practice is to assign a default large value to λ. Their result can be summarized as follows.

Theorem 3.9 ([61]). *There is a family of step-wise sampling-based reductions $\{\mathcal{F}_\lambda\}_{\lambda\in(0,1)}$ such that for any MDP $\mathcal{M}$ and LTL specification φ, there exists a $\lambda_{\mathcal{M},\varphi} \in (0,1)$ such that for all $\lambda \geq \lambda_{\mathcal{M},\varphi}$, letting $(\bar{\mathcal{M}}_\lambda, \varphi'_\lambda, f_\lambda) = \mathcal{F}_\lambda(\mathcal{M}, \varphi)$, we have $f_\lambda(\Pi_{opt}(\bar{\mathcal{M}}_\lambda, \varphi'_\lambda)) \subseteq \Pi_{opt}(\mathcal{M}, \varphi)$ and $\varphi'_\lambda = R_\lambda : S \times A \times S \to \mathbb{R}$ is a limit-average reward specification.*

Hahn et al. [62] show that the above approach can be modified to get less sparse rewards with similar guarantees using two discount factors $\gamma_1 < 1$ and $\gamma_2 = 1$ (where $\gamma_2 = 1$ is only used in steps at which the reward is zero).

Another approach [28] with an optimality preservation guarantee reduces LTL specifications to discounted rewards with two discount factors $\gamma_1 < \gamma_2 < 1$ which are applied in different states. This approach uses the product $\mathcal{M} \times \mathcal{A}_\varphi$ as $\bar{\mathcal{M}}$ and assigns a reward of $1 - \gamma_1$ to the accepting states (where discount factor γ_1 is applied) and 0 to the remaining states (where discount factor γ_2 is applied). Applying Theorem 3.8 we get the following result as a corollary of the optimality preservation guarantee of this approach.

Theorem 3.10 ([28]). *There is a family of step-wise sampling-based reductions $\{\mathcal{F}_\gamma\}_{\gamma\in(0,1)}$ such that for any MDP $\mathcal{M}$ and LTL specification φ, there exists $\gamma_{\mathcal{M},\varphi} \in (0,1)$ such that for all $\gamma \geq \gamma_{\mathcal{M},\varphi}$, letting $(\bar{\mathcal{M}}_\gamma, \varphi'_\gamma, f_\gamma) = \mathcal{F}_\gamma(\mathcal{M}, \varphi)$, we have $f_\gamma(\Pi_{opt}(\bar{\mathcal{M}}_\gamma, \varphi'_\gamma)) \subseteq \Pi_{opt}(\mathcal{M}, \varphi)$ and $\varphi'_\gamma = (R_\gamma, \gamma)$ is a discounted-sum reward specification.*

Similar to [61], the optimality preservation guarantee only applies to large enough γ, and the lower bound on γ depends on the transition probability function P.

It is unknown if there exists an optimality preserving step-wise sampling-based reduction from LTL specifications to reward-based specifications that is completely independent of P.

Open Problem 2. *Does there exist an optimality preserving step-wise sampling-based reduction $\mathcal{F}$ such that for any RL task $(\mathcal{M}, \varphi)$ where φ is an LTL specification, letting $(\bar{\mathcal{M}}, \varphi', f) = \mathcal{F}(\mathcal{M}, \varphi)$, we have that φ' is a reward-based specification (either limit-average or discounted-sum)?*

3.2. Robustness

A key property of discounted reward specifications that is exploited by RL algorithms is *robustness*. In this section, we discuss the concept of robustness for specifications as well as reductions. We show that robust reductions from LTL specifications to discounted rewards are not possible due to the fact that LTL specifications are not robust.

3.2.1. Robust Specifications

A specification φ is said to be *robust* [108] if an optimal policy for φ in an estimate $\mathcal{M}'$ of the MDP $\mathcal{M}$ achieves close to optimal performance in $\mathcal{M}$. Formally, an MDP $\mathcal{M} = (S, A, s_0, P, L)$ is said to be δ-*close* to another MDP $\mathcal{M}' = (S, A, s_0, P', L)$ if their states, actions, initial states, and labeling functions are identical and their transition probabilities differ by at most a δ amount—i.e.,

$$|P(s, a, s') - P'(s, a, s')| \leq \delta$$

for all $s, s' \in S$ and $a \in A$.

Definition 3.11. *A specification φ is* robust *if for any MDP $\mathcal{M}$ for which φ is a valid specification and $\varepsilon > 0$, there exists a $\delta_{\mathcal{M},\varepsilon} > 0$ such that if MDP $\mathcal{M}'$ is $\delta_{\mathcal{M},\varepsilon}$-close to $\mathcal{M}$, then an optimal policy in $\mathcal{M}'$ is an ε-optimal policy in $\mathcal{M}$—i.e., $\Pi_{opt}(\mathcal{M}', \varphi) \subseteq \Pi^{\varepsilon}_{opt}(\mathcal{M}, \varphi)$.*

The simulation lemma in [92] proves that discounted-sum rewards are robust. On the other hand, [108] shows that language-based specifications, even safety specifications, are not robust. Here, we give a slightly modified example (that we will use later) to show that the specification $\varphi = \mathcal{L}_{\texttt{safe}}(\{b\})$ is not robust which also shows that limit-average rewards are not robust.

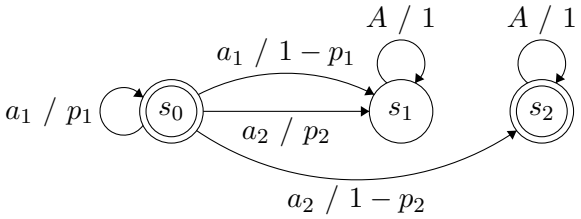


Figure 3.3: Example showing non-robustness of $\mathcal{L}_{\texttt{safe}}(\{b\})$.

Theorem 3.12 ([108])**.** *There exists an MDP $\mathcal{M}$ and a safety specification φ such that, for any $\delta > 0$, there is an MDP $\mathcal{M}_\delta$ that is δ-close to $\mathcal{M}$ which satisfies $\Pi_{opt}(\mathcal{M}_\delta, \varphi) \cap \Pi^{\varepsilon}_{opt}(\mathcal{M}, \varphi) = \emptyset$ for all $\varepsilon < 1$.*

Proof. Consider the MDP $\mathcal{M}$ in Figure 3.3 with $p_1 = p_2 = 1$; the double circles denote states where b holds. Then, an optimal policy for $\varphi = \mathcal{L}_{\texttt{safe}}(\{b\})$ always selects action a_1 and achieves a satisfaction probability of 1. Now let $\mathcal{M}_\delta$ denote the same MDP with $p_1 = p_2 = 1 - \delta$. Then, any optimal policy for φ in $\mathcal{M}_\delta$ must select a_2 almost surely, which is not optimal for $\mathcal{M}$. In fact, such a policy achieves a satisfaction probability of 0 in $\mathcal{M}$. Therefore, we have $\Pi_{\text{opt}}(\mathcal{M}_\delta, \varphi) \cap \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}, \varphi) = \emptyset$ for any $\delta > 0$ and any $\varepsilon < 1$. □

3.2.2. Robust Reductions

In our discussion of reductions, we were interested in optimality preserving sampling-based reductions mapping an RL task $(\mathcal{M}, \varphi)$ to another task $(\bar{\mathcal{M}}, \varphi')$. However, in the learning setting, if we use a PAC-MDP algorithm to compute a policy $\bar{\pi}$ for $(\bar{\mathcal{M}}, \varphi')$, it might be the case that $\bar{\pi} \notin \Pi_{\text{opt}}(\bar{\mathcal{M}}, \varphi')$. Therefore, we cannot conclude anything useful about the optimality of the corresponding policy $f(\bar{\pi})$ in $\mathcal{M}$ w.r.t. φ. Ideally, we would like to ensure that for any $\varepsilon > 0$ there is a $\varepsilon' > 0$ such that an ε'-optimal policy for $(\bar{\mathcal{M}}, \varphi')$ corresponds to an ε-optimal policy for $(\mathcal{M}, \varphi)$.

Definition 3.13. *A step-wise sampling-based reduction $\mathcal{F}$ is robust if for any RL task $(\mathcal{M}, \varphi)$ with $(\bar{\mathcal{M}}, \varphi', f) = \mathcal{F}(\mathcal{M}, \varphi)$ and any $\varepsilon > 0$, there is an $\varepsilon' > 0$ such that $f(\Pi^{\varepsilon'}_{opt}(\bar{\mathcal{M}}, \varphi')) \subseteq \Pi^{\varepsilon}_{opt}(\mathcal{M}, \varphi)$.*

Observe that for any optimal policy $\bar{\pi} \in \Pi_{\text{opt}}(\bar{\mathcal{M}}, \varphi')$ for $\bar{\mathcal{M}}$ and φ', we have $f(\bar{\pi}) \in \bigcap_{\varepsilon > 0} \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}, \varphi) = \Pi_{\text{opt}}(\mathcal{M}, \varphi)$; hence, a robust reduction is also optimality preserving. Although a robust reduction

is preferred when translating LTL specifications to discounted-sum rewards, the following theorem shows that such a reduction is not possible.

Theorem 3.14. *Let $\mathcal{P} = \{b\}$ and $\varphi = \mathcal{L}_{\texttt{safe}}(\{b\})$. Then, there does not exist a robust step-wise sampling-based reduction $\mathcal{F}$ with the property that for any given $\mathcal{M}$, if $(\bar{\mathcal{M}}, \varphi', f) = \mathcal{F}(\mathcal{M}, \varphi)$, then φ' is a robust specification and $\Pi_{opt}(\bar{\mathcal{M}}, \varphi') \neq \emptyset$.*

Proof. Consider the MDP $\mathcal{M} = (S, A, s_0, P, L)$ in Figure 3.3 with $p_1 = p_2 = 1$, and consider any $\varepsilon < 1$. From Theorem 3.12, we know that for any $\delta > 0$ there is an MDP $\mathcal{M}_\delta = (S, A, s_0, P_\delta, L)$ that is δ-close to $\mathcal{M}$ such that $\Pi_{\text{opt}}(\mathcal{M}_\delta, \varphi) \cap \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}, \varphi) = \emptyset$. For the sake of contradiction, suppose that such a reduction exists. Then, since $\mathcal{M}$ and $\mathcal{M}_\delta$ represent the same input (S, A, s_0, L, φ), the reduction outputs the same tuple $(\bar{S}, \bar{A}, \bar{s}_0, \bar{L}, f, \beta, \alpha, q_1, q_2, \varphi')$ in both cases. Furthermore, from Equation 3.2 it follows, that the new transition probability functions $\bar{P}$ and $\bar{P}_\delta$ corresponding to P and P_δ differ by at most a δ amount—i.e., $|\bar{P}(\bar{s}, \bar{a}, \bar{s}') - \bar{P}_\delta(\bar{s}, \bar{a}, \bar{s}')| \leq \delta$ for all $\bar{s}, \bar{s}' \in \bar{S}$ and $\bar{a} \in \bar{A}$. Let $\bar{\mathcal{M}}$ and $\bar{\mathcal{M}}_\delta$ be the MDPs corresponding to $\bar{P}$ and $\bar{P}_\delta$.

Let $\varepsilon' > 0$ be such that $f(\Pi^{\varepsilon'}_{\text{opt}}(\bar{\mathcal{M}}, \varphi')) \subseteq \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}, \varphi)$. Since the specification φ' is robust, there is a $\delta = \delta_{\bar{\mathcal{M}}, \varepsilon'} > 0$ such that $\Pi_{\text{opt}}(\bar{\mathcal{M}}_\delta, \varphi') \subseteq \Pi^{\varepsilon'}_{\text{opt}}(\bar{\mathcal{M}}, \varphi')$. Let $\bar{\pi} \in \Pi_{\text{opt}}(\bar{\mathcal{M}}_\delta, \varphi')$ be an optimal policy for $\bar{\mathcal{M}}_\delta$ w.r.t. φ'. Now, since the reduction is optimality preserving, we have $f(\bar{\pi}) \in \Pi_{\text{opt}}(\mathcal{M}_\delta, \varphi)$. But then, we also have $f(\bar{\pi}) \in \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}, \varphi)$, which contradicts our assumption on $\mathcal{M}_\delta$. □

We observe that the above result holds when the reduction is allowed to take at most one step in $\mathcal{M}$ during a step in $\bar{\mathcal{M}}$ (and can be generalized to a bounded number of steps). This leads to the following open problem.

Open Problem 3. *Does there exist a robust sampling-based reduction $\mathcal{F}$ such that for any RL task $(\mathcal{M}, \varphi)$, where φ is an LTL specification, letting $(\bar{\mathcal{M}}, \varphi', f) = \mathcal{F}(\mathcal{M}, \varphi)$, we have that φ' is a discounted reward specification (allowing $\bar{\mathcal{M}}$ to take unbounded number of steps in $\mathcal{M}$ per transition)?*

Note that even if such a reduction is possible, simulating $\bar{\mathcal{M}}$ would be computationally hard since there might be no bound on the time it takes for a step in $\bar{\mathcal{M}}$ to occur.

We formalized a notion of sampling-based reduction for MDPs with unknown transition probabilities. Although reducing LTL specifications to discounted rewards is a natural approach towards obtaining learning algorithms for LTL specifications, we showed that step-wise sampling-based reductions are insufficient to obtain learning algorithms with guarantees. This leads us to the natural question of whether it is possible to design learning algorithms for LTL specifications with guarantees. Unfortunately, it turns out that it is not possible to obtain PAC-MDP algorithms for safety specifications.

Theorem 3.15. *There does not exist a PAC-MDP algorithm for the class of safety specifications.*

Theorem 3.14 shows that it is not possible to obtain a PAC-MDP algorithm for safety specifications by simply applying a step-wise sampling-based reduction followed by a PAC-MDP algorithm for discounted-sum reward specifications. Also, Theorem 3.14 does not follow from Theorem 3.15 because, the definition of a robust reduction allows the maximum value of ε' that satisfies $f(\Pi^{\varepsilon'}_{\text{opt}}(\bar{\mathcal{M}}, \varphi')) \subseteq \Pi^{\varepsilon}_{\text{opt}}(\bar{\mathcal{M}}, \varphi)$ to depend on the transition probability function P of $\mathcal{M}$. However the sample complexity function h of a PAC-MDP algorithm (Definition 2.2) should be independent of P.

Intuitively, Theorem 3.15 follows from that fact that, when learning from simulation, it is highly likely that the learning algorithm will encounter identical transitions when the underlying MDP is modified slightly. This makes it impossible to infer an ε-optimal policy using a number of samples that is independent of the transition probabilities since safety specifications are not robust.

Proof. Suppose there is a PAC-MDP algorithm $\mathcal{A}$ for the class of safety specifications. Consider $\mathcal{P} = \{b\}$ and the family of MDPs shown in Figure 3.4 where double circles denote states at which b holds. Let $\varphi = \mathcal{L}_{\texttt{safe}}(\{b\})$ and $0 < \varepsilon < \frac{1}{2}$. For any $\delta > 0$, we use $\mathcal{M}^1_\delta$ to denote the MDP with $p_1 = 1$ and $p_2 = 1 - \delta$, and $\mathcal{M}^2_\delta$ to denote the MDP with $p_1 = 1 - \delta$ and $p_2 = 1$. Finally, let $\mathcal{M}$ denote the MDP with $p_1 = p_2 = 1$. We first show the following.

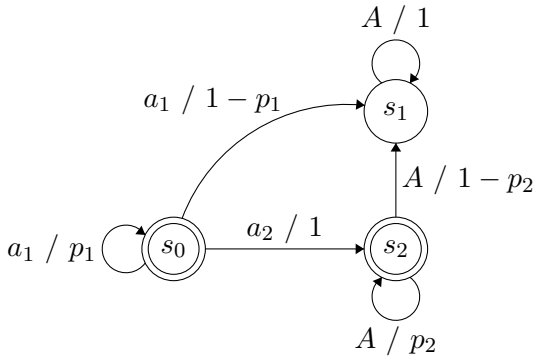


Figure 3.4: A class of MDPs for showing no PAC-MDP algorithm exists for safety specifications.

Lemma 3.16. *For any $\delta \in (0,1)$, we have $\Pi^{\varepsilon}_{opt}(\mathcal{M}^1_\delta, \varphi) \cap \Pi^{\varepsilon}_{opt}(\mathcal{M}^2_\delta, \varphi) = \emptyset$.*

Proof. Suppose $\pi \in \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}^1_\delta, \varphi)$ is an ε-optimal policy for $\mathcal{M}^1_\delta$ w.r.t. φ. Let $x_i = \pi((s_0a_1)^i s_0)(a_1)$ denote the probability that π chooses a_1 after i self-loops in s_0. Then $J_\varphi^{\mathcal{M}^1_\delta}(\pi) = \lim_{t\to\infty} \prod_{i=0}^{t} x_i$ since choosing a_2 in s_0 leads to eventual violation of the safety specification. The policy π^*_1 that always chooses a_1 achieves a value of $J_\varphi^{\mathcal{M}^1_\delta}(\pi^*_1) = \mathcal{J}^*(\mathcal{M}^1_\delta, \varphi) = 1$. Since $\pi \in \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}^1_\delta, \varphi)$ we have $\lim_{t\to\infty} \prod_{i=0}^{t} x_i \geq 1-\varepsilon$. Therefore $\prod_{i=0}^{t} x_i \geq 1-\varepsilon$ for all $t \in \mathbb{N}$ since $z_t = \prod_{i=0}^{t} x_i$ is a non-increasing sequence.

Now let $E_t = \texttt{Cyl}((s_0a_1)^t s_1)$ denote the set of all runs that reach s_1 after exactly t steps while staying in s_0 until then. We have

$$\mathcal{D}_\pi^{\mathcal{M}^2_\delta}(E_t) = (1-p_1)p_1^{t-1}\prod_{i=0}^{t-1} x_i \geq \delta(1-\delta)^{t-1}(1-\varepsilon).$$

Since $\{E_t\}_{t=1}^{\infty}$ are pairwise disjoint sets, letting $E = \bigcup_{t=1}^{\infty} E_t$, we have

$$\mathcal{D}_\pi^{\mathcal{M}^2_\delta}(E) = \sum_{t=1}^{\infty} \mathcal{D}_\pi^{\mathcal{M}^2_\delta}(E_t) \geq \sum_{t=1}^{\infty} \delta(1-\delta)^{t-1}(1-\varepsilon) = 1-\varepsilon.$$

But we have that $E \subseteq B = \{\zeta \in \mathcal{Z}(S,A) \mid L(\zeta) \notin \mathcal{L}_{\texttt{safe}}(\{b\})\}$ and hence $J_\varphi^{\mathcal{M}^2_\delta}(\pi) = 1 - \mathcal{D}_\pi^{\mathcal{M}^2_\delta}(B) \leq 1 - \mathcal{D}_\pi^{\mathcal{M}^2_\delta}(E) \leq \varepsilon$. Any policy π^*_2 that picks a_2 in the first step achieves $J_\varphi^{\mathcal{M}^2_\delta}(\pi^*_2) = \mathcal{J}^*(\mathcal{M}^2_\delta, \varphi) = 1$. Since $\varepsilon < \frac{1}{2}$, we have $J_\varphi^{\mathcal{M}^2_\delta}(\pi) \leq \varepsilon < \frac{1}{2} < 1-\varepsilon = \mathcal{J}^*(\mathcal{M}^2_\delta, \varphi) - \varepsilon$ which implies $\pi \notin \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}^2_\delta, \varphi)$.

Therefore $\Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}^1_\delta, \varphi) \cap \Pi^{\varepsilon}_{\text{opt}}(\mathcal{M}^2_\delta, \varphi) = \emptyset$ for all $\delta \in (0,1)$. □

Now let h be the sample complexity function of $\mathcal{A}$ as in Definition 2.2. We let $p = 0.1$ and $N = h(|S|, |A|, |\varphi|, \frac{1}{p}, \frac{1}{\varepsilon})$. We let $K = 2N + 1$ and choose $\delta \in (0,1)$ such that $(1-\delta)^K \geq 0.9$. Let $\{\pi_n\}_{n=1}^{\infty}$ denote the sequence of output policies of $\mathcal{A}$ when run on $\mathcal{M}$ with the precision $\varepsilon < \frac{1}{2}$ and $p = 0.1$. For $j \in \{1,2\}$, let E_j denote the event that at most N out of the first K policies $\{\pi_n\}_{n=1}^{K}$ are *not* ε-optimal for $\mathcal{M}^j_\delta$ (when $\mathcal{A}$ is run on $\mathcal{M}$). Then we have $\Pr^{\mathcal{M}}_{\mathcal{A}}(E_1) + \Pr^{\mathcal{M}}_{\mathcal{A}}(E_2) \leq 1$ because E_1 and E_2 are disjoint events (due to Lemma 3.16).

For $j \in \{1,2\}$, we let $\{\pi^j_n\}_{n=1}^{\infty}$ be the sequence of output policies of $\mathcal{A}$ when run on $\mathcal{M}^j_\delta$ with the same precision ε and $p = 0.1$. Let F_j denote the event that at most N out of the first K policies $\{\pi^j_n\}_{n=1}^{K}$ are *not* ε-optimal for $\mathcal{M}^j_\delta$ (when $\mathcal{A}$ is run on $\mathcal{M}^j_\delta$). Then PAC-MDP guarantee of $\mathcal{A}$ gives us that $\Pr^{\mathcal{M}^j_\delta}_{\mathcal{A}}(F_j) \geq 0.9$ for $j \in \{1,2\}$. Now let G_j denote the event that the the first K samples from $\mathcal{M}^j_\delta$ correspond to the deterministic transitions in $\mathcal{M}$—i.e., taking a_1 in s_0 leads to s_0 and taking any action in s_2 leads to s_2. We have that $\Pr^{\mathcal{M}^j_\delta}_{\mathcal{A}}(G_j) \geq (1-\delta)^K \geq 0.9$ for $j \in \{1,2\}$.

Applying union bound, we get that $\Pr^{\mathcal{M}^j_\delta}_{\mathcal{A}}(F_j \wedge G_j) \geq 0.8$ for $j \in \{1,2\}$. The probability of any execution (sequence of output policies, actions taken, resets performed and transitions observed) of $\mathcal{A}$ on $\mathcal{M}^j_\delta$ that satisfies the conditions of F_j and G_j is less than or equal to the probability of obtaining the same execution when $\mathcal{A}$ is run on $\mathcal{M}$ and furthermore such an execution also satisfies the conditions of E_j. Therefore, we have $\Pr^{\mathcal{M}}_{\mathcal{A}}(E_j) \geq \Pr^{\mathcal{M}^j_\delta}_{\mathcal{A}}(F_j \wedge G_j) \geq 0.8$ for $j \in \{1,2\}$. But this contradicts the fact that $\Pr^{\mathcal{M}}_{\mathcal{A}}(E_1) + \Pr^{\mathcal{M}}_{\mathcal{A}}(E_2) \leq 1$. □

We can also conclude that PAC-MDP algorithms do not exist for limit-average rewards since safety specifications can be encoded using limit-average rewards. Our proof of Theorem 3.15 can be modified to show the result for reachability as well.

A concurrent work [157] characterizes the class of LTL specifications for which PAC-MDP algorithms exist. An LTL formula φ is *finitary* if there exists a horizon H such that infinite length words sharing the same prefix of length H are either all accepted or all rejected by φ. Then, their result can be

summarized as follows.

Theorem 3.17 ([157]). *There exists a PAC-MDP algorithm for an LTL specification φ if and only if φ is finitary.*

Next, to the best of our knowledge, it is unknown if there is a learning algorithm that converges in the limit for the class of LTL specifications.

Open Problem 4. *Does there exist a learning algorithm that converges in the limit for the class of LTL specifications?*

Observe that algorithms that converge in the limit do not necessarily have a bound on the number of samples needed to learn an ε-optimal policy; instead, they only guarantee that the values of the policies $\{J_\varphi^{\mathcal{M}}(\pi_n)\}_{n=1}^{\infty}$ converge to the optimal value $\mathcal{J}^*(\mathcal{M}, \varphi)$ almost surely. Therefore, the rate of convergence can be arbitrarily small and can depend on the transition probability function P.

3.4. Summary

We have established a formal framework for sampling-based reductions of RL tasks. Given an RL task (an MDP and a specification), the goal is to generate another RL task such that the transformation preserves optimal solutions and is (optionally) robust. A key challenge is that the transformation must be defined without the knowledge of the transition probabilities.

This framework offers a unified view of the literature on RL from logical specifications, in which an RL task with a logical specification is transformed to one with a reward-based specification. We defined optimality preserving as well as robust sampling-based reductions of RL tasks. Specification translations are a special form of sampling-based reductions in which the underlying MDP is not altered. We showed that specification translations from LTL to reward machines with discounted-sum objectives do not preserve optimal solutions. This motivated the need for transformations in which the underlying MDP may be altered. By revisiting such transformations from existing literature within our framework, we exposed the nuances in their theoretical guarantees about optimality preservation. Specifically, known transformations from LTL specifications to rewards are not

strictly optimality preserving sampling-based reductions since they depend on parameters which are not available in the RL setting such as some information about the transition probabilities of the MDP. We showed that LTL specifications, which are non-robust, cannot be robustly transformed to a robust specification, such as discounted-sum rewards. We also proved that there are LTL specifications, including simple safety specifications, that do not admit PAC-MDP learning algorithms.

Finally, we are left with multiple open problems. Notably, it is unknown whether there exists a learning algorithm for LTL that converges in the limit and does not depend on any unavailable information about the MDP. However, existing algorithms for learning from LTL specifications have been demonstrated to be effective in practice, even for continuous state MDPs. This shows that there is a gap between the theory and practice suggesting that we need better measures for theoretical analysis of such algorithms; for instance, realistic MDPs may have additional structure that makes learning possible.

3.5. Related Work

A recent paper [49] proposes a PAC algorithm for LTL specifications under the assumption that the structure of the MDP $\mathcal{M}$ (transitions with non-zero probability) is known. Concurrent to this work, Yang et al. [157] show that PAC algorithms do not exist for any *non-finitary* LTL objective. Closely related to this work is the work on expressivity of discounted rewards [3] which studies whether certain kinds of tasks can be encoded using discounted rewards. There are a couple of key differences to this work. First, they do not consider reductions that involve modifying the underlying MDP $\mathcal{M}$. Second, the tasks considered are based on explicit orderings among policies or trajectories rather than succinct formal specifications.

CHAPTER 4

SPECTRL: A Task Specification Language

In Chapter 3, we demonstrated the difficulty in obtaining theoretical guarantees for RL from LTL specifications. The primary hurdle lies in the fact that temporal logics such as LTL were designed to specify properties about *infinite-horizon* behaviour of systems. However, in many practical applications such as robotics, it is sufficient to specify the desired behaviour over a fixed finite duration—e.g., the robot must reach a particular location *within H steps*. Such *finite-horizon* specifications help us circumvent the theoretical hardness results and obtain PAC learning algorithms; for example, by assigning a reward of 1 when the specification is satisfied and 0 otherwise. Nonetheless, such simple reward schemes are usually not practical and do not enable efficient learning for complex tasks.

In this chapter, we define a simple specification language based on a fragment of LTL_f which can be used to specify complex control tasks. Our language allows the user to specify objectives and safety constraints as logical predicates over states, and then compose these primitives sequentially or as disjunctions. We then show that we can generate *well-shaped* reward functions for tasks specified in our language and demonstrate empirically that the generated rewards can be used in conjunction with exiting RL algorithms to train policies to perform complex tasks in continuous-state environments.

4.1. Motivation

As a simple example, consider the task in Figure 4.1, where the state is the robot position and its remaining fuel, the action is a (bounded) robot velocity, and the task is

> "Reach target q, then reach target p, while maintaining positive fuel and avoiding obstacle O."

To encode this task using rewards, we would have to combine rewards for (i) reaching q, and then reaching p (where "reach x" denotes the task of reaching an ε-box around x—the regions

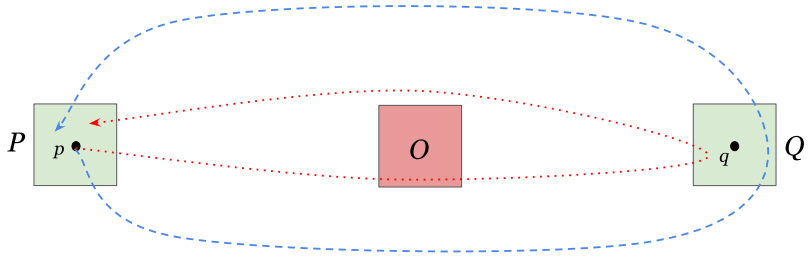


Figure 4.1: Example control task. The blue dashed trajectory satisfies the specification φ_{ex} (ignoring the fuel budget), whereas the red dotted trajectory does not satisfy φ_{ex} as it passes through the obstacle.

corresponding to p and q are denoted by P and Q respectively), (ii) avoiding region O, and (iii) maintaining positive fuel, into a single reward function. Furthermore, we would have to extend the state space to keep track of whether q has been reached—otherwise, the control policy would not know whether the current goal is to move towards q or p. Finally, we might need to shape the reward to assign partial credit for getting closer to q, or for reaching q without reaching p. This task can be expressed in our language as

$$\varphi_{\text{ex}} \;=\; \texttt{achieve}\;(\texttt{reach}\;q;\;\texttt{reach}\;p)\;\texttt{ensuring}\;(\texttt{avoid}\;O \wedge \texttt{fuel} > 0), \tag{4.1}$$

where `fuel` is the component of the state space keeping track of how much fuel is remaining.

The principle underlying our approach is that in many applications, users have in mind a sequence of high-level actions that are needed to accomplish a given task. For example, φ_{ex} may encode the scenario where the user wants a quadcopter to fly to a location q, take a photograph, and then return back to its owner at position p, while avoiding a building O and without running out of battery. Alternatively, a user may want to program a warehouse robot to go to the next room, pick up a box, and then bring this item back to the first room. In addition to specifying sequences of tasks, users can also specify choices between multiple tasks (e.g., bring back any box).

A key aspect of our approach is to allow the user to specify a task without providing the low-level rewards. Instead, analogous to how a compiler generates machine code from a program written by the user, we propose a compiler for our language that takes the user-provided task specification and

generates a reward function.

A key challenge is that our specifications may encode rewards that are not Markov—e.g., in φ_{ex}, the robot needs memory that keeps track of whether its current goal is `reach` q or `reach` p. Thus, our compiler automatically extends the state space using a *task monitor*, which is an automaton that keeps track of which subtasks have been completed.[3] Furthermore, this automaton may have nondeterministic transitions; thus, our compiler also extends the action space with actions for choosing state transitions. Intuitively, there may be multiple points in time at which a subtask is considered completed, and the robot must choose which one to use.

Another challenge is that the naïve choice of rewards—i.e., reward 1 if the task is completed and 0 otherwise—can be very sparse, especially for complex tasks. Thus, our compiler automatically performs two kinds of reward shaping based on the structure of the specification—it assigns partial credit for (i) partially accomplishing intermediate subtasks, and (ii) for completing more subtasks. For deterministic MDPs, our reward shaping is guaranteed to preserve the optimal policy; we empirically find it also works well for stochastic MDPs.

4.2. Task Specification Language

Specification language. Intuitively, a specification φ in our language, SPECTRL, is a logical formula specifying whether a given run ζ successfully accomplishes the desired task—in particular, given an MDP $\mathcal{M} = (S, A, s_0, P)$ it can be interpreted as a function $\varphi : \mathcal{Z}_f(S, A) \to \mathbb{B}$, where $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$, defined by

$$\varphi(\zeta) = \mathbb{1}[\zeta \text{ successfully achieves the task}],$$

where $\mathbb{1}$ is the indicator function. Formally, the user first defines a set of *atomic predicates*[4] $\mathcal{P}_0$, where every $p \in \mathcal{P}_0$ is associated with a function $[\![p]\!] : S \to \mathbb{B}$ such that $[\![p]\!](s)$ indicates whether s

[3] Intuitively, this construction is analogous to compiling a regular expression to a finite state automaton.
[4] Note that predicates here are defined in a slightly different way as compared to Chapter 2.

satisfies p. For example, given $s' \in S$, the atomic predicate

$$[\![\texttt{reach}\ s']\!](s) \ = \ (\|s - s'\|_\infty < 1)$$

indicates whether the robot is in a state near s', and given a rectangular region $O \subseteq S$, the atomic predicate

$$[\![\texttt{avoid}\ O]\!](s) \ = \ (s \notin O)$$

indicates if the robot is avoiding O. In general, the user can define a new atomic predicate as an arbitrary function $[\![p]\!] : S \to \mathbb{B}$. Next, *predicates* $b \in \mathcal{P}$ are conjunctions and disjunctions of atomic predicates. In particular, the syntax of predicates is given by

$$b \ ::= \ p \mid b_1 \wedge b_2 \mid b_1 \vee b_2,$$

where $p \in \mathcal{P}_0$. Similar to atomic predicates, each predicate $b \in \mathcal{P}$ corresponds to a function $[\![b]\!] : S \to \mathbb{B}$, defined recursively by $[\![b_1 \wedge b_2]\!](s) = [\![b_1]\!](s) \wedge [\![b_2]\!](s)$ and $[\![b_1 \vee b_2]\!](s) = [\![b_1]\!](s) \vee [\![b_2]\!](s)$. Finally, the syntax of our specifications is given by [5]

$$\varphi \ ::= \ \texttt{achieve}\ b \mid \varphi_1\ \texttt{ensuring}\ b \mid \varphi_1; \varphi_2 \mid \varphi_1\ \texttt{or}\ \varphi_2,$$

where $b \in \mathcal{P}$. Intuitively, the first construct means that the robot should try to reach a state s such that $[\![b]\!](s) = \texttt{true}$. The second construct says that the robot should try to satisfy φ_1 while always staying in states s such that $[\![b]\!](s) = \texttt{true}$. The third construct says the robot should try to satisfy task φ_1 and then task φ_2. The fourth construct means that the robot should try to satisfy either task φ_1 or task φ_2. Formally, we associate a function $[\![\varphi]\!] : \mathcal{Z}_f(S, A) \to \mathbb{B}$ with φ, mapping finite

[5] Here, `achieve` and `ensuring` correspond to the "eventually" and "always" operators in temporal logic.

runs to Boolean values, recursively as follows:

$$\begin{aligned}
[\![\texttt{achieve}\ b]\!](\zeta) &= \exists\ i \le t,\ [\![b]\!](s_i) \\
[\![\varphi\ \texttt{ensuring}\ b]\!](\zeta) &= [\![\varphi]\!](\zeta)\ \wedge\ (\forall\ i \le t,\ [\![b]\!](s_i)) \\
[\![\varphi_1; \varphi_2]\!](\zeta) &= \exists\ i < t,\ ([\![\varphi_1]\!](\zeta_{0:i})\ \wedge\ [\![\varphi_2]\!](\zeta_{i+1:t})) \\
[\![\varphi_1\ \texttt{or}\ \varphi_2]\!](\zeta) &= [\![\varphi_1]\!](\zeta)\ \vee\ [\![\varphi_2]\!](\zeta),
\end{aligned}$$

where $\zeta = s_0 a_0 s_1 \dots a_{t-1} s_t$. The function $[\![\varphi]\!]$ can be naturally extended to infinite runs; for an infinite run $\zeta \in \mathcal{Z}(S, A)$ we have $[\![\varphi]\!](\zeta) = \texttt{true}$ if and only if there is a $t \ge 0$ such that $[\![\varphi]\!](\zeta_{0:t}) = \texttt{true}$. A run ζ (finite or infinite) *satisfies* φ if $[\![\varphi]\!](\zeta) = \texttt{true}$, which is denoted $\zeta \models \varphi$.

Objective function. In this chapter, we study the finite-horizon setting where there is a fixed horizon H; however, the reward generation and reward shaping techniques presented here are also useful in a slightly more general setting that we study in the next chapter. Given an MDP $\mathcal{M}$, a horizon $H \in \mathbb{N}$ and a SPECTRL specification φ, the objective function $J^{\mathcal{M}}_{\varphi,H}$ is given by

$$J^{\mathcal{M}}_{\varphi,H}(\pi) = \Pr_{\zeta \sim \mathcal{D}^{\mathcal{M}}_{\pi}} \big[[\![\varphi]\!](\zeta_{0:H}) = \texttt{true} \big].$$

4.3. Compilation and Learning Algorithms

In this section, we describe our algorithm for reducing a SPECTRL specification φ for a given MDP (S, A, s_0, P) to an *augmented MDP* $\tilde{\mathcal{M}}$ and a reward function $\tilde{R}$. This is essentially a simulation-based reduction in the finite-horizon setting. At a high level, our algorithm extends the state space S to keep track of completed subtasks and constructs a reward function $R : \mathcal{Z}_f(S, A) \to \mathbb{R}$ encoding φ. A key feature of our algorithm is that the user has control over the compilation process—we provide a natural default compilation strategy, but the user can extend or modify our approach to improve the performance of the RL algorithm.

Quantitative semantics. So far, we have associated specifications φ with Boolean semantics (i.e., $[\![\varphi]\!](\zeta) \in \mathbb{B}$). A naïve strategy is to assign rewards to runs based on whether they satisfy φ:

$$R(\zeta) = \begin{cases} 1 & \text{if } \zeta \models \varphi \\ 0 & \text{otherwise.} \end{cases}$$

However, it is usually difficult to learn a policy to maximize this reward due to its discrete nature. A common strategy is to provide a *shaped reward* that quantifies the "degree" to which ζ satisfies φ. Our algorithm uses an approach based on *quantitative semantics* for temporal logic [40, 46, 110]. In particular, we associate an alternate interpretation of a specification φ as a real-valued function $[\![\varphi]\!]_q : \mathcal{Z}_f(S, A) \to \mathbb{R}$. To do so, the user provides quantitative semantics for atomic predicates $p \in \mathcal{P}_0$—in particular, they provide a function $[\![p]\!]_q : S \to \mathbb{R}$ that quantifies the degree to which p holds for $s \in S$. For example, we can use

$$\begin{aligned} [\![\texttt{reach } s']\!]_q(s) &= 1 - d_\infty(s, s') \\ [\![\texttt{avoid } O]\!]_q(s) &= d_\infty(s, O), \end{aligned}$$

where d_∞ is the L_∞ distance between points, with the usual extension to sets. These semantics should satisfy $[\![p]\!]_q(s) > 0$ if and only if $[\![p]\!](s) = \texttt{true}$, and a larger value of $[\![p]\!]_q$ should correspond to an increase in the "degree" to which p holds. Then, the quantitative semantics for predicates $b \in \mathcal{P}$ are $[\![b_1 \wedge b_2]\!]_q(s) = \min\{[\![b_1]\!]_q(s), [\![b_2]\!]_q(s)\}$ and $[\![b_1 \vee b_2]\!]_q(s) = \max\{[\![b_1]\!]_q(s), [\![b_2]\!]_q(s)\}$. Assuming $[\![p]\!]_q$ satisfies the above properties, then $[\![b]\!]_q > 0$ if and only if $[\![b]\!] = \texttt{true}$.

In principle, we could now define quantitative semantics for specifications φ. For a run ζ of length

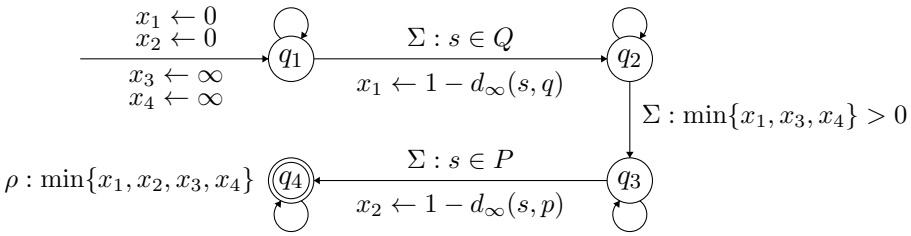


Figure 4.2: An example of a task monitor. Final states are labeled with rewards (prefixed with "ρ :"). Transitions are labeled with transition conditions (prefixed with "Σ :"), as well as register update rules. A transition from q_2 to q_4 is omitted for clarity. The two updates $x_3 \leftarrow \min\{x_3, d_\infty(s, O)\}$ and $x_4 \leftarrow \min\{x_4, \texttt{fuel}(s)\}$ are applied in every transition and are also omitted for clarity.

t we have,

$$
\begin{aligned}
[\![\texttt{achieve}\ b]\!]_q(\zeta) &= \max_{i \le t} [\![b]\!]_q(s_i) \\
[\![\varphi\ \texttt{ensuring}\ b]\!]_q(\zeta) &= \min\{[\![\varphi]\!]_q(\zeta),\ [\![b]\!]_q(s_0),\ \ldots,\ [\![b]\!]_q(s_t)\} \\
[\![\varphi_1; \varphi_2]\!]_q(\zeta) &= \max_{i < t} \min\{[\![\varphi_1]\!]_q(\zeta_{0:i}),\ [\![\varphi_2]\!]_q(\zeta_{i+1:t})\} \\
[\![\varphi_1\ \texttt{or}\ \varphi_2]\!]_q(\zeta) &= \max\{[\![\varphi_1]\!]_q(\zeta),\ [\![\varphi_2]\!]_q(\zeta)\}.
\end{aligned}
$$

Then, it is easy to show that $[\![\varphi]\!](\zeta) = \texttt{true}$ if and only if $[\![\varphi]\!]_q(\zeta) > 0$, so we could define a reward function $R(\zeta) = [\![\varphi]\!]_q(\zeta)$. However, one of our key goals is to extend the state space so the policy knows which subtasks have been completed. On the other hand, the semantics $[\![\varphi]\!]_q$ quantify over all possible ways that subtasks could have been completed in hindsight (i.e., once the entire trajectory is known). For example, there may be multiple points in a trajectory when a subtask $\texttt{reach}\ q$ could be considered as completed. Below, we describe our construction of the reward function, which is based on $[\![\varphi]\!]_q$, but applied to a single choice[6] of time steps at which each subtask is completed.

Task monitor. Intuitively, a *task monitor* is a finite-state automaton (FSA) that keeps track of which subtasks have been completed and which constraints are still satisfied. Unlike an FSA, its transitions may depend on the state $s \in S$ of a given MDP. Also, since we are using quantitative semantics, the task monitor has to keep track of the degree to which subtasks are completed and the degree to which constraints are satisfied; thus, it includes *registers* that keep track of the these

[6]Chosen by the RL agent.

values. A key challenge is that the task monitor is nondeterministic; as we describe below, we let the policy resolve the nondeterminism, which corresponds to choosing which subtask to complete at each step.

Formally, a task monitor is a tuple $\mathcal{T} = (Q, X, \Sigma, U, \Delta, q_0, v_0, F, \rho)$. First, Q is a finite set of *monitor states*, which are used to keep track of which subtasks have been completed. Also, X is a finite set of registers, which are variables used to keep track of the degree to which the specification holds so far. Given an MDP $\mathcal{M} = (S, A, s_0, P)$, an *augmented state* is a tuple $(s, q, v) \in S \times Q \times V$, where $V = \mathbb{R}^X$—i.e., an MDP state $s \in S$, a monitor state $q \in Q$, and a vector $v \in V$ encoding the value of each register in the task monitor. An augmented state can be viewed as a state in the product of $\mathcal{M}$ and $\mathcal{T}$.

The transitions Δ of the task monitor depend on the augmented state; thus, they need to specify two pieces of information: (i) conditions on the MDP states and registers for the transition to be enabled, and (ii) how the registers are updated. To handle (i), we consider a set Σ of predicates over $S \times V$, and to handle (ii), we consider a set U of functions $u : S \times V \to V$. Then, $\Delta \subseteq Q \times \Sigma \times U \times Q$ is a finite set of (nondeterministic) transitions, where $(q, \sigma, u, q') \in \Delta$ encodes *augmented transitions* $(s, q, v) \xrightarrow{a} (s', q', u(s, v))$, where $s \xrightarrow{a} s'$ is an MDP transition, which can be taken as long as $\sigma(s, v) = \texttt{true}$. Finally, $v_0 \in \mathbb{R}^X$ is the vector of initial register values, $F \subseteq Q$ is a set of final monitor states, and ρ is a reward function $\rho : S \times F \times V \to \mathbb{R}$.

Given an MDP $\mathcal{M} = (S, A, s_0, P)$ and a specification φ, our algorithm constructs a task monitor $\mathcal{T}_\varphi = (Q, X, \Sigma, U, \Delta, q_0, v_0, F, \rho)$ whose states and registers keep track which subtasks of φ have been completed. Our task monitor construction algorithm is analogous to compiling a regular expression to an FSA. More specifically, it is analogous to algorithms for compiling temporal logic formulas to automata [146]. We detail this algorithm in Section 4.4. The underlying graph of a task monitor constructed from any given specification is acyclic (ignoring self loops) and final states correspond to sink vertices with no outgoing edges (except a self loop).

As an example, the task monitor for φ_{ex} is shown in Figure 4.2. It has monitor states $Q =$

$\{q_1, q_2, q_3, q_4\}$ and registers $X = \{x_1, x_2, x_3, x_4\}$. The monitor states encode when the robot (i) has not yet reached q (q_1), (ii) has reached q, but has not yet returned to p (q_2 and q_3), and (iii) has returned to p (q_4); q_3 is an intermediate monitor state used to ensure that the constraints are satisfied before continuing. Register x_1 records $[\![\texttt{reach } q]\!](s) = 1 - d_\infty(s, q)$ when transitioning from q_1 to q_2, and x_2 records $[\![\texttt{reach } p]\!]_q = 1 - d_\infty(s, p)$ when transitioning from q_3 to q_4. Register x_3 keeps track of the minimum value of $[\![\texttt{avoid } O]\!](s) = d_\infty(s, O)$ over states s in the run, and x_4 keeps track of the minimum value of $[\![\texttt{fuel} > 0]\!](s)$ over states s in the run.

Augmented MDP. Given an MDP $\mathcal{M}$, a specification φ, and its task monitor $\mathcal{T}_\varphi$, our algorithm constructs an *augmented MDP*, $\tilde{\mathcal{M}} = (\tilde{S}, \tilde{A}, \tilde{s}_0, \tilde{P})$ and a (run-based) reward function $\tilde{R}$. Intuitively, if $\tilde{\pi}^*$ is a good policy (one that achieves a high expected reward) for the augmented MDP, then runs generated using $\tilde{\pi}^*$ should satisfy φ with high probability.

In particular, we have $\tilde{S} = S \times Q \times V$ and $\tilde{s}_0 = (s_0, q_0, v_0)$. The transitions $\tilde{P}$ are based on P and Δ. However, the task monitor transitions Δ may be nonderministic. To resolve this nondeterminism, we require that the policy decides which task monitor transitions to take. In particular, we extend the actions $\tilde{A} = A \times A_\varphi$ to include a component $A_\varphi = \Delta$ indicating which one to take at each step. An *augmented action* $(a, \tau) \in \tilde{A}$, where $\tau = (q, \sigma, u, q')$, is only available in augmented state $\tilde{s} = (s, q, v)$ if $\sigma(s, v) = \texttt{true}$. Then, the *augmented transition probability* is given by,

$$\tilde{P}((s, q, v),\ (a, (q, \sigma, u, q'))),\ (s', q', u(s, v))) = P(s, a, s').$$

Next, an *augmented run* of length t is a sequence $\tilde{\zeta} = (s_0, q_0, v_0) \xrightarrow{a_0} \ldots \xrightarrow{a_{t-1}} (s_t, q_t, v_t)$ of augmented transitions. The *projection* $\text{proj}(\tilde{\zeta}) = s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{t-1}} s_t$ of $\tilde{\zeta}$ is the corresponding (normal) run. Then, the *augmented rewards*

$$\tilde{R}(\tilde{\zeta}) = \begin{cases} \rho(s_t, q_t, v_t) & \text{if } q_t \in F \\ -\infty & \text{otherwise} \end{cases}$$

are constructed based on F and ρ. The augmented rewards satisfy the following property.

Theorem 4.1. *For any MDP $\mathcal{M}$, specification φ, and run ζ of $\mathcal{M}$ of length t, ζ satisfies φ if and only if there exists an augmented run $\tilde{\zeta}$ of length $t+1$ such that (i) $\tilde{R}(\tilde{\zeta}) > 0$, and (ii) proj$(\tilde{\zeta}_{0:t}) = \zeta$.*

This theorem follows by structural induction on φ and Lemmas 4.3 and 4.4 in Section 4.4. Thus, if we use RL to learn an optimal *augmented policy* $\tilde{\pi}^*$ over augmented states, then $\tilde{\pi}^*$ is likely to generate runs $\tilde{\zeta}$ such that proj$(\tilde{\zeta}_{0:t})$ satisfies φ.

Reward shaping. As discussed before, our algorithm constructs a shaped reward function that provides "partial credit" based on the degree to which φ is satisfied. We have already described one step of reward shaping—i.e., using quantitative semantics instead of the Boolean semantics. However, the augmented rewards $\tilde{R}$ are $-\infty$ unless a run reaches a final state of the task monitor. Thus, our algorithm performs an additional step of reward shaping—in particular, it constructs a reward function $\tilde{R}_s$ that gives partial credit for accomplishing subtasks in the MDP.

For a non-final monitor state q, let $\alpha : S \times Q \times V \to \mathbb{R}$ be defined by

$$\alpha(s, q, v) = \max_{(q,\sigma,u,q') \in \Delta,\ q' \neq q} [\![\sigma]\!]_q(s, v).$$

Intuitively, α quantifies how "close" an augmented state $\tilde{s} = (s, q, v)$ is to transitioning to another augmented state with a different monitor state. Then, our algorithm assigns partial credit to augmented states where α is larger.

However, to ensure that a good policy according to the shaped rewards $\tilde{R}_s$ is also a good policy according to $\tilde{R}$, it does so in a way that preserves the ordering of the cumulative rewards for runs—i.e., for two length t runs $\tilde{\zeta}$ and $\tilde{\zeta}'$, it guarantees that if $\tilde{R}(\tilde{\zeta}) > \tilde{R}(\tilde{\zeta}')$, then $\tilde{R}_s(\tilde{\zeta}) > \tilde{R}_s(\tilde{\zeta}')$.

To this end, we assume that we are given a lower bound C_ℓ on the final reward achieved when reaching a final monitor state—i.e., $C_\ell < \tilde{R}(\tilde{\zeta})$ for all $\tilde{\zeta}$ with final state $\tilde{s}_t = (s_t, q_t, v_t)$ such that $q_t \in F$ is a final monitor state. Furthermore, we assume that we are given an upper bound C_u on the absolute value of α over non-final monitor states—i.e., $C_u \geq |\alpha(s, q, v)|$ for any augmented state such that $q \notin F$.

Now, for any $q \in Q$, let d_q be the length of the longest path from q_0 to q in the graph of $\mathcal{T}_\varphi$ (ignoring self loops in Δ) and $D = \max_{q \in Q} d_q$. Given an augmented run $\tilde{\zeta}$, let $\tilde{s}_i = (s_i, q_i, v_i)$ be the first augmented state in $\tilde{\zeta}$ such that $q_i = q_{i+1} = ... = q_t$. Then, the shaped reward is

$$\tilde{R}_s(\tilde{\zeta}) = \begin{cases} \max_{i \leq j < t} \alpha(s_j, q_t, v_j) + 2C_u \cdot (d_{q_t} - D) + C_\ell & \text{if } q_t \notin F \\ \tilde{R}(\tilde{\zeta}) & \text{otherwise.} \end{cases}$$

If $q_t \notin F$, then the first term of $\tilde{R}_s(\tilde{\zeta})$ computes how close $\tilde{\zeta}$ was to transitioning to a new monitor state. The second term ensures that moving closer to a final state always increases reward. Finally, the last term ensures that rewards $\tilde{R}(\tilde{\zeta})$ for $q_t \in F$ are always higher than rewards for $q_t \notin F$. The following theorem justifies our reward shaping mechanism.

Theorem 4.2. *For two finite augmented runs $\tilde{\zeta}, \tilde{\zeta}'$,*

1. *if $\tilde{R}(\tilde{\zeta}) > \tilde{R}(\tilde{\zeta}')$, then $\tilde{R}_s(\tilde{\zeta}) > \tilde{R}_s(\tilde{\zeta}')$, and*
2. *if $\tilde{\zeta}$ and $\tilde{\zeta}'$ end in distinct non-final monitor states q_t and q'_t with $d_{q_t} > d_{q'_t}$, then we have $\tilde{R}_s(\tilde{\zeta}) \geq \tilde{R}_s(\tilde{\zeta}')$.*

Proof. The proof follows from the definitions of constants C_u and C_ℓ.

1. Let $\tilde{\zeta}$, $\tilde{\zeta}'$ be two augmented rollouts such that $\tilde{R}(\tilde{\zeta}) > \tilde{R}(\tilde{\zeta}')$. There are three cases to consider:

   - If both $\tilde{\zeta}$ and $\tilde{\zeta}'$ end in final monitor states, we have

   $$\tilde{R}_s(\tilde{\zeta}) = \tilde{R}(\tilde{\zeta}) > \tilde{R}(\tilde{\zeta}') = \tilde{R}_s(\tilde{\zeta}').$$

   - If $\tilde{\zeta}$ ends in a non-final monitor state, $\tilde{R}(\tilde{\zeta}) = -\infty$ and hence the claim is vacuously true.

   - If $\tilde{\zeta}$ ends in a final monitor state but $\tilde{\zeta}'$ ends in a monitor state $q'_T \notin F$, we have

$$
\begin{aligned}
\tilde{R}_s(\tilde{\zeta}') &= \max_{i' \le j < t} \alpha(s'_j, q'_t, v'_j) + 2C_u \cdot (d_{q_t} - D) + C_\ell \\
&\le \max_{i' \le j < t} \alpha(s'_j, q'_t, v'_j) - 2C_u + C_\ell && (d_{q_t} \le D - 1) \\
&\le C_\ell && (C_u \text{ is an upper bound on } \alpha) \\
&< \tilde{R}(\tilde{\zeta}) = \tilde{R}_s(\tilde{\zeta}). && (C_\ell \text{ is a lower bound on } \tilde{R})
\end{aligned}
$$

2. Let $\tilde{\zeta}$, $\tilde{\zeta}'$ be two augmented rollouts ending in distinct (non-final) monitor states q_t and q'_t such that $d_{q_t} > d_{q'_t}$. Then,

$$
\begin{aligned}
\tilde{R}_s(\tilde{\zeta}) &= \max_{i \le j < t} \alpha(s_j, q_t, v_j) + 2C_u \cdot (d_{q_t} - D) + C_\ell \\
&\ge \max_{i \le j < t} \alpha(s_j, q_t, v_j) + 2C_u + 2C_u \cdot (d_{q'_t} - D) + C_\ell && (d_{q_t} \ge d_{q'_t} - 1 \;\&\; C_u \ge 0) \\
&\ge C_u + 2C_u \cdot (d_{q'_t} - D) + C_\ell && (C_u \text{ is an upper bound on } -\alpha) \\
&\ge \max_{i' \le j < t} \alpha(s'_j, q'_t, v'_j) + 2C_u \cdot (d_{q'_t} - D) + C_\ell && (C_u \text{ is an upper bound on } \alpha) \\
&= \tilde{R}_s(\tilde{\zeta}').
\end{aligned}
$$

This concludes the proof. □

Reinforcement learning. Once our algorithm has constructed an augmented MDP $\tilde{\mathcal{M}}$, it can use any RL algorithm to learn an *augmented policy* $\tilde{\pi} : \tilde{S} \to \tilde{A}$ for the augmented MDP:

$$
\tilde{\pi}^* \in \arg\max_{\tilde{\pi}} \mathbb{E}_{\tilde{\zeta} \sim \mathcal{D}^{\tilde{\mathcal{M}}}_{\tilde{\pi}}}[\tilde{R}_s(\tilde{\zeta}_{0:H+1})]
$$

We solve this RL problem using augmented random search (ARS) [111].

After computing $\tilde{\pi}^*$, we can convert $\tilde{\pi}^*$ to a *projected policy* $\pi^* = \text{proj}(\tilde{\pi}^*)$ for the original MDP by integrating $\tilde{\pi}^*$ with the task monitor $\mathcal{T}_\varphi$, which keeps track of the information needed for $\tilde{\pi}^*$ to make

decisions. More precisely, $\text{proj}(\tilde{\pi}^*)$ includes internal memory that keeps track of the current monitor state and register value $(q_i, v_i) \in Q \times V$. It initializes this memory to the initial monitor state q_0 and initial register valuation v_0. Given an augmented action $(a, (q, \sigma, u, q')) = \tilde{\pi}^*((s_t, q_i, v_i))$, it updates this internal memory using the rules $q_{i+1} = q'$ and $v_{i+1} = u(s_i, v_i)$.

Finally, we use a neural network architecture similar to neural module networks [17, 18], where different neural networks accomplish different subtasks in φ. In particular, an augmented policy $\tilde{\pi}$ is a set of neural networks $\{N_q \mid q \in Q\}$, where Q are the monitor states in $\mathcal{T}_\varphi$. Each N_q takes as input $(s, v) \in S \times V$ and outputs an augmented action $N_q(s, v) = (a, a') \in A \times \mathbb{R}^k$, where k is the out-degree of q in $\mathcal{T}_\varphi$ and the transitions out of q are $\{(q, \sigma_1, u_1, q'_1), \ldots, (q, \sigma_k, u_k, q'_k)\}$; then, $\tilde{\pi}(s, q, v) = (a, \tau)$ with $\tau = \arg\max_{i \in \texttt{valid}(s,v)} \; a'_i$ where $\texttt{valid}(s, v) = \{i \mid \sigma_i(s, v) = \texttt{true}\}$.

4.4. Task Monitor Construction Algorithm

In this section, we detail our algorithm for constructing a task monitor $\mathcal{T}_\varphi$ for a given specification φ. Our construction algorithm proceeds recursively on the structure of φ. Implicitly, our algorithm maintains the property that every monitor-state q has a self-transition $(q, \texttt{true}, u, q)$; here, the update function u is the identity by default, but may be modified as part of the construction.

Notation. We use $\sqcup$ to denote the disjoint union. Given $v \in \mathbb{R}^X$, $v' \in \mathbb{R}^{X'}$, we define $v'' = v \oplus v' \in \mathbb{R}^{X \sqcup X'}$ to be their concatenation—i.e.,

$$v''(x) = \begin{cases} v(x) & \text{if } x \in X \\ v'(x) & \text{otherwise.} \end{cases}$$

Given $v \in \mathbb{R}^X$ and $Y \subseteq X$, we define $v \downarrow_Y \in \mathbb{R}^Y$ to be the restriction of v to Y. Given $v \in \mathbb{R}^X$ and $Y \supseteq X$, we define $v' = \texttt{extend}(v)_Y \in \mathbb{R}^Y$ to be

$$v'(y) = \begin{cases} v(y) & \text{if } y \in X \\ 0 & \text{otherwise.} \end{cases}$$

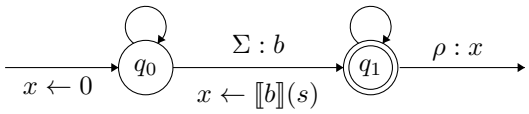
We drop the subscript Y when it is clear from context. Finally, given $v \in \mathbb{R}^X$ and $k \in \mathbb{R}$, we define $v' = v[x \mapsto k] \in V$ to be

$$v(x') = \begin{cases} v(x') & \text{if } x' \neq x \\ k & \text{otherwise.} \end{cases}$$

Also, recall that a predicate $b \in \mathcal{P}$ is defined over states $s \in S$; it can straightforwardly be extended to a predicate in Σ over $(s, v) \in S \times V$ by ignoring v. Note that every predicate $b \in \mathcal{P}$ is a negation free Boolean combination of atomic predicates $p \in \mathcal{P}_0$.

Finally, the definition of Σ depends on the set X of registers in the task monitor. When necessary, we use the notation Σ_X to make this dependence explicit. Finally, for $X \subseteq X'$, any $\sigma \in \Sigma_X$ can be interpreted as a predicate in $\Sigma_{X'}$ by ignoring the components of X' not in X.

Objectives. Consider the case $\varphi = \texttt{achieve}\ b$, where $b \in \mathcal{P}$. For this specification, our algorithm constructs the following task monitor:



The initial state is marked with an arrow into the state. Final states are double circles. Predicates $\sigma \in \Sigma$ labeling a transition appear prefixed by "Σ :". Rewards ρ labeling a state appear prefixed by "ρ :". Self loops are associated with the true predicate (omitted). Updates $u \in U$ are by default the identity function. Intuitively, the state q_0 on the left indicates that subtask b is not yet completed, and the state q_1 on the right indicates that b is completed, and x_1 records the degree to which b is satisfied while transitioning from q_0 to q_1.

Constraints. Consider the case $\varphi = \varphi_1\ \texttt{ensuring}\ b$, where $b \in \mathcal{P}$. Let

$$\mathcal{T}_{\varphi_1} = (Q_1, X_1, \Sigma, U_1, \Delta_1, q_1^0, v_1^0, F_1, \rho_1).$$

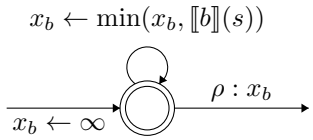


Figure 4.3: Task monitor for `ensuring` b.

Then, $\mathcal{T}_\varphi$ is the product of $\mathcal{T}_\varphi$ and $\mathcal{T}_{\texttt{ensuring}\, b}$ defined as

$$\mathcal{T}_\varphi = (Q_1, X_1 \sqcup \{x_b\}, \Sigma, U, \Delta, q_1^0, v_0, F_1, \rho),$$

where $\mathcal{T}_{\texttt{ensuring}\, b}$ is shown in Figure 4.3. Then, we have $(q, \sigma, u', q') \in \Delta$ if and only if there is a transition $(q, \sigma, u, q') \in \Delta_1$ such that

$$u'(s, v) = \texttt{extend}(u(s, v \downarrow_{X_1}))[x_b \mapsto \min(v(x_b), [\![b]\!](s))].$$

Furthermore, the initial register valuation is $v_0 = \texttt{extend}(v_1^0)[x_b \mapsto \infty]$ and the reward function ρ is

$$\rho(s, q, v) = \min\{\rho_1(s, q, v \downarrow_{X_1}), v(x_b)\}.$$

Intuitively, x_b encodes the minimum degree to which b is satisfied during a run.

Sequencing. An overview of the constructions for sequencing and choice operators is provided in Figure 4.4. Consider $\varphi = \varphi_1; \varphi_2$. Intuitively, $\mathcal{T}_\varphi$ is constructed by concatenating the registers of $\mathcal{T}_{\varphi_1}$ and $\mathcal{T}_{\varphi_2}$ (extending the update functions u as needed), and adding transitions (q, σ, u, q_0) from each final state q of $\mathcal{T}_{\varphi_1}$ to the initial state q_0 of $\mathcal{T}_{\varphi_2}$, where $\sigma = \texttt{true}$ and u is the identity on registers for $\mathcal{T}_{\varphi_1}$ and sets the registers of $\mathcal{T}_{\varphi_2}$ to their initial values. A subtle issue is that transitioning from φ_1 to φ_2 takes one time step, yet it should take zero time steps. Therefore, we add transitions from each final state of $\mathcal{T}_{\varphi_1}$ to all successors of the initial state of $\mathcal{T}_{\varphi_2}$.

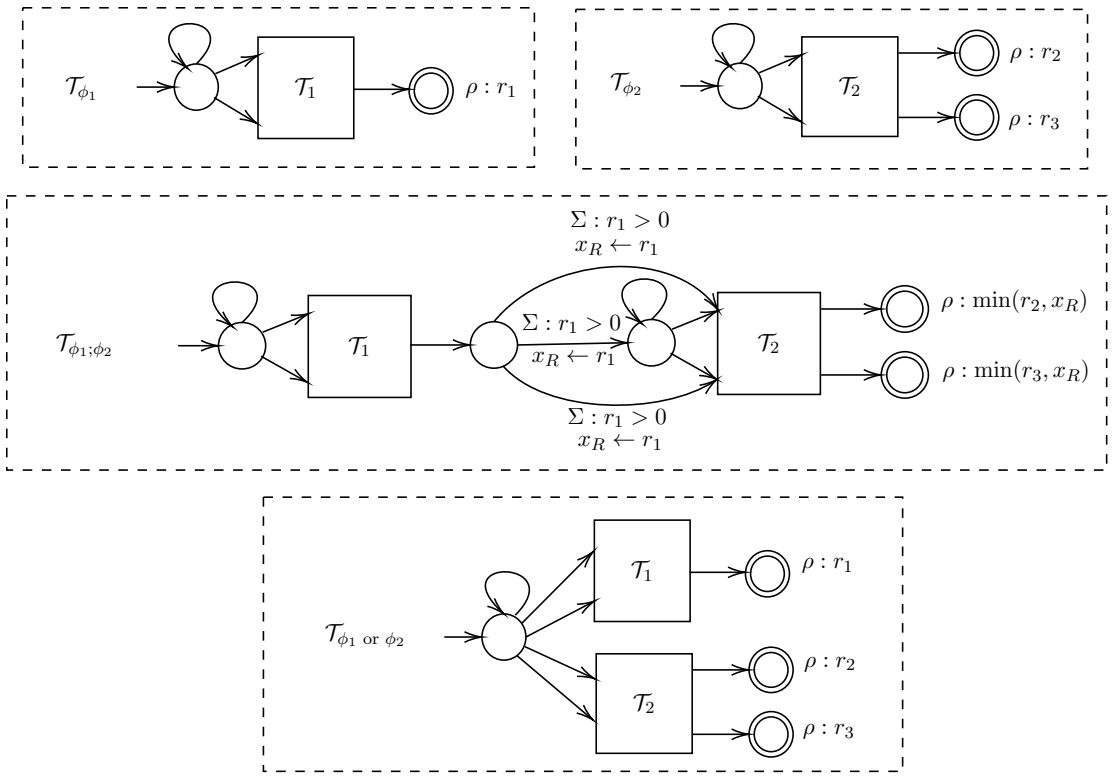


Figure 4.4: Overview of monitor construction for sequencing and choice operators.

More precisely, let

$$\mathcal{T}_{\varphi_1} = (Q_1, X_1, \Sigma, U_1, \Delta_1, q_1^0, v_1^0, F_1, \rho_1),$$
$$\mathcal{T}_{\varphi_2} = (Q_2, X_2, \Sigma, U_2, \Delta_2, q_2^0, v_2^0, F_2, \rho_2).$$

Assume without loss of generality that $X_2 \subseteq X_1$. Then,

$$\mathcal{T}_{\varphi} = (Q_1 \sqcup Q_2, X_1 \sqcup \{x_R\}, \Sigma, U, \Delta, q_1^0, v_0, F_2, \rho).$$

Here, $\Delta = \Delta_1' \cup \Delta_2' \cup \Delta_{1\to 2}$, where $(q, \sigma, u', q') \in \Delta_i'$ if there exists $(q, \sigma, u, q') \in \Delta_i$ such that

$$u'(s, v) = u(s, v \downarrow_{X_i}) \oplus v \downarrow_{X \setminus X_i},$$

and $(q, \sigma' \wedge \sigma_R, u', q') \in \Delta_{1\to 2}$ if $q \in F_1$ and there exists $(q_2^0, \sigma, u, q') \in \Delta_2$ such that the atomic

predicate σ_R is given by $[\![\sigma_R]\!](s, v) = \rho_1(s, q, v \downarrow_{X_1}) > 0$, the predicate σ' is given by $[\![\sigma']\!](s, v) = [\![\sigma]\!](s, v_2^0)$, and u' is given by

$$u'(s, v) = \texttt{extend}(u(s, v_2^0))[x_R \mapsto \rho_1(s, q, v \downarrow_{X_1})].$$

The initial register valuation is $v_0 = \texttt{extend}(v_1^0)$, and for any $q \in F_2$, the reward function ρ is

$$\rho(s, q, v) = \min\{\rho_2(s, q, v \downarrow_{X_2}), v(x_R)\}.$$

Choice. Consider the case $\varphi = \varphi_1 \texttt{ or } \varphi_2$. Intuitively, $\mathcal{T}_\varphi$ is constructed by combining the initial states of $\mathcal{T}_{\varphi_1}$ and $\mathcal{T}_{\varphi_2}$ into a single initial state q_0, and concatenating their registers. The transitions from q_0 are the union of the transitions from the initial states of $\mathcal{T}_{\varphi_1}$ and $\mathcal{T}_{\varphi_2}$. More precisely, let

$$\mathcal{T}_{\varphi_1} = (Q_1, X_1, \Sigma, U_1, \Delta_1, q_1^0, v_1^0, F_1, \rho_1),$$
$$\mathcal{T}_{\varphi_2} = (Q_2, X_2, \Sigma, U_2, \Delta_2, q_2^0, v_2^0, F_2, \rho_2).$$

This construction assumes that there are self loops on the initial states of $\mathcal{T}_{\varphi_1}$ and $\mathcal{T}_{\varphi_2}$. Then,

$$\mathcal{T}_\varphi = (Q, X_1 \sqcup X_2, \Sigma, U, \Delta, q_0, v_1^0 \oplus v_2^0, F_1 \sqcup F_2, \rho).$$

Here,

$$Q = (Q_1 \setminus \{q_1^0\}) \sqcup (Q_2 \setminus \{q_2^0\}) \sqcup \{q_0\},$$

and $\Delta = \Delta_1' \cup \Delta_2' \cup \Delta_0$, where where $(q, \sigma, u', q') \in \Delta_i'$ if $q \neq q_0$ and there is a transition $(q, \sigma, u, q') \in \Delta_i$ such that

$$u'(s, r, v) = \texttt{extend}(u(s, r, v \downarrow_{X_i})).$$

Also, let $(q_1^0, \top, u_1^0, q_1^0) \in \Delta_1$ and $(q_2^0, \top, u_2^0, q_2^0) \in \Delta_2$ be the self loops on the initial states of $\mathcal{T}_{\varphi_1}$ and $\mathcal{T}_{\varphi_2}$ respectively. Let

$$u_0(s, r, v) = u_1^0(s, r, v \downarrow_{X_1}) \oplus u_2^0(s, r, v \downarrow_{X_2}).$$

Then, $(q_0, \sigma, u', q) \in \Delta_0$ if either (i) $(q_0, \sigma, u', q) = (q_0, \top, u_0, q_0)$, or (ii) there exists $i \in \{1, 2\}$ such that $(q_i^0, \sigma, u, q) \in \Delta_i$, where $q \in Q_i \setminus \{q_i^0\}$ and

$$u'(s, r, v) = \texttt{extend}(u(s, r, v \downarrow_{X_i})).$$

The reward function ρ for $q \in F_i$ is given by

$$\rho(s, q, v) = \rho_i(s, q, v \downarrow_{X_i}).$$

4.4.1. Properties of the Constructed Task Monitor

The task monitor $\mathcal{T}_\varphi = (Q, X, \Sigma, U, \Delta, q_0, v_0, F, \rho)$ constructed from a SPECTRL specification φ admits some key properties that enable us to check satisfaction of φ and also assign shape rewards to any finite run ζ as described in Section 4.3. First, the following lemma follows by structural induction on φ.

Lemma 4.3. *For $\sigma \in \Sigma$, $[\![\sigma]\!](s, v) = \texttt{true}$ if and only if $[\![\sigma]\!]_q(s, v) > 0$.*

Next, let $G_{\mathcal{T}_\varphi}$ denote the underlying state transition graph of the task monitor $\mathcal{T}_\varphi$. Then we have the following lemma.

Lemma 4.4. *The task monitor $\mathcal{T}_\varphi$ constructed by our algorithm satisfies the following properties.*

1. *The only cycles in $G_{\mathcal{T}_\varphi}$ are self loops.*
2. *The finals states are precisely those states from which there are no outgoing edges except for self loops in $G_{\mathcal{T}_\varphi}$.*

3. *In $G_{\mathcal{T}_\varphi}$, every state is reachable from the initial state and for every state there is a final state that is reachable from it.*

4. *For any pair of states q and q', there is at most one transition from q to q'.*

5. *There is a self loop on every state q given by a transition $(q, \top, u, q)$ for some update function u where $\top$ denotes the predicate that assigns* `true` *to all $(s, v) \in S \times V$.*

The first three properties ensure progress when switching from one monitor state to another. The last two properties enable simpler composition of task monitors. The lemma follows by structural induction on φ.

4.5. Experiments

Setup. We implemented our algorithm in a tool, also called SPECTRL[7], and used it to learn policies for a variety of specifications. We consider a dynamical system with states $S = \mathbb{R}^2 \times \mathbb{R}$, where $(x, r) \in S$ encodes the robot position x and its remaining fuel r, actions $A = [-1, 1]^2$ where an action $a \in A$ is the robot velocity, and transitions $f(x, r, a) = (x + a + \epsilon, r - 0.1 \cdot |x_1| \cdot \|a\|_2)$, where $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ and the fuel consumed is proportional to the product of speed and distance from the y-axis. The initial state is $s_0 = (5, 0, 7)$, and the horizon is $H = 40$.

In Figure 4.5, we consider the following specifications, where $O = [4, 6] \times [4, 6]$:

- $\varphi_1 =$ `achieve reach` $(5, 10)$ `ensuring` (`avoid` O)
- $\varphi_2 =$ `achieve reach` $(5, 10)$ `ensuring` (`avoid` $O \wedge (r > 0)$)
- $\varphi_3 =$ `achieve` (`reach` $[(5, 10); (5, 0)]$) `ensuring avoid` O
- $\varphi_4 =$ `achieve` (`reach` $(5, 10)$ `or reach` $(10, 0)$; `reach` $(10, 10)$) `ensuring avoid` O
- $\varphi_5 =$ `achieve` (`reach` $[(5, 10); (5, 0); (10, 0)]$) `ensuring avoid` O
- $\varphi_6 =$ `achieve` (`reach` $[(5, 10); (5, 0); (10, 0); (10, 10)]$) `ensuring avoid` O
- $\varphi_7 =$ `achieve` (`reach` $[(5, 10); (5, 0); (10, 0); (10, 10); (0, 0)]$) `ensuring avoid` O

where `achieve` $(b; b')$ denotes `achieve` b; `achieve` b' and the abbreviation `reach` $[p_1; p_2]$ denotes `reach` p_1; `reach` p_2. For all specifications, each N_q has two fully connected hidden layers with

[7] The implementation can be found at https://github.com/keyshor/spectrl_tool.

30 neurons each and ReLU activations, and tanh function as its output layer. We compare our algorithm to [102] (TLTL), which directly uses the quantitative semantics of the specification as the reward function (with ARS as the learning algorithm), and to the constrained cross entropy method (CCE) [154], which is an RL algorithm for learning policies to perform tasks with constraints. We used neural networks with two hidden layers and 50 neurons per layer for both the baselines.

Results. Figure 3 shows learning curves of SPECTRL (our tool), TLTL, and CCE. In addition, it shows SPECTRL without reward shaping (Unshaped), which uses rewards $\tilde{R}$ instead of $\tilde{R}_s$. These plots demonstrate the ability of SPECTRL to outperform similar approaches previously proposed. For specifications $\varphi_1, ..., \varphi_5$, the curve for SPECTRL gets close to 100% in all executions, and for φ_6 and φ_7, it gets close to 100% in 4 out of 5 executions. The performance of CCE drops when multiple constraints (here, obstacle and fuel) are added (i.e., φ_2). TLTL performs similar to SPECTRL on tasks φ_1, φ_3 and φ_4 (at least in some executions), but SPECTRL converges faster for φ_1 and φ_4.

Since TLTL and CCE use a single neural network to encode the policy as a function of state, they perform poorly in tasks that require memory—i.e., φ_5, φ_6, and φ_7. For example, to satisfy φ_5, the action that should be taken at $s = (5, 0)$ depends on whether $(5, 10)$ has been visited. In contrast, SPECTRL performs well on these tasks since its policy is based on the monitor state.

These results also demonstrate the importance of reward shaping. Without it, ARS cannot learn unless it randomly samples a policy that reaches final monitor state. Reward shaping is especially important for specifications that include many sequencing operators $(\varphi; \varphi')$—i.e., specifications φ_5, φ_6, and φ_7.

Figure 4.6 (left) shows how sample complexity grows with the number of nested sequencing operators $(\varphi_1, \varphi_3, \varphi_5, \varphi_6, \varphi_7)$. Each curve indicates the average number of samples needed to learn a policy that achieves a satisfaction probability $\geq \tau$. SPECTRL scales well with the size of the specification.

Cartpole. Finally, we applied SPECTRL to a different control task—namely, to learn a policy for the version of cart-pole in OpenAI Gym, in which we used continuous actions instead of discrete actions. The specification is to move the cart to the right and move back left without letting the

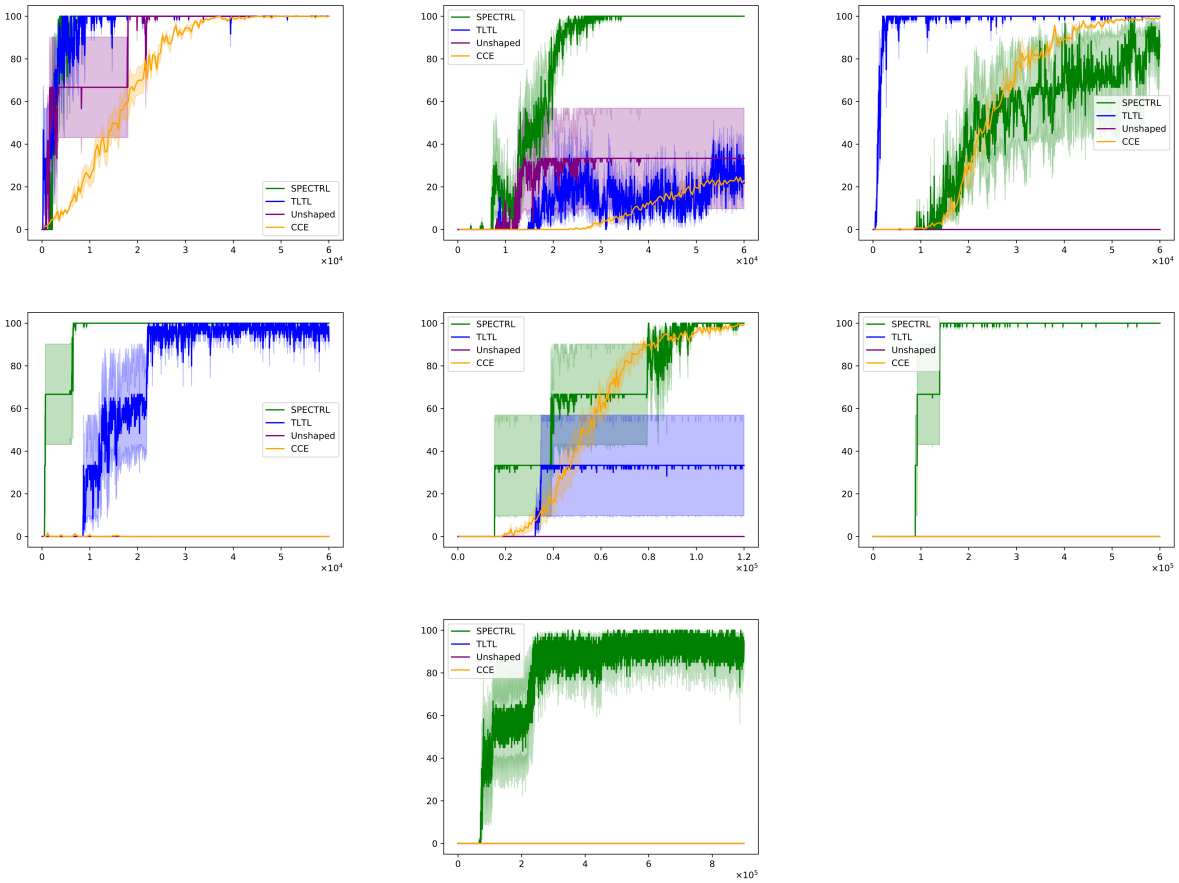


Figure 4.5: Learning curves for φ_1, φ_2 and φ_3 (top, left to right), and φ_4, φ_5, and φ_6 (middle, left to right) and, φ_7 (bottom), for SPECTRL (green), TLTL (blue), CCE (yellow), and SPECTRL without reward shaping (purple). The x-axis shows the number of sample trajectories, and the y-axs shows the probability of satisfying the specification (estimated using samples). To exclude outliers, we omitted one best and one worst run out of the 5 runs. The plots are the average over the remaining 3 runs with error bars indicating one standard deviation around the average.

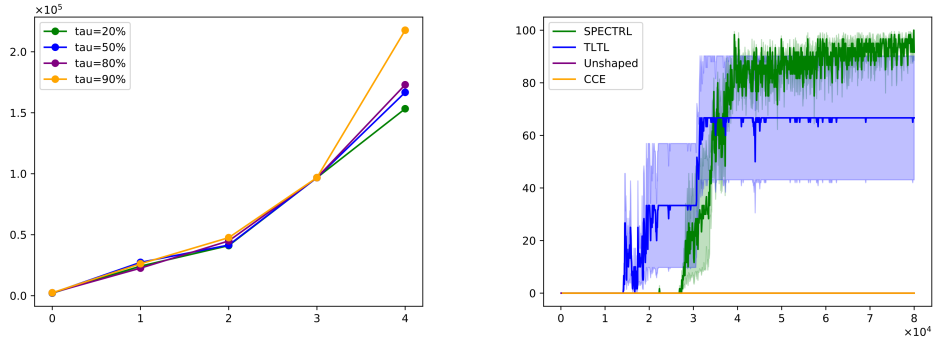


Figure 4.6: Sample complexity curves (left) with number of nested sequencing operators on the x-axis and average number of samples to converge on the y-axis. Learning curve for cartpole example (right).

pole fall. The formal specification is given by

$$\varphi = \texttt{achieve}\ (\texttt{reach}\ 0.5; \texttt{reach}\ 0.0)\ \texttt{ensuring}\ \texttt{balance}$$

where the predicate `balance` holds when the vertical angle of the pole is smaller than $\pi/15$ in absolute value. Figure 4.6 (right) shows the learning curve for this task averaged over 3 runs of the algorithm along with the three baselines. TLTL is able to learn a policy to perform this task, but it converges slower than SPECTRL; CCE is unable to learn a policy satisfying this specification.

4.6. Summary

We have proposed a language for formally specifying control tasks and an algorithm to learn policies to perform tasks specified in the language. Our algorithm first constructs a task monitor from the given specification, and then uses the task monitor to assign shaped rewards to runs of the system. Furthermore, the monitor state is also given as input to the controller, which enables our algorithm to learn policies for non-Markovian specifications. Finally, we implemented our approach in a tool called SPECTRL, which enables the users to *program* what the agent needs to do at a high level; then, it automatically learns a policy that tries to best satisfy the user intent. We also demonstrated that SPECTRL can be used to learn policies for complex specifications, and that it can outperform baselines for generating shaped rewards from temporal specifications.

4.7. Related Work

Imitation learning enables users to specify tasks by providing *demonstrations* of the desired task [119, 1, 160, 132, 69]. However, in many settings, it may be easier for the user to directly specify the task—e.g., when programming a warehouse robot, it may be easier to specify waypoints describing paths the robot should take than to manually drive the robot to obtain demonstrations. Also, unlike imitation learning, our language allows the user to specify global safety constraints on the robot. Indeed, we believe our approach complements imitation learning, since the user can specify some parts of the task in our language and others using demonstrations.

Another approach is for the user to provide a *policy sketch*—i.e., a string of tokens specifying a sequence of subtasks [18]. However, tokens have no meaning, except equal tokens represent the same task. Thus, policy sketches cannot be compiled to a reward function, which must be provided separately.

Our specification language is based on *temporal logic* [127], a language of logical formulas for specifying constraints over (typically, infinite) sequences of events happening over time. For example, temporal logic allows the user to specify that a logical predicate must be satisfied at some point in time (e.g., "eventually reach state q") or that it must always be satisfied (e.g., "always avoid an obstacle"). In our language, these notions are represented using the `achieve` and `ensuring` operators, respectively. Our language restricts temporal logic in a way that enables us to perform reward shaping, and also adds useful operators such as sequencing that allow the user to easily express complex control tasks.

Reward machines have been proposed as a high-level way to specify tasks [76]. In their work, the user provides a specification in the form of a finite state machine along with reward functions for each state. Then, they propose an algorithm for learning multiple tasks simultaneously by applying the Q-learning updates across different specifications. At a high level, these reward machines are similar to the task monitors defined in our work. However, we differ from their approach in two ways. First, in contrast to their work, the user only needs to provide a high-level logical specification; we

automatically generate a task monitor from this specification. Second, our notion of task monitor has a finite set of registers that can store real values; in contrast, their finite state reward machines cannot store quantitative information. There has also been work on automatically constructing reward machines from logical specifications [32]; however, reward machines generated this way produce sparse rewards.

CHAPTER 5

DIRL: A Compositional RL Algorithm

In Chapter 4, we introduced a specification language called SPECTRL for specifying complex long-horizon tasks and provided an algorithm to automatically generate shaped rewards for SPECTRL specifications. A key feature of the approach is that it enables the user to specify tasks *compositionally*—i.e., the user can independently specify a set of short-term subgoals, and then ask the robot to perform a complex task that involves achieving some of these subgoals. In principle, this exposes the compositional structure of the specified task—i.e., the decomposition of the task into a set of subtasks—to the reinforcement learning algorithm. However, existing approaches for learning from high-level specifications, including our approach in the previous chapter, do not exploit this structure during training which is often handled by an off-the-shelf RL algorithm. Recent works based on Reward Machines [76, 77] have proposed RL algorithms that exploit the structure of the specification to improve learning. However, these algorithms are based on model-free RL at both the high- and low-levels instead of model-based RL. Model-free RL has been shown to outperform model-based approaches on low-level control tasks [35]; however, at the high-level, it is unable to exploit the large amount of available structure. Thus, these approaches scale poorly to long-horizon tasks involving complex decision making.

In this chapter, we introduce DIRL, a novel compositional RL algorithm that leverages the structure in the specification to decompose the policy synthesis problem into a high-level planning problem and a set of low-level control problems. Then, it interleaves model-based high-level planning with model-free RL to compute a policy that tries to maximize the probability of satisfying the specification. In more detail, our algorithm begins by converting the user-provided specification into an abstract graph whose edges encode the subtasks, and whose vertices encode regions of the state space where each subtask is considered achieved. Then, it uses a Djikstra-style forward graph search algorithm to compute a sequence of subtasks for achieving the specification, aiming to maximize the success probability. Rather than compute a policy to achieve each subtask beforehand, it constructs them

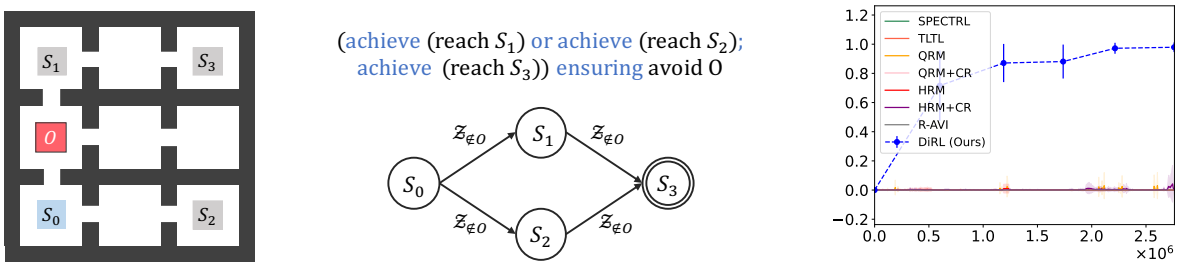


Figure 5.1: Left: The 9-rooms environment, with initial region S_0 in the bottom-left, an obstacle O in the middle-left, and three subgoal regions S_1, S_2, S_3 in the remaining corners. Middle top: A user-provided specification φ_{ex}. Middle bottom: The abstract graph $\mathcal{G}_{\text{ex}}$ DIRL constructs for φ_{ex}. Right: Learning curves for our approach and some baselines; x-axis is number of steps and y-axis is probability of achieving φ_{ex}.

on-the-fly for a subtask as soon as Djikstra's algorithm requires the cost of that subtask.

5.1. Overview

Illustrative example. Consider an RL-agent in the environment of interconnected rooms in Figure 5.1. The agent is initially in the blue box, and their goal is to navigate to either the top-left room S_1 or the bottom-right room S_2, followed by the top-right room S_3, all the while avoiding the red block O. This goal is formally captured by the SPECTRL specification φ_{ex} (middle top). This specification is comprised of four simpler RL subtasks—namely, navigating between the corner rooms while avoiding the obstacle. Our approach, DIRL, leverages this structure to improve learning. First, based on the specification alone, it constructs the abstract graph $\mathcal{G}_{\text{ex}}$ (see middle bottom) whose vertices represent the initial region and the three subgoal regions, and the edges correspond to subtasks (labeled with a safety constraint that must be satisfied).

However, $\mathcal{G}_{\text{ex}}$ by itself is insufficient to determine the optimal path—e.g., it does not know that there is no path leading directly from S_2 to S_3, which is a property of the environment. These differences can be represented as (*a priori* unknown) edge costs in $\mathcal{G}_{\text{ex}}$. At a high level, DIRL trains a policy π_e for each edge e in $\mathcal{G}_{\text{ex}}$, and sets the cost of e to be $c(e;\pi_e) = -\log P(e;\pi_e)$, where $P(e;\pi_e)$ is the probability that π_e succeeds in achieving e. For instance, for the edge $S_0 \to S_1$, π_e is trained to reach S_1 from a random state in S_0 while avoiding O. Then, a naïve strategy for identifying the optimal path is to (i) train a policy π_e for each edge e, (ii) use it to estimate the edge cost $c(e;\pi_e)$,

and (iii) run Djikstra's algorithm with these costs.

One challenge is that π_e depends on the initial states used in its training—e.g., training π_e for $e = S_1 \to S_3$ requires a distribution over S_1. Using the wrong distribution can lead to poor performance due to distribution shift; furthermore, training a policy for all edges may unnecessarily waste effort training policies for unimportant edges. To address these challenges, DIRL interweaves training policies with the execution of Djikstra's algorithm, only training π_e once Djikstra's algorithm requires the cost of edge e. This strategy enables DIRL to scale to complex tasks; in our example, it quickly learns a policy that satisfies the specification with high probability. These design choices are validated empirically—as shown in Figure 5.1, DIRL quickly learns to achieve the specification, whereas it is beyond the reach of existing approaches.

Contributions. In summary, we make the following contributions.

- We propose a novel compositional algorithm to learn policies in continuous-state environments from complex high-level specifications that interleaves high-level model-based planning with low-level RL.
- We present a theoretical analysis of our algorithm showing that it aims to maximize a lower bound on the satisfaction probability of the specification.
- We perform an empirical evaluation demonstrating that our algorithm outperforms several state-of-the-art algorithms for learning from high-level specifications.

5.1.1. Problem Setting

Given an MDP[8] $\mathcal{M} = (S, A, \eta, P)$ with unknown transitions and a SPECTRL specification (see Section 4.2) φ, our goal is to compute a policy $\pi^* : \mathcal{Z}_f(S, A) \to A$ such that

$$\pi^* \in \arg\max_{\pi} \Pr_{\zeta \sim \mathcal{D}_\pi^{\mathcal{M}}}[\zeta \models \varphi],$$

[8]Continuous-state MDP with a distribution over initial states $\eta : S \to \mathbb{R}_{\geq 0}$ (i.e., $\eta(s)$ is the probability density of the initial state being s) instead of a fixed initial state s_0.

where $\mathcal{D}_\pi^{\mathcal{M}}$ is the distribution over infinite trajectories (runs) generated by π. Similar to the focus of the previous chapter, we want to learn a policy π^* that maximizes the probability that a sampled run ζ satisfies the specification φ. In the rest of the chapter, we use $\mathcal{Z}_f$ and $\mathcal{Z}$ to denote $\mathcal{Z}_f(S, A)$ and $\mathcal{Z}(S, A)$ respectively.

We consider the reinforcement learning setting in which we do not know the probabilities P but instead only have access to a simulator of $\mathcal{M}$. Typically, we can only sample trajectories of $\mathcal{M}$ starting at an initial state $s_0 \sim \eta$. Some parts of our algorithm are based on an assumption that we can sample trajectories starting at any state that has been observed before. For example, if taking action a_0 in s_0 leads to a state s_1, we can store s_1 and obtain future samples starting at s_1.

Assumption 5.1. *We can sample from $p(\cdot \mid s, a) = P(s, a, \cdot)$ for any previously observed state s and any action a.*

The rest of this chapter is organized as follows. In Section 5.2, we define the *abstract reachability* problem and reduce the policy synthesis problem for SPECTRL specifications to abstract reachability. Next, in Section 5.3, we present an algorithm for the abstract reachability problem. In Section 5.4, we present experiments that demonstrate that our approach is capable of learning policies to perform complex tasks in high-dimensional environments.

5.2. Abstract Reachability

In this section, we describe how to reduce the RL problem for a given MDP $\mathcal{M}$ and specification φ to a reachability problem on a directed acyclic graph (DAG) $\mathcal{G}_\varphi$, augmented with information connecting its edges to finite runs in $\mathcal{M}$. In Section 5.3, we describe how to exploit the compositional structure of $\mathcal{G}_\varphi$ to learn efficiently.

5.2.1. Abstract Reachability Problem

We begin by defining the *abstract reachability* problem, and describe how to reduce the problem of learning from a SPECTRL specification to abstract reachability. At a high level, abstract reachability is defined as a graph reachability problem over a directed acyclic graph (DAG) whose vertices correspond to *subgoal regions*—a subgoal region $X \subseteq S$ is a subset of the state space S. As

discussed below, in our reduction, these subgoal regions are derived from the given specification φ. The constructed graph structure also encodes the relationships between subgoal regions.

Definition 5.1. An *abstract graph* $\mathcal{G} = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$ is a directed acyclic graph (DAG) with vertices U, (directed) edges $E \subseteq U \times U$, initial vertex $u_0 \in U$, final vertices $F \subseteq U$, subgoal region map $\beta : U \to 2^S$ such that for each $u \in U$, $\beta(u)$ is a subgoal region,[9] and *safe trajectories* $\mathcal{Z}_{\texttt{safe}} = \bigcup_{e \in E} \mathcal{Z}^e_{\texttt{safe}}$, where $\mathcal{Z}^e_{\texttt{safe}} \subseteq \mathcal{Z}_f$ denotes the safe trajectories for edge $e \in E$.

Intuitively, (U, E) is a standard DAG, and u_0 and F define a graph reachability problem for (U, E). Furthermore, β and $\mathcal{Z}_{\texttt{safe}}$ connect (U, E) back to the original MDP $\mathcal{M}$; in particular, for an edge $e = u \to u'$, $\mathcal{Z}^e_{\texttt{safe}}$ is the set of trajectories in $\mathcal{M}$ that can be used to transition from $\beta(u)$ to $\beta(u')$.

Definition 5.2. An infinite trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ in $\mathcal{M}$ satisfies *abstract reachability* for $\mathcal{G}$ (denoted $\zeta \models \mathcal{G}$) if there is a sequence of indices $0 = i_0 \leq i_1 < \cdots < i_k$ and a path $\rho = u_0 \to u_1 \to \cdots \to u_k$ in $\mathcal{G}$ such that

- $u_k \in F$,
- for all $j \in \{0, \ldots, k\}$, we have $s_{i_j} \in \beta(u_j)$, and
- for all $j < k$, letting $e_j = u_j \to u_{j+1}$, we have $\zeta_{i_j : i_{j+1}} \in \mathcal{Z}^{e_j}_{\texttt{safe}}$.

The first two conditions state that the trajectory should visit a sequence of subgoal regions corresponding to a path from the initial vertex to some final vertex, and the last condition states that the trajectory should be composed of subtrajectories that are safe according to $\mathcal{Z}_{\texttt{safe}}$.

Definition 5.3. Given MDP $\mathcal{M}$ with unknown transitions and abstract graph $\mathcal{G}$, the *abstract reachability problem* is to compute a policy $\tilde{\pi} : \mathcal{Z}_f \to A$ such that $\tilde{\pi} \in \arg\max_\pi \Pr_{\zeta \sim \mathcal{D}^{\mathcal{M}}_\pi}[\zeta \models \mathcal{G}]$.

In other words, the goal is to find a policy for which the probability that a generated trajectory satisfies abstract reachability is maximized.

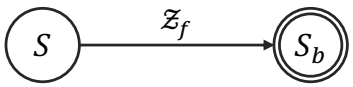


Figure 5.2: Abstract graph for `achieve` b.

5.2.2. Reduction to Abstract Reachability

In this section, we describe how to reduce the RL problem for a given MDP $\mathcal{M} = (S, A, \eta, P)$ and a specification φ to an abstract reachability problem for $\mathcal{M}$ by constructing an abstract graph $\mathcal{G}_\varphi$ inductively from φ. At a high level, the construction works as follows. First, for each predicate b, we define the corresponding subgoal region $S_b = \{s \in S \mid s \models b\}$ denoting the set of states at which b holds. Next, the abstract graph $\mathcal{G}_\varphi$ for $\varphi = \texttt{achieve}\ b$ is shown in Figure 5.2. All trajectories in $\mathcal{Z}_f$ are considered safe for the edge $e = u_0 \to u_1$ and the only final vertex is u_1 with $\beta(u_1) = S_b$. The abstract graph for a specification of the form $\varphi = \varphi_1\ \texttt{ensuring}\ b$ is obtained by taking the graph $\mathcal{G}_{\varphi_1}$ and replacing the set of safe trajectories $\mathcal{Z}^e_{\texttt{safe}}$, for each $e \in E$, with the set $\mathcal{Z}^e_{\texttt{safe}} \cap \mathcal{Z}_b$, where $\mathcal{Z}_b = \{\zeta \in \mathcal{Z}_f \mid \forall i\ .\ s_i \models b\}$ is the set of trajectories in which all states satisfy b. For the sequential specification $\varphi = \varphi_1; \varphi_2$, we construct $\mathcal{G}_\varphi$ by adding edges from every final vertex of $\mathcal{G}_{\varphi_1}$ to every vertex of $\mathcal{G}_{\varphi_2}$ that is a neighbor of its initial vertex. Finally, choice $\varphi = \varphi_1\ \texttt{or}\ \varphi_2$ is handled by merging the initial vertices of the graphs corresponding to the two sub-specifications. Figure 5.1 shows an example abstract graph. The labels on the vertices are regions in the environment. All trajectories that avoid hitting the obstacle O are safe for all edges.

Definitions. We start with some definitions that are needed for describing the full construction. Given two sets of finite trajectories $\mathcal{Z}_1, \mathcal{Z}_2 \subseteq \mathcal{Z}_f$, let us denote by $\mathcal{Z}_1 \circ \mathcal{Z}_2$ the concatenation of the two sets—i.e.,

$$\mathcal{Z}_1 \circ \mathcal{Z}_2 = \left\{ \zeta \in \mathcal{Z}_f \;\middle|\; \exists i < t\ .\ \zeta_{0:i} \in \mathcal{Z}_1\ \wedge\ \zeta_{(i+1):t} \in \mathcal{Z}_2 \right\}.$$

In addition to the abstract graph $\mathcal{G} = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$ we also construct a set of safe *terminal trajectories* $\mathcal{Z}_{\texttt{term}} = \bigcup_{u \in F} \mathcal{Z}^u_{\texttt{term}}$ where $\mathcal{Z}^u_{\texttt{term}} \subseteq \mathcal{Z}_f$ is the set of terminal trajectories for the final vertex $u \in F$. Now, we define what it means for a finite trajectory ζ to satisfy the pair $(\mathcal{G}, \mathcal{Z}_{\texttt{term}})$.

[9] We do not require that the subgoal regions partition the state space or that they be non-overlapping.

Definition 5.4. A finite trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t$ in $\mathcal{M}$ satisfies the pair $(\mathcal{G}, \mathcal{Z}_{\texttt{term}})$ (denoted $\zeta \models (\mathcal{G}, \mathcal{Z}_{\texttt{term}})$) if there is a sequence of indices $0 = i_0 \leq i_1 < \cdots < i_k \leq t$ and a path $\rho = u_0 \to u_1 \to \cdots \to u_k$ in $\mathcal{G}$ such that

- $u_k \in F$,
- for all $j \in \{0, \ldots, k\}$, we have $s_{i_j} \in \beta(u_j)$,
- for all $j < k$, letting $e_j = u_j \to u_{j+1}$, we have $\zeta_{i_j:i_{j+1}} \in \mathcal{Z}^{e_j}_{\texttt{safe}}$, and
- $\zeta_{i_k:t} \in \mathcal{Z}^{u_k}_{\texttt{term}}$.

We now outline the inductive construction of the pair $(\mathcal{G}_\varphi, \mathcal{Z}_{\texttt{term},\varphi})$ from a specification φ such that any finite trajectory $\zeta \in \mathcal{Z}_f$ satisfies φ if and only if ζ satisfies $(\mathcal{G}_\varphi, \mathcal{Z}_{\texttt{term},\varphi})$.

Objectives ($\varphi = \texttt{achieve}\ b$). The abstract graph is $\mathcal{G}_\varphi = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$ where

- $U = \{u_0, u_b\}$ with $\beta(u_0) = S$ and $\beta(u_b) = S_b = \{s \mid s \models b\}$,
- $E = \{u_0 \to u_b\}$,
- $F = \{u_b\}$ and,
- $\mathcal{Z}^{(u_0,u_b)}_{\texttt{safe}} = \mathcal{Z}^{u_b}_{\texttt{term}} = \mathcal{Z}_f$.

Constraints ($\varphi = \varphi_1\ \texttt{ensuring}\ b$). Let the abstract graph for φ_1 be $\mathcal{G}_{\varphi_1} = (U_1, E_1, u^1_0, F_1, \beta_1, \mathcal{Z}_{\texttt{safe},1})$ and the terminal trajectories be $\mathcal{Z}_{\texttt{term},1}$. Then, the abstract graph for φ is $\mathcal{G}_\varphi = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$ where

- $U = U_1$, $u_0 = u^1_0$, $E = E_1$ and $F = F_1$.
- $\beta(u) = \beta_1(u) \cap S_b$ for all $u \in U \setminus \{u_0\}$ where $S_b = \{s \mid s \models b\}$, and $\beta(u_0) = S$.
- $\mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}^e_{\texttt{safe},1} \cap \mathcal{Z}_b$ for all $e \in E$ where

$$\mathcal{Z}_b = \{\zeta \in \mathcal{Z}_f \mid \forall i \,.\, s_i \models b\}.$$

- $\mathcal{Z}^u_{\texttt{term}} = \mathcal{Z}^u_{\texttt{term},1} \cap \mathcal{Z}_b$ for all $u \in F$.

Sequencing ($\varphi = \varphi_1; \varphi_2$). Let the abstract graph for φ_i be $\mathcal{G}_{\varphi_i} = (U_i, E_i, u_0^i, F_i, \beta_i, \mathcal{Z}_{\texttt{safe},i})$ and the terminal trajectories be $\mathcal{Z}_{\texttt{term},i}$ for $i \in \{1, 2\}$. The abstract graph $\mathcal{G}_\varphi = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$ is constructed as follows.

- $U = U_1 \sqcup U_2 \setminus \{u_0^2\}$.
- $E = E_1 \sqcup E_2' \sqcup E_{1\to 2}$ where

$$E_2' = \{u \to u' \in E_2 \mid u \neq u_0^2\} \quad \text{and}$$

$$E_{1\to 2} = \{u^1 \to u^2 \mid u^1 \in F_1 \;\&\; u_0^2 \to u^2 \in E_2\}.$$

- $u_0 = u_0^1$ and $F = F_2$.
- $\beta(u) = \beta_i(u)$ for all $u \in U_i$ and $i \in \{1, 2\}$.
- The safe trajectories are given by
  - $\mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}^e_{\texttt{safe},1}$ for all $e \in E_1$,
  - $\mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}^e_{\texttt{safe},2}$ for all $e \in E_2'$ and,
  - $\mathcal{Z}^{u^1\to u^2}_{\texttt{safe}} = \mathcal{Z}^{u^1}_{\texttt{term},1} \circ \mathcal{Z}^{u_0^2\to u^2}_{\texttt{safe},2}$ for all $u^1 \to u^2 \in E_{1\to 2}$.
- $\mathcal{Z}^u_{\texttt{term}} = \mathcal{Z}^u_{\texttt{term},2}$ for all $u \in F$.

Choice ($\varphi = \varphi_1$ `or` φ_2). Let the abstract graph for φ_i be $\mathcal{G}_{\varphi_i} = (U_i, E_i, u_0^i, F_i, \beta_i, \mathcal{Z}_{\texttt{safe},i})$ and the terminal trajectories be $\mathcal{Z}_{\texttt{term},i}$ for $i \in \{1, 2\}$. The abstract graph for φ is $\mathcal{G}_\varphi = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$ where:

- $U = \left(U_1 \setminus \{u_0^1\}\right) \sqcup \left(U_2 \setminus \{u_0^2\}\right) \sqcup \{u_0\}$.

- $E = E_1' \sqcup E_2' \sqcup E_0$ where

$$E_i' = \{u \to u' \in E_i \mid u \neq u_0^i\} \quad \text{and}$$

$$E_0 = \{u_0 \to u^i \mid i \in \{1,2\} \;\&\; u_0^i \to u^i \in E_i\}.$$

- $F = F_1 \sqcup F_2$.
- $\beta(u) = \beta_i(u)$ for all $u \in U_i$, $i \in \{1,2\}$ and $\beta(u_0) = S$.
- The safe trajectories are given by
  - $\mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}^e_{\texttt{safe},i}$ for all $e \in E_i'$ and $i \in \{1,2\}$,
  - $\mathcal{Z}^{u_0 \to u^i}_{\texttt{safe}} = \mathcal{Z}^{u_0^i \to u^i}_{\texttt{safe},i}$ for all $u_0 \to u^i \in E_0$ with $u^i \in U_i$.
- $\mathcal{Z}^u_{\texttt{term}} = \mathcal{Z}^u_{\texttt{term},i}$ for all $u \in F_i$ and $i \in \{1,2\}$.

The constructed pair $(\mathcal{G}_\varphi, \mathcal{Z}_{\texttt{term},\varphi})$ has the following important properties which enable us to justify the reduction.

Lemma 5.5. *For any* SPECTRL *specification φ, the following hold.*

- *For any finite trajectory $\zeta \in \mathcal{Z}_f$, $\zeta \models \varphi$ if and only if $\zeta \models (\mathcal{G}_\varphi, \mathcal{Z}_{\texttt{term},\varphi})$.*
- *For any final vertex u of $\mathcal{G}_\varphi$ and any state $s \in \beta(u)$, the length-1 trajectory $\zeta = s$ is contained in $\mathcal{Z}^u_{\texttt{term},\varphi}$.*

Proof. Follows from the above construction by structural induction on φ. □

We now have the following key guarantee.

Theorem 5.6. *Given a* SPECTRL *specification φ, we can construct an abstract graph $\mathcal{G}_\varphi$ such that, for every infinite trajectory $\zeta \in \mathcal{Z}$, we have $\zeta \models \varphi$ if and only if $\zeta \models \mathcal{G}_\varphi$. Furthermore, the number*

of vertices in $\mathcal{G}_\varphi$ is $O(|\varphi|)$ where $|\varphi|$ is the size of the specification φ.

Proof. Let $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ be an infinite trajectory. First we show that $\zeta \models \varphi$ if and only if $\zeta \models \mathcal{G}_\varphi$.

($\implies$) Suppose $\zeta \models \varphi$. Then, there is a $t \geq 0$ such that $\zeta_{0:t} \models \varphi$. From Lemma 5.5, we get that $\zeta_{0:t} \models (\mathcal{G}_\varphi, \mathcal{Z}_{\texttt{term},\varphi})$ which implies that $\zeta \models \mathcal{G}_\varphi$.

($\impliedby$) Suppose $\zeta \models \mathcal{G}_\varphi$. Then, let $0 = i_0 \leq i_1 < \cdots < i_k$ be a sequence of indices realizing a path $u_0 \to \cdots \to u_k$ to a final vertex u_k in $\mathcal{G}_\varphi$. Since $s_{i_k} \in \beta(u_k)$, from Lemma 5.5 we have $\zeta_{i_k:i_k} \in \mathcal{Z}^{u_k}_{\texttt{term},\varphi}$ and hence $\zeta_{0:i_k} \models (\mathcal{G}_\varphi, \mathcal{Z}_{\texttt{term},\varphi})$. From Lemma 5.5, we conclude that $\zeta_{0:i_k} \models \varphi$ and therefore $\zeta \models \varphi$.

Next, it follows by a straightforward induction on φ that the number of vertices in $\mathcal{G}_\varphi$ is at most $|\varphi| + 1$ where $|\varphi|$ is the number of operators (`achieve`, `ensuring`, ;, `or`) in φ. □

As a consequence, we can solve the reinforcement learning problem for φ by solving the abstract reachability problem for $\mathcal{G}_\varphi$. As described below, we leverage the structure of $\mathcal{G}_\varphi$ in conjunction with reinforcement learning to do so.

5.3. Compositional Reinforcement Learning

In this section, we propose a compositional approach for learning a policy to solve the abstract reachability problem for MDP $\mathcal{M}$ (with unknown transition probabilities) and abstract graph $\mathcal{G}$.

5.3.1. Overview

At a high level, our algorithm proceeds in three steps:

- For each edge $e = u \to u'$ in $\mathcal{G}$, use RL to learn a neural network (NN) policy π_e to try and transition the system from any state $s \in \beta(u)$ to some state $s' \in \beta(u')$ in a safe way according to $\mathcal{Z}^e_{\text{safe}}$. Importantly, this step requires a distribution η_u over initial states $s \in \beta(u)$.

- Use sampling to estimate the probability $P(e; \pi_e, \eta_u)$ that π_e safely transitions from $\beta(u)$ to $\beta(u')$ when the distribution over initial states in $\beta(u)$ is given by η_u.

Algorithm 2 Compositional reinforcement learning algorithm for solving abstract reachability.

function DIRL($\mathcal{M}$, $\mathcal{G}$)
  Initialize processed vertices $U_p \leftarrow \varnothing$
  Initialize $\Gamma_{u_0} \leftarrow \{u_0\}$, and $\Gamma_u \leftarrow \varnothing$ for $u \neq u_0$
  Initialize edge policies $\Pi \leftarrow \varnothing$
  while true do
    $u \leftarrow$ NEARESTVERTEX($U \setminus U_p, \Gamma, \Pi$)
    $\rho_u \leftarrow$ SHORTESTPATH(Γ_u)
    $\eta_u \leftarrow$ REACHDISTRIBUTION(ρ_u, Π)
    if $u \in F$ **then return** PATHPOLICY(ρ_u, Π)
    for $e = u \to u' \in$ Outgoing(u) **do**
      $\pi_e \leftarrow$ LEARNPOLICY(e, η_u)
      Add $\rho_u \circ e$ to $\Gamma_{u'}$ and π_e to Π
    end for
    Add u to U_p
  end while
end function

- Use Djikstra's algorithm in conjunction with the edge costs $c(e) = -\log(P(e; \pi_e, \eta_u))$ to compute a path $\rho^* = u_0 \to u_1 \to \cdots \to u_k$ in $\mathcal{G}$ that minimizes $c(\rho) = -\sum_{j=0}^{k-1} \log(P(e_j; \pi_j, \eta_j))$, where $e_j = u_j \to u_{j+1}$, $\pi_j = \pi_{e_j}$, and $\eta_j = \eta_{u_j}$.

Then, we could choose π to be the sequence of policies $\pi_1, ..., \pi_{k-1}$—i.e., execute each policy π_j until it reaches $\beta(u_{j+1})$, and then switch to π_{j+1}.

There are two challenges that need to be addressed in realizing this approach effectively. First, it is unclear what distribution to use as the initial state distribution η_u to train π_e. Second, it might be unnecessary to learn all the policies since a subset of the edges might be sufficient for the reachability task. Our algorithm (Algorithm 2) addresses these issues by lazily training π_e—i.e., only training π_e when the edge cost $c(e)$ is needed by Djikstra's algorithm.

In more detail, DIRL iteratively processes vertices in $\mathcal{G}$ starting from the initial vertex u_0, continuing until it processes a final vertex $u \in F$. It maintains the property that for every u it processes, it has already trained policies for all edges along some path ρ_u from u_0 to u. This property is satisfied by u_0 since there is a path of length zero from u_0 to itself. In Algorithm 2, Γ_u is the set of all paths from u_0 to u discovered so far, $\Gamma = \bigcup_u \Gamma_u$, and $\Pi = \{\pi_e \mid e = u \to u' \in E, u \in U_p\}$ is the set of all

edge policies trained so far. In each iteration, DIRL processes an unprocessed vertex u nearest to u_0, which it discovers using NEARESTVERTEX, and performs the following steps:

1. SHORTESTPATH selects the shortest path from u_0 to u in Γ_u, denoted $\rho_u = u_0 \to \cdots \to u_k = u$.

2. REACHDISTRIBUTION computes the distribution η_u over states in $\beta(u)$ induced by using the sequence of policies $\pi_{e_0}, ..., \pi_{e_{k-1}} \in \Pi$, where $e_j = u_j \to u_{j+1}$ are the edges in ρ_u.

3. For every edge $e = u \to u'$, LEARNPOLICY learns a policy π_e for e using η_u as the initial state distribution, and adds π_e to Π and $\rho_{u'}$ to $\Gamma_{u'}$, where $\rho_{u'} = u_0 \to \cdots \to u \to u'$; π_e is trained to ensure that the resulting trajectories from $\beta(u)$ to $\beta(u')$ are in $\mathcal{Z}^e_{\texttt{safe}}$ with high probability.

5.3.2. Definitions and Notation

Edge costs. We begin by defining the edge costs used in Djikstra's algorithm. Given a policy π_e for edge $e = u \to u'$, and an initial state distribution η_u over the subgoal region $\beta(u)$, the cost $c(e)$ of e is the negative log probability that π_e safely transitions the system from $s_0 \sim \eta_u$ to $\beta(u')$. First, we say a trajectory ζ starting at s_0 *achieves* an e if it safely reaches $\beta(u')$—formally:

Definition 5.7. An infinite trajectory $\zeta = s_0 \to s_1 \to \cdots$ *achieves* edge $e = u \to u'$ in $\mathcal{G}$ (denoted $\zeta \models e$) if (i) $s_0 \in \beta(u)$, and (ii) there exists i (constrained to be positive if $u \neq u_0$) such that $s_i \in \beta(u')$ and $\zeta_{0:i} \in \mathcal{Z}^e_{\texttt{safe}}$; we denote the smallest such i by $i(\zeta, e)$.

Then, the probability that π achieves e from an initial state $s_0 \sim \eta_u$ is

$$P(e; \pi_e, \eta_u) = \Pr_{s_0 \sim \eta_u, \zeta \sim \mathcal{D}_{\pi_e, s_0}} [\zeta \models e],$$

where $\mathcal{D}_{\pi_e, s_0}$ is the distribution over infinite trajectories induced by using π_e from initial state s_0. Finally, the cost of edge e is $c(e) = -\log P(e; \pi_e, \eta_u)$. Note that $c(e)$ is nonnegative for any edge e.

Path policies. Given edge policies Π along with a path $\rho = u_0 \to u_1 \to \cdots \to u_k = u$ in $\mathcal{G}$, we define a *path policy* π_ρ to navigate from $\beta(u_0)$ to $\beta(u)$. In particular, π_ρ executes $\pi_{u_j \to u_{j+1}}$ (starting from $j = 0$) until reaching $\beta(u_{j+1})$, after which it increments $j \leftarrow j + 1$ (unless $j = k$). That is, π_ρ

is designed to achieve the sequence of edges in ρ. Note that π_ρ is stateful since it internally keeps track of the index j of the current policy.

Induced distribution. Let path $\rho = u_0 \to \cdots \to u_k = u$ from u_0 to u be such that edge policies for all edges along the path have been trained. The induced distribution η_ρ is defined inductively on the length of ρ. Formally, for the zero length path $\rho = u_0$ (so $u = u_0$), we define $\eta_\rho = \eta$ to be the initial state distribution of the MDP $\mathcal{M}$. Otherwise, we have $\rho = \rho' \circ e$, where $e = u' \to u$. Then, we define η_ρ to be the state distribution over $\beta(u)$ induced by using π_e from $s_0 \sim \eta_{\rho'}$ conditioned on $\zeta \models e$. Formally, η_ρ is the probability distribution over $\beta(u)$ such that for a set of states $S' \subseteq \beta(u)$, the probability of S' according to η_ρ is

$$\Pr_{s \sim \eta_\rho}[s \in S'] = \Pr_{s_0 \sim \eta_{\rho'}, \zeta \sim \mathcal{D}_{\pi_e, s_0}} \left[s_{i(\zeta,e)} \in S' \mid \zeta \models e \right].$$

Path costs. The cost of a path $\rho = u_0 \to \cdots \to u_k = u$ is $c(\rho) = -\sum_{j=0}^{k-1} \log P(e_j; \pi_{e_j}, \eta_{\rho_{0:j}})$ where $e_j = u_j \to u_{j+1}$ is the j-th edge in ρ, and $\rho_{0:j} = u_0 \to \cdots \to u_j$ is the j-th prefix of ρ.

5.3.3. Algorithm Details

DIRL interleaves Djikstra's algorithm with using RL to train policies π_e. Note that the edge weights to run Dijkstra's are not given *a priori* since the edge policies and initial state/induced distributions are unknown. Instead, they are computed on-the-fly beginning from the subgoal region u_0 using Algorithm 2. We describe each subprocedure below.

Processing order (NEARESTVERTEX). On each iteration, DIRL chooses the vertex u to process next to be an unprocessed vertex that has the shortest path from u_0—i.e.,

$$u \in \arg\min_{u' \in U \setminus U_p} \min_{\rho \in \Gamma_{u'}} c(\rho).$$

This choice is an important part of Djikstra's algorithm. For a graph with fixed costs, it ensures that the computed path ρ_u to each vertex u is minimized. While the costs in our setting are not fixed since they depend on η_u, this strategy remains an effective heuristic.

Shortest path computation (ShortestPath). This subroutine returns a path of minimum cost, $\rho_u \in \arg\min_{\rho \in \Gamma_u} c(\rho)$. These costs can be estimated using Monte Carlo sampling.

Initial state distribution (ReachDistribution). A key choice DiRL makes is what initial state distribution η_u to choose to train policies π_e for outgoing edges $e = u \to u'$. DiRL chooses the initial state distribution $\eta_u = \eta_{\rho_u}$ to be the distribution of states reached by the path policy π_{ρ_u} from a random initial state $s_0 \sim \eta$.[10]

Learning an edge policy (LearnPolicy). Now that the initial state distribution η_u is known, we describe how DiRL learns a policy π_e for a single edge $e = u \to u'$. At a high level, it trains π_e using a standard RL algorithm, where the rewards $\mathbb{1}(\zeta \models e)$ are designed to encourage π_e to safely transition the system to a state in $\beta(u')$. To be precise, DiRL uses RL to compute $\pi_e \in \arg\max_\pi P(e; \pi, \eta_u)$. Shaped rewards can be used to improve learning; see subsection 5.3.4.

Constructing a path policy (PathPolicy). Given edge policies Π along with a path $\rho = u_0 \to \cdots \to u$, where $u \in F$ is a final vertex, DiRL returns the path policy π_ρ.

Theoretical Guarantee. We guarantee that minimizing the path cost $c(\rho)$ corresponds to maximizing a lower bound on the objective of the abstract reachability problem. Formally, we have the following theorem.

Theorem 5.8. *Given a path policy π_ρ corresponding to a path $\rho = u_0 \to \cdots \to u_k = u$, where $u \in F$, we have* $\Pr_{\zeta \sim \mathcal{D}^{\mathcal{M}}_{\pi_\rho}}[\zeta \models \mathcal{G}] \geq \exp(-c(\rho))$.

Proof. Let the abstract graph be $\mathcal{G} = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$. Let us first define what it means for a rollout to achieve a path in $\mathcal{G}$.

Definition 5.9. *We say that an infinite trajectory ζ achieves the path ρ (denoted $\zeta \models \rho$) if $\zeta \models \mathcal{G}_\rho$ where $\mathcal{G}_\rho = (U_\rho, E_\rho, u_0, \{u_k\}, \beta \downarrow \rho, \mathcal{Z}_{\texttt{safe}} \downarrow \rho)$ with $U_\rho = \{u_j \mid 0 \leq j \leq k\}$, $E_\rho = \{u_j \to u_{j+1} \mid 0 \leq$*

[10] This choice is the distribution of states reaching u by the path policy π_ρ eventually returned by DiRL. Thus, it ensures that the training and test distributions for edge policies in π_ρ are equal.

$j < k\}$ *and* $\beta \downarrow \rho$ *and* $\mathcal{Z}_{\texttt{safe}} \downarrow_\rho$ *are* β *and* $\mathcal{Z}_{\texttt{safe}}$ *restricted to the vertices and the edges of* $\mathcal{G}_\rho$*, respectively.*

From the definition it is clear that for any infinite trajectory ζ, if $\zeta \models \rho$ then $\zeta \models \mathcal{G}$ and therefore

$$\Pr_{\zeta \sim \mathcal{D}^{\mathcal{M}}_{\pi_\rho}}[\zeta \models \mathcal{G}] \geq \Pr_{\zeta \sim \mathcal{D}^{\mathcal{M}}_{\pi_\rho}}[\zeta \models \rho]. \tag{5.1}$$

Let us now define a slightly stronger notion of achieving an edge.

Definition 5.10. *An infinite trajectory* $\zeta = s_0 \to s_1 \to \cdots$ *is said to greedily achieve the path* ρ *(denoted* $\zeta \models_g \rho$*) if there is a sequence of indices* $0 = i_0 \leq i_1 < \cdots < i_k$ *such that for all* $j < k$*,*

- $\zeta_{i_j:\infty} \models e_j = u_j \to u_{j+1}$ *and,*
- $i_{j+1} = i(\zeta_{i_j:\infty}, e_j)$*,*

where $\zeta_{i_j:\infty} = s_{i_j} \to s_{i_j+1} \to \cdots$*.*

That is, $\zeta \models_g \rho$ if a partition of ζ realizing ρ can be be constructed greedily by picking i_{j+1} to be the smallest index $i \geq i_j$ (strictly bigger if $j > 0$) such that $s_i \in \beta(u_{j+1})$ and $\zeta_{i_j:i} \in \mathcal{Z}^{e_j}_{\texttt{safe}}$. Since $\zeta \models_g \rho$ implies $\zeta \models \rho$, we have

$$\Pr_{\zeta \sim \mathcal{D}^{\mathcal{M}}_{\pi_\rho}}[\zeta \models \rho] \geq \Pr_{\zeta \sim \mathcal{D}^{\mathcal{M}}_{\pi_\rho}}[\zeta \models_g \rho]. \tag{5.2}$$

Let $\rho_{j:k}$ denote the j-th suffix of ρ. We can decompose the probability $\Pr_{\zeta \sim \mathcal{D}_{\pi_\rho}}[\zeta \models_g \rho]$ as follows.

$$\begin{aligned}
\Pr_{\zeta \sim \mathcal{D}_{\pi_\rho}}[\zeta \models_g \rho] &= \Pr_{\zeta \sim \mathcal{D}_{\pi_\rho}}[\zeta \models e_0 \ \wedge \ \zeta_{i(\zeta,e_0):\infty} \models_g \rho_{1:k}] \\
&= \Pr_{\zeta \sim \mathcal{D}_{\pi_{e_0}}}[\zeta \models e_0] \cdot \Pr_{\zeta \sim \mathcal{D}_{\pi_\rho}}[\zeta_{i(\zeta,e_0):\infty} \models_g \rho_{1:k} \mid \zeta \models e_0] \\
&= P(e_0; \pi_{e_0}, \eta_0) \cdot \Pr_{s_0 \sim \eta_{\rho_{0:1}}, \zeta \sim \mathcal{D}_{\pi_{\rho_{1:k}}, s_0}}[\zeta \models_g \rho_{1:k}]
\end{aligned}$$

where the last equality followed from the definition of $\eta_{\rho_{0:1}}$ and the Markov property of $\mathcal{M}$. Applying the above decomposition recursively, we get

$$\begin{aligned}
\Pr_{\zeta \sim \mathcal{D}_{\pi_\rho}}[\zeta \models_g \rho] &= \prod_{j=0}^{k-1} P(e_j; \pi_{e_j}, \eta_{\rho_{0:j}}) \\
&= \exp(\log(\prod_{j=0}^{k-1} P(e_j; \pi_{e_j}, \eta_{\rho_{0:j}}))) \\
&= \exp(-(-\sum_{j=0}^{k-1} \log P(e_j; \pi_{e_j}, \eta_{\rho_{0:j}}))) \\
&= \exp(-c(\rho)).
\end{aligned}$$

Therefore, from Equations 5.1 and 5.2, we get the required bound. □

5.3.4. Shaped Rewards for Learning Edge Policies

To improve learning, we use shaped rewards for learning each edge policy π_e. To enable reward shaping, we assume that the atomic predicates additionally have a *quantitative semantics*—i.e., each atomic predicate $p \in \mathcal{P}_0$ is associated with a function $[\![p]\!]_q : S \to \mathbb{R}$. To ensure compatibility with the Boolean semantics, we assume that

$$[\![p]\!](s) = \big([\![p]\!]_q(s) > 0\big). \tag{5.3}$$

For example, given a state $s \in S$, the atomic predicate

$$[\![\texttt{reach } s]\!]_q(s') \;=\; 1 - \|s' - s\|$$

indicates whether the system is in a state near s w.r.t. some norm $\|\cdot\|$. Recall from the previous chapter that we can extend the quantitative semantics to predicates $b \in \mathcal{P}$ by recursively defining $[\![b_1 \wedge b_2]\!]_q(s) = \min\{[\![b_1]\!]_q(s), [\![b_2]\!]_q(s)\}$ and $[\![b_1 \vee b_2]\!]_q(s) = \max\{[\![b_1]\!]_q(s), [\![b_2]\!]_q(s)\}$. This preserves (5.3)—i.e., $b \models s$ if and only if $[\![b]\!]_q(s) > 0$.

In addition to quantitative semantics, we make use of the following property to define shaped

rewards.

Lemma 5.11. *The abstract graph $\mathcal{G}_\varphi = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$ of a specification φ satisfies the following:*

- *For every non-initial vertex $u \in U \setminus \{u_0\}$, there is a predicate $b \in \mathcal{P}$ such that $\beta(u) = S_b = \{s \mid s \models b\}$.*
- *For every $e \in E$, either $\mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}_b = \{\zeta \in \mathcal{Z} \mid \forall i \,.\, s_i \models b\}$ for some $b \in \mathcal{P}$ or $\mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}_{b_1} \circ \mathcal{Z}_{b_2}$ for some $b_1, b_2 \in \mathcal{P}$.*

Proof sketch. We prove a stronger property that, in addition to the above, requires that for any $e = u_0 \to u \in E$, $\mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}_b$ for some $b \in \mathcal{P}$ and for any final vertex u, $\mathcal{Z}^u_{\texttt{term},\varphi} = \mathcal{Z}_b$ for some $b \in \mathcal{P}$. This stronger property follows from a straightforward induction on φ. □

We are now ready to define shaped rewards that can be used to train a policy π_e for an edge $e = u \to u'$ in $\mathcal{G}_\varphi$. The rewards are given by

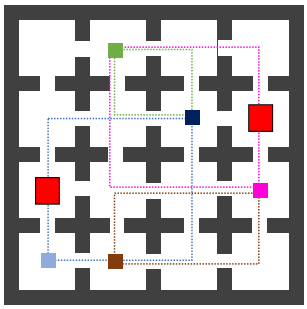
$$R_{\text{step}}(s, a, s') = R_{\text{reach}}(s, a, s') + R_{\text{safe}}(s, a, s').$$

Intuitively, the first term encodes a reward for reaching $\beta(u')$, and the second term encodes a reward for maintaining safety. By Lemma 5.11, $\beta(u') = S_b$ for some $b \in \mathcal{P}$. Then, we define

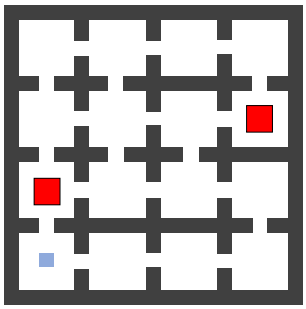
$$R_{\text{reach}}(s, a, s') = [\![b]\!]_q(s').$$

The safety reward is defined by

$$R_{\text{safe}}(s, a, s') = \begin{cases} \min\{0, [\![b]\!]_q(s')\} & \text{if } \mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}_b \\ \min\{0, [\![b \vee b']\!]_q(s')\} & \text{if } \mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}_b \circ \mathcal{Z}_{b'} \;\&\; \psi_b \\ \min\{0, [\![b']\!]_q(s')\} & \text{if } \mathcal{Z}^e_{\texttt{safe}} = \mathcal{Z}_b \circ \mathcal{Z}_{b'} \;\&\; \neg\psi_b. \end{cases}$$



(a) 16-Rooms: All doors open



(b) 16-Rooms: Some doors open

Figure 5.3: 16-Rooms Environments. Blue square indicates the initial room. Red squares represent obstacles. (a) illustrates the segments in the specifications.

Here, ψ_b is an internal state keeping track of whether b has held so far—i.e., $\psi_b \leftarrow \psi_b \wedge [\![b]\!](s)$ at state s. Intuitively, the first case is the simpler case, which checks if every state in the trajectory satisfies b, and the latter two cases handle a sequence where b should hold for the first part of the trajectory, and b' should hold for the remainder.

5.4. Experiments

We implemented our approach in a tool called DIRL[11] and empirically evaluated our tool on two classes of continuous control environments, namely, the Rooms and the Fetch environments.

5.4.1. Rooms Environment

We considered environments with several interconnected rooms. The rooms are separated by thick walls and are connected through bi-directional doors. The environments are a 9-Rooms environment, (Figure 5.1), a 16-Rooms environment with all doors open (Figure 5.3a), and a 16-Rooms environment with some doors open (Figure 5.3b). The red blocks indicate obstacles. A robot can pass through those rooms by moving around the red blocks. The robot is initially placed randomly in the center of the room with the blue box (bottom-left corner). The robot has a state $(x, y) \in \mathbb{R}^2$ encoding its 2D position. At any step, it can perform an action $(v, \theta) \in \mathbb{R}^2$ (encoding speed and direction) which leads to a transition $s' \sim \mathcal{N}(s + (v\cos(\theta), v\sin(\theta)), \delta I)$ where $\delta > 0$.

Rooms are identified by a tuple (r, c) denoting the room in the r-th row and c-th column. We use

[11] Our implementation is available at https://github.com/keyshor/dirl.

the convention that the bottom-left corner is room (0,0). Predicate `reach` (r, c) is interpreted as reaching the center of the (r, c)-th room and predicate `avoid` (r, c) is interpreted as avoiding the center of the (r, c)-th room. For clarity, we omit the word `achieve` from specifications of the form `achieve` b denoting such a specification using just the predicate b.

Specifications in the 9-Rooms Environment

1. $\varphi_1 :=$ `reach` $(2, 0)$; `reach` $(0, 0)$

   This specification is difficult for standard RL algorithms that do not store whether the first subtask has been achieved. In these cases, a stateless policy will not be able to determine whether to move upwards or downwards. In contrast, DIRL (as well as SPECTRL and RM based approaches) augments the state space to automatically keep track of which subtasks have been achieved so far.

2. $\varphi_2 :=$ `reach` $(2, 0)$ `or` `reach` $(0, 2)$

3. $\varphi_3 := \varphi_2$; `reach` $(2, 2)$

   This specification combines two choices of similar difficulty yet only one is favorable to fulfilling the specification since the direct path to the top-right corner from the bottom-right one is obstructed by walls.

4. $\varphi_4 :=$ `reach` $(2, 0)$ `ensuring avoid` $(1, 0)$

5. $\varphi_5 := \varphi_4$ `or` `reach` $(0, 2)$; `reach` $(2, 2)$

   This specification is similar to φ_3 except that the choices are of unequal difficulty due to the placement of the red obstacle. In this case, the non-greedy choice is favorable for completing the task.

Specifications in the 16-Rooms Environments

We describe the five specifications used in the 16-rooms environments, which are designed to increase in difficulty. First, we define a *segment* as the following specification: Given the current location of

the agent, the goal is to reach a room diagonally opposite to it by visiting at least one of the rooms at the remaining two corners of the rectangle formed by the current room and the goal room—e.g., in the 9-Rooms environment, to visit S_3 from the initial room, the agent must visit either S_1 or S_2 first. Then, we design specifications of varying sizes by sequencing several segments one after the other. In addition, the agent must always avoid the obstacles in the environment. We studied five such specifications, one half-segment and specifications up to four segments (φ_1 to φ_5), as illustrated in Figure 5.3a and described below.

1. φ_1 corresponds to a *half-segment*—i.e., φ_1 is simply a choice between (0,2) and (2,0).

2. φ_2 is the first segment that goes from (0,0) to (2,2)

3. φ_3 augments φ_2 with a second segment from (2,2) to (3,1).

4. φ_4 augments φ_3 with a segment from (3,1) to (1,3)

5. φ_5 augments φ_4 with a segment from (1,3) to (0,1)

5.4.2. Fetch Environment

We also evaluated our approach in the Fetch-Pick-And-Place environment in OpenAI Gym [31], consisting of a robotic arm that can grasp objects as well as a cuboidal block to manipulate (visualized in Figure 5.4). The state space is $\mathbb{R}^{25}$, which includes components encoding the gripper position, the (relative) position of the block, and the distance between the gripper fingers. The action space is $\mathbb{R}^4$, where the first 3 components encode the target gripper position and the last encodes the target gripper width. The block's initial position is a random location on a table. Let us denote by $s_r = (s_r^x, s_r^y, s_r^z) \in \mathbb{R}^3$ the position of the gripper, $s_o \in \mathbb{R}^3$ the relative position of the object (black block) w.r.t. the gripper, $s_g \in \mathbb{R}^3$ the goal location (red sphere) and $s_w \in \mathbb{R}$ the width of the gripper. Let c denote the width of the object and $z_\epsilon = (0, 0, \epsilon + c)$ for $\epsilon > 0$. Then, we define the following predicates.

- *NearObj* holds true in states in which the gripper is wide open, aligned with the object and

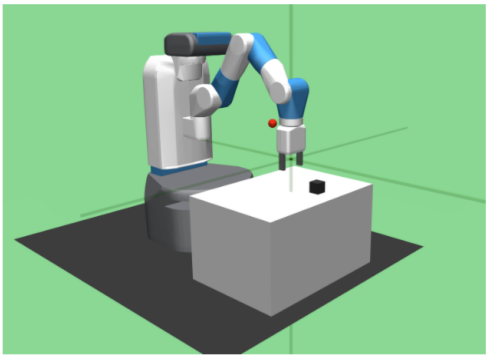


Figure 5.4: Fetch robotic arm.

is slightly above the object.

$$\text{NearObj}(s) = \left(\|s_o + z_\epsilon\|_2^2 + (s_w - 2c)^2 < \delta_1\right)$$

- *HoldingObj* holds true in states in which the gripper is close to the object and its width is close to the object's width.

$$\text{HoldingObj}(s) = \left(\|s_o\|_2^2 + (s_w - c)^2 < \delta_2\right)$$

- *LiftedObj* holds true in states in which the object is above the surface level of the table.

$$\text{LiftedObj}(s) = \left(s_r^z + s_o^z > \delta_3\right)$$

- *ObjAt*$[g]$ holds true in states in which the object is close to g.

$$\text{ObjAt}[g](s) = \left(\|s_r + s_o - g\|_2^2 < \delta_4\right)$$

Then the specifications we studied are the following.[12]

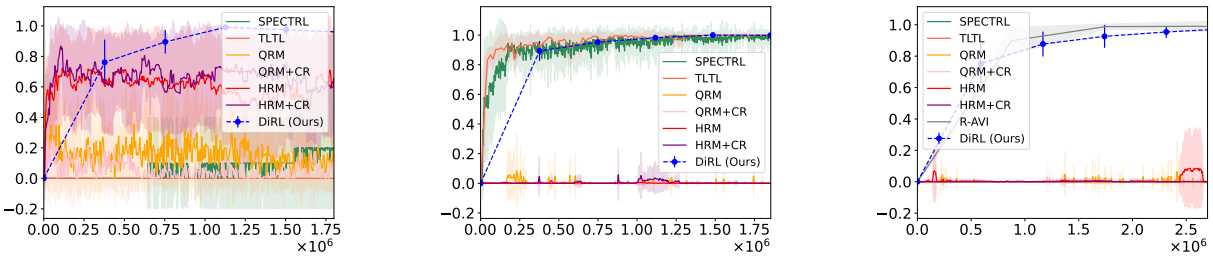
[12]We denote `achieve` b using just the predicate b.

- PickAndPlace: NearObj; HoldingObj; LiftedObj; ObjAt[s_g].
- PickAndPlaceStatic: NearObj; HoldingObj; LiftedObj; ObjAt[g_1] where g_1 is a fixed goal.
- PickAndPlaceChoice: $\big($NearObj; HoldingObj; LiftedObj$\big)$; $\big($(ObjAt[g_1]; ObjAt[g_2]) `or` (ObjAt[g_3]; ObjAt[g_4]$\big)$.

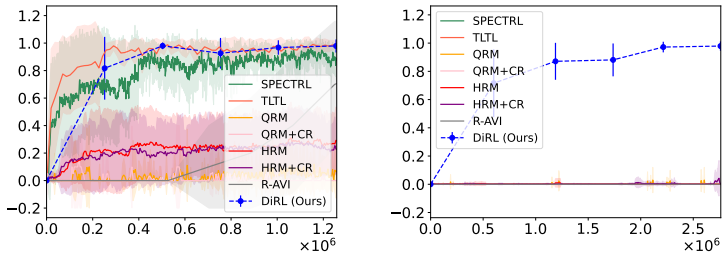
Baselines. We compared our approach to four state-of-the-art algorithms for learning from specifications, SPECTRL [84], QRM [76], HRM [77], and a TLTL [102] based approach, as well as a state-of-the-art hierarchical RL algorithm, R-AVI [86], that leverages state abstractions. We used publicly available implementations of SPECTRL, QRM, HRM and R-AVI. For QRM and HRM, we manually encoded the tasks as reward machines with continuous rewards. The variants QRM+CR and HRM+CR use counterfactual reasoning to reuse samples during training. Our implementation of TLTL uses the quantitative semantics defined in [102] with ARS to learn a single policy for each task. We used the subgoal regions and the abstract graph generated by our algorithm as inputs to R-AVI. Since R-AVI only supports disjoint subgoal regions and furthermore assumes the ability to sample from any subgoal region, we only ran R-AVI on supported benchmarks. The learning curves for R-AVI denote the probability of reaching the final goal region in the y-axis which is an upper bound on the probability of satisfying the specification.

Setup. Our tool learns the low-level NN policies for edges using an off-the-shelf RL algorithm. For the Rooms environment, we trained these policies using ARS [111] with shaped rewards; each one is a fully connected NN with 2 hidden layers of 30 neurons each. For the Fetch environment, we trained policies using TD3 [50] with shaped rewards; each one is a fully connected NN with 2 hidden layers of 256 neurons each.

For each specification in an environment, we first construct its abstract graph. In DIRL, each edge policy π_e is trained using k episodes of interactions with the environment. For the purpose of generating a learning curve, we run DIRL for each specification with several values of k. For each k value, we plot the sum total of the samples taken to train all edge policies against the probability with which the computed policy reaches a final subgoal region. For a fair comparison with the baselines, if each episode for learning an edge policy in DIRL is run for m steps, we run the episodes



(a) Specification φ_1 (b) Specification φ_2 (c) Specification φ_3

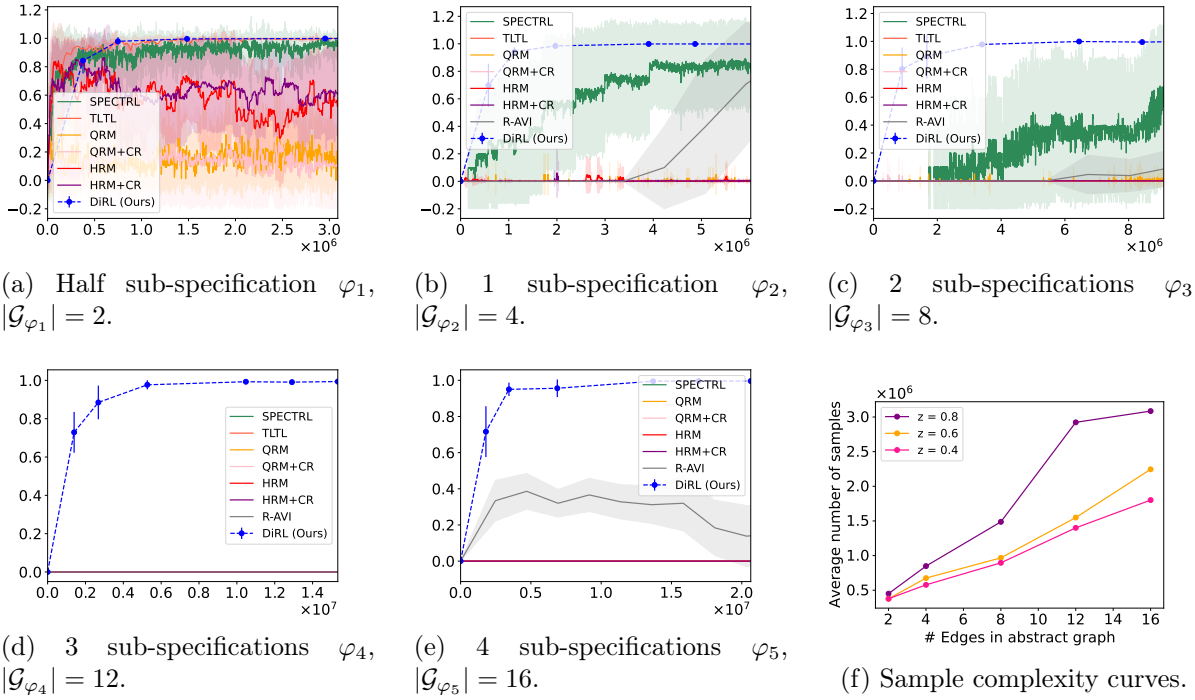


(d) Specification φ_4 (e) Specification φ_5

Figure 5.5: Learning curves for 9-Rooms environment with different specifications. x-axis denotes the number of samples (steps) and y-axis denotes the estimated probability of success. Results are averaged over 10 runs with error bars indicating $\pm$ standard deviation.

of the baselines for $m \cdot d + c$ steps, where d is the maximum path length to reach a final vertex in the abstract graph of the specification and $c > 0$ is a buffer. Intuitively, this approach ensures that all tools get a similar number of steps in each episode to learn the specification.

Results. The learning curves for the 9-Rooms environment are shown in Figure 5.5. The learning curves for the 16-Rooms environment with all open doors and the constrained 16-Rooms environment with some open doors are shown in Figure 5.6 and Figure 5.7, respectively. Focusing on Figure 5.6, we see that none of the baselines scale beyond ϕ_2 (one segment), while DiRL quickly converges to high-quality policies for all specifications. The TLTL baseline performs poorly since most of these tasks require stateful policies, which it does not support. Though SPECTRL can learn stateful policies, it scales poorly since (i) it does not decompose the learning problem into simpler ones, and (ii) it does not integrate model-based planning at the high-level. Reward Machine based approaches (QRM and HRM) are also unable to handle complex specifications, likely because they are completely based on model-free RL, and do not employ model-based planning at the high-level. Although R-



(a) Half sub-specification φ_1, $|\mathcal{G}_{\varphi_1}| = 2$.

(b) 1 sub-specification φ_2, $|\mathcal{G}_{\varphi_2}| = 4$.

(c) 2 sub-specifications φ_3, $|\mathcal{G}_{\varphi_3}| = 8$.

(d) 3 sub-specifications φ_4, $|\mathcal{G}_{\varphi_4}| = 12$.

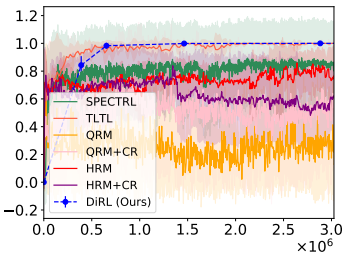
(e) 4 sub-specifications φ_5, $|\mathcal{G}_{\varphi_5}| = 16$.

(f) Sample complexity curves.

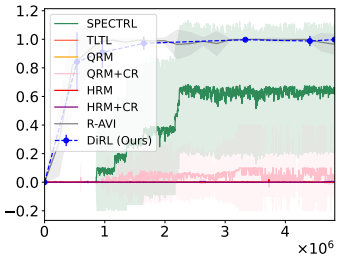
Figure 5.6: (a)-(e) Learning curves for 16-Rooms environment with different specifications increasing in complexity from from (a) to (e). x-axis denotes the number of samples (steps) and y-axis denotes the estimated probability of success. Results are averaged over 10 runs with error bars indicating $\pm$ standard deviation. (f) shows the average number of samples (steps) needed to achieve a success probability $\geq z$ (y-axis) as a function of the size of the abstract graph $|\mathcal{G}_\varphi|$.

AVI uses model-based planning at the high-level in conjunction with low-level RL, it does not scale to complex specifications since it trains all edge policies multiple times (across multiple iterations) with different initial state distributions; in contrast, our approach trains any edge policy at most once. The results are similar for the other Rooms environments as well. Our experiments also demonstrate the robustness of our tool on different specifications and environments. For instance, in the 16-Rooms environment with blocked doors, fewer policies satisfy the specification, which makes learning more challenging but DIRL is still able to learn high-quality policies for all the specifications.

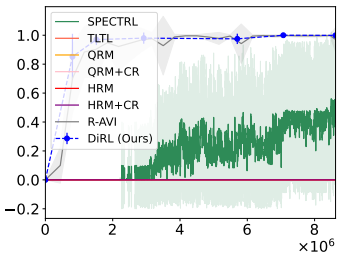
We summarize the scalability of DIRL in Figure 5.6f, where we show the average number of steps needed to achieve a given success probability z as a function of the number of edges in $\mathcal{G}_\varphi$ (denoted by $|\mathcal{G}_\varphi|$). As can be seen, the sample complexity of DIRL scales roughly linearly in the graph size.



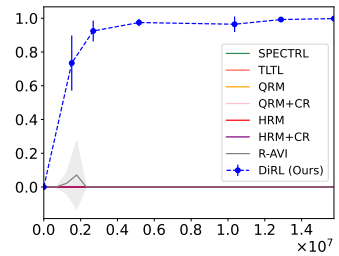
(a) Half-segment spec. φ_1, $|\mathcal{G}_{\varphi_1}| = 2$.



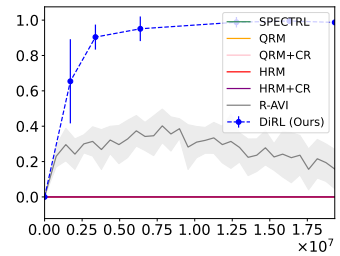
(b) 1-segment spec. φ_2, $|\mathcal{G}_{\varphi_2}| = 4$.



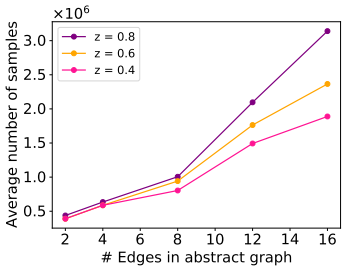
(c) 2-segment spec. φ_3, $|\mathcal{G}_{\varphi_3}| = 8$.



(d) 3-segment spec. φ_4, $|\mathcal{G}_{\varphi_4}| = 12$.



(e) 4-segment spec. φ_5, $|\mathcal{G}_{\varphi_5}| = 16$.



(f) Sample Complexity

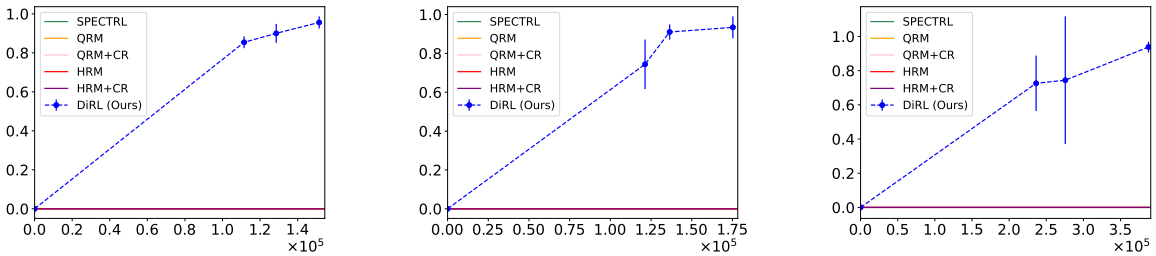
Figure 5.7: (a)-(e) Learning curves for 16-Rooms environment with some blocked doors (Figure 5.3b) with different specifications increasing in complexity from (a) to (e). x-axis denotes the number of samples (steps) and y-axis denotes the estimated probability of success. Results are averaged over 10 runs with error bars indicating $\pm$ standard deviation. (f) shows the average number of samples (steps) needed to achieve a success probability $\geq z$ (y-axis) as a function of the size of the abstract graph $|\mathcal{G}_\varphi|$.

Intuitively, each subtask takes a constant number of steps to learn, so the total number of steps required is proportional to $|\mathcal{G}_\varphi|$.

Next, we can see the results for the Fetch environment in Figure 5.8. The trends are similar to before—DIRL leverages compositionality to quickly learn effective policies, whereas the baselines are ineffective. The last task is especially challenging, taking DIRL somewhat longer to solve, but it ultimately achieves similar effectiveness. These results demonstrate that DIRL can scale to complex specifications even in challenging environments with high-dimensional state spaces.

5.5. Discussion

We have proposed DIRL, a reinforcement learning algorithm for logical specifications that leverages the compositional structure of the specification to decouple high-level planning and low-level control.



(a) PickAndPlace (b) PickAndPlaceStatic (c) PickAndPlaceChoice

Figure 5.8: Learning curves for Fetch environment; x-axis denotes the total number of samples (steps) and y-axis denotes the estimated probability of success. Results are averaged over 5 runs with error bars indicating ± standard deviation.

Our experiments demonstrate that DIRL can effectively solve complex continuous control tasks, significantly improving over existing approaches. Logical specifications are a promising approach to enable users to more effectively specify robotics tasks; by enabling more scalable learning of these specifications, we are directly enabling users to specify more complex objectives through the underlying specification language. While we have focused on SPECTRL specifications, we believe our approach can also enable the incorporation of more sophisticated features into the underlying language, such as conditionals (i.e., only perform a subtask upon observing some property of the environment) and iterations (i.e., repeat a subtask until some objective is met).

A key limitation of DIRL is that it assumes the ability to sample trajectories starting at any state $s \in S$ that has been observed before, whereas in some cases it might only be possible to obtain trajectories starting at some initial state. One way to overcome this limitation is to use the learnt path policies for sampling—i.e., in order to sample a state from a subgoal region $\beta(u)$ corresponding to a vertex u in the abstract graph, we could sample an initial state $s_0 \sim \eta$ from $\beta(u_0)$ and execute the path policy π_{ρ_u} corresponding to the shortest path ρ_u from u_0 to u starting at s_0. Upon successfully reaching $\beta(u)$ (we can restart the sampling procedure if $\beta(u)$ is not reached), the system will be in a state $s \sim \eta_u$ in $\beta(u)$ from which we can simulate the system further.

Another limitation of our approach is that we only consider path policies. It is possible that an optimal policy must follow different high-level plans from different states within the same subgoal

region. We believe this limitation can be addressed in future work by modifying our algorithm appropriately.

5.6. Related Work

There has been recent interest in combining high-level planning with reinforcement learning [2, 86, 45]. These approaches all target MDPs with reward functions, whereas we target MDPs with logical task specifications. Furthermore, in our setting, the high-level structure is derived from the given specification, whereas in existing approaches it is manually provided. Illanes et al. [78] propose an RL algorithm for reachability tasks that uses high-level planning to guide low-level RL; however, unlike our approach, they assume that a high-level model is given and high-level planning is not guided by the learned low-level policies.

CHAPTER 6

A Framework for Multi-Agent RL from Temporal Specifications

In the previous chapters, we studied the use of temporal specifications in reinforcement learning with a focus on the single-agent setting—i.e., there is one agent interacting with a stochastic environment whose transition probabilities are unknown. However, recent successes of RL has lead to an increased interest in applying RL to multi-agent systems in which multiple agents are interacting with the same stochastic environment. In the multi-agent setting, we are once again faced with issues regarding specifying the training objective using reward functions—for example, sparsity of rewards and scalability with respect to task complexity. In particular, in non-cooperative systems, each agent is trying to achieve its own goal [93]; for such systems, we need to devise a separate reward function for each agent.

In this chapter, we study the use of formal specifications in multi-agent reinforcement learning (MARL); in particular, we focus on non-cooperative systems in which each agent has its own specification. The goal in MARL is typically to learn a policy for each agent such that the joint strategy forms a Nash equilibrium. Existing approaches mostly focus on computing an arbitrary Nash equilibrium. However, in many settings, the user is a social planner trying to optimize the overall social welfare of the system, and most existing approaches are not designed to optimize such additional criteria while making sure that the resulting joint strategy is a Nash equilibrium.

6.1. Overview

We propose a novel multi-agent RL framework for learning policies from high-level specifications (one specification per agent) such that the resulting joint policy (i) has high social welfare, and (ii) is an ϵ-Nash equilibrium (for a given ϵ). We formulate this problem as a constrained optimization problem where the goal is to maximize social welfare under the constraint that the joint policy is an ϵ-Nash equilibrium.

Our algorithm for solving this optimization problem uses an enumerative search strategy. First, it

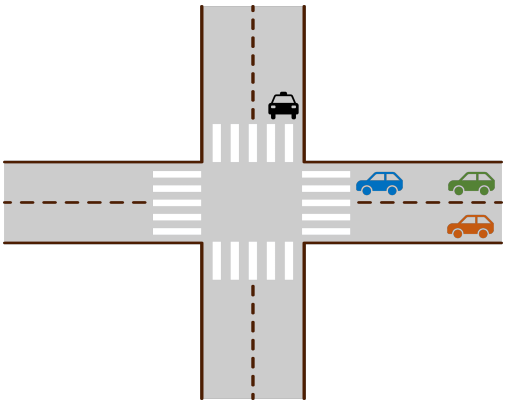


Figure 6.1: Intersection Example

enumerates candidate policies in decreasing order of social welfare. To ensure a tractable search space, it restricts to policies that conform to the structure of the user-provided specification. Then, for each candidate policy, it uses an explore-then-exploit self-play RL algorithm [20] to compute *punishment strategies* that are triggered when some agent deviates from the original joint policy. It also computes the maximum benefit each agent derives from deviating, which can be used to determine whether the joint policy augmented with punishment strategies forms an ϵ-Nash equilibrium; if so, it returns the joint policy.

Intuitively, the enumerative search tries to optimize social welfare, whereas the self-play RL algorithm checks whether the ϵ-Nash equilibrium constraint holds. Since this RL algorithm comes with PAC (Probably Approximately Correct) guarantees, our algorithm is guaranteed to return an ϵ-Nash equilibrium with high probability. In summary, we make the following contributions.

- We study the problem of maximizing social welfare under the constraint that the policies form an ϵ-NE. To the best of our knowledge, this problem has not been studied before in the context of learning (beyond single-step games).
- We provide an enumerate-and-verify framework for solving the said problem.
- We propose a verification algorithm with a probabilistic soundness guarantee in the RL setting of probabilistic systems with unknown transition probabilities.

Motivating example. Consider the road intersection scenario in Figure 6.1. There are four cars; three are traveling east to west and one is traveling north to south. At any stage, each car can either move forward one step or stay in place. Suppose each car's specification is as follows:

- *Black car:* Cross the intersection before the green and orange cars.
- *Blue car:* Cross the intersection before the black car and stay a car length ahead of the green and orange cars.
- *Green car:* Cross the intersection before the black car.
- *Orange car:* Cross the intersection before the black car.

We also require that the cars do not crash into one another. Clearly, not all agents can achieve their goals. The next highest social welfare is for three agents to achieve their goals. In particular, one possibility is that all cars except the black car achieve their goals. However, the corresponding joint policy requires that the black car does not move, which is not a Nash equilibrium—there is always a gap between the blue car and the other two cars behind, so the black car can deviate by inserting itself into the gap to achieve its own goal. Our algorithm uses self-play RL to optimize the policy for the black car, and finds that the other agents cannot prevent the black car from improving its outcome in this way. Thus, it correctly rejects this joint policy. Eventually, our algorithm computes a Nash equilibrium in which the black and blue cars achieve their goals.

6.2. Definitions

In this section, we provide some definitions and describe the multi-agent RL problem.

6.2.1. Markov Game.

Similar to the use of MDPs in the single-agent setting, the environment is modeled as a Markov game in the multi-agent setting. An n-agent Markov game[13] is a tuple $\mathcal{M} = (S, A, s_0, P, H)$ with a finite set of states S, actions $A = A_1 \times \cdots \times A_n$ where A_i is a finite set of actions available to agent i, transition probabilities $P(s, a, s')$ for $s, s' \in S$ and $a \in A$, finite horizon H, and initial state

[13]Here we fix a finite horizon H in advance.

s_0 [106]. A finite *trajectory* $\zeta \in \mathcal{Z} = (S \times A)^* \times S$ is a finite sequence[14] $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t$ where $s_k \in S$, $a_k \in A$; we use $|\zeta| = t$ to denote the length of the trajectory ζ and $a_k^i \in A_i$ to denote the action of agent i in a_k.

For any $i \in [n]$, let $\mathcal{D}(A_i)$ denote the set of distributions over A_i—i.e., $\mathcal{D}(A_i) = \{\Delta : A_i \to [0,1] \mid \sum_{a_i \in A_i} \Delta(a_i) = 1\}$. A *policy* for agent i is a function $\pi_i : \mathcal{Z} \to \mathcal{D}(A_i)$ mapping trajectories to distributions over actions. A policy π_i is *deterministic* if for every $\zeta \in \mathcal{Z}$, there is an action $a_i \in A_i$ such that $\pi_i(\zeta)(a_i) = 1$; in this case, we also use $\pi_i(\zeta)$ to denote the action a_i. A *joint policy* $\pi : \mathcal{Z} \to \mathcal{D}(A)$ maps finite trajectories to distributions over joint actions. We use $(\pi_1, \ldots, \pi_n)$ to denote the joint policy in which agent i chooses its action in accordance to π_i. We denote by $\mathcal{D}_\pi$ the distribution over H-length trajectories in $\mathcal{M}$ induced by π.

Similar to DIRL, some parts of our algorithm are based on an assumption which allows us to obtain sample trajectories starting at any state that has been observed before.

Assumption 6.1. *We can obtain samples from $P(s, a, \cdot)$ for any previously observed state s and any action a.*

6.2.2. Specification Language and Abstract Graph

We consider the specification language SPECTRL to express agent specifications (see Section 4.2). We choose SPECTRL since we can construct an abstract graph from any SPECTRL specification which exposes the structure in the specification. However, we believe that our framework can be adapted to other specification languages as well.

Abstract Graphs. We showed in the previous chapter that SPECTRL specifications can be represented by *abstract graphs* which are DAG-like structures in which each vertex represents a set of states (called subgoal regions) and each edge represents a set of concrete trajectories that can be used to transition from the source vertex to the target vertex without violating safety constraints. Since we are interested in the finite-horizon setting, we also have terminal safe trajectories for each final vertex f (denoted by $\mathcal{Z}^f_{\texttt{term}}$ in the previous chapter). Here, we incorporate the terminal safe

[14] We simply use $\mathcal{Z}$ to also denote the set of finite trajectories $\mathcal{Z}_f$ in this chapter since we have a fixed finite horizon.

trajectories $\mathcal{Z}_{\texttt{term}}$ within $\mathcal{Z}_{\texttt{safe}}$.

Definition 6.1. An *abstract graph* $\mathcal{G} = (U, E, u_0, F, \beta, \mathcal{Z}_{\texttt{safe}})$ is a directed acyclic graph (DAG) with vertices U, (directed) edges $E \subseteq U \times U$, initial vertex $u_0 \in U$, final vertices $F \subseteq U$, subgoal region map $\beta : U \to 2^S$ such that for each $u \in U$, $\beta(u)$ is a subgoal region, and *safe trajectories* $\mathcal{Z}_{\texttt{safe}} = \bigcup_{e \in E} \mathcal{Z}^e_{\texttt{safe}} \cup \bigcup_{f \in F} \mathcal{Z}^f_{\texttt{safe}}$, where $\mathcal{Z}^e_{\texttt{safe}} \subseteq \mathcal{Z}$ denotes the safe trajectories for edge $e \in E$ and $\mathcal{Z}^f_{\texttt{safe}} \subseteq \mathcal{Z}$ denotes the terminal safe trajectories for final vertex $f \in F$.

The satisfaction of an abstract graph by a finite trajectory $\zeta \in \mathcal{Z}$ is defined according to Definition 5.4 (in which $\mathcal{Z}^f_{\texttt{term}}$ denotes $\mathcal{Z}^f_{\texttt{safe}}$). Recall that for every SPECTRL specification φ, we can construct an abstract graph $\mathcal{G}_\varphi$ such that for every trajectory $\zeta \in \mathcal{Z}$, $\zeta \models \varphi$ if and only if $\zeta \models \mathcal{G}_\varphi$.

6.2.3. Nash Equilibrium and Social Welfare

Given a Markov game $\mathcal{M}$ with unknown transitions and SPECTRL specifications $\varphi_1, \ldots, \varphi_n$ for the n agents respectively, the score of agent i from a joint policy π is given by

$$J_i(\pi) = \Pr_{\zeta \sim \mathcal{D}_\pi}[\zeta \models \varphi_i].$$

Our goal is to compute a *high-value* ϵ-Nash equilibrium in $\mathcal{M}$ w.r.t these scores. Given a joint policy $\pi = (\pi_1, \ldots, \pi_n)$ and an alternate policy π'_i for agent i, let (π_{-i}, π'_i) denote the joint policy $(\pi_1, \ldots, \pi'_i, \ldots, \pi_n)$. Then, a joint policy π is an *ϵ-Nash equilibrium* if for all agents i and all alternate policies π'_i, $J_i(\pi) \geq J_i((\pi_{-i}, \pi'_i)) - \epsilon$. Our goal is to compute a joint policy π that maximizes the social welfare given by

$$\texttt{welfare}(\pi) = \frac{1}{n} \sum_{i=1}^{n} J_i(\pi)$$

subject to the constraint that π is an ϵ-Nash equilibrium.

6.3. Our Framework

Our framework for computing a high-welfare ϵ-Nash equilibrium consists of two phases. The first phase is a *prioritized enumeration* procedure that learns deterministic joint policies in the environment and ranks them in decreasing order of social welfare. The second phase is a *verification*

phase that checks whether a given joint policy can be extended to an ϵ-Nash equilibrium by adding punishment strategies. A policy is returned if it passes the verification check in the second phase. Algorithm 3 summarizes our framework.

For the enumeration phase, it is impractical to enumerate all joint policies even for small environments, since the total number of deterministic joint policies is $\Omega(|A|^{|S|^{H-1}})$, which is $\Omega(2^{n|S|^{H-1}})$ if each agent has atleast two actions. Thus, in the prioritized enumeration phase, we apply a specification-guided heuristic to reduce the number of joint policies considered. The resulting search space is independent of $|S|$ and H, depending only on the specifications $\{\varphi_i\}_{i\in[n]}$. Since the transition probabilities are unknown, these joint policies are trained using an efficient compositional RL approach.

Since the joint policies are trained cooperatively, they are typically not ϵ-Nash equilibria. Hence, in the verification phase, we use a probably approximately correct (PAC) procedure (Algorithm 6) to determine whether a given joint policy can be modified by adding *punishment strategies* to form an ϵ-Nash equilibrium. Our approach is to reduce this problem to solving two-agent zero-sum games. The key insight is that for a given joint policy to be an ϵ-Nash equilibrium, unilateral deviations by any agent must be successfully punished by the coalition of all other agents. In such a *punishment game*, the deviating agent attempts to maximize its score while the coalition of other agents attempts to minimize its score, leading to a competitive min-max game between the agent and the coalition. If the deviating agent can improve its score by a margin $\geq \epsilon$, then the joint policy cannot be extended to an ϵ-Nash equilibrium. Alternatively, if no agent can increase its score by a margin $\geq \epsilon$, then the joint policy (augmented with punishment strategies) is an ϵ-Nash equilibrium. Thus, checking if a joint policy can be converted to an ϵ-Nash equilibrium reduces to solving a two-agent zero-sum game for each agent. Each punishment game is solved using a self-play RL algorithm for learning policies in min-max games with unknown transitions [20], after converting specification-based scores to reward-based scores. While the initial joint policy is deterministic, the punishment strategies can be probabilistic.

Overall, we provide the guarantee that with high probability, if our algorithm returns a joint policy,

Algorithm 3 HIGHNASHSEARCH
Inputs: Markov game (with unknown transition probabilities) $\mathcal{M}$ with n-agents, agent specifications $\varphi_1, \ldots, \varphi_n$, Nash factor ϵ, precision δ, failure probability p.
Outputs: ϵ-NE, if found.

```
PrioritizedPolicies ← PRIORITIZEDENUMERATION(M, φ_1, ..., φ_n)
for joint policy π ∈ PrioritizedPolicies do
    // Can π be extended to an ε-NE?
    isNash, τ ← VERIFYNASH(M, π, φ_1, ··· , φ_n, ε, δ, p)
    if isNash then return π ⋈ τ // Add punishment strategies
end for
return No ε-NE found
```

it will be an ϵ-Nash equilibrium.

6.4. Prioritized Enumeration

In this section, we describe our specification-guided compositional RL algorithm for learning a finite number of deterministic joint policies in an unknown environment under Assumption 6.1. These policies are then ranked in decreasing order of their (estimated) social welfare.

6.4.1. Overview

Our learning algorithm harnesses the structure of specifications, exposed by their abstract graphs, to curb the number of joint policies to learn. For every set of *active agents* $B \subseteq [n]$, we construct a product abstract graph, from the abstract graphs of all active agents' specifications. A property of this product is that if a trajectory ζ in $\mathcal{M}$ corresponds to a path in the product that ends in a final state then ζ satisfies the specification of *all* active agents. Then, our procedure learns one joint policy for every path in the product graph that reaches a final state. Intuitively, policies learned using the product graph corresponding to a set of active agents B aim to maximize satisfaction probabilities of all agents in B. By learning joint policies for every set of active agents, we are able to learn policies under which some agents may not satisfy their specifications. This enables learning joint policies in non-cooperative settings. Note that the number of paths (and hence the number of policies considered) is independent of $|S|$ and H, and depends only on the number of agents and their specifications.

One caveat is that the number of paths may be exponential in the number of states in the product

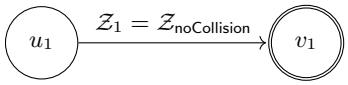


Figure 6.2: Abstract Graph of black car.

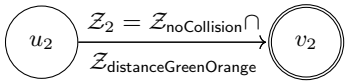


Figure 6.3: Abstract Graph of blue car.

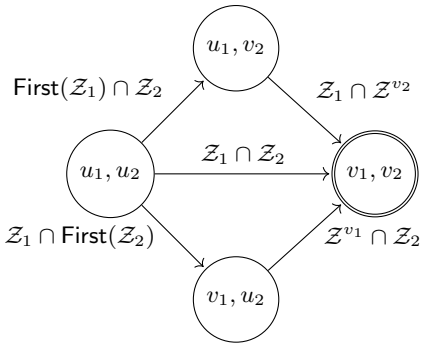


Figure 6.4: Product Abstract Graph of black and blue cars. $\mathcal{Z}^{v_1}$ and $\mathcal{Z}^{v_2}$ refer to safe trajectories after the black and blue cars have reached their final states, respectively.

graph. It would be impractical to naïvely learn a joint policy for every path. Instead, we design an efficient compositional RL algorithm that learns a joint policy for each edge in the product graph; these edge policies are then composed together to obtain joint policies for paths in the product graph.

6.4.2. Product Abstract Graph

Let $\varphi_1, \ldots, \varphi_n$ be the specifications for the n-agents, respectively, and let $\mathcal{G}_i = (U_i, E_i, u_0^i, F_i, \beta_i, \mathcal{Z}_{\texttt{safe},i})$ be the abstract graph of specification φ_i in the environment $\mathcal{M}$. We construct a product abstract graph for every set of active agents in $[n]$. The product graph for a set of active agents $B \subseteq [n]$ is used to learn joint policies which satisfy the specification of all agents in B with high probability.

Definition 6.2. *Given a set of agents $B = \{i_1, \ldots, i_m\} \subseteq [n]$, the product graph $\mathcal{G}_B = (\overline{U}, \overline{E}, \overline{u}_0, \overline{F}, \overline{\beta}, \overline{\mathcal{Z}}_{\texttt{safe}})$ is the asynchronous product of $\mathcal{G}_i$ for all $i \in B$, with*

- $\overline{U} = \prod_{i \in B} U_i$ *is the set of product vertices,*
- *An edge $e = (u_{i_1}, \ldots, u_{i_m}) \to (v_{i_1}, \ldots, v_{i_m}) \in \overline{E}$ if at least for one agent $i \in B$ the edge $u_i \to v_i \in E_i$ and for the remaining agents, $u_i = v_i$,*
- $\overline{u}_0 = (u_0^{i_1}, \ldots, u_0^{i_m})$ *is the initial vertex,*

- $\overline{F} = \Pi_{i \in B} F_i$ *is the set of final vertices,*
- $\overline{\beta} = (\beta_{i_1}, \ldots, \beta_{i_m})$ *is the collection of concretization maps, and*
- $\overline{\mathcal{Z}}_{\texttt{safe}} = (\mathcal{Z}_{\texttt{safe},i_1}, \ldots, \mathcal{Z}_{\texttt{safe},i_m})$ *is the collection of safe trajectories.*

We denote the i-th component of a product vertex $\overline{u} \in \overline{U}$ by u_i for agent $i \in B$. Similarly, the i-th component in an edge $e = \overline{u} \to \overline{v}$ is denoted by $e_i = u_i \to v_i$ for $i \in B$; note that e_i can be a self loop which is not an edge in $\mathcal{G}_i$. For an edge $e \in \overline{E}$, we denote the set of agents $i \in B$ for which $e_i \in E_i$, and not a self loop, by $\mathsf{progress}(e)$.

Abstract graphs of the black car and the blue car from the motivating example are shown in Figures 6.2 and 6.3 respectively. The vertex v_1 denotes the subgoal region $\beta_{\text{black}}(v_1)$ consisting of states in which the black car has crossed the intersection but the orange and green cars have not. The subgoal region $\beta_{\text{blue}}(v_2)$ is the set of states in which the blue car has crossed the intersection but the black car has not. $\mathcal{Z}_1$ denotes trajectories in which the black car does not collide and $\mathcal{Z}_2$ denotes trajectories in which the blue car does not collide and stays a car length ahead of the orange and green cars. The product abstract graph for the set of active agents $B = \{\text{black, blue}\}$ is shown in Fig 6.4. The safe trajectories on the edges reflect the notion of *achieving* a product edge which we discuss below.

A trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t$ *achieves* an edge $e = \overline{u} \to \overline{v}$ in $\mathcal{G}_B$ if all progressing agents $i \in \mathsf{progress}(e)$ reach their target subgoal region $\beta_i(v_i)$ along the trajectory and the trajectory is safe for all agents in B. For a progressing agent $i \in \mathsf{progress}(e)$, the initial segment of the rollout until the agent reaches its subgoal region should be safe with respect to the edge e_i. After that, the rollout should be safe with respect to every future possibility for the agent. This is required to ensure continuity of the rollout into adjacent edges in the product graph $\mathcal{G}_B$. For the same reason, we require that the entire rollout is safe with respect to all future possibilities for non-progressing agents. Note that we are not concerned with non-active agents in $[n] \setminus B$. In order to formally define this notion, we need to setup some notation.

For a predicate $b \in \mathcal{P}$, let the set of safe trajectories w.r.t. b be given by

$$\mathcal{Z}_b = \{\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t \in \mathcal{Z} \mid \forall\ 0 \leq k \leq t, s_k \models b\}.$$

We known that safe trajectories along an edge in an abstract graph constructed from a SPECTRL specification is either of the form $\mathcal{Z}_b$ or $\mathcal{Z}_{b_1} \circ \mathcal{Z}_{b_2}$, where $b, b_1, b_2 \in \mathcal{P}$ and $\circ$ denotes concatenation (see Lemma 5.11). In addition, for every final vertex f, $\mathcal{Z}^f_{\texttt{safe}}$ is of the form $\mathcal{Z}_b$ for some $b \in \mathcal{P}$. We define First as follows:

$$\mathsf{First}(\mathcal{Z}') = \begin{cases} \mathcal{Z}_b, & \text{if } \mathcal{Z}' = \mathcal{Z}_b \\ \mathcal{Z}_{b_1}, & \text{if } \mathcal{Z}' = \mathcal{Z}_{b_1} \circ \mathcal{Z}_{b_2} \end{cases}$$

We are now ready to define the notion of satisfiability of a product edge.

Definition 6.3. A rollout $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_k$ *achieves an edge* $e = \overline{u} \to \overline{v}$ in $\mathcal{G}_B$ (denoted $\zeta \models_B e$) if

1. for all progressing agents $i \in \mathsf{progress}(e)$, there exists an index $k_i \leq k$ such that $s_{k_i} \in \beta_i(v_i)$ and $\zeta_{0:k_i} \in \mathcal{Z}^{e_i}_{\texttt{safe},i}$. If $v_i \in F_i$ then $\zeta_{k_i:k} \in \mathcal{Z}^{v_i}_{\texttt{safe},i}$. Otherwise, $\zeta_{k_i:k} \in \mathsf{First}(\mathcal{Z}^{v_i \to w_i}_{\texttt{safe},i})$ for all $w_i \in \mathsf{outgoing}(v_i)$. Furthermore, we require $k_i > 0$ if $u_i \neq u^i_0$.

2. for all non-progressing agents $i \in B \setminus \mathsf{progress}(e)$, if $u_i \notin F_i$, $\zeta \in \mathsf{First}(\mathcal{Z}^{u_i \to w_i}_{\texttt{safe},i})$ for all $w_i \in \mathsf{outgoing}(u_i)$. Otherwise (if $u_i \in F_i$), $\zeta \in \mathcal{Z}^{u_i}_{\texttt{safe},i}$

We can now define what it means for a trajectory to achieve a path in the product graph $\mathcal{G}_B$.

Definition 6.4. Given $B \subseteq [n]$, a rollout $\zeta = s_0 \to \cdots \to s_t$ *achieves* a path $\rho = \overline{u}_0 \to \cdots \to \overline{u}_\ell$ in $\mathcal{G}_B$ (denoted $\zeta \models_B \rho$) if there exists indices $0 = k_0 \leq k_1 \leq \cdots \leq k_\ell \leq t$ such that (i) $\overline{u}_\ell \in \overline{F}$, (ii) $\zeta_{k_z:k_{z+1}}$ achieves $\overline{u}_z \to \overline{u}_{z+1}$ for all $0 \leq z < \ell$, and (iii) $\zeta_{k_\ell:t} \in \mathcal{Z}^{u_{\ell,i}}_{\texttt{safe},i}$ for all $i \in B$.

Theorem 6.5. *Let $\rho = \overline{u_0} \to \overline{u_1} \to \cdots \to \overline{u_\ell}$ be a path in the product abstract graph $\mathcal{G}_B$ for $B \subseteq [n]$. Suppose trajectory $\zeta \models_B \rho$. Then $\zeta \models \varphi_i$ for all $i \in B$.*

Proof. We show that if $\zeta \models_B \rho$ then every agent $i \in B$ achieves a path in the abstract graph $\mathcal{G}_i$ of its specification φ_i. Let $0 = k_0 \leq k_1 \leq \cdots \leq k_\ell \leq t$ be the indices that satisfy the criteria in Definition 6.4.

For agent $i \in B$, let indices $0 \leq z_0 < \cdots < z_x < \ell$ be such that the agent makes progress along the edge $e_{z_y} = \overline{u}_{z_y} \to \overline{u}_{z_y+1}$ for $0 \leq y \leq x$. Note that agents can make progress only till they reach a final state in their abstract graph. So, $(u_{z_y})_i \notin F_i$ and $(u_{z_x+1})_i \in F_i$ for all $0 \leq y \leq x$. Further note, $(u_{z_0})_i = u_0^i$ and $(u_{z_{y+1}})_i = (u_{z_y+1})_i$ for $0 \leq y < x$ since the agent i has not made any progress in between.

Let $\zeta_y = \zeta_{k_{z_y}:k_{z_y+1}}$ be the sub-trajectory that achieves the edge e_{z_y} for $0 \leq y \leq x$, i.e. $\zeta_y \models_B e_{z_y}$ for $0 \leq y \leq x$. Since, agent i has made progress along e_{z_y}, using Definition 6.3, we can obtain indices $0 = p_0 \leq \cdots < p_{x+1} \leq t$ on the trajectory such that

- $p_y \leq k_{z_y} \leq p_{y+1}$ for $0 \leq y < x$,
- $s_{p_{y+1}} \in \beta_i((u_{z_y+1})_i)$ for all $0 \leq y \leq x$,
- $\zeta_{k_{z_y}:p_{y+1}} \in \mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i}$ for all $0 \leq y \leq x$,
- $\zeta_{p_{x+1}:k_{z_x+1}} \models \mathcal{Z}_{\texttt{safe},i}^{(u_{z_x+1})_i}$, and
- $\zeta_{p_{y+1}:k_{z_y+1}} \in \mathsf{First}(\mathcal{Z}_{\texttt{safe},i}^{(u_{z_y+1})_i \to w_i})$ for $w_i \in \mathsf{outgoing}((u_{z_y+1})_i)$ and $0 \leq y < x$.

  This is because $(u_{z_y})_i \notin F_i$ and $(u_{z_x+1})_i \in F_i$ for all $0 \leq y \leq x$, as observed earlier.

Additionally, by using, $(u_{z_y+1})_i = (u_{z_{y+1}})_i$ for $0 \leq y < x$ since the agent i has not made any progress in between, the above is simplified to: there exists indices $0 = p_0 \leq \cdots < p_{x+1} \leq t$ on the trajectory such that

- $p_y \leq k_{z_y} \leq p_{y+1}$ for $0 \leq y \leq x$,
- $s_{p_{y+1}} \in \beta_i((u_{z_{y+1}})_i)$ for all $0 \leq y \leq x$,

- $\zeta_{k_{z_y}:p_{y+1}} \in \mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i}$ for all $0 \leq y \leq x$,
- $\zeta_{p_{x+1}:k_{z_{x+1}}} \models \mathcal{Z}_{\texttt{safe},i}^{(u_{z_x+1})_i}$, and
- $\zeta_{p_{y+1}:k_{z_{y+1}}} \in \mathsf{First}(\mathcal{Z}_{\texttt{safe},i}^{(e_{z_{y+1}})_i})$ for $0 \leq y < x$.

  This is because $(e_{z_{y+1}})_i$ is an outgoing edge from $(u_{z_{y+1}})_i$.

Our goal is to show that $\zeta_{p_y:p_{y+1}} \in \mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i}$ for $0 \leq y \leq x$. We already know that $\zeta_{k_{z_y}:p_{y+1}} \in \mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i}$ for all $0 \leq y \leq x$. Observing the fact that for any edge e of $\mathcal{G}_i$ the set $\mathsf{First}(\mathcal{Z}_{\texttt{safe},i}^{e})$ is closed under concatenation, it is sufficient to show $\zeta_{p_y:k_{z_y}} \in \mathsf{First}(\mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i})$ for $0 \leq y \leq x$. We do this in two cases.

- **Case** $y = 0$: In this case $\zeta_{p_0:k_{z_0}} = \zeta_{k_0:k_{z_0}}$. So, it has made no progress along the path till $s_{k_{z_0}}$. So, the non-progressing agent will ensure its trajectory is safe with respect to outgoing edges from vertex $(u_{z_0})_i = (u_0)_i$. Now, $(e_{z_0})_i$ is an outgoing edge from $(u_{z_0})_i$. So, we get that $\zeta_{0:k_{z_0}} \in \mathsf{First}(\mathcal{Z}_{\texttt{safe},i}^{(e_{z_0})_i})$.
- **Case** $0 < y \leq x$: Here, we know that $\zeta_{p_y:k_{z_{y-1}+1}} \in \mathsf{First}(\mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i})$. So, it is sufficient to show that $\zeta_{k_{z_{y-1}+1}:k_{z_y}} \in \mathsf{First}(\mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i})$. This is true because agent i is has not progressed on any of the edges between $\overline{u}_{z_{y-1}+1}$ and $\overline{u}_{z_y}$. The i^{th} component of all vertices between these states is $(u_{z_y})_i$, since agent i will not change its vertex. Now $(e_{z_y})_i$ is an outgoing edge from $(u_{z_y})_i$. So, in particular, we get that $\zeta_{k_{z_{y-1}+1}:k_{z_y}} \in \mathsf{First}(\mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i})$.

Now consider the path $(u_0)_i \to (u_{z_0})_i \to (u_{z_1})_i \to (u_{z_x})_i \to (u_{z_x+1})_i \in F_i$ in graph $\mathcal{G}_i$. There exists indices $0 = p_0 \leq p_1 < ... < p_{x+1} \leq t$ such that

- $s_{p_0} = s_0 \in \beta_i((u_0)_i)$ (from Definition 6.4), $s_{p_y} \in \beta_i((u_{z_y})_i)$ for $0 < y \leq x$ and $s_{p_{x+1}} \in \beta_i((u_{z_x+1})_i)$ (from inferences made above),
- $\zeta_{p_y:p_{y+1}} \in \mathcal{Z}_{\texttt{safe},i}^{(e_{z_y})_i}$ for $0 \leq y \leq x$, and
- $\zeta_{p_{x+1}:t} \in \mathcal{Z}_{\texttt{safe},i}^{(u_{z_x+1})_i}$ because agent i cannot progress further after visiting state $(u_{z_x+1})_i \in F_i$.

Thus, for agent i, $\zeta \models \mathcal{G}_i$, which implies $\zeta \models \varphi_i$. □

We can conclude from the above theorem that joint policies that maximize the probability of achieving paths (to final vertices) in the product abstract graph $\mathcal{G}_B$ have high social welfare w.r.t. the active agents B.

6.4.3. Compositional RL Algorithm

Our compositional RL algorithm learns joint policies corresponding to paths in product abstract graphs. For every $B \subseteq [n]$, it learns a joint policy π_e for each edge in the product abstract graph $\mathcal{G}_B$, which is a (deterministic) policy that maximizes the probability of achieving e from a given initial state distribution. We assume all agents are acting cooperatively; thus, we treat the agents as one and use single-agent RL to learn each edge policy. We will check whether any deviation to this co-operative behaviour by any agent can be punished by the coalition of other agents in the verification phase. The reward function is designed to capture the reachability objective of progressing agents and the safety objective of all active agents.

The edges are learned in topological order, allowing us to learn an induced state distribution for each product vertex $\overline{u}$ prior to learning any edge policies from $\overline{u}$; this distribution is used as the initial state distribution when learning outgoing edge policies from $\overline{u}$. In more detail, the distribution for the initial vertex of $\mathcal{G}_B$ is taken to be the initial state distribution of the environment; for every other product vertex, the distribution is the average over distributions induced by executing edge policies for all incoming edges. This is possible because the product graph is a DAG.

Given edge policies Π along with a path $\rho = \overline{u}_0 \to \overline{u}_1 \to \cdots \to \overline{u}_\ell = \overline{u} \in \overline{F}$ in $\mathcal{G}_B$, we define a *path policy* π_ρ to navigate from $\overline{u}_0$ to $\overline{u}$. In particular, π_ρ executes $\pi_{e[z]}$, where $e[z] = \overline{u}_z \to \overline{u}_{z+1}$ (starting from $z = 0$) until the resulting trajectory achieves $e[z]$, after which it increments $z \leftarrow z + 1$ (unless $z = \ell$). That is, π_ρ is designed to achieve the sequence of edges in ρ. Note that π_ρ is a finite-state deterministic joint policy in which vertices on the path correspond to the memory states that keep track of the index of the current policy. This way, we obtain finite-state joint policies by learning edge policies only.

This process is repeated for all sets of active agents $B \subseteq [n]$. These finite-state joint policies are

Algorithm 4 PRIORITIZEDENUMERATION
Inputs: n-agent environment $\mathcal{M}$ and agent specifications $\varphi_1, \ldots, \varphi_n$
Output: Ranking scheme

Initialize pathPolicyWelfareMaxHeap $\leftarrow \varnothing$
for $i \in [n]$ **do** $\mathcal{G}_i \leftarrow \mathsf{AbstractGraph}(\varphi_i)$
for $B \in 2^{[n]}$ **do**
  $\mathcal{G}_B \leftarrow \mathsf{ProductAbstractGraph}(\mathcal{G}_1, \ldots \mathcal{G}_n, B)$
  Initialize path policies $\Pi(\overline{u}_0) \leftarrow \{\varepsilon\}$ and $\Pi(\overline{u}) \leftarrow \varnothing$ if $\overline{u} \neq \overline{u}_0$
  Initialize state distribution $\Gamma(\overline{u}_0) \leftarrow \{\mathsf{At}(s_0)\}$ and $\Gamma(\overline{u}) \leftarrow \varnothing$ if $\overline{u} \neq \overline{u}_0$
  topoSortedVertexStack $\leftarrow \mathsf{TopologicalSort}(\overline{U}, \overline{E})$
  while topoSortedVertexStack $\neq \varnothing$ **do**
    $\overline{u} \leftarrow$ topoSortedVertexStack.pop()
    $\eta_{\overline{u}} \leftarrow \mathsf{AverageDistribution}(\Gamma(\overline{u}))$
    for $e = \overline{u} \to \overline{v} \in \mathsf{outgoing}(\overline{u})$ **do**
      $\pi_e \leftarrow \mathsf{LearnEdgePolicy}(e, \eta_{\overline{u}})$
      $\eta_{\overline{v},e} \leftarrow \mathsf{ReachDistribution}(e, \pi_e, \eta_{\overline{u}})$
      Add $\eta_{\overline{v},e}$ to $\Gamma(\overline{v})$
      for $\pi_\rho \in \Pi(\overline{u})$ **do** Add $\pi_\rho \circ \pi_e$ to $\Pi(\overline{v})$
    end for
    if $\overline{u} \in F$ **then**
      for $\pi_\rho \in \Pi(\overline{u})$ **do**
        $c \leftarrow \mathsf{EstimatePolicyWelfare}(\pi_\rho, \varphi_1, \ldots, \varphi_n)$
        Add (π_ρ, c) to pathPolicyWelfareMaxHeap
      end for
    end if
  end while
end for
return pathPolicyWelfareMaxHeap

then ranked by estimating their social welfare on several simulations.

6.4.4. Complete Enumeration Algorithm

The complete algorithm for prioritized enumeration is outlined in Algorithm 4. The details of non-standard functions are given below.

AverageDistribution: The set $\Gamma(\overline{u})$ consists of as many state distributions as there are incoming edges into the state when $\overline{u} \neq \overline{u_0}$. When $\overline{u} = \overline{u}_0$, $\Gamma(\overline{u})$ only contains the distribution corresponding to the initial state distribution of the underlying environment. The function AverageDistribution computes an initial state distribution for the input abstract product vertex $\overline{u}$ by taking an average of all of the distributions in $\Gamma(\overline{u})$.

LearnEdgePolicy: Use (single agent) RL to learn a co-operative joint policy π_e so that π_e achieves the edge $e = \overline{u} \to \overline{v}$ from the given initial state distribution $\eta_{\overline{u}}$. We use single-agent RL, specifically Q-learning, to learn a co-operative joint policy to achieve an edge with the reward $\mathbb{1}(\zeta \models_B e)$. Precisely, we learn π_e such that

$$\pi_e \in \arg\max_{\pi} \Pr_{s_0 \sim \eta_{\overline{u}}, \zeta \sim \mathcal{D}_{\pi, s_0}} [\zeta \models_B e]$$

where $\zeta \sim \mathcal{D}_{\pi,s_0}$ is the trajectory sampled by executing policy π from state s_0.

ReachDistribution: Given an edge $e = \overline{u} \to \overline{v}$, edge policy π_e, and initial state distribution $\eta_{\overline{u}}$, this function evaluates the state distribution induced on $\overline{v}$ upon executing policy π_e with an initial state distribution $\eta_{\overline{u}}$. Formally, for any $s \in \mathcal{S}$

$$\Pr_{s' \sim \eta_{\overline{v},e}} [s = s'] = \Pr_{s_0 \sim \eta_{\overline{u}}, \zeta \sim \mathcal{D}_{\pi_e, s_0}} \left[s = s_k \mid \zeta_{0:k} \models_B e \text{ and } \forall k' < k, \zeta_{0:k'} \not\models_B e \right]$$

where $\zeta \sim \mathcal{D}_{\pi_e,s_0}$ is the trajectory sampled from executing policy π_e from state s_0 and k is the length of the smallest prefix of ζ that achieves e.

EstimatePolicyWelfare: Once a path policy π_ρ is learnt, we estimate the probability of satisfaction of the specifications $J_i(\pi_\rho)$ for all agents i using Monte-Carlo sampling. Then $\texttt{welfare}(\pi_\rho)$ is computed by taking the mean of the probabilities of satisfaction of agent specifications.

6.5. Nash Equilibria Verification

The prioritized enumearation phase produces a list of path policies which are ranked by the total sum of scores. Each path policy is deterministic and also finite state. Since the joint policies are trained cooperatively, they are typically not ϵ-Nash equilibria. Thus, our verification algorithm not only tries to prove that a given joint policy is an ϵ-Nash equilibrium, but also tries to modify it so it satisfies this property. In particular, our verification algorithm attempts to modify a given joint policy by adding *punishment strategies* so that the resulting policy is an ϵ-Nash equilibrium.

Concretely, it takes as input a finite-state deterministic joint policy $\pi = (M, \alpha, \sigma, m_0)$ where M is a finite set of *memory states*, $\alpha : S \times A \times M \to M$ is the memory update function, $\sigma : S \times M \to A$

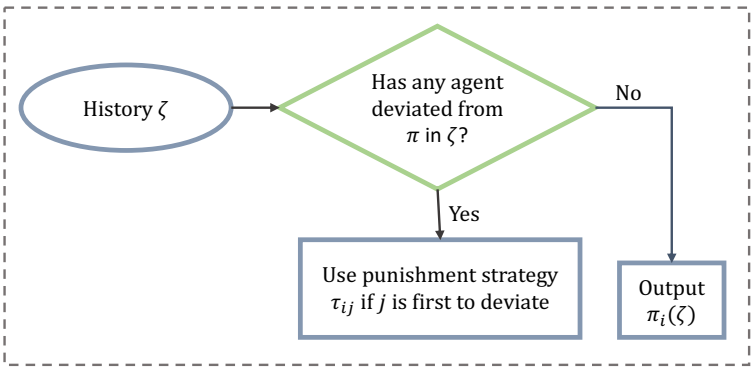


Figure 6.5: π_i augmented with punishment strategies.

maps states to (joint) actions and m_0 is the initial policy state. The *extended memory update function* $\hat{\alpha} : \mathcal{Z} \to M$ is given by $\hat{\alpha}(\epsilon) = m_0$ and $\hat{\alpha}(\zeta s_t a_t) = \alpha(s_t, a_t, \hat{\alpha}(\zeta))$. Then, π is given by $\pi(\zeta s_t) = \sigma(s_t, \hat{\alpha}(\zeta))$. The policy π_i of agent i simply chooses the i^{th} component of $\pi(\zeta)$ for any history ζ.

The verification algorithm learns one punishment strategy $\tau_{ij} : \mathcal{Z} \to \mathcal{D}(A_i)$ for each pair (i, j) of agents. As outlined in Figure 6.5, the modified policy for agent i uses π_i if every agent j has taken actions according to π_j in the past. In case some agent j' has taken an action that does not match the output of $\pi_{j'}$, then agent i uses the punishment strategy τ_{ij}, where j is the agent that deviated the earliest (ties broken arbitrarily). The goal of verification is to check if there is a set of punishment strategies $\{\tau_{ij} \mid i \neq j\}$ such that after modifying each agent's policy to use them, the resulting joint policy is an ϵ-Nash equilibrium.

6.5.1. Problem Formulation

We denote the set of all punishment strategies of agent i by $\tau_i = \{\tau_{ij} \mid j \neq i\}$. We define the composition of π_i and τ_i to be the policy $\tilde{\pi}_i = \pi_i \bowtie \tau_i$ such that for any trajectory $\zeta = s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{t-1}} s_t$, we have

- $\tilde{\pi}_i(\zeta) = \pi_i(\zeta)$ if for all $0 \leq k < t$, $a_k = \pi(\zeta_{0:k})$—i.e., no agent has deviated so far,
- $\tilde{\pi}_i(\zeta) = \tau_{ij}(\zeta)$ if there is a k such that (i) $a_k^j \neq \pi_j(\zeta_{0:k})$ and (ii) for all $\ell < k$, $a_\ell = \pi(\zeta_{0:\ell})$. If there are multiple such j's, an arbitrary but consistent choice is made (e.g., the smallest j).

Given a finite-state deterministic joint policy π, the verification problem is to check if there exists a set of punishment strategies $\tau = \bigcup_i \tau_i$ such that the joint policy $\tilde{\pi} = \pi \bowtie \tau = (\pi_1 \bowtie \tau_1, \ldots, \pi_n \bowtie \tau_n)$ is an ϵ-Nash equilibrium. In other words, the problem is to check if there exists a policy $\tilde{\pi}_i$ for each agent i such that (i) $\tilde{\pi}_i$ follows π_i as long as no other agent j deviates from π_j and (ii) the joint policy $\tilde{\pi} = (\tilde{\pi}_1, \ldots, \tilde{\pi}_n)$ is an ϵ-Nash equilibrium.

6.5.2. High-Level Procedure

Our approach is to compute the best set of punishment strategies τ^* w.r.t. π and check if $\pi \bowtie \tau^*$ is an ϵ-Nash equilibrium. The best punishment strategy against agent j is the one that minimizes its incentive to deviate. To be precise, we define the best response of j with respect to a joint policy $\pi' = (\pi'_1, \ldots, \pi'_n)$ to be $\mathrm{br}_j(\pi') \in \arg\max_{\pi''_j} J_j(\pi'_{-j}, \pi''_j)$. Then, the best set of punishment strategies τ^* w.r.t. π is one that minimizes the value of $\mathrm{br}_j(\pi \bowtie \tau)$ for all $j \in [n]$. To be precise, define $\tau[j] = \{\tau_{ij} \mid i \neq j\}$ to be the set of punishment strategies *against* agent j. Then, we want to compute τ^* such that for all j,

$$\tau^* \in \arg\min_{\tau} J_j((\pi \bowtie \tau)_{-j}, \mathrm{br}_j(\pi \bowtie \tau)). \tag{6.1}$$

We observe that for any two sets of punishment strategies τ, τ' with $\tau[j] = \tau'[j]$ and any policy π'_j, we have $J_j((\pi \bowtie \tau)_{-j}, \pi'_j) = J_j((\pi \bowtie \tau')_{-j}, \pi'_j)$. This is because, for any τ, punishment strategies in $\tau \setminus \tau[j]$ do not affect the behaviour of the joint policy $((\pi \bowtie \tau)_{-j}, \pi'_j)$, since no agent other than agent j will deviate from π. Hence, $\mathrm{br}_j(\pi \bowtie \tau)$ as well as $J_j((\pi \bowtie \tau)_{-j}, \mathrm{br}_j(\pi \bowtie \tau))$ are independent of $\tau \setminus \tau[j]$; therefore, we can separately compute $\tau^*[j]$ (satisfying Equation 6.1) for each j and take $\tau^* = \bigcup_j \tau^*[j]$. We now have the following theorem.

Theorem 6.6. *Given a finite-state deterministic joint policy $\pi = (\pi_1, \ldots, \pi_n)$, if there is a set of punishment strategies τ such that $\pi \bowtie \tau$ is an ϵ-Nash equilibrium, then $\pi \bowtie \tau^*$ is an ϵ-Nash equilibrium, where τ^* is the set of best punishment strategies w.r.t. π. Furthermore, $\pi \bowtie \tau^*$ is an ϵ-Nash equilibrium iff for all j,*

$$J_j((\pi \bowtie \tau^*)_{-j}, \mathrm{br}_j(\pi \bowtie \tau^*)) - \epsilon \leq J_j(\pi \bowtie \tau^*) = J_j(\pi).$$

Proof. It follows from the definition of $\pi \bowtie \tau$ that the distribution over H-length trajectories induced by $\pi \bowtie \tau$ in $\mathcal{M}$ is the same as the one induced by π since τ is never triggered when all agents are following π. Therefore, $J_j(\pi \bowtie \tau) = J_j(\pi)$ for all j and τ. Now, suppose there is a τ such that $\pi \bowtie \tau$ is an ϵ-Nash equilibrium. Then for all j,

$$J_j((\pi \bowtie \tau)_{-j}, \mathrm{br}_j(\pi \bowtie \tau)) \leq J_j(\pi \bowtie \tau) + \epsilon = J_j(\pi) + \epsilon.$$

But $\tau^*[j]$ minimizes the LHS of above equation which is independent of $\tau \setminus \tau[j]$. Hence, for all j,

$$\begin{aligned} J_j((\pi \bowtie \tau^*)_{-j}, \mathrm{br}_j(\pi \bowtie \tau^*)) &\leq J_j((\pi \bowtie \tau)_{-j}, \mathrm{br}_j(\pi \bowtie \tau)) \\ &\leq J_j(\pi) + \epsilon \\ &= J_j(\pi \bowtie \tau^*) + \epsilon. \end{aligned}$$

Therefore, $\pi \bowtie \tau^*$ is an ϵ-Nash equilibrium. The rest of the theorem follows from the definition of ϵ-Nash equilibrium. □

Thus, to solve the verification problem, it suffices to compute (or estimate), for all j, the optimal deviation scores

$$\mathtt{dev}_j^{\pi} = \min_{\tau[j]} \max_{\pi'_j} J_j((\pi \bowtie \tau)_{-j}, \pi'_j). \tag{6.2}$$

6.5.3. Reduction to Min-Max Games

Next, we describe how to reduce the computation of optimal deviation scores to a standard self-play RL setting. We first translate the problem from the specification setting to a reward-based setting using *reward machines*[15].

Reward Machines. A *reward machine (RM)* [76] is a tuple $\mathcal{R} = (Q, \delta_u, \delta_r, q_0)$ where Q is a finite set of states, $\delta_u : S \times A \times Q \to Q$ is the state transition function, $\delta_r : S \times Q \to [-1, 1]$ is the reward function and q_0 is the initial RM state. Given a trajectory $\zeta = s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{t-1}} s_t$, the reward assigned by $\mathcal{R}$ to ζ is $\mathcal{R}(\zeta) = \sum_{k=0}^{t-1} \delta_r(s_k, q_k)$, where $q_{k+1} = \delta_u(s_k, a_k, q_k)$ for all k. For any

[15] We use slightly different and simplified notation for reward machines in this chapter for clarity.

SPECTRL specification φ, we can construct an RM such that the reward assigned to a trajectory ζ indicates whether ζ satisfies φ. We provide the construction in subsection 6.5.5.

Theorem 6.7. *Given any* SPECTRL *specification φ, we can construct an RM $\mathcal{R}_\varphi$ such that for any trajectory ζ of length $t+1$, $\mathcal{R}_\varphi(\zeta) = \mathbf{1}(\zeta_{0:t} \models \varphi)$.*

For an agent j, let $\mathcal{R}_j$ denote $\mathcal{R}_{\varphi_j} = (Q_j, \delta_u^j, \delta_r^j, q_0^j)$. Letting $\tilde{\mathcal{D}}_\pi$ be the distribution over length $H+1$ trajectories induced by using π, we have $\mathbb{E}_{\zeta \sim \tilde{\mathcal{D}}_\pi}[\mathcal{R}_j(\zeta)] = J_j(\pi)$. The deviation values defined in Eq. 6.2 are now min-max values of expected reward, except that it is not in a standard min-max setting since the policy of every non-deviating agent $i \neq j$ is constrained to be of the form $\pi_i \bowtie \tau_i$. This issue can be handled by considering a product of $\mathcal{M}$ with the reward machine $\mathcal{R}_j$ and the finite-state joint policy π. This is formalized in the following theorem.

Theorem 6.8. *Given a finite-state deterministic joint policy $\pi = (M, \alpha, \sigma, m_0)$, for any agent j, we can construct a simulator for an augmented two-player zero-sum Markov game $\mathcal{M}_j^\pi$ (with rewards) which has the following properties.*

- *The number of states in $\mathcal{M}_j^\pi$ is at most $2|S||M||Q_j|$.*
- *The actions of player 1 is A_j, and the actions of player 2 is $A_{-j} = \prod_{i \neq j} A_i$.*
- *The min-max value of the two player game corresponds to the deviation cost of j, i.e.,*

$$\texttt{dev}_j^\pi = \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2),$$

  where $\bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2) = \mathbb{E}\big[\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) \mid \bar{\pi}_1, \bar{\pi}_2\big]$ is the expected sum of rewards w.r.t. the distribution over $(H+1)$-length trajectories generated by the joint policy $(\bar{\pi}_1, \bar{\pi}_2)$ in $\mathcal{M}_j^\pi$.

- *Given any policy $\bar{\pi}_2$ for player 2 in $\mathcal{M}_j^\pi$, we can construct a set of punishment strategies $\tau[j] = \text{PUNSTRAT}(\bar{\pi}_2)$ against agent j in $\mathcal{M}$ such that*

$$\max_{\bar{\pi}_1} \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2) = \max_{\pi_j'} J_j((\pi \bowtie \tau[j])_{-j}, \pi_j').$$

Given an estimate $\tilde{\mathcal{M}}$ of $\mathcal{M}$, we can also construct an estimate $\tilde{\mathcal{M}}_j^\pi$ of $\mathcal{M}_j^\pi$.

We omit the superscript π from $\mathcal{M}_j^\pi$ when there is no ambiguity. We denote by ConstructGame$(\tilde{\mathcal{M}}, j, \mathcal{R}_j, \pi)$ the product construction procedure that returns $\tilde{\mathcal{M}}_j^\pi$.

Proof of Theorem 6.8. For an agent j, the two-player zero-sum game $\mathcal{M}_j$ is defined by $\mathcal{M}_j = (S_j, A_j, A_{-j}, s_0^j, P_j, H+1, R_j)$ with rewards where,

- The set of states S_j is the product $S_j = S \times M \times Q_j \times \{\bot, \top\}$.
- The set of actions of the max-agent is A_j and the set of actions of the min-agent is $A_{-j} = \prod_{i \neq j} A_i$. Given $a_j \in A_j$ and $a_{-j} \in A_{-j}$, we denote the joint action by $(a_j, a_{-j}) \in A$.
- The last component of a state denotes whether agent j has deviated from π_j in the past or not. Intuitively, $\bot$ implies that agent j has *not* deviated from π_j in the past and $\top$ implies that it has deviated from π_j in the past. We define an update function f_j which is used to update this information at every step. The deviation update function $f_j : S \times A \times M \times \{\bot, \top\} \to \{\bot, \top\}$ is defined by $f_j(s, a, m, \top) = \top$ and $f_j(s, a, m, \bot) = \bot$ if $a_j = \sigma(s, m)_j$ and $\top$ otherwise.

  The transitions of $\mathcal{M}_j$ are such that the action of the min-agent a_{-j} is ignored and replaced with the output of π_{-j} until agent j deviates from π_j (or equivalently, until the last component of the state is $\top$). The transition probabilities are given by

  - $P_j((s, m, q, b), a, (s', m', q', b')) = P(s, (a_j, \sigma(s, m)_{-j}), s')$ if $m' = \alpha(s, (a_i, \sigma(s, m)_{-j}), m)$, $q' = \delta_u(s, (a_i, \sigma(s, m)_{-j}), q)$, $b = \bot$ and $b' = f_j(s, a, m, b)$.
  - $P_j((s, m, q, b), a, (s', m', q', b')) = P(s, a, s')$ if $m' = \alpha(s, a, m)$, $q' = \delta_u(s, a, q)$, $b = \top$ and $b' = f_j(s, a, m_j, b)$.
  - $P_j((s, m, q, b), a, (s', m', q', b')) = 0$ otherwise.
- The initial state is $s_0^j = (s_0, m_0, q_0^j, \bot)$.

- The rewards are given by $R_j((s, m, q, b), a) = \delta_r^j(s, q)$.

Let us denote by $\bar{\pi}_1$ and $\bar{\pi}_2$ the policies of the max-agent and the min-agent respectively. Then the expected reward attained by the max-agent is

$$\bar{J}_j(\bar{\pi}_1, \bar{\pi}_2) = \mathbb{E}\Big[\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) \mid \bar{\pi}_1, \bar{\pi}_2\Big]$$

where the expectation is w.r.t. the distribution over trajectories of length $H+1$ generated by using $(\bar{\pi}_1, \bar{\pi}_2)$ in $\mathcal{M}_j$. Given a trajectory $\bar{\zeta} = \bar{s_0} \xrightarrow{a_0} \bar{s_1} \xrightarrow{a_1} \ldots \xrightarrow{a_{t-1}} \bar{s}_t$ in $\mathcal{M}_j$, we denote by $\bar{\zeta} \downarrow_{\mathcal{M}}$ the trajectory projected to the state space of $\mathcal{M}$—i.e., $\bar{\zeta} \downarrow_{\mathcal{M}} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{t-1}} s_t$. From Theorem 6.7 and the above definition of $\mathcal{M}_j$ it follows that for any trajectory $\bar{\zeta}$ in $\mathcal{M}_j$ of length $H+1$ we have

$$\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) = \mathcal{R}_j(\bar{\zeta} \downarrow_{\mathcal{M}}) = \mathbb{1}(\bar{\zeta}_{0:H} \downarrow_{\mathcal{M}} \models \varphi_j).$$

Let $\mathcal{D}^{\mathcal{M}}[\pi]$ denote the distribution over length H trajectories in $\mathcal{M}$ generated by π and $\mathcal{D}^{\mathcal{M}_j}[\bar{\pi}]$ denote the distribution over length $H+1$ trajectories in $\mathcal{M}_j$ generated by $\bar{\pi}$. It is easy to see that any policy π'_j for agent j in $\mathcal{M}$ can be interpreted as a policy $g(\pi'_j)$ for the max-agent in $\mathcal{M}_j$ and any policy $\bar{\pi}_1$ for the max-agent in $\mathcal{M}_j$ can be interpreted as a policy $g'(\bar{\pi}_1)$ for agent j in $\mathcal{M}$. Since the actions of the min-agent in $\mathcal{M}_j$ are only taken into account after agent j deviates from π_j, we also have that any policy of the form $(\pi \bowtie \tau)_{-j}$ for the punishing agents in $\mathcal{M}$ corresponds to a policy $h(\tau)$ of the min-agent in $\mathcal{M}_j$ and any policy $\bar{\pi}_2$ of the min-agent in $\mathcal{M}_j$ corresponds to a policy $(\pi \bowtie h'(\bar{\pi}_2))_{-j}$ for the punishing agents in $\mathcal{M}$. Furthermore the mappings g, g', h, h' satisfy the property that for any trajectory $\bar{\zeta}$ of length $H+1$ in $\mathcal{M}_j$, we have

- for any τ and π'_j in $\mathcal{M}$, $\mathcal{D}^{\mathcal{M}}[(\pi \bowtie \tau)_{-j}, \pi'_j](\bar{\zeta} \downarrow_{\mathcal{M}}) = \mathcal{D}^{\mathcal{M}_j}[g(\pi'_j), h(\tau)](\bar{\zeta})$ and,
- for any $\bar{\pi}_1$ and $\bar{\pi}_2$ in $\mathcal{M}$, $\mathcal{D}^{\mathcal{M}}[(\pi \bowtie h'(\bar{\pi}_2))_{-j}, g'(\bar{\pi}_1)](\bar{\zeta} \downarrow_{\mathcal{M}}) = \mathcal{D}^{\mathcal{M}_j}[\bar{\pi}_1, \bar{\pi}_2](\bar{\zeta})$.

Therefore we have

$$\begin{aligned}
\mathtt{dev}_j^{\pi} &= \min_{\tau[j]} \max_{\pi'_j} J_j((\pi \bowtie \tau)_{-j}, \pi'_j) \\
&= \min_{\tau[j]} \max_{\pi'_j} \mathbb{E}_{\zeta \sim \mathcal{D}^{\mathcal{M}_{[(\pi \bowtie \tau)_{-j}, \pi'_j]}}} \Big[\zeta \models \varphi_i \Big] \\
&= \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \mathbb{E}_{\bar{\zeta} \sim \mathcal{D}^{\mathcal{M}_j[\bar{\pi}_1, \bar{\pi}_2]}} \Big[\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) \Big] \\
&= \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}_j(\bar{\pi}_1, \bar{\pi}_2).
\end{aligned}$$

It is easy to see that the function h' has the desired properties of PunStrat. Finally, we observe that given a simulator for $\mathcal{M}$ it is straightforward to construct a simulator for $\mathcal{M}_j$. Similarly, given an estimate $\tilde{\mathcal{M}}$ of $\mathcal{M}$ we can use the above definition of $\mathcal{M}_j$ to construct an estimate $\tilde{\mathcal{M}}_j$ of $\mathcal{M}_j$. □

6.5.4. Solving Min-Max Games

The min-max game $\mathcal{M}_j$ can be solved using self-play RL algorithms. Many of these algorithms provide probabilistic approximation guarantees for computing the min-max value of the game. We use a model-based algorithm, similar to the one proposed in [20], that first estimates the model $\mathcal{M}_j$ and then solves the game in the estimated model.

Estimating the Markov Game. One approach is to use existing algorithms for reward-free exploration to estimate the model [83], but this approach requires estimating each $\mathcal{M}_j$ separately. Under Assumption 6.1, we provide a simpler and more sample-efficient algorithm, called BFS-Estimate, for estimating $\mathcal{M}$ which is outlined in Algorithm 5. BFS-Estimate performs a search over the transition graph of $\mathcal{M}$ by exploring previously seen states in a breadth first manner. In order to figure out all outgoing edges from a state s, multiple samples are collected by taking each possible action K-times from s. Newly discovered states are then added to the state space of $\tilde{\mathcal{M}}$ and the collected samples are used to estimate transition probabilities. The value K is defined in line 2 of the algorithm in which $|Q|$ denotes the maximum size of the state space of reward machine $\mathcal{R}_j$ for any agent j—i.e., $|Q| = \max_j |Q_j|$. BFS-Estimate has the following approximation guarantee.

Algorithm 5 BFS-Estimate

Inputs: Precision δ, failure probability p.

Outputs: Estimated model $\tilde{\mathcal{M}}$ of $\mathcal{M}$.

$\tilde{M} \leftarrow (\tilde{S} = \{s_0\}, A, s_0, \tilde{P} = \emptyset, H)$

$K \leftarrow \left\lceil \frac{2|S|^2|M|^2|Q|^2H^4}{\delta^2} \log\left(\frac{2|S|^2|A|}{p}\right) \right\rceil$

`queue` $\leftarrow [s_0]$

while $\neg$ `queue.isempty()` **do**

  $s \leftarrow$ `queue.pop()`

  for $a \in A$ **do**

    // Initialize number of visits to each state

    $N \leftarrow$ `empty-map()`

    for $s' \in \tilde{S}$ **do**

      $N[s'] \leftarrow 0$

    end for

    // Obtain K samples for the state-action pair (s, a)

    for $x \in \{1, \ldots, K\}$ **do**

      $s' \sim P(s, a, \cdot)$

      // Add any newly discovered state to $\tilde{S}$ and the map N

      if $s' \notin \tilde{S}$ **then**

        $\tilde{S} \leftarrow \tilde{S} \cup \{s'\}$

        `queue.add`(s')

        $N[s'] \leftarrow 0$

      end if

      // Increment number of visits to s'

      $N[s'] \leftarrow N[s'] + 1$

    end for

    // Store estimated transition probabilities in $\tilde{P}$

    for $s' \in \tilde{S}$ **do**

      $\tilde{P}(s, a, s') \leftarrow \frac{N[s']}{K}$

    end for

  end for

end while

return $\tilde{\mathcal{M}}$

Lemma 6.9. *With probability at least* $1-p$*, for all* $s \in \tilde{S}$*,* $a \in A$ *and* $s' \in S$*,*

$$\left|\tilde{P}(s,a,s') - P(s,a,s')\right| \leq \varepsilon = \frac{\delta}{2|S||M||Q|H^2}$$

where $\tilde{P}(s,a,s')$ *is taken to be* 0 *if* $s' \notin \tilde{S}$*.*

Proof. For any given $s \in \tilde{S}$, $a \in A$ and $s' \in S$, the probability $\tilde{P}(s,a,s')$ is estimated using K independent samples from $P(s,a,\cdot)$. Therefore, using Chernoff bounds, we get

$$\Pr\left[\left|\tilde{P}(s,a,s') - P(s,a,s')\right| > \varepsilon\right] \leq 2e^{-2K\varepsilon^2}$$

Applying union bound over all triples $(s,a,s') \in \tilde{S} \times A \times S$ and substituting the values of K and ε, we get

$$\begin{aligned}\Pr\left[\bigcup_{s,a,s'}\left\{\left|\tilde{P}(s,a,s') - P(s,a,s')\right| > \varepsilon\right\}\right] &\leq 2|S|^2|A|e^{-2K\varepsilon^2} \\ &\leq 2|S|^2|A|e^{-\log(\frac{2|S|^2|A|}{p})} = p.\end{aligned}$$

Hence we obtained the desired bound. □

Obtaining estimates of $\mathcal{M}_j^\pi$. After estimating $\mathcal{M}$, for any finite-state deterministic joint policy π and any agent j, we perform the product construction outlined in Section 6.5.3 with the estimated model $\tilde{\mathcal{M}}$ to obtain an estimate $\tilde{\mathcal{M}}_j^\pi$ of the punishment game $\mathcal{M}_j^\pi$. The constructed model $\tilde{\mathcal{M}}_j^\pi$ can be used to estimate $\mathtt{dev}_j^\pi$ as claimed in the following theorem.

Theorem 6.10. *For any* $\delta > 0$ *and* $p \in (0,1]$*,* BFS-ESTIMATE$(\mathcal{M}, \delta, p)$ *computes an estimate* $\tilde{\mathcal{M}}$ *of* $\mathcal{M}$ *using* $O\left(\frac{|S|^3|M|^2|Q|^4|A|H^4}{\delta^2}\log\left(\frac{|S||A|}{p}\right)\right)$ *sample steps such that with probability at least* $1-p$*, for any finite-state deterministic joint policy* π *and any agent* j*,*

$$\left|\min_{\bar{\pi}_2}\max_{\bar{\pi}_1}\bar{J}^{\tilde{\mathcal{M}}_j^\pi}(\bar{\pi}_1,\bar{\pi}_2) - \mathtt{dev}_j^\pi\right| \leq \delta,$$

Algorithm 6 VERIFYNASH

Inputs: Finite-state deterministic joint policy π, specifications φ_j for all j, Nash factor ϵ, precision δ, failure probability p.

Outputs: `True` or `False` along with a set of punishment strategies τ.

```
existsNE ← True
τ ← ∅
M̃ ← BFS-ESTIMATE(M, δ, p) // Only run if M has not been estimated before.
for agent j ∈ {1, ..., n} do
    R_j ← CONSTRUCTRM(φ_j)
    M̃_j ← CONSTRUCTGAME(M̃, j, R_j, π)
    deṽ_j ← min_{π̄2} max_{π̄1} J̄^{M̃_j}(π̄1, π̄2)
    π̄2* ← arg min_{π̄2} max_{π̄1} J̄^{M̃_j}(π̄1, π̄2)
    existsNE ← existsNE ∧ (deṽ_j ≤ J_j(π) + ε − δ)
    τ ← τ ∪ PUNSTRAT(π̄2*)
end for
return existsNE, τ
```

where $\bar{J}^{\tilde{\mathcal{M}}_j^{\pi}}(\bar{\pi}_1, \bar{\pi}_2)$ *is the expected reward over length* $H+1$ *trajectories generated by* $(\bar{\pi}_1, \bar{\pi}_2)$ *in* $\tilde{\mathcal{M}}_j^{\pi}$. *Furthermore, letting* $\bar{\pi}_2^* \in \arg\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$ *and* $\tau[j] = \text{PUNSTRAT}(\bar{\pi}_2^*)$, *we have*

$$\left| \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j^{\pi}}(\bar{\pi}_1, \bar{\pi}_2^*) - \max_{\pi_j'} J_j((\pi \bowtie \tau[j])_{-j}, \pi_j') \right| \leq \delta. \tag{6.3}$$

Proof. Since the transition probabilities in $\tilde{\mathcal{M}}_j^{\pi}$ are inherited from $\tilde{\mathcal{M}}$, from Lemma 6.9 we have that with probability at least $1-p$, for any π, j, $\bar{s}, \bar{s}' \in S_j$ and $a \in A$ such that $\bar{s}$ is in the state space of $\tilde{\mathcal{M}}_j^{\pi}$,

$$\left|\tilde{P}_j(\bar{s}, a, \bar{s}') - P_j(\bar{s}, a, \bar{s}')\right| \leq \varepsilon = \frac{\delta}{2|S||M||Q|H^2}$$

where $\tilde{P}_j$ represents the transition probabilities of $\tilde{\mathcal{M}}_j$[16]. Consider the model $\tilde{\mathcal{M}}_j'$ which is a modification of $\tilde{\mathcal{M}}_j$ that has state space S_j (state space of $\mathcal{M}_j$) and for any $\bar{s}, \bar{s}' \in S_j$ and $a \in A$, its transition probabilities are defined by $\tilde{P}_j'(\bar{s}, a, \bar{s}') = \tilde{P}_j(\bar{s}, a, \bar{s}')$ if $\bar{s}$ is in the state space of $\tilde{\mathcal{M}}_j$ and $P_j(\bar{s}, a, \bar{s}')$ otherwise. We have $\left|\tilde{P}_j'(\bar{s}, a, \bar{s}') - P_j(\bar{s}, a, \bar{s}')\right| \leq \varepsilon$ for all $\bar{s}, \bar{s}' \in S_j$ and $a \in A$. For any two policies $\bar{\pi}_1$ and $\bar{\pi}_2$ of the max-agent and the min-agent in $\mathcal{M}_j$ respectively, we denote by $\bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$ and $\bar{J}^{\tilde{\mathcal{M}}_j'}(\bar{\pi}_1, \bar{\pi}_2)$ the expected reward over $H+1$ length trajectories generated by $(\bar{\pi}_1, \bar{\pi}_2)$ in $\tilde{\mathcal{M}}_j$ and $\tilde{\mathcal{M}}_j'$ respectively. Then $\bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) = \bar{J}^{\tilde{\mathcal{M}}_j'}(\bar{\pi}_1, \bar{\pi}_2)$ because both the models assign the same

[16] Omitting the superscript π in $\tilde{\mathcal{M}}_j^{\pi}$

probability to all runs as any run that leaves the state space of $\tilde{\mathcal{M}}_j$ has probability zero in both the models. Now we can apply Lemma 4 of [30] to conclude that $|\bar{J}^{\tilde{\mathcal{M}}'_j}(\bar{\pi}_1, \bar{\pi}_2) - \bar{J}^{\pi}_j(\bar{\pi}_1, \bar{\pi}_2)| \leq \delta$ and hence $|\bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \bar{J}^{\pi}_j(\bar{\pi}_1, \bar{\pi}_2)| \leq \delta$ for any $\bar{\pi}_1$ and $\bar{\pi}_2$. This implies that for any $\bar{\pi}_2$ we have

$$|\max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \max_{\bar{\pi}_1} \bar{J}^{\pi}_j(\bar{\pi}_1, \bar{\pi}_2)| \leq \delta \tag{6.4}$$

and therefore can conclude that

$$|\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\pi}_j(\bar{\pi}_1, \bar{\pi}_2)| \leq \delta.$$

Applying Theorem 6.8 we get

$$|\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \mathtt{dev}^{\pi}_j| \leq \delta.$$

Now, let $\bar{\pi}^*_2 = \arg\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$ and $\tau[j] = \textsc{PunStrat}(\bar{\pi}^*_2)$. Then from Equation 6.4 we can conclude that $|\max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}^*_2) - \max_{\bar{\pi}_1} \bar{J}^{\pi}_j(\bar{\pi}_1, \bar{\pi}^*_2)| \leq \delta$. Using Theorem 6.8 we get

$$\left| \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}^{\pi}_j}_j(\bar{\pi}_1, \bar{\pi}^*_2) - \max_{\pi'_j} J_j((\pi \bowtie \tau[j])_{-j}, \pi'_j) \right| \leq \delta.$$

Finally the total number of samples used is at most $|S||A|K = O\left(\frac{|S|^3|M|^2|Q|^4|A|H^4}{\delta^2} \log\left(\frac{|S||A|}{p}\right)\right)$. $\square$

Our full verification algorithm is summarized in Algorithm 6. For each j, the min-max game $\tilde{\mathcal{M}}_j$ is solved in polynomial time using value iteration [20] to compute an estimate $\tilde{\mathtt{dev}}_j$ of $\mathtt{dev}^{\pi}_j$ which is used in Line 9 of Algorithm 6 to check whether agent j can successfully deviate from π_j. It checks if $\tilde{\mathtt{dev}}_j \leq J_j(\pi) + \epsilon - \delta$ for all j, and returns `True` if so and `False` otherwise. It also simultaneously computes the punishment strategies τ using the optimal policies for player 2 in the punishment games. Note that BFS-Estimate is called only once (i.e., the first time VerifyNash is called) and the obtained estimate $\tilde{\mathcal{M}}$ is stored and used for verification of every candidate policy π. The following soundness guarantee follows from Theorem 6.10.

Corollary 6.11 (Soundness)**.** *For any $p \in (0, 1]$, $\varepsilon > 0$ and $\delta \in (0, \varepsilon)$, with probability at least $1-p$, if* HighNashSearch *returns a joint policy $\tilde{\pi}$ then $\tilde{\pi}$ is an ϵ-Nash equilibrium.*

Proof. From Theorem 6.10 we get that with probability at least $1 - p$, if a policy $\tilde{\pi} = \pi \bowtie \tau$ is returned by our algorithm, then for all j we have

$$\begin{aligned}
\max_{\pi'_j} J_j((\pi \bowtie \tau)_{-j}, \pi'_j) &\leq \tilde{\texttt{dev}}_j + \delta && \text{(Equation 6.3)} \\
&\leq J_j(\pi) + \epsilon && \text{(Line 9 of Algorithm 6)} \\
&= J_j(\pi \bowtie \tau) + \epsilon.
\end{aligned}$$

Therefore, $\pi \bowtie \tau$ is an ϵ-Nash equilibrium. □

6.5.5. Reward Machine Construction

In this section, we detail the construction of reward machines from Spectrl specifications such that the reward of any finite-length trajectory is 1 if the trajectory satisfies the specification and 0 otherwise. This section can be skipped without loss of continuity and is provided for completeness.

We begin by constructing deterministic finite-state automata (DFA) that accepts all trajectories which satisfy the specification. Next, we will convert the DFA into a reward machine with the desired reward function.

DFA Construction. A finite-state automaton is a tuple $D = (Q, \mathcal{B}, \Delta, q_{\mathsf{init}}, F)$ where Q is a finite-set of states, $\mathcal{B}$ is a finite-set of propositions, q_{init} is the initial state, and $F \subseteq Q$ is the set of accepting states. The transition relation is defined as $\Delta \subseteq Q \times \mathsf{Formula}(\mathcal{B}) \times Q$ where $\mathsf{Formula}(\mathcal{B})$ is the set of boolean formulas over propositions $\mathcal{B}$. A finite-state automaton is *deterministic* if every assignment $\sigma \in 2^{\mathcal{P}}$ can transition to a unique state from every state, i..e, if for all states $q \in Q$ and assignments $\sigma \in 2^{\mathcal{B}}$, $|\{q' \mid (q, b, q') \in \Delta \text{ and } \sigma \models b\}| \leq 1$. Otherwise, it is *non-deterministic*. Every non-deterministic finite-state automata (NFA) can be converted to a deterministic finite-state automata (DFA). A *run* of a *word* (sequence of assignments over $\mathcal{B}$) given by $w = w_0 \dots w_m \in (2^{\mathcal{B}})^*$ is a sequence of states $\rho = q_0 \dots q_{m+1}$ such that $q_0 = q_{\mathsf{init}}$ and there exists $(q_i, b_i, q_{i+1}) \in \Delta$ such

that $w_i \models b_i$ for all $0 \leq i < m$. A run $q_0 \dots q_{m+1}$ is accepting if $q_{m+1} \in F$. A word w is accepted by D if it has an accepting run.

Let SPECTRL specifications be defined over the set of basic predicates $\mathcal{P}_0$. We define a labelling function $L : S \to 2^{\mathcal{P}_0}$ such that $L(s) = \{p \mid [\![p]\!](s) = \texttt{true}\}$. Given a trajectory $\zeta = s_0, \dots, s_t$ in the environment $\mathcal{M}$, let its proposition sequence $\mathcal{L}(\zeta)$ be given by $\mathcal{L}(s_0), \dots, \mathcal{L}(s_t)$.

Lemma 6.12. *Given* SPECTRL *specification φ, we can construct a DFA D_φ such that a trajectory $\zeta \models \varphi$ iff $\mathcal{L}(\zeta)$ is accepted by D_φ.*

Proof. We use structural induction on SPECTRL specifications to construct the desired DFA. The construction is very similar to the construction of finite-state automata from regular expressions, and hence details of proof have been skipped [71]. The construction is given below:

Eventually ($\varphi ::= \texttt{achieve}\ b$). Construct finite-state automata $D_\varphi = (\{q_{\mathsf{init}}, q\}, \mathcal{P}_0, \Delta, q_{\mathsf{init}}, \{q\})$ where

$$\Delta = \{(q_{\mathsf{init}}, \neg b, q_{\mathsf{init}}), (q_{\mathsf{init}}, b, q), (q, \mathsf{True}, q)\}.$$

Clearly, D_φ is deterministic because the only state from which more that two transitions emanate is q_{init} and these transitions are defined on predicate functions that negate one another (b and $\neg b$).

Always ($\varphi ::= \varphi_1\ \texttt{ensuring}\ b$). Let the DFA for φ_1 be $D_1 = (Q, \mathcal{P}_0, \Delta_1, q_{\mathsf{init}}, F)$. Then the DFA for φ is given by $D_\varphi = (Q, \mathcal{P}_0, \Delta, q_{\mathsf{init}}, F)$ where

$$\Delta = \{(q, b \wedge b', q') \mid (q, b', q') \in \Delta_1\}.$$

D_φ is deterministic because D_1 is deterministic.

Sequencing ($\varphi_1; \varphi_2$). Let the DFA for φ_1 and φ_2 be $D_1 = (Q_1, \mathcal{P}_0, \Delta_1, q^1_{\mathsf{init}}, F_1)$ and $D_2 = (Q_2, \mathcal{P}_0, \Delta_2, q^2_{\mathsf{init}}, F_2)$, respectively. Construct the NFA $N_\varphi = (Q_1 \sqcup Q_2, \mathcal{P}_0, \Delta, q^1_{\mathsf{init}}, F_2)$ with

$$\Delta = \Delta_1 \cup \Delta_2 \cup \bigcup_{f \in F_1} \mathsf{DivertAwayFrom}(f)$$

where $\mathsf{DivertAwayFrom}(f) = \{(q_1^1, b, q_{\mathsf{init}}^2)\} \mid (q_1^1, b, f) \in \Delta_1\}$. Essentially, transitions in $\mathsf{DivertAwayFrom}(f)$ divert all incoming transitions to $f \in F_1$ to the initial state of the second DFA. Then, the DFA D_φ is obtained from determinization of N_φ.

Choice ($\varphi ::= \varphi_1$ `or` φ_2). Let the DFA for φ_1 and φ_2 be $D_1 = (Q_1, \mathcal{P}_0, \Delta_1, q_{\mathsf{init}}^1, F_1)$ and $D_2 = (Q_2, \mathcal{P}_0, \Delta_2, q_{\mathsf{init}}^2, F_2)$, respectively. Construct NFA $N_\varphi = (Q_1 \sqcup Q_2 \setminus \{q_{\mathsf{init}}^2\}, \mathcal{P}_0, \Delta, q_{\mathsf{init}}^1, F_1 \sqcup F_2)$ where

$$\begin{aligned}\Delta = \ & \Delta_1 \cup \Delta_2 \setminus \{(q_1^2, b, q_2^2) \mid (q_1^2, b, q_2^2) \in \Delta_2 \text{ and } q_1^2 = q_{\mathsf{init}}^2\} \\ & \cup \{(q_{\mathsf{init}}^1, b, q_2^2) \mid (q_1^2, b, q_2^2) \in \Delta_2 \text{ and } q_1^2 = q_{\mathsf{init}}^2\}.\end{aligned}$$

Then, the DFA D_φ is obtained from determinization of N_φ.

Lastly, we can extend DFA D_φ to make it *complete*, i.e., if for all states $q \in Q$ and assignments $\sigma \in 2^{\mathcal{P}_0}$, $|\{q' \mid (q, b, q') \in \Delta \text{ and } \sigma \models b\}| = 1$. □

Reward Machine. Given a SPECTRL specification φ, let $D_\varphi = (Q, \mathcal{P}_0, \Delta, q_{\mathsf{init}}, F)$ be the DFA such that a trajectory $\zeta \models \varphi$ iff $L(\zeta)$ is accepted by the DFA D_φ, where $L : \mathcal{S} \to 2^{\mathcal{P}_0}$ is the labelling function. WLOG, assume $\mathcal{D}_\varphi$ is complete. We construct a reward machine $\mathcal{R}_\varphi = (Q \cup \{\mathsf{dead}\}, \delta_u, \delta_r, q_{\mathsf{init}})$ where the state transition function $\delta_u : S \times A \times Q \to Q$ is defined as

$$\delta_u(s, -, q) = q' \text{where } (q, b, q') \in \Delta \text{ s.t. } L(s) \models b$$

and the reward function $\delta_r : S \times Q \to [-1, 1]$ is given by

$$\delta_r(s, q) = \begin{cases} 1 & \text{if } q \notin F, q' = \delta_u(s, -, q') \text{ and } q' \in F \\ -1 & \text{if } q \in F, q' = \delta_u(s, -, q') \text{ and } q' \notin F \\ 0 & \text{otherwise} \end{cases}$$

Observe that the above functions are well defined since the DFA is deterministic and complete.

Proof of Theorem 6.7. Let $\mathcal{R}_\varphi$ be as constructed above. Let $\zeta = s_0, s_1, \ldots s_t, s_{t+1}$. Then, by con-

struction a run $\rho = q_0, q_1 \dots q_{t+1}$ of $L(\zeta_{0:t})$ in D_φ is also a run of ζ in $\mathcal{R}_\varphi$. Then, the reward function is design so that (a) each time the run visits a state in F from a non-accepting state, it will receive a reward of 1, (b) each time the run visits a state in $Q \setminus F$ from a state in F, and it receives -1, and (c) 0 otherwise.

Suppose $\zeta_{0:t}$ does not satisfy φ. Then ρ is not an accepting run in D_φ. Then, each time the run visits a state in F, the run will exit states in F after a finite amount of time. Thus, either ζ receives a reward of 0 or it receives a reward of 1 and -1 an equal number of times. In this case, $\mathcal{R}_\varphi(\zeta) = 0$ since the $+1$s and -1s will cancel each other out.

Suppose $\zeta \models \varphi$. Then, ρ is an accepting run in D_φ. Let k be the largest index such that $q_k \notin F$ and $q_\ell \in F$ for all $k < \ell \leq t+1$. So, $\delta_r(s_k, q_k) = 1$ and $\delta_r(s_\ell, q_\ell) = 0$ for all $k < \ell \leq t$. Additionally, the run $q_0, \dots, q_k$ is not an accepting run in $\mathcal{D}_\varphi$. Thus, the trajectory $\zeta_{0:k-1}$ does not satisfy φ, thus $\mathcal{R}_\varphi(\zeta_{0:k-1}) = 0$. So, $\mathcal{R}_\varphi(\zeta) = \Sigma_{z=0}^{t} \delta_r(s_z, q_z) = \Sigma_{z=0}^{k-1} \delta_r(s_z, q_z) + \delta_r(s_k, q_k) + \Sigma_{z=k+1}^{t} \delta_r(s_z, q_z)$. Since $\Sigma_{z=0}^{k-1} \delta_r(s_z, q_z) = \Sigma_{z=k+1}^{t} \delta_r(s_z, q_z) = 0$, we get that $\mathcal{R}_\varphi(\zeta) = 1$. □

6.6. Complexity

In this section, we analyze the time and sample complexity of the complete MARL algorithm in terms of the number of agents n, size of the specification $|\varphi| = \max_{i \in [n]} |\varphi_i|$, number of states in the environment $|S|$, number of joint actions $|A|$, time horizon H, precision δ and failure probability p.

Sample Complexity. We know that the number of edges in the abstract graph $\mathcal{G}_i$ corresponding to specification φ_i is $O(|\varphi_i|^2)$. Hence for any set of active agents B, the number of edges in the product abstract graph $\mathcal{G}_B$ is $O(|\varphi|^{2|B|})$. Hence total number of edge policies learned by our compositional RL algorithm is $\sum_{B \subseteq [n]} O((|\varphi|^2)^{|B|}) = O((|\varphi|^2 + 1)^n)$. We learn each edge using a fixed number of sample steps C, which is a hyperparameter.

The number of samples used in the verification phase is the same as the number used by BFS-ESTIMATE. The maximum size of a candidate policy output by the enumeration algorithm $|M|$ is at most the length of the longest path in a product abstract graph. Since the maximum path length in a single abstract graph $\mathcal{G}_i$ is bounded by $|\varphi_i|$ and at least one agent must progress along

every edge in a product graph, the maximum length of a path in any product graph is at most $n|\varphi|$. Also, the number of states in the reward machine $\mathcal{R}_j$ corresponding to $|\varphi_j|$ is $O(2^{|\varphi_j|})$. Hence, from Theorem 6.10 we get that the total number of sample steps used by our algorithm is $O\big((|\varphi|^2+1)^n C + \frac{2^{4|\varphi|}|S|^3 n^2 |\varphi|^2 |A| H^4}{\delta} \log\big(\frac{|S||A|}{p}\big)\big)$.

Time Complexity. As with sample complexity, the time required to learn all edge policies is $O((|\varphi|^2+1)^n(C+|A|))$ where the term $|A|$ is added to account for the time taken to select an action from A during exploration (we use Q-learning with ε-greedy exploration for learning edge policies). Similarly, time taken for constructing the reward machines and running BFS-Estimate is $O(\frac{2^{4|\varphi|}|S|^3 n^2 |\varphi|^2 |A| H^4}{\delta} \log\big(\frac{|S||A|}{p}\big))$.

The total number of path policies considered for a given set of active agents B is bounded by the number of paths in the product abstract graph $\mathcal{G}_B$ that terminate in a final product state. First, let us consider paths in which exactly one agent progresses in each edge. The number of such paths is bounded by $(|B||\varphi|)^{|B||\varphi|}$ since the length of such paths is bounded by $|B||\varphi|$ and there are at most $|B||\varphi|$ choices at each step—i.e., progressing agent j and next vertex of the abstract graph $\mathcal{G}_{\varphi_j}$. Now, any path in $\mathcal{G}_B$ can be constructed by merging adjacent edges along such a path (in which at most one agent progresses at any step). The number of ways to merge edges along such a path is bounded by the number of groupings of edges along the path into at most $|B||\varphi|$ groups which is bounded by $(|B||\varphi|)^{|B||\varphi|}$. Therefore, the total number of paths in $\mathcal{G}_B$ is at most $2^{2|B||\varphi|\log(n|\varphi|)}$. Finally, the total number of path policies considered is at most $\sum_{B\subseteq[n]} 2^{2|B||\varphi|\log(n|\varphi|)} \leq ((n|\varphi|)^{2|\varphi|}+1)^n = O(2^{2n|\varphi|\log(2n|\varphi|)})$.

Now, for each path policy π, the verification algorithm solves $\tilde{\mathcal{M}}_j^\pi$ using value iteration which takes $O(|\tilde{S}||A|Hf(|A|)) = O(2^{|\varphi|}n|\varphi||S||A|Hf(|A|))$ time, where $f(|A|)$ is the time required to solve a linear program of size $|A|$. Also accounting for the time taken to sort the path policies, we arrive at a time complexity bound of $2^{O(n|\varphi|\log(n|\varphi|))}\text{poly}(|S|,|A|,H,\frac{1}{p},\frac{1}{\delta})$.

It is worth noting that the procedure halts as soon as our verification procedure successfully verifies a policy; this leads to early termination for cases where there is a high value ϵ-Nash equilibrium

(among the policies considered). Furthermore, our verification algorithm runs in polynomial time and therefore one could potentially improve the overall time complexity by reducing the search space in the prioritized enumeration phase—e.g., by using domain specific insights.

6.7. Experiments

We evaluated our algorithm on finite state environments and a variety of specifications, aiming to answer the following:

- Can our approach be used to learn ϵ-Nash equilibria?
- Can our approach learn policies with high social welfare?

We compared our approach to two baselines described below, using two metrics: (i) the social welfare $\texttt{welfare}(\pi)$ of the learned joint policy π, and (ii) an estimate of the minimum value of ϵ for which π forms an ϵ-Nash equilibrium:

$$\epsilon_{\min}(\pi) = \max\{J_i(\pi_{-i}, \mathrm{br}_i(\pi)) - J_i(\pi) \mid i \in [n]\}.$$

Here, $\epsilon_{\min}(\pi)$ is estimated using single agent RL (specifically, Q-learning) to compute $\mathrm{br}_i(\pi)$ for each agent i.

6.7.1. Environments and Specifications

Single Lane Environment. The environment consists of k agents along a straight track of length l. All agents are initially placed at the 0^{th} location and the destination is at the l^{th} location. In a single step, each agent can either move forward one location (with a failure probability of 0.05) or remain in its current position. We create competitive and co-operative scenarios through agent specifications. For example, we can create competitive scenarios in which an agent meets its specification only if it reaches the final location before all other or a set of other agents. We can also create co-operative scenarios in which the i-th agent must reach its goal before the j-th agent, for various pairs of agents (i, j). In these cases, the highest social welfare would occur when the agents manage to coordinate so that all ordering constraints are satisfied (we ensure that there are

no cycles in the ordering constraints). The specifications are described below.

φ^1 All agents should reach the final state

φ^2 Agent 1 should reach its destination before Agent 0. Agent 2 must reach the destination.

φ^3 Agent 1 should reach its destination before Agent 0. All agents must reach their destination.

φ^4 Agent 0 and Agent 1 are competing to reach the destination first. Only the agent reaching first meets the specification. Agent 2's goal is to reach the destination.

φ^5 Agent 0 should reach the mid-point before Agent 1. However, Agent 1 should reach the final destination before Agent 0. All agents should eventually reach the final destination.

φ^6 Agent 0 should reach the mid-point before Agent 1. However, Agent 1 and Agent 2 should reach the final destination before Agent 0. All agents should reach the final destination.

Intersection Environment. This is the environment illustrated in Figure 6.1, which consists of k-cars (agents) at a 2-way intersection of which k_1 and k_2 cars are placed along the N-S and E-W axes, respectively. The state consists of the location of all cars where the location of a single car is a non-negative integer. 1 corresponds to the intersection, 0 corresponds to the location one step towards the south or west of the intersection (depending on the car) and locations greater than 1 are to the east or north of the intersection. Each agent has two actions. `STAY` stays at the current position. `MOVE` decreases the position value by 1 with probability 0.95 and stays with probability 0.05. The specifications and the corresponding initial states are described below.

φ^1 Two N-S cars both starting at 3 and one E-W car starting at 2. N-S cars' goal is to reach 0 before the E-W car without collision. E-W car's goal is to reach 0 before both N-S agents without collision.

φ^2 Same as motivating example except that blue car is not required to stay a car length away from green and orange cars.

φ^3 Same as motivating example.

φ^4 Two N-S agents (0 and 1) both starting at 3 and one E-W agent (2) starting at 3. Agent 0's task is to reach 0 before other two agents. Agent 1's task is to reach 0. Agent 2's task is to reach 0 before agent 1. All agents must avoid collision.

φ^5 Two N-S cars starting at 2 and 3 and three E-W cars all starting at 2, 3 and 4, respectively. N-S cars' goal is to reach 0 before the E-W cars without collision. E-W cars' goal is to reach 0 before both N-S cars without collision.

Gridworld Environment. The environment is a 4×4 discrete grid with 2-agents. The agents are initially placed at opposite corners of the grid. In every step, each agent can either move in one of four cardinal directions (with a failure probability) or remain in position. The task of each agent is to visit a series of locations on the grid while ensuring no collision between the agents. The agents must learn to coordinate between themselves to accomplish their tasks. As an example, each agent's task is to visit any one of the other two corners in the grid. In this case, the agents must learn to choose and navigate to different corners to minimize their risk of collision. This specification can be increased in length (thus, in complexity) by sequencing visits to more locations on the grid. All scenarios are cooperative. The specifications are described below.

φ^1 Swap the positions of both agents without collision.

φ^2 Both agents should choose to visit one of the other two corners of the grid without collision.

φ^3 Append φ^2 with the specification to reach the corner that is diagonally opposite the initial position of the agent without collision.

φ^4 Append φ^3 with the agents swapping their positions without collision.

φ^5 Append φ^4 with the agents swapping their positions again without collision.

Algorithm 7 Nash Value Iteration
Inputs: n-agent Markov game $\mathcal{M}$ with rewards, horizon H.
Outputs: Nash equilibrium joint policy $\pi = (\pi_1, \ldots, \pi_n)$.

Initialize joint policy $\pi = (\pi_1, \ldots, \pi_n)$
Initialize value function $V : S \times [H+1] \to \mathbb{R}^n$ to be the zero map
for $t \in \{H, H-1, \ldots, 1\}$ **do**
  for $s \in S$ **do**
    Initialize step game $G_s^t : A \to \mathbb{R}^n$
    for $a = (a_1, \ldots, a_n) \in A$ **do**
      $G_s^t(a_1, \ldots, a_n) = R(s, a) + \mathbb{E}_{s' \sim P(s,a,\cdot)}[V(s', t+1)]$
    end for
    $(d_1, d_2, \ldots, d_n) \leftarrow$ Best-Nash-General-Sum$(G_s^t) \in \mathcal{D}(A_1) \times \cdots \times \mathcal{D}(A_n)$
    $V(s,t) \leftarrow \mathbb{E}_{a_1 \sim d_1, a_2 \sim d_2, \ldots, a_n \sim d_n}[G_s^t(a_1, \ldots, a_n)]$
    $\pi(s,t) \leftarrow (d_1, d_2, \ldots, d_n)$
  end for
end for
return π

6.7.2. Baselines

We compared our NE computation method (HighNashSearch) to two approaches (maqrm and nvi) for learning in non-cooperative games. Both maqrm and nvi learn from rewards as opposed to specification; thus, we supply rewards in the form of reward machines constructed from the specifications. nvi is guaranteed to return an ϵ-Nash equilibrium with high probability, but maqrm is not guaranteed to do so.

Nash Value Iteration. This baseline first computes an estimate $\tilde{\mathcal{M}}$ of $\mathcal{M}$ using BFS-Estimate (Algorithm 5) and then computes a product of $\tilde{\mathcal{M}}$ with the reward machines corresponding to the agent specifications in order to define rewards at every step. It then solves the resulting general sum game $\tilde{\mathcal{M}}'$ using value iteration. The value iteration procedure is outlined in Algorithm 7 which uses Best-Nash-General-Sum to solve n-player general-sum strategic games (one-step games) at each step. When there are multiple Nash equilibria for a step game, Best-Nash-General-Sum chooses one with the highest social welfare (for that step). In our experiments, we used the library `gambit` [112] for solving the step games.

Multi-agent QRM. The second baseline (Algorithm 8) is a multi-agent variant of QRM [76, 77]. We learn one $\mathcal{Q}$-function for each agent. The $\mathcal{Q}$-function for the i-th agent, denoted $\mathcal{Q}_i : S \times$

Algorithm 8 Multi-agent QRM

Inputs: n-agent Markov game $\mathcal{M} = (S, A = \Pi_{i\in[n]} A_i, s_0, P, H)$, agent specifications $\varphi_1, \dots \varphi_n$, learning rate $\alpha \in (0, 1]$, discount factor $\gamma \in (0, 1]$, $\varepsilon \in (0, 1]$

Outputs: Joint policy $\pi = (\pi_1, \dots, \pi_n)$.

for $i \in [n]$ **do** $(Q_i, \delta_u^i, \delta_r^i, q_0^i) \leftarrow \mathsf{RewardMachine}(\varphi_i)$
// Initialize environment state, reward machines state, and Q-functions
Current state $s \leftarrow s_0$
for $i \in [n]$ **do** $q_i \leftarrow q_0^i$
for $i \in [n]$ **do** Initialize $\mathcal{Q}_i(s, (q_1, \dots, q_n), a_i)$ for all states $s \in S, q_i \in Q_i$, and actions $a_i \in A_i$
for $l \in \{0, \dots, N\}$ **do**
  // Sample actions from policy derived from Q-functions
  for $i \in [n]$ **do** choose action $a_i \in A_i$ at $(s, (q_1, \dots, q_n))$ using exploration policy derived from $\mathcal{Q}_i$ (e.g., ε-greedy)
  // Take a step in environment and the reward machines
  Take action $a = (a_1, \dots a_n)$ in $\mathcal{M}$ and observe the next state s'
  for $i \in [n]$ **do** compute the reward $r_i \leftarrow \delta_r^i(s, q_i)$ and next RM state $q_i' \leftarrow \delta_u^i(s, a, q_i)$
  // Update all Q-functions
  if s' is terminal **then**
    for $i \in [n]$ **do** $\mathcal{Q}_i(s, (q_1, \dots, q_n), a) \xleftarrow{\alpha} r_i$
  else
    for $i \in [n]$ **do** $\mathcal{Q}_i(s, (q_1, \dots, q_n), a_i) \xleftarrow{\alpha} r_i + \gamma \cdot \max_{a_i' \in A_i} \mathcal{Q}_i(s', (q_1', \dots, q_n'), a_i')$
  end if
  if s' is terminal **then**
    // Reset environment state and reward machines state
    $s \leftarrow s_0$ and **for** $i \in [n]$ **do** $q_i \leftarrow q_0^i$
  else
    $s \leftarrow s'$ and **for** $i \in [n]$ **do** $q_i \leftarrow q_i'$
  end if
end for
for $i \in [n]$ **do** $\pi_i \leftarrow$ Best action policy derived from $\mathcal{Q}_i$
return $(\pi_1, \dots, \pi_n)$

$\Pi_{i\in[n]} Q_i \to A_i$, can be used to derive the best action for the i-th agent from the current state of the environment and reward machines of all agents. In every step, $\mathcal{Q}_i$ is used to sample an action a_i for the i-th agent. The joint action $(a_i)_{i\in[n]}$ is used to take a step in the environment and all reward machines. Finally, each $\mathcal{Q}_i$ is individually updated according to the reward gained by the i-th agent. For notational convenience, we let $q \xrightarrow{\alpha} q'$ denote $q \leftarrow (1 - \alpha) \cdot q + \alpha \cdot q'$.

Results. Our results are summarized in Tables 6.1, 6.2 and 6.3. We focus on the Intersection environment (Table 6.2) in the discussion here. For each specification, we ran all algorithms 10 times with a timeout of 24 hours. Along with the average social welfare and $\epsilon_{\min}$, we also report

Spec.	Num. of agents	Algorithm	`welfare`(π) (avg ± std)	$\epsilon_{\min}(\pi)$ (avg ± std)	Num. of runs terminated	Avg. num. of sample steps (in millions)
φ^1	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.02
		NVI	1.00 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	1.00 ± 0.00	0.01 ± 0.00	10	4.00
φ^2	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.01
		NVI	1.00 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	0.66 ± 0.00	0.99 ± 0.00	10	4.00
φ^3	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.50
		NVI	1.00 ± 0.00	0.00 ± 0.00	8	5.00
		MAQRM	0.81 ± 0.01	0.56 ± 0.03	10	4.00
φ^4	3	HighNashSearch	0.67 ± 0.00	0.00 ± 0.00	10	3.03
		NVI	0.33 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	0.63 ± 0.00	0.57 ± 0.01	10	4.00
φ^5	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.60
		NVI	0.33 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	0.62 ± 0.01	0.47 ± 0.02	10	4.00
φ^6	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.68
		NVI	0.00 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	0.44 ± 0.01	0.55 ± 0.02	10	4.00

Table 6.1: Results for all specifications in Single Lane Environment. Total of 10 runs per benchmark. Timeout = 24 hrs.

the average number of sample steps taken in the environment as well as the number of runs that terminated before timeout. For a fair comparison, all approaches were given a similar number of samples from the environment.

Nash equilibrium. Our approach learns policies that have low values of $\epsilon_{\min}$, indicating that it can be used to learn ϵ-Nash equilibria for small values of ϵ. NVI also has similar values of ϵ, which is expected since NVI provides guarantees similar to our approach w.r.t. Nash equilibria computation. On the other hand, MAQRM learns policies with large values of $\epsilon_{\min}$, implying that it fails to converge to a Nash equilibrium in most cases.

Social Welfare. Our experiments show that our approach consistently learns policies with high social welfare compared to the baselines. For instance, φ^3 corresponds to the specifications in the motivating example for which our approach learns a joint policy that causes both blue and black cars to achieve their goals. Although NVI succeeds in learning policies with high social welfare for some specifications (φ^1, φ^3, φ^4), it fails to do so for others (φ^2, φ^5). Experiments in the Single Lane and Gridworld environments indicate that NVI can achieve high social welfare (similar to our approach)

Spec.	Num. of agents	Algorithm	`welfare`(π) (avg ± std)	$\epsilon_{\min}(\pi)$ (avg ± std)	Num. of terminated runs	Avg. num. of sample steps (in millions)
φ^1	3	HighNashSearch	**0.33 ± 0.00**	**0.00 ± 0.00**	10	1.78
		nvi	0.32 ± 0.00	**0.00 ± 0.00**	10	1.92
		maqrm	0.18 ± 0.01	0.51 ± 0.01	10	2.00
φ^2	4	HighNashSearch	**0.55 ± 0.10**	**0.01 ± 0.02**	10	11.53
		nvi	0.04 ± 0.01	0.02 ± 0.01	10	12.60
		maqrm	0.12 ± 0.01	0.20 ± 0.03	10	15.00
φ^3	4	HighNashSearch	**0.49 ± 0.01**	**0.00 ± 0.01**	10	11.26
		nvi	0.45 ± 0.01	**0.00 ± 0.01**	10	12.60
		maqrm	0.11 ± 0.01	0.22 ± 0.02	10	15.00
φ^4	3	HighNashSearch	0.90 ± 0.15	**0.00 ± 0.00**	10	2.16
		nvi	**0.98 ± 0.00**	**0.00 ± 0.00**	4	2.18
		maqrm	0.23 ± 0.01	0.39 ± 0.04	10	2.00
φ^5	5	HighNashSearch	**0.58 ± 0.02**	**0.00 ± 0.00**	10	62.17
		nvi	0.05 ± 0.01	0.01 ± 0.01	7	80.64
		maqrm	Timeout	Timeout	0	Timeout

Table 6.2: Results for all specifications in Intersection Environment. Total of 10 runs per benchmark. Timeout = 24 hrs.

for specifications in which all agents can successfully achieve their goals (cooperative scenarios). However, in many other scenarios in which only some of the agents can fulfill their objectives, our approach achieves higher social welfare.

6.8. Discussion

We have proposed a framework for maximizing social welfare under the constraint that the joint policy should form an ϵ-Nash equilibrium. Our approach involves learning and enumerating a small set of finite-state deterministic policies in decreasing order of social welfare and then using a self-play RL algorithm to check if they can be extended with punishment strategies to form an ϵ-Nash equilibrium. Our experiments demonstrate that our approach is effective in learning Nash equilibria with high social welfare.

One limitation of our approach is that our algorithm does not have any guarantee regarding optimality with respect to social welfare. The policies considered by our algorithm are chosen heuristically based on the specifications, which may lead to scenarios where we miss high welfare solutions. For example, φ^2 in the Intersection environment corresponds to specifications in the motivating example except that the blue car is not required to stay a car length ahead of the other two cars. In this

Spec.	Num. of agents	Algorithm	`welfare`(π) (avg ± std)	$\epsilon_{\min}(\pi)$ (avg ± std)	Num. of runs terminated	Avg. num. of sample steps (in millions)
φ^1	2	HighNashSearch	0.95 ± 0.02	0.00 ± 0.00	10	18.86
		nvi	1.00 ± 0.00	0.00 ± 0.00	10	22.40
		maqrm	Timeout	Timeout	10	Timeout
φ^2	2	HighNashSearch	0.99 ± 0.01	0.00 ± 0.00	10	29.74
		nvi	1.00 ± 0.00	0.00 ± 0.00	8	38.40
		maqrm	Timeout	Timeout	10	Timeout
φ^3	2	HighNashSearch	0.98 ± 0.02	0.00 ± 0.00	10	48.40
		nvi	1.00 ± 0.00	0.00 ± 0.00	4	57.60
		maqrm	Timeout	Timeout	10	Timeout
φ^4	2	HighNashSearch	0.94 ± 0.02	0.00 ± 0.00	10	65.91
		nvi	0.94 ± 0.01	0.00 ± 0.00	7	92.80
		maqrm	Timeout	Timeout	10	Timeout
φ^5	2	HighNashSearch	0.86 ± 0.05	0.00 ± 0.00	10	87.05
		nvi	0.13 ± 0.01	0.00 ± 0.00	10	128.00
		maqrm	Timeout	Timeout	10	Timeout

Table 6.3: Results for all specifications in Gridworld Environment. Total of 10 runs per benchmark. Timeout = 24 hrs.

scenario, it is possible for three cars to achieve their goals in an equilibrium solution if the blue car helps the cars behind by staying in the middle of the intersection until they catch up. Such a joint policy is not among the set of policies considered; therefore, our approach learns a solution in which only two cars achieve their goals. We believe that such limitations can be overcome in future work by modifying the various components within our enumerate-and-verify framework.

6.9. Related Work

Multi-agent RL. There has been work on learning Nash equilibria in the multi-agent RL setting [72, 73, 107, 126, 129, 6]; however, these approaches focus on learning an arbitrary equilibrium and do not optimize social welfare. There has also been work on studying weaker notions of equilibria in this context [161, 58], as well as work on learning Nash equilibria in two agent zero-sum games [20, 153, 106].

Equilibrium in Markov games. There has been work on computing Nash equilibrium in Markov games [93, 134], including work on computing ϵ-Nash equilibria from logical specifications [34, 33], as well as recent work focusing on computing welfare-optimizing Nash equilibria from temporal specifications [97, 98]; however, all these works focus on the planning setting where the transition

probabilities are known. Checking for existence of Nash equilibrium, even in deterministic games, has been shown to be NP-complete for reachability objectives [27].

Social welfare. There has been work on computing welfare maximizing Nash equilibria for bimatrix games, which are two-player one-step Markov games with known transitions [37, 68]; in contrast, we study this problem in the context of general Markov games.

CHAPTER 7

Compositional Verification of Neural Network Controllers

The previous chapters focused on designing and analysing reinforcement learning algorithms for synthesizing policies from complex temporal specifications. We primarily studied the single-task setting in which the training objective encodes only one task. However, there are many scenarios in which we want to train a single policy to perform multiple tasks—e.g., we might want to train a controller for an autonomous racing car that works in all race tracks. Furthermore, in realistic settings with continuous state spaces, the trained policy is either a single neural network (NN) or contains many neural network components. In such cases, we often do not have strong guarantees regarding the safety of the learned controllers—e.g., there is no evidence for why an NN policy trained using reinforcement learning for autonomous driving will never cause a crash.

In this chapter, we present a compositional approach to multi-task learning and utilize our framework to also design an algorithm to verify the safety of the trained policy *across all tasks*. We focus on closed-loop safety, where the goal is to ensure that the controller, composed with a model of the robot dynamics and its environment, is safe over the entire planning horizon—e.g., that an autonomous car does not run into an obstacle, or a walking robot does not fall over.

7.1. Overview

A key challenge is proving safety for the full closed-loop system. One approach is to unroll the safety property over a finite horizon [79]. However, this approach becomes intractable as the planning horizon becomes large. In particular, existing verification algorithms rely on overapproximating the dynamics [36], and the approximation error accumulates over the horizon. Thus, very precise abstractions are required to verify safety for long horizons. An alternative approach is to establish the existence of an inductive invariant such as a Lyapunov function [38, 143] or a control barrier function [128, 10]. This strategy reduces the problem to a verification problem over a single step, since it suffices to prove that a candidate invariant is inductive and that it implies safety. However, establishing such an invariant can be intractable for high-dimensional state spaces, especially when

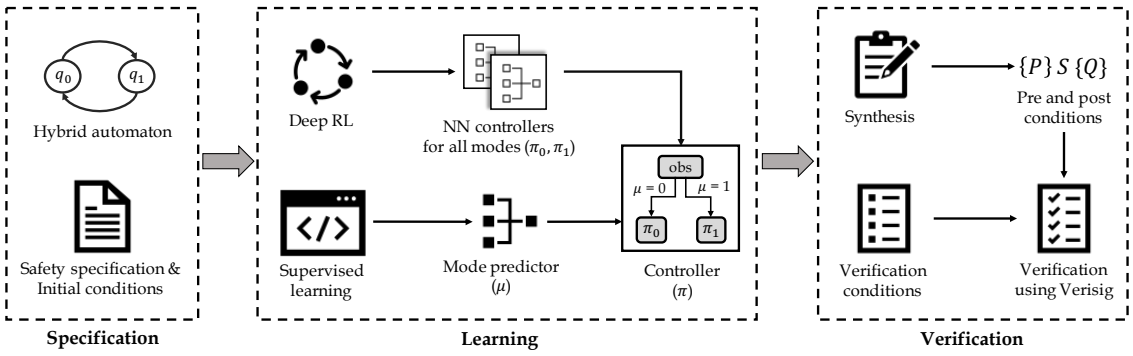


Figure 7.1: An overview of our compositional learning and verification framework.

using neural network controllers with many parameters.

These challenges are further exacerbated for real-world robotics systems, which are typically only partially observable (e.g., the inputs to the NN are LiDAR scans), and the geometry of the environment is a priori unknown (e.g., the robot is acting in a building with an unknown layout of hallways).

To address the above challenges, we propose a framework for compositional learning and verification of NN controllers[17] (Figure 7.1). Our framework is inspired by classical techniques such as Hoare logic [70] for compositional program verification. The idea is to verify a program by decomposing it into modular components, devising verification conditions (VCs) for all components that suffice to prove safety, and then proving that each VC holds for its respective component.

In particular, our framework leverages this strategy to both learn an NN controller to solve a given control task and verify the learned controller. Following the theme of compositionality in the previous chapters, we first decompose the task into a sequence of sub-tasks, where each sub-task is associated with a precondition (e.g., the region of the state space where the robot starts) and a postcondition (e.g., the region where the robot ends up). This decomposition is designed to satisfy two properties:

[17] Although the proposed framework can be used with any verification tool, we use Verisig [79] for closed-loop verification. Since Verisig supports fully-connected NNs with sigmoid/tanh activations, we focus on this class of NNs as well.

- **Mode safety and progress:** For any single sub-task, using the NN controller from any state satisfying the precondition should safely transition the system to a state satisfying the postcondition within some bounded number of steps.

- **Switching safety:** The postcondition of one sub-task should imply the precondition of the next.

As long as these two properties are satisfied, the NN controller is guaranteed to be safe for the entire planning horizon. Furthermore, these two properties are sufficient to guarantee a particular liveness property which states that any finite sequence of sub-tasks will be completed eventually. Intuitively, our strategy combines verification over a finite horizon (i.e., mode safety) with establishing inductive invariants (i.e., switching safety), except that the inductive invariants are established at the level of sub-tasks rather than individual steps in the system. Formally, we model the system as a hybrid automaton [11, 120, 10]—i.e., a model of the system is a set of modes of operation, with differential equations specifying the state dynamics of each mode; in our approach, the discrete transitions encode switching from one sub-task to the next. Many practical control tasks can be decomposed in such a way—e.g., navigation problems can be decomposed into sequences of sub-goals.

Given a hybrid automaton, our framework performs the following steps:

- **Compositional learning:** First, it learns a separate NN controller for each mode, using shaped rewards to encourage it to satisfy mode safety and progress. An added advantage of this approach is that we can use simpler NNs that are easier to both train and verify.

- **Pre/postcondition synthesis:** Next, it synthesizes *candidate pre/postconditions* (i.e., a candidate pair of pre- and postconditions for each mode) that satisfy switching safety and are consistent with a set of traces obtained by simulating the system with the learned controllers.

- **Compositional verification:** Finally, it uses hybrid systems verification tools [36, 79] to independently check mode safety and progress for each mode.

The second step builds on recent work on invariant synthesis [54]. In particular, our synthesis algo-

rithm uses testing to identify *implication examples* that connect the different (pre/postcondition) sets, and then tries to synthesize candidate pre/postconditions consistent with these examples.

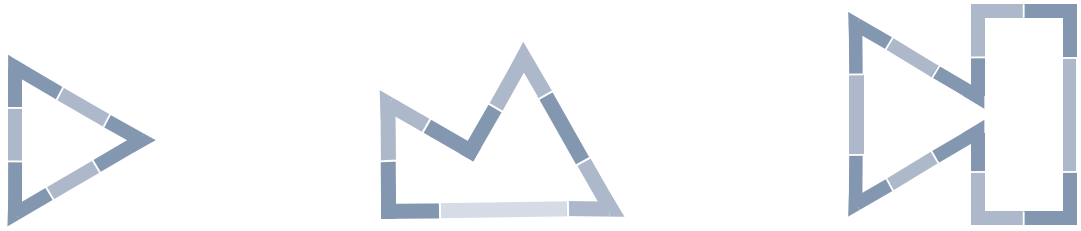
One challenge is that in partially observed environments, the controller may not know when one sub-task has been completed and/or what the next sub-task is. To address this issue, we additionally train a *mode predictor*, which is a separate NN that predicts whether the postcondition for the current sub-task holds in the current state and if so, predicts the next sub-task. This mode predictor is incorporated into the overall controller. To ensure correctness, the safety and progress conditions are verified with respect to the full compositional controller (including the mode predictor). For instance, consider a robot navigating in a building with an unknown layout; then, it may not know if the next segment is to go straight, turn left, or turn right. Our approach naturally handles this setting since it proves safety for arbitrary compositions of the sub-tasks as long as the switching safety property is satisfied. Thus, the sequence of sub-tasks can be chosen dynamically based on observations of the environment—e.g., if a robot comes to a left turn at the end of a hallway, then the mode predictor would determine that the next sub-task is to make that left turn. Therefore, our framework enables us to learn and verify a controller that generalizes to multiple tasks composed of the same set of sub-tasks.

We evaluate our approach on a challenging benchmark—namely, a simulation model of the F1/10 autonomous racing system [121], where the goal is for an NN-controlled car to complete a track without crashing into the walls. Verifying safety for this system has received recent attention [79]; however, these approaches do not scale to verifying safety beyond short time horizons on a single, predefined track, due to two main reasons. First, the controller must rely on high-dimensional LiDAR observations of the environment, which poses challenges for scalability. Second, we ideally want to ensure safety for a wide variety of complex track geometries. As a consequence, this system is beyond the reach of existing state-of-the-art verification techniques.

We demonstrate that our framework can successfully learn and verify an NN controller for this system, by decomposing tracks into sequences of individual segments. In particular, we consider sub-tasks that include going straight or executing four different kinds of turns, and verify safety for

(a) Straight (b) Right (c) Left (d) Sharp Right (e) Sharp Left

Figure 7.2: Different types of track segments.



(a) (b) (c)

Figure 7.3: Example tracks decomposed into segments.

any sequence of such sub-tasks. We also provide evidence that training a monolithic controller for an example track is significantly harder than our compositional learning approach.

In summary, our main contributions are:

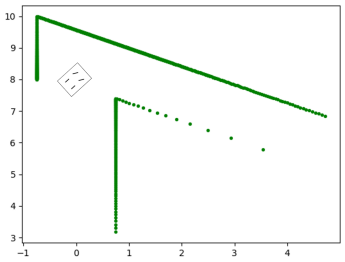
- A framework for compositional verification of NN controllers for hybrid systems (Section 7.2).
- An algorithm for automatically inferring pre/postconditions given a controller π, as well as a compositional learning algorithm for training π.
- An extensive evaluation[18] via a case study based on a model of the F1/10 autonomous car (Sections 7.4 & 7.5).

7.1.1. Motivating Example

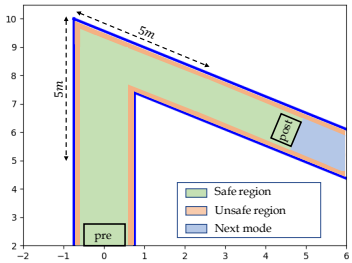
We now give a brief overview of our approach using the F1/10 autonomous racing system as a motivating example.

F1/10 car. The objective is to safely navigate the autonomous F1/10 car along a racing track to complete a lap as quickly as possible. The safety property states that the car should not crash into the track walls. Ignoring modes for now, the state space is $\mathcal{X} \subseteq \mathbb{R}^4$ (a state $x \in \mathcal{X}$ denotes the

[18] Our implementation is available at https://github.com/keyshor/autonomous_car_verification.



(a) A LiDAR scan.



(b) Different regions in the segment.

Figure 7.4: A sharp right turn.

2D position, speed, and angle of the car), the action space is $\mathcal{U} \subseteq \mathbb{R}^2$ (an action $u \in \mathcal{U}$ consists of acceleration and steering angle), and the dynamics are the bicycle dynamics [131].

We assume the track is decomposed into a sequence of segments, where each segment is either a straight track, a left/right turn, or a sharp left/right turn as shown in Figure 7.2; these are the five *modes* of the system. Our goal is not to learn and verify a controller for a given specific track, but rather to learn and verify a controller that works for *all* tracks constructed by composing these segments. Some example tracks are shown in Figure 7.3.

The F1/10 car observes its environment with a LiDAR sensor, which uses laser rays to determine the distance to the nearest obstacle along different directions. In particular, it produces an observation $o \in \mathcal{O} \subseteq \mathbb{R}^m$, where each $o_i \in \mathbb{R}$ corresponds to an angle $\psi \in [-135, 135]$ and denotes the distance from the car position to the nearest wall in the direction $\vartheta + \psi$, where ϑ is the angle the car is currently facing. An example of a scan is shown in Figure 7.4a; each green point is the obstacle observed by one of the $m = 1081$ LiDAR rays.

Control problem. Our goal is to learn a controller $\pi : \mathcal{O} \to \mathcal{U}$ that maps LiDAR observations to actions. Designing a safe controller for the F1/10 car is challenging due to the high-dimensional observation space. One approach is to train a neural network (NN) controller π using reinforcement learning, and then verify post-hoc that π is safe. This technique has been used to verify that the car can safely navigate a right turn [80]. However, existing verification approaches [42, 79, 144] do not scale to more complex tasks such as the tracks in Figure 7.3—even when the track is known

ahead of time—due to the long planning horizon.

Compositional verification (fully observed). For now, let us assume that the controller π is given and that the state is fully observed, and describe how we verify that π is safe. We also assume that we are given a *pre-region* and a *post-region* for each mode, which are subsets of the state space such that the car always starts in the pre-region of the mode and ends in its post-region. Intuitively, membership in the pre-region (resp., post-region) corresponds to the precondition (resp., postcondition) for that mode. An example of the pre- and post-regions for the sharp right turn mode is shown in Figure 7.4b. These regions are chosen so that the system immediately and safely transitions from the post-region of any mode q to the pre-region of some subsequent mode q' (i.e., *switching safety*). If we know the sequence of track segments, then the choice of q' is unique. In our case, since we do not know the sequence of track segments a priori, we prove switching safety for *every* pair of modes q, q', which, together with mode safety and progress guarantees that the car safely completes *any* track consisting of an arbitrary sequence of these five kinds of segments. Finally, to prove mode safety and progress, it suffices to verify that π safely navigates the car from the pre-region of each mode to the corresponding post-region without crashing.

Compositional verification (partially observed). Verification is more challenging when the state is partially observed—e.g., π only has access to LiDAR observations. We assume π is decomposed into a *mode predictor* μ together with a controller π^q for each mode q. Then, π uses π^q, where q is the predicted mode at the current step.

Importantly, we do not assume that the mode predictor is correct; thus, π may use the incorrect controller. For example, in the case of the sharp right turn, if the LiDAR range is smaller than the distance of the corner from the entry region, there will be regions where the mode predictor cannot distinguish the sharp turn segment from a straight segment using just LiDAR observations (see Figure 7.4b). Thus, we need to prove that the full controller π is correct, even if μ is wrong. This involves simultaneously reasoning about the controllers $\pi^{q''}$ for *all* modes q'', along with the mode predictor μ.

Compositional learning. We use deep reinforcement learning to train one neural network controller π^q for each mode q to drive the car from the pre-region to the post-region. Since the controller can only observe the LiDAR observations, we also train a mode predictor that predicts the current mode from the observations. We can do so using supervised learning from observations encountered while training π^q.

Importantly, we find that our compositional approach benefits not only verification but also learning. In particular, we can train simpler neural networks with fewer parameters, and training is less likely to get stuck at local maxima that are characteristic of long planning horizons.

Candidate pre/post-region synthesis. Finally, manually specifying the pre- and post-regions for each mode can be challenging. We propose an algorithm for automatically inferring these regions. Our algorithm, based on invariant inference [48, 135, 54], alternates between synthesizing *candidate pre/post-regions* that are consistent with all the example traces generated so far, and generating new example traces using π.

In particular, the synthesis algorithm uses the example traces to identify both *unsafe examples* z from which π is known to be unsafe, and *implication examples* $z \to z'$, which say that z' is reachable from z using π. Then, it represents the pre- and post-regions as boxes in $\mathbb{R}^n$, and infers a set of boxes that are consistent with the identified examples. Finally, it uses the inferred pre/post-regions to try and verify that π is safe.

7.2. Compositional Verification

In this section, we describe our framework for compositional verification of controllers. Our model of the system is based on hybrid automata [120, 12, 11] tailored to our setting. We define safety and liveness in our context and show that we can reduce safety and liveness to a set of verification conditions (VCs) that are local to the modes of the hybrid automaton and can be checked using existing verification tools.

7.2.1. Problem Formulation

Dynamics. We consider a hybrid dynamical system[19] with states $z \in \mathcal{Z}$ and actions $\mathcal{U} \subseteq \mathbb{R}^k$. We assume the state space has structure $\mathcal{Z} = \mathcal{Q} \times \mathcal{X}$, where $\mathcal{Q}$ is a finite set of *modes* and $\mathcal{X} \subseteq \mathbb{R}^n$ is the continuous component of the state space. We denote the states in mode q by $\mathcal{Z}^q = \{q\} \times \mathcal{X}$. Within a mode $q \in \mathcal{Q}$, the dynamics are given by a function $f : \mathcal{Z} \times \mathcal{U} \to \mathbb{R}^n$; in particular, the system evolves according to the differential equation $\dot{x}(t) = f(z(t), u(t))$ (with respect to time t). When there is no ambiguity, we simply write $\dot{x} = f(z, u)$. The mode transitions are given by a relation $\mathcal{T} \subseteq \mathcal{Z} \times \mathcal{Z}$, where an edge $z \to z' \in \mathcal{T}$ means the system can transition from state z to state z'. We let $\mathcal{Z}_F = \{z \in \mathcal{Z} \mid \exists z' \in \mathcal{Z} \text{ s.t. } z \to z' \in \mathcal{T}\}$ denote the set of states where mode transitions can occur. The mode transitions are assumed to be *urgent*—i.e., a mode transition occurs as soon as the system reaches some $z \in \mathcal{Z}_F$; we assume that $\mathcal{Z}_F$ is closed so this property is well-defined.

Intuitively, the corresponding discrete time dynamics are given by $z_+ = z'$ if $z \to z' \in \mathcal{T}$ and $z_+ = (q, x + f(z, u) \cdot \Delta t)$ otherwise. Note that the mode transitions are nondeterministic, since the condition $z \to z' \in \mathcal{T}$ may be satisfied by multiple $z' \in \mathcal{Z}$. This nondeterminism is needed to capture settings where the sequence of sub-tasks is a priori unknown. In our F1/10 example, at a state z about to exit the current mode, transitions $z \to (q', x')$ exist for all modes $q' \in \{\text{straight}, \text{left turn}, ...\}$. Finally, our goal is to control the system based on observations $o \in \mathcal{O} \subseteq \mathbb{R}^m$; in particular, an observation function $h : \mathcal{Z} \to \mathcal{O}$ maps states to observations. If the system is fully observable, $\mathcal{O}$ can be taken to be $\mathcal{Z}$ with $h(z) = z$ for all $z \in \mathcal{Z}$.

We formally represent the dynamical system as a hybrid automaton which is defined as:

Definition 7.1. *A* hybrid automaton $\mathcal{A}$ *is a tuple* $\mathcal{A} = (\mathcal{Q}, \mathcal{X}, \mathcal{U}, \mathcal{T}, \mathcal{O}, f, h)$.

Control. A controller is a function $\pi : \mathcal{O} \to \mathcal{U}$, where $u = \pi(h(z))$ specifies the action to use in state z. We use $f(z, \pi, t) \in \mathcal{Z}$ to denote the state reached at time $t \in \mathbb{R}_{\geq 0}$ by evolving the system according to $\dot{x} = f(z, \pi(h(z)))$. Furthermore, let $F(z, \pi, t) \subseteq \mathcal{Z}$ denote the set of states visited until time t—i.e., $F(z, \pi, t) = \{f(z, \pi, t') \mid 0 \leq t' \leq t\}$.

[19]The environment is no longer an MDP as we study the verification problem in the continuous-time setting.

We decompose π into controllers $\pi^q : \mathcal{O} \to \mathcal{U}$ designed to be used in mode $q \in \mathcal{Q}$, and a *mode predictor* $\mu : \mathcal{O} \to \mathcal{Q}$ that predicts the current mode. Then, we have $\pi(o) = \pi^q(o)$ where $q = \mu(o)$. We do not assume that the mode predictor is always correct—i.e., we may have $\mu(o) = q$ even though the current mode is $q' \neq q$, in which case π would use the wrong controller.

Trajectories. Next, we describe the space of trajectories that may be generated by a given controller π. Since the dynamics are continuous-time, the trajectory is a curve in the state space parameterized by time $t \in \mathbb{R}_{\geq 0}$. However, formally reasoning about this representation is difficult. Instead, we represent a trajectory as an infinite sequence $\rho = (z_0 \xrightarrow{t_0} z_1 \xrightarrow{t_1} \cdots)$, where $t_i \in \mathbb{R}_{\geq 0}$ for all $i \in \mathbb{N}$. In particular, an edge $z_i \xrightarrow{t_i} z_{i+1}$ in ρ says that the system transitions from z_i to z_{i+1} in time t_i. For clarity, we omit the t_i's from ρ when it is not needed. There are two kinds of transitions $z_i \to z_{i+1}$ that can occur:

- **Continuous transition:** This kind of transition occurs when $z_i \notin \mathcal{Z}_F$. Then, the system evolves according to the continuous dynamics f—i.e., $z_{i+1} = f(z_i, \pi, t_i)$, where $t_i > 0$. We assume that no mode transition is triggered—i.e., $f(z_i, \pi, t) \notin \mathcal{Z}_F$ for all $t \in [0, t_i)$. We denote such a transition by $z_i \to_f z_{i+1}$.

- **Mode transition:** This kind of transition occurs when $z_i \in \mathcal{Z}_F$. Then, the system instantaneously transitions to some z_{i+1} such that $z_i \to z_{i+1} \in \mathcal{T}$—i.e., $t_i = 0$. We denote such a transition by $z_i \to_{\mathcal{T}} z_{i+1}$.

We assume all trajectories are *non-Zeno*—i.e., $\sum_{i=0}^{\infty} t_i = \infty$. It is only necessary to consider Zeno trajectories if subsequent mode transitions can occur after arbitrarily small amounts of time, which cannot happen if the system requires a minimum amount of time before triggering the next mode transition. In our F1/10 example, the car must traverse an entire segment to trigger another mode transition, which cannot happen arbitrarily quickly since velocity is bounded from above.

Correctness properties. We consider a safety property specified as a region $\mathcal{Z}_{\text{safe}} \subseteq \mathcal{Z}$ in which we expect the system to stay. In addition, we assume given a set of initial states $\mathcal{Z}_0 \subseteq \mathcal{Z}_{\text{safe}}$ from which we want to ensure safety.

Definition 7.2. *A controller π is* safe *for a hybrid automaton $\mathcal{A}$ if for any trajectory ρ starting from $z_0 \in \mathcal{Z}_0$, for all $i \in \mathbb{N}$, we have $f(z_i, \pi, t) \in \mathcal{Z}_{safe}$ for all $t \in [0, t_i]$.*

That is, the system should be safe for the duration of any trajectory generated using π from an initial state. Next, liveness says the system should switch modes infinitely often.

Definition 7.3. *A controller π is* live *for a hybrid automaton $\mathcal{A}$ if for any trajectory ρ starting from $z_0 \in \mathcal{Z}_0$, we have $z_i \to_{\mathcal{T}} z_{i+1}$ for infinitely many $i \in \mathbb{N}$.*

7.2.2. Verification Conditions

Our verification algorithm reduces the problem of verifying safety and liveness to a set of verification conditions (VCs).

Pre- and post-regions. Following our compositional approach, our VCs decompose the problem into properties of individual modes or pairs of modes. For each mode q, we assume given a *pre-region* $\mathcal{X}^q_{\text{pre}} \subseteq \mathcal{X}$ and a *post-region* $\mathcal{X}^q_{\text{post}} \subseteq \mathcal{X}$. In addition, we define $\mathcal{Z}^q_{\text{pre}} = \{q\} \times \mathcal{X}^q_{\text{pre}}$ and $\mathcal{Z}^q_{\text{post}} = \{q\} \times \mathcal{X}^q_{\text{post}}$. Intuitively, the precondition (resp., postcondition) for q is membership in its pre-region (resp., post-region). We require that pre- and post-regions satisfy the following conditions, which we call *compatibility conditions (CCs)* since they are not checked by the verifier, but are directly enforced when we generate the pre/post regions.

Definition 7.4 (CC 1)**.** We have $\mathcal{Z}_0 \subseteq \bigcup_{q \in \mathcal{Q}} \mathcal{Z}^q_{\text{pre}}$.

That is, every initial state is contained in a pre-region.

Definition 7.5 (CC 2)**.** We have $\bigcup_{q \in \mathcal{Q}} \mathcal{Z}^q_{\text{post}} \subseteq \mathcal{Z}_F$.

That is, every state in the post-region triggers a mode transition; intuitively, the post-region should only include states that "exit" the mode. Now, we have two kinds of VCs:

- **Mode safety and progress:** For each mode $q \in \mathcal{Q}$, the system safely transitions from $\mathcal{Z}^q_{\text{pre}}$ to $\mathcal{Z}^q_{\text{post}}$.
- **Switching safety:** For each pair of modes $q, q' \in \mathcal{Q}$ with a mode transition $(q, x) \to (q', x') \in$

$\mathcal{T}$, the system safely transitions from $\mathcal{Z}^q_{\text{post}}$ to $\mathcal{Z}^{q'}_{\text{pre}}$.

First, our VC for mode safety and progress is:

Definition 7.6 (VC 1)**.** *For any $z \in \mathcal{Z}^q_{pre}$, there exists $t \in \mathbb{R}_{>0}$ such that $f(z, \pi, t) \in \mathcal{Z}^q_{post}$, $F(z, \pi, t) \subseteq \mathcal{Z}_{safe}$, and $f(z, \pi, t') \notin \mathcal{Z}_F$ for all $t' \in [0, t)$.*

That is, π safely transitions the system from any state in the pre-region of mode q to the post-region of q. The last condition is needed to ensure that the system does not trigger a mode transition $z \to z' \in \mathcal{T}$ at some state $z \notin \mathcal{Z}^q_{\text{post}}$. That is, $f(z, \pi, t)$ is the first state reached that triggers a mode transition (such a state exists since we have assumed $\mathcal{Z}_F$ is closed).

Remark 7.7. *Although VC 1 is local to a mode $q \in \mathcal{Q}$, it is a property of the full controller π which includes the mode predictor μ and controllers $\pi^{q'}$ for all $q' \in \mathcal{Q}$.*

Next, our VC for switching safety is:

Definition 7.8 (VC 2)**.** *For all $z \in \mathcal{Z}^q_{post}$ and all $z \to z' \in \mathcal{T}$, we have $z' \in \mathcal{Z}^{q'}_{pre}$ for some $q' \in \mathcal{Q}$.*

That is, for every state z in a post-region and every mode transition $z \to z'$, the target state z' is contained in the pre-region of another mode q'.

Together, CCs 1 & 2 and VCs 1 & 2 imply that π is safe and live for $\mathcal{A}$. First, CC1 ensures that the initial states satisfy the precondition of some mode q. Then, VC 1 says that the precondition of mode q implies the postcondition of mode q. Next, VC 2 and CC 2 together say that the postcondition of mode q implies the precondition of another mode q'.

Theorem 7.9. *Given controller π for hybrid automaton $\mathcal{A}$, if CCs 1 & 2 and VCs 1 & 2 hold, then π is safe and live for $\mathcal{A}$.*

Proof. Let π be a compositional controller such that VCs 1 and 2 hold. Let ρ be a non-Zeno

trajectory of the automaton generated by π,

$$\rho = (z_0 \xrightarrow{t_0} z_1 \xrightarrow{t_1} \cdots).$$

Let us the denote the cumulative times using $T_i = \sum_{j=0}^{i-1} t_i$. We define $\mathcal{Z}_{\text{pre}} = \cup_{q \in \mathcal{Q}} \mathcal{Z}^q_{\text{pre}}$ and $\mathcal{Z}_{\text{post}} = \cup_{q \in \mathcal{Q}} \mathcal{Z}^q_{\text{post}}$. Note that VC 1 implies $\mathcal{Z}_{\text{pre}} \subseteq \mathcal{Z}_{\texttt{safe}}$ and WLOG we can also take $\mathcal{Z}_{\text{post}} = \mathcal{Z}_{\text{post}} \cap \mathcal{Z}_{\texttt{safe}} \subseteq \mathcal{Z}_{\texttt{safe}}$. Let $I_{\mathcal{T}}$ denote the set of indices at which a mode transition occurs from a state in $\mathcal{Z}_{\text{post}}$—i.e., $I_{\mathcal{T}} = \{i \mid z_i \to_{\mathcal{T}} z_{i+1} \text{ and } z_i \in \mathcal{Z}_{\text{post}}\}$. We first show that $I_{\mathcal{T}}$ is infinite, thereby proving liveness.

Lemma 7.10. *$I_{\mathcal{T}}$ contains infinitely many indices.*

Proof. We give a proof by contradiction. Suppose $I_{\mathcal{T}}$ is finite. If $I_{\mathcal{T}} = \emptyset$, let $z = z_0 \in \mathcal{Z}_0 \subseteq \mathcal{Z}_{\text{pre}}$. If $I_{\mathcal{T}}$ is nonempty, let $i_{\max}$ be the largest index in $I_{\mathcal{T}}$ and $z = z_{i_{\max}+1}$. We have $z_{i_{\max}} \to_{\mathcal{T}} z_{i_{\max}+1}$ and from VC 2 we get that $z = z_{i_{\max}+1} \in \mathcal{Z}_{\text{pre}}$.

In either case, we have $z = z_i \in \mathcal{Z}^q_{\text{pre}}$ for some $q \in \mathcal{Q}$ and $i \geq 0$. From VC 1 we get that there is a $t \in \mathbb{R}_{\geq 0}$ such that $f(z, \pi, t) \in \mathcal{Z}^q_{\text{post}}$ and for all $t' \in [0, t)$, $f(z, \pi, t') \notin \mathcal{Z}_F$. Since the run ρ is non-Zeno, there is an index $j \geq i$ such that $T_j - T_i \leq t \leq T_{j+1} - T_i$. Since $f(z_i, \pi, t') \notin \mathcal{Z}_F$ if $t' < t$, we get that $z_k \to_f z_{k+1}$ for all k with $i \leq k \leq j-1$. We now have two cases to consider.

- Case 1: $t = T_j - T_i$. In this case, $z_j = f(z_i, \pi, t) \in \mathcal{Z}^q_{\text{post}}$. Since $z_j \in \mathcal{Z}_F$ we must have $z_j \to_{\mathcal{T}} z_{j+1}$ in ρ and hence $j \in I_{\mathcal{T}}$.

- Case 2: $t > T_j - T_i$. In this case, $z_j = f(z_i, \pi, T_j - T_i) \notin \mathcal{Z}_F$. Hence we must have $z_j \to_f z_{j+1}$ in ρ. From the definition of $\to_f$, it follows that $f(z_i, \pi, t') \notin \mathcal{Z}_F$ for all $t' < T_{j+1} - T_i$. Therefore $t = T_{j+1} - T_i$ and $z_{j+1} = f(z_i, \pi, t) \in \mathcal{Z}^q_{\text{post}}$ and we can conclude that $j + 1 \in I_{\mathcal{T}}$

Therefore, in either case, we reach a contradiction as we showed that there is a $k \in I_{\mathcal{T}}$ with $k \geq i = i_{\max} + 1 > i_{\max}$. □

We now prove safety.

Lemma 7.11. *For all $i \geq 0$, $f(z_i, \pi, t) \in \mathcal{Z}_{\texttt{safe}}$ for all $t \in [0, t_i]$.*

Proof. Let $i_1 < i_2 < \cdots$ be the ordered sequence of indices in $I_{\mathcal{T}}$. Let $i_0 = -1$. We show that for all $k \geq 0$ and any $i \geq 0$ with $i_k \leq i < i_{k+1}$, $f(z_i, \pi, t') \in \mathcal{Z}_{\texttt{safe}}$ for all $t' \in [0, t_i]$.

Let $z = z_{i_k+1} \in \mathcal{Z}_{\text{pre}}$ (VC 2). Then by VC 1, there is a $t \in \mathbb{R}_{\geq 0}$ such that $f(z, \pi, t) \in \mathcal{Z}_{\text{post}}$, $F(z, \pi, t) \subseteq \mathcal{Z}_{\texttt{safe}}$ and for all $t' \in [0, t)$, $f(z, \pi, t') \notin \mathcal{Z}_F$. Let $j \geq i_k + 1$ be the smallest index after i_k such that $z_j \in \mathcal{Z}_F$. Such a j exists since $z_{i_{k+1}} \in \mathcal{Z}_F$. Let $T = T_j - T_{i_k+1}$. Then we have $z_i \to_f z_{i+1}$ for all i with $i_k + 1 \leq i < j$. Hence $z_j = f(z, \pi, T)$ and for all $t' < T$, $f(z, \pi, t') \notin \mathcal{Z}_F$. From this we can conclude that $t = T$ and $z_j = f(z, \pi, t) \in \mathcal{Z}_{\text{post}}$. Therefore, j is also the smallest index after i_k such that $j \in I_{\mathcal{T}}$, so $j = i_{k+1}$. Now for any i with $i_k + 1 \leq i < i_{k+1}$, we have $f(z_i, \pi, t') = f(z, \pi, t' + T_i - T_{i_k+1}) \in \mathcal{Z}_{\texttt{safe}}$ for all $t' \in [0, t_i]$ since $t' + T_i - T_{i_k+1} \leq t$. If $k > 0$, we have $z_{i_k} \in \mathcal{Z}_{\text{post}} \subseteq \mathcal{Z}_{\texttt{safe}}$ and therefore $f(z_{i_k}, \pi, t') = z_{i_k} \in \mathcal{Z}_{\texttt{safe}}$ for all $t' \in [0, t_i] = \{0\}$. □

Lemmas 7.10 and 7.11 together imply the theorem. □

7.2.3. Checking Verification Conditions

We now describe how we check each of the VCs for a given controller π and a hybrid automaton $\mathcal{A}$.

Verification condition 1. We observe that VC 1 is local to the dynamics of a single mode—i.e., it suffices to verify a safe reachabililty property for the dynamics $\dot{x} = f(z, \pi(z))$, where the mode of z does not change. Thus, we can drop the mode and express these dynamics as $\dot{x} = f^q(x, \bar{\pi}(x))$, where $f^q : \mathcal{X} \times \mathcal{U} \to \mathbb{R}^n$ and we have defined $\bar{\pi} : \mathcal{X} \to \mathcal{U}$ by $\bar{\pi}(x) = \pi(h(q, x))$. We let $F^q(x, \bar{\pi}, t) \subseteq \mathcal{X}$ denote the trajectory generated by evolving the system according to this differential equation from state $x \in \mathcal{X}$ for time $t \in \mathbb{R}_{\geq 0}$.

Although verification tools like Verisig can only check safety properties, we can encode the reachability condition as a safety condition by considering a time-limit $T_{\max}$ within which we require the

Algorithm 9 Compositional learning and synthesis. *Inputs:* Hybrid automaton $\mathcal{A}$ and initial candidate pre/post-regions B_0. *Output:* A verified controller π or `FAIL`. *Hyperparameters:* Number of synthesis iterations $K \in \mathbb{N}$.

function LearnController($\mathcal{A}, B_0$)
  $\pi \leftarrow \mathsf{Learn}(\mathcal{A}, B_0)$
  $B_\pi \leftarrow \mathsf{Synthesize}(\mathcal{A}, \pi, B_0)$
  if $B_\pi = \varnothing$ **then return** `FAIL`
  if $\mathsf{Verify}(\mathcal{A}, \pi, B_\pi)$ **then return** π
  return `FAIL`
end function

function Synthesize($\mathcal{A}, \pi, B$)
  $E \leftarrow \varnothing$
  for $i \in \{1, \ldots, K\}$ **do**
    $E \leftarrow E \cup \mathsf{Test}(\mathcal{A}, \pi, B)$
    $B \leftarrow \mathsf{Infer}(E)$
    if $B = \varnothing$ **then return** $\varnothing$
  end for
  return B
end function

system to reach $\mathcal{X}^q_{\text{post}}$ when started in any state in $\mathcal{X}^q_{\text{pre}}$. Note that the mode predictor has a discrete output; we model each output as a separate mode of the hybrid system.

Verification condition 2. Next, for VC 2, we need to check that for all $q, q' \in \mathcal{Q}$, we have $\{x' \mid (q, x) \to (q', x') \in \mathcal{T},\ x \in \mathcal{X}^q_{\text{post}}\} \subseteq \mathcal{X}^{q'}_{\text{pre}}$. In other words, every state reachable from $x \in \mathcal{X}^q_{\text{post}}$ is contained in $\mathcal{X}^{q'}_{\text{pre}}$ for some $q' \in \mathcal{Q}$. This check is problem-specific. For instance, in our F1/10 example, the transitions $(x, q) \to (x', q') \in \mathcal{T}$ involve an affine change of coordinates—i.e., $x' = A^{q \to q'} x + b^{q \to q'}$. Thus, assuming $\mathcal{X}^q_{\text{post}}$ and $\mathcal{X}^{q'}_{\text{pre}}$ are represented by convex polytopes P^q_{post} and $P^{q'}_{\text{pre}}$, respectively, then we can verify VC2 by checking for $q, q' \in \mathcal{Q}$, whether $A^{q \to q'} P^q_{\text{post}} + b^{q \to q'} \subseteq P^{q'}_{\text{pre}}$. To check this, it suffices to check that each vertex of the polytope $A^{q \to q'} P_{\text{post}} + b^{q \to q'}$ is contained in $P^{q'}_{\text{pre}}$, which corresponds to checking feasibility of a system of linear inequalities, which we can do efficiently via linear programming.

7.3. Compositional Learning and Synthesis

Our overall framework is summarized in Algorithm 9. Suppose we are given initial *pre/post-regions* B_0—i.e., a pre- and a post-region for every mode $q \in \mathcal{Q}$. Then, the method consists of the following

steps:

- **Learning:** Train a controller π that tries to drive the system from every state in the pre-region of each mode q to the post-region of q, where we use the pre/post regions in B_0.
- **Pre/post-region synthesis:** Synthesize new candidate pre/post-regions B_π for π.
- **Verification:** Use the algorithm in Section 7.2 with B_π to try and prove that π is safe and live.

A natural choice for the initial pre/post-regions is to take $\mathcal{Z}^q_{\text{pre}} = \mathcal{Z}_0 \cap \mathcal{Z}^q$ and $\mathcal{Z}^q_{\text{post}} = \mathcal{Z}_F \cap \mathcal{Z}^q$ for all $q \in \mathcal{Q}$. The above procedure can fail because of two reasons: either synthesis fails (i.e., no set of pre/post-regions consistent with the generated examples exists) or verification fails. In either case, we retry the above steps with modified rewards for learning and/or a different choice of initial pre/post-regions. In our experiments, we retried our procedure (Algorithm 9) a few (3-4) times with different reward functions until we were able to verify the learned controller.

The subroutine for synthesizing a candidate set of pre/post-regions alternates between the following two steps:

- **Testing:** Generate new examples using testing.
- **Inference:** Infer a candidate set of pre/post-regions B based on examples E generated so far.

The examples E include both *implication examples* $z \to z' \in \mathcal{Z}^2$ such that z' is reachable from z using π, and *unsafe examples* $z \in \mathcal{Z}$ that reach an unsafe state using π.

Below, we describe our pre/post-region inference algorithm (Section 7.3.1) and our testing algorithm (Section 7.3.2), as well as our compositional learning algorithm (Section 7.3.3).

7.3.1. Pre/Post-Region Inference

Problem formulation. We describe our algorithm for inferring pre- and post-regions given a set of examples. First, we represent the regions using boxes–i.e., products of intervals.

Definition 7.12. *A* box $b \in \mathcal{B}$ *in* $\mathbb{R}^n$ *is defined by* $b = \prod_{i=1}^{n}[x_i, y_i] \subseteq \mathbb{R}^n$*, where* $x_i \leq y_i$ *for all* $i \in \{1, ..., n\}$.

We synthesize a set of boxes $B = \{b_\alpha \mid \alpha \in \{\text{pre}, \text{post}\} \times Q\}$ denoting the pre- and post-regions of all the modes. For now, we assume given *lower and upper bounds* $b_\alpha^\perp, b_\alpha^\top$ for all $\alpha \in A = \{\text{pre}, \text{post}\} \times Q$. As discussed below, these bounds are used to enforce CCs 1 & 2. Then, our goal is to find boxes b_α for all $\alpha \in A$ satisfying $b_\alpha^\perp \subseteq b_\alpha \subseteq b_\alpha^\top$, such that taking $\mathcal{X}_{\text{pre}}^q = b_{(\text{pre},q)}$ and $\mathcal{X}_{\text{post}}^q = b_{(\text{post},q)}$, VCs 1 & 2 are satisfied. We denote the set of lower and upper bounds by $B^\perp$ and $B^\top$ respectively.

First, we describe the kinds of examples that are available. Examples are states (or pairs of states) that encode a *necessary* condition for the VCs to hold—i.e., if the invariant does not satisfy an example, then it cannot possibly satisfy the VCs, but the converse is not true. First, we have states from which using π is unsafe.

Definition 7.13. *An* unsafe example *is a pair* (α, x) *where* $\alpha = (pre, q) \in A$ *and* $x \in \mathcal{X}$ *such that there exists* $t \in \mathbb{R}_{\geq 0}$ *with* $f((q, x), \pi, t) \notin \mathcal{Z}_{\texttt{safe}}$ *and* $f((q, x), \pi, t') \notin \mathcal{Z}_F$ *for all* $t' \in [0, t)$.

Next, we have examples that correspond to pairs of states z and z' where z' is reachable from z.

Definition 7.14. *An* implication example *is a pair* $(\alpha, x) \to (\alpha, x')$ *with* $\alpha, \alpha' \in A$ *and* $x, x' \in \mathcal{X}$ *such that either (i)* $\alpha = (post, q)$ *and* $\alpha' = (pre, q')$*, with* $(q, x) \to (q', x') \in \mathcal{T}$*, or (ii)* $\alpha = (pre, q)$ *and* $\alpha' = (post, q)$*, and there exists* $t \in \mathbb{R}_{\geq 0}$ *with* $(q, x') = f((q, x), \pi, t) \in \mathcal{Z}_F$*,* $F((q, x), \pi, t) \subseteq \mathcal{Z}_{\texttt{safe}}$ *and* $f((q, x), \pi, t') \notin \mathcal{Z}_F$ *for all* $t' \in [0, t)$.

Given these two kinds of examples, our goal is to synthesize a candidate set of boxes that is consistent with them—i.e., it excludes examples that are inconsistent with our VCs.

Definition 7.15. *Given lower and upper bounds* $B^\perp, B^\top$*, unsafe examples* C *and implication examples* I*, a candidate set of boxes* B *is* consistent *if the following hold:*

- *For all* $(\alpha, x) \in C$*, we have* $x \notin b_\alpha$.
- *For all* $(\alpha, x) \to (\alpha', x') \in I$*,* $x \in b_\alpha \Rightarrow x' \in b_{\alpha'}$.

Algorithm 10 Pre/post-region inference. *Inputs:* Implication & unsafe examples E. *Output:* Candidate pre/post-regions B. *Hyperparamters:* $B^\bot, B^\top$

```
1: function Infer(E)
2:    I, C ← E
3:    for α ∈ A do
4:       X⁺_α ← ∅
5:       X⁻_α ← {x | (α, x) ∈ C}
6:    end for
7:    while true do
8:       for α ∈ A do
9:          b_α ← ConsistentBox(X⁺_α, X⁻_α, b^⊥_α, b^⊤_α)
10:         if b_α = ∅ then return ∅
11:      end for
12:      ψ ← true
13:      for (α, x) → (α′, x′) ∈ I do
14:         if x ∈ b_α and x′ ∉ b_α′ then
15:            X⁺_α′ ← X⁺_α′ ∪ {x′}
16:            ψ ← false
17:         end if
18:      end for
19:      if ψ then return {b_α | α ∈ A}
20:   end while
21: end function
```

- *For all $\alpha \in A$, we have $b^\bot_\alpha \subseteq b_\alpha \subseteq b^\top_\alpha$.*

Furthermore, B is minimal *if for any candidate set of boxes $\tilde{B}$ satisfying these conditions, $b_\alpha \subseteq \tilde{b}_\alpha$ for all $\alpha \in A$.*

Given bounds $B^\bot$, $B^\top$, unsafe examples C, and implication examples I, the Infer subroutine used in Algorithm 9 returns a minimal consistent candidate set of boxes (if one exists, returning $\varnothing$ otherwise).

Algorithm. Next, we describe our algorithm for synthesizing minimal set of boxes given a set of examples. This algorithm is outlined in Algorithm 10. Our approach is to reduce the synthesis problem to the following:

Definition 7.16 (Consistent Box)**.** *Given positive examples $X^+ \subseteq \mathbb{R}^n$, negative examples $X^- \subseteq \mathbb{R}^n$*

and boxes $b^\perp, b^\top$*, the* (minimal) consistent box *is*

$$b^* = \arg\min_{b \in \mathcal{B}} \prod_{i=1}^{n} (y_i - x_i) \quad \textit{subj. to} \quad X^+ \subseteq b, \quad b \cap X^- = \varnothing, \quad b^\perp \subseteq b \subseteq b^\top.$$

That is, the goal is to find the smallest box that includes X^+ and excludes X^-. This problem can be solved efficiently—in particular, let $b = \prod_{i=1}^{n} [x_i, y_i]$, where $x_i = \min\{x'_i \mid x' \in X^+\} \cup \{x_i^\perp\}$ and $y_i = \max\{x'_i \mid x' \in X^+\} \cup \{y_i^\perp\}$ where $b^\perp = \prod_{i=1}^{n} [x_i^\perp, y_i^\perp]$. Then, return b if $b \cap X^- = \varnothing$ and $b \subseteq b^\top$; otherwise, we return $\varnothing$ (i.e., no such box exists).

Our synthesis algorithm initializes positive examples $X_\alpha^+ = \varnothing$, and negative examples X_α^- to be the unsafe examples, for each $\alpha \in A$. Then, at each iteration, it independently synthesizes a consistent box b_α to be the minimal consistent box[20] for positive examples X_α^+, negative examples X_α^-, and boxes $b_\alpha^\perp, b_\alpha^\top$. Next, it handles implication examples in I by expanding the sets X_α^+ for $\alpha \in A$. In particular, it checks if any of the implication examples $(\alpha, x) \to (\alpha', x') \in I$ violate the current candidate invariant—i.e., $x \in b_\alpha$ but $x' \notin b_{\alpha'}$. If so, it requires that $x' \in b_{\alpha'}$ by adding x' to $X_{\alpha'}^+$. It continues the iterative process until either all examples in I are satisfied, in which case it returns the current candidate boxes B, or the consistent box subroutine fails, in which case it returns $\varnothing$.

Suppose there exists a set of minimal consistent boxes $\{b_\alpha^* \mid \alpha \in A\}$. Then, our algorithm maintains the invariant that the current candidate boxes $\{b_\alpha \mid \alpha \in A\}$ are contained in the minimal consistent boxes—i.e., $b_\alpha \subseteq b_\alpha^*$ for all $\alpha \in A$. Therefore, when dealing with an inconsistent implication example $(\alpha, x) \to (\alpha', x') \in I$ with $x \in b_\alpha$, we can infer that $x' \in b_{\alpha'}^*$ and hence it correctly adds x' to $X_{\alpha'}^+$, forcing $b_{\alpha'}$ in the next iteration to include x'. Since we deal with any implication example at most once and we deal with at least one implication example in every iteration (except the last iteration), we have:

Theorem 7.17. *Algorithm 10 terminates after at most* $|I|$ *iterations and computes a set of minimal consistent boxes if one exists and returns* $\varnothing$ *otherwise.*

[20] Although X_α^+ is initialized to $\varnothing$, b_α is not empty since it has to contain $b_\alpha^\perp$.

Algorithm 11 Testing to check verification conditions. *Inputs:* Hybrid automaton $\mathcal{A}$, NN controller π, and candidate pre/post-regions B. *Output:* Implication & unsafe examples E. *Hyperparameters:* Horizon $T \in \mathbb{R}_{>0}$, iterations $K \in \mathbb{N}$.

function Test($\mathcal{A}, \pi, B$)
  $E \leftarrow \varnothing$
  for $i \in \{1, ..., K\}$ **do**
    $\alpha \leftarrow (\beta, q) \sim \text{Uniform}(A)$
    $z \leftarrow (q, x)$ where $x \sim \mathcal{P}(b_\alpha)$
    if $\beta = \text{pre}$ **then**
      $\zeta \leftarrow F(z, \pi, T)$ (stop as soon as ζ enters $\mathcal{Z}_F$)
      $z' \leftarrow (q, x') = f(z, \pi, T)$
      if $\zeta \not\subseteq \mathcal{Z}_{\texttt{safe}}$ **then** $E.C.\mathsf{Add}((\alpha, x))$
      if $z' \notin \mathcal{Z}^q_{\text{post}}$ **then** $E.I.\mathsf{Add}\big((\alpha, x) \to ((\text{post}, q), x')\big)$
    else
      $z' \leftarrow (q', x') \sim \mathcal{P}(\{z' \mid z \to z' \in \mathcal{T}\})$
      if $z' \notin \mathcal{Z}^{q'}_{\text{pre}}$ **then** $E.I.\mathsf{Add}\big((\alpha, x) \to ((\text{pre}, q'), x')\big)$
    end if
  end for
  return E
end function

Choosing upper and lower bounds. Finally, we use the upper and lower bounds to handle CCs 1 & 2. First, CC 1 says that for every state $(q, x) \in \mathcal{Z}_0$, we have $x \in b_\alpha$ where $\alpha = (\text{pre}, q)$. Thus, to ensure this condition holds, it suffices to choose $b^\perp_\alpha$ such that $\mathcal{X}^q_0 \subseteq b^\perp_\alpha$ for all $q \in \mathcal{Q}$. Similarly, CC 2 says that for every $x \in b_\alpha$ with $\alpha = (\text{post}, q)$, we have $(q, x) \in \mathcal{Z}_F$; thus, it suffices to choose $b^\top_\alpha \subseteq \mathcal{X}^q_F$ for all $q \in \mathcal{Q}$.

7.3.2. Testing

Our testing subroutine takes as input candidate pre/post-regions B and uses simulated trajectories from random start states to try and discover examples that are inconsistent with our VCs. Our testing algorithm is summarized in Algorithm 11.

At a high level, it samples trajectories $\zeta \subseteq \mathcal{Z}$ starting from random states $z = (q, x)$, where $\alpha = (\beta, q) \sim \mathsf{Uniform}(A)$, and $x \sim \mathcal{P}(b_\alpha)$—e.g., we can take $\mathcal{P}(b_\alpha)$ to be the uniform distribution over b_α. Then, it checks whether ζ is an unsafe or an implication example that is inconsistent with B; if so, it adds z to $E.C$ and/or $z \to z'$ (z' is the last state in ζ) to $E.I$, respectively. Finally, it returns the set of examples E which is then used by our pre/post-region inference algorithm.

Algorithm 12 Compositional learning to try and satisfy verification conditions. *Inputs:* Hybrid automaton $\mathcal{A}$, candidate pre/post-regions B. *Output:* Compositional controller π.

function Learn($\mathcal{A}, B$)
  for $q \in \mathcal{Q}$ **do**
    $p(x) = p^q(x)$ where $x \sim \mathcal{P}(b_{(\text{pre},q)})$
    $r(x) = r^q(x, b_{(\text{post},q)})$
    $\pi^q \leftarrow \mathsf{ReinforcementLearning}(f^q, p(x), r(x))$
  end for
  $p(q, x) = p(q)p(x)$ where $q \sim \text{Uniform}(\mathcal{Q})$, $x \sim \mathcal{P}_{\pi^q}$
  $\mu \leftarrow \mathsf{SupervisedLearning}(p(z), h(z))$
  return π
end function

7.3.3. Controller & Mode Predictor Learning

We describe our approach for learning the compositional controller π, which involves learning the controller π^q for each mode $q \in \mathcal{Q}$ as well as learning the mode predictor μ. Our approach is summarized in Algorithm 12.

Controllers. First, we use reinforcement learning to learn the controllers π^q for each mode q. We parameterize $\pi^q = \pi^q_\theta$ as a neural network $\pi^q_\theta : \mathcal{O} \to \mathcal{U}$ mapping observations to actions. The inputs to the reinforcement learning algorithm are the dynamics f^q for mode q, a distribution $p(x)$ over initial states x, and a reward function $r : \mathcal{X} \to \mathbb{R}$. For the initial state distribution, we assume given a distribution $\mathcal{P}(b_{(\text{pre},q)})$ over the pre-region of q—e.g., the uniform distribution. The reward function should encourage the system to reach the next region. We can use any reinforcement learning algorithm in conjunction with these inputs to learn π^q. We use the twin delayed deep deterministic policy gradient (TD3) algorithm [50], which is a more stable variant of the popular deep deterministic policy gradient (DDPG) algorithm [104].

Mode predictor. Next, we learn the mode predictor using supervised learning. To do so, we need to construct a training set consisting of input-output examples $(o, q) \in \mathcal{O} \times \mathcal{Q}$, where observation o is the input and mode q is the ground truth mode. To do so, we sample states $z = (q, x)$, compute the observations $o = h(z)$, and then construct the training examples (o, q). For the distribution $p(z) = p(q, x)$ over states z, we use the uniform distribution over q and the distribution $\mathcal{P}_{\pi^q}$ over x visited by the controller π^q. The reason we use this distribution over x is that it is the distribution

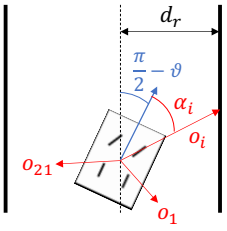


Figure 7.5: LiDAR observation for a straight segment.

of x values that the mode predictor will encounter when running π. Finally, we parameterize μ using a neural network $\mu_\theta : \mathcal{O} \times \mathcal{Q} \to [0, 1]$ (i.e., predict the probability $\mu_\theta(q \mid o)$ of mode $q \in \mathcal{Q}$ given observation $o \in \mathcal{O}$).

7.4. System Modeling

We briefly describe the F1/10 car model used in our evaluation, and how we train the controllers π^q and the mode predictor μ.

Dynamics model. We use the model in [80]. We use vector notations $\vec{x} \in \mathcal{X}$ and $\vec{u} \in \mathcal{U}$ for clarity. The car dynamics are given by a kinematic bicycle model with 4D state space $\vec{x} = (x, y, \vartheta, v) \in \mathcal{X} \subseteq \mathbb{R}^4$, including 2D position (x, y), orientation ϑ, and velocity v. The actions are $\vec{u} = (a, \phi) \in \mathcal{U} \subseteq \mathbb{R}^2$, where a denotes throttle and ϕ is the orientation of the front wheels. We assume throttle is constant at $a = 16$ (resulting in a top speed of 2.4m/s), whereas ϕ is set by the controller at a sampling rate of 10Hz. The dynamics are governed by the following differential equations (with respect to time):

$$\begin{aligned} \dot{x} &= v \cdot \cos(\vartheta) \qquad & \dot{v} &= -c_a \cdot v + c_a \cdot c_m \cdot (a - c_h) \\ \dot{y} &= v \cdot \sin(\vartheta) \qquad & \dot{\vartheta} &= \frac{v}{\ell} \cdot \tan(\phi) \end{aligned} \tag{7.1}$$

where $c_a = 1.633$ is the car's acceleration constant, $c_m = 0.2$ is its motor constant, $c_h = 4$ is its hysteresis constant, and $\ell = 0.45$ is the its length. We consider two different observation models.

State-feedback system First, we consider a variant of the F1/10 car with state-feedback—i.e., $\mathcal{O} = \mathcal{Z}$ and the controller $\pi^q : \mathcal{Z} \to \mathcal{U}$ has access to the true state of the car; similarly, the mode predictor $\mu : \mathcal{Z} \to \mathcal{Q}$ outputs the true mode $\mu(q, x) = q$. This setting allows us to evaluate the controllers in isolation of the mode detector.

LiDAR observation model. Next, we consider a LiDAR based observation model. A LiDAR scan consists of a number of laser rays emanating at a range of degrees with respect to the car's orientation. For each ray, the car receives the distance to the nearest object reached by the ray, or the maximum LiDAR range of 5m if no obstacle is in that range. The controller has access to the LiDAR measurements only and cannot observe the position, orientation or the velocity of the car. Similar to prior work [80], we focus on a LiDAR scan with 21 rays since the complexity of the verification task increases exponentially with the number of rays. This gives us a 21-dimensional observation $o \in \mathbb{R}^{21}$. The rays range from -115 to 115 degrees relative to the car's orientation—i.e., there are rays at $-115, -103.5, \ldots, 115$ degrees relative to the car's orientation. Each LiDAR ray can be modeled as a function of the car's state relative to the current track segment. Figure 7.5 illustrates the scenario of a ray reaching the right wall in a straight segment. The specific equation for such a ray is

$$o_i = h(\vec{x})_i = \frac{d_r}{\cos(\vartheta - \alpha_i)},$$

where d_r is the distance to the right wall, and α_i is the relative angle (in radians) of ray i with respect to the car's orientation ϑ. Rays for other walls and segments can be modeled similarly, depending on which walls are in range.

Tracks. We consider tracks consisting of a sequence of segments, each corresponding to one of five modes: right and left 90-degree turns, right and left 120-degree turns, and straight segments. Each segment is 1.5m wide and is of a fixed length. Straight segments can be of arbitrary lengths but must be sufficiently long to allow for an inductive proof of our VCs; see Section 7.5. The segments are lined up with the end of one segment meeting the start of the next one. We represent each segment as having coordinates where the top-most corner is at the origin. Then, a mode transition $z \to z' \in \mathcal{T}$ is an (instantaneous) affine change of coordinates[21] to bring the car into this coordinate system. Furthermore, there is a mode transition from any state at the end of any segment to a state at the start of every segment, thereby modeling all possible tracks in a single hybrid automaton.

[21] The post-region of one segment is contained within the pre-region of the next segment (after change of coordinates) since mode transitions are instantaneous and do not involve movement of the car.

Safety. The safety property is that the car should not run into any of the walls. We model the car as a square of size $\gamma = 0.15m$ and the walls as line segments. Then, the car should not intersect the wall—i.e., $\mathcal{X}^q_{\texttt{safe}} = \{\vec{x} \in \mathcal{X} \mid \forall w \in \mathsf{walls}[q] \;.\; \|(x, y) - (w_x, w_y)\|_\infty \geq \gamma\}$.

Controller. For the state-feedback system, each controller has 5 inputs: the x and y distances to each of the two corners in the turn and the car's orientation relative to the segment. For LiDAR-feedback, each controller has 21 inputs corresponding to the LiDAR rays. We use reinforcement learning to train the controllers π^q. We represent the policy as an NN $\pi^q = \pi^q_\theta$ with two fully connected layers with tanh activations and 16 neurons per layer for the state-feedback system and 64 neurons per layer for the LiDAR system. We use a uniform distribution on the pre-region as the initial state distribution. We use a reward function that aims to achieve two goals: (i) stay in the safe region, (ii) stay in regions where we can compose the different verification results. The second goal is necessary for our compositional approach to work, since we need the car to visit the post-region when started in the pre-region. To achieve this goal, we train controllers that stay in the middle of each segment after turns, with the exception of the sharp turns, where it seems challenging to train controllers to stay in the middle. For instance, the reward function used for a right turn (the left-turn case is symmetric) is

$$r(\vec{x}, \vec{u}) = \begin{cases} g_s - g_i \cdot \phi^2 - g_m \cdot d(\vec{x}) & \text{if before turn} \\ g_s - g_h \cdot h(\vec{x}) & \text{if during turn} \\ g_s + g_f \cdot \delta(\vec{x}, \vec{u}) - g_m \cdot d(\vec{x}) & \text{if after turn} \\ g_c & \text{if crash,} \end{cases}$$

where $g_s = 5$ is a reward for each safe step, g_i is a loss penalizing high steering angle ϕ, g_m is a penalty on the car's distance $d(\vec{x})$ from the middle of the lane, $g_h = 3$ is a loss penalizing the difference $h(\vec{x})$ between the car's orientation and the turn angle (either 90 or 120 degrees), $g_f = 10$ is a reward for the distance $\delta(\vec{x}, \vec{u})$ covered on the current step after the turn in the new segment direction, and $g_c = 100$ is a penalty for crashing. The values g_i and g_m depend on the mode and are chosen as follows: (i) $g_i = -0.05, g_m = -2$ for state-feedback; (ii) $g_i = 0, g_m = -3$ for a

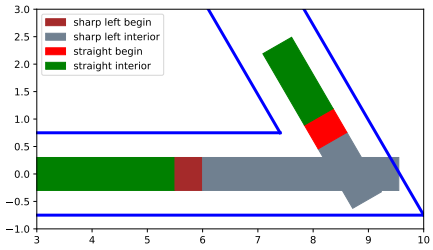


Figure 7.6: Regions for training the mode predictor

90-degree turn and LiDAR-feedback; and (iii) $g_i = -0.05, g_m = -0.5$ for a 120-degree turn and LiDAR-feedback.

Mode predictor. We decompose the mode predictor into two parts: (i) a *new mode predictor* $\mu^p : \mathcal{O} \to \mathcal{Q}$ and (ii) an *exit detector* $\mu^q : \mathcal{O} \to \{0, 1\}$, one for each mode q. Intuitively, μ^p is used to determine the mode q the system is about to enter; once q is determined, the corresponding μ^q is run until it predicts that system has exited mode q (at which point μ^p is run again). Since standard control systems are sampled periodically, let o_k denote the observation at sampling step k. Then, the output of the overall mode predictor at step k, q_k, is defined as follows:

$$\begin{aligned} q_k &= q_{k-1} && \text{if } \mu^{q_{k-1}}(o_k) = 0 \\ q_k &= \mu^p(o_k) && \text{if } \mu^{q_{k-1}}(o_k) = 1, \end{aligned} \tag{7.2}$$

where $q_0 = \mu^p(o_0)$. This decomposition simplifies mode predictor training since each individual NN is trained either only on data from one mode (in the case of μ^q) or on data from the pre-regions of all the modes (in the case of μ^p). Specifically, we divide each track segment into two regions: one consisting of the 50cm at the beginning of the segment, and the other of the rest of the segment; examples are shown in Figure 7.6. Each exit detector μ^q is trained to predict 0 (i.e., "not exited") on LiDAR scans taken in its own mode q (both in the beginning region and the remainder region) and 1 (i.e., "exited") on scans from the beginning region of other modes $q' \neq q$. The new mode detector μ^p is trained to predict the mode in which the LiDAR scan was taken, with half the training examples from beginning regions of each mode and half from remaining regions. This strategy allows the mode predictor to recover from incorrect predictions by μ^p—i.e., if $q_k = \mu^p(o_k)$ is an error, then

μ^{q_k} should predict that q_k is wrong at the next step (i.e., $\mu^{q_k}(o_{k+1})$ should be 1) and ask μ^p to update its prediction. In summary, the mode predictor consists of a total of six NNs: a *new mode predictor* and five *exit detectors*, one for each track segment. All NNs have two hidden layers, with 32 neurons per layer; the hidden layers have tanh activations, whereas the output layer is linear. We train them with a batch size of 32 and learning rate of 0.001. Training is fairly fast and takes a few minutes per NN.

Verification. We use the Verisig tool [79] for verification. Verisig verifies neural networks with smooth activation functions (e.g., sigmoid, tanh) by transforming the networks into hybrid systems. The neural network hybrid system is then composed with the dynamics model, thereby converting the closed-loop problem into a hybrid system verification problem that is solved by Flow* [36].

7.5. Experimental Results

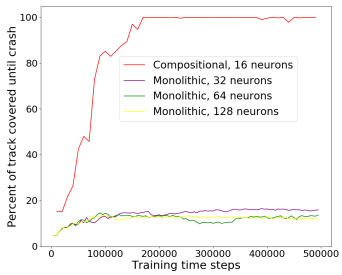
We evaluate our framework on the F1/10 car, aiming to address the following research questions:

- Can our compositional learning strategy improve the scalability of reinforcement learning?
- Can our compositional verification algorithm be used to prove that the learned controller safe and live for arbitrary sequences of track segments?

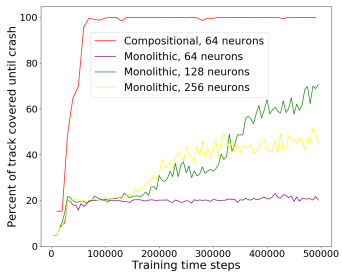
7.5.1. Benefits of Compositional Learning

For both state-feedback and LiDAR systems, we trained two controllers: one for the 90-degree right turn and one for the 120-degree right turn. Since left and right turns are symmetric, we use the right-turn controller for a left turn by reflecting the observations and negating the control input. We also use the 90-degree controller in straight segments, since it is able to steer the car close to the middle.

To illustrate the benefit of compositional learning, we trained a single NN controller for the full track in Figure 7.3c. We used increasingly larger NNs (with 32, 64, 128 neurons per layer for state-feedback and 64, 128 and 256 neurons per layer for observation-feedback); however, none safely completed a lap in the entire track. Figures 7.7a & 7.7b show the performance of these controllers along with the performance of the compositional controller (the individual controllers combined with

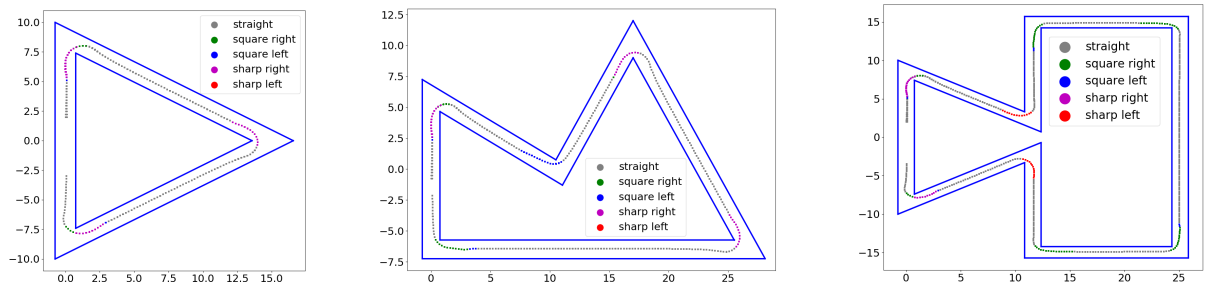


(a) State-feedback system.



(b) LiDAR-feedback system.

Figure 7.7: Training evolution for state- and LiDAR-feedback controllers. The “Compositional" controller curve shows the combined number of training steps for controllers trained on each individual turn, whereas the “Monolithic" controllers are trained on the track from Figure 7.3c. All NNs have two fully connected layers, with the number of neurons per layer indicated in the legend. Results are averaged over five runs per setup.



(a) (b) (c)

Figure 7.8: Example trajectories with LiDAR-feedback using the compositional controller. The color of each position indicates the mode predictor output.

a pre-trained mode predictor) on the full track, as a function of the number of training steps. As expected, training is fast and stable for our compositional controller, whereas the monolithic ones are unable to converge to a stable policy. While it may be possible to train a monolithic controller using a larger NN or a different reward function, our results provide evidence that the compositional approach is simpler and requires less expert domain knowledge, both in reinforcement learning and in the specific system.

Our compositional controller performs well (and can be verified, as shown in the verification experiments below) on *all* tracks constructed using the five kinds of segments. Figure 7.8 shows the

simulated trajectories of the compositional controller on the tracks in Figure 7.3. While the mode predictor sometimes predicts the wrong mode when far from the turn, it eventually switches to the correct one selecting the appropriate controller for the remainder of the turn.

7.5.2. Pre/Post-Region Synthesis

Our synthesis algorithm is used to compute pre/post-regions for all the modes. We abuse notation and use y to denote the y-distance (in meters) from the start of the segment and x to denote the distance from the left wall. The synthesized pre-region is the same for all the modes because we have implication examples from the post-region of every mode to the pre-region of each mode. The pre-region computed for the LiDAR-feedback system is given by $x \in [0.75, 0.83]$, $y \in [0, 0.24]$, $\vartheta \in [\frac{\pi}{2} - 0.0042, \frac{\pi}{2} + 0.002]$, and $v \in [2.4, 2.4]$. The post-regions are the corresponding boxes at the end of each segment. For example, the post-region computed for the 90-degree right turn is given by $x \in [8, 8.24]$, $y \in [5.67, 5.75]$, $\vartheta \in [-0.0042, 0.002]$ and $v \in [2.4, 2.4]$.

7.5.3. Verification Results for LiDAR-Feedback System

Verifying safety for the LiDAR-feedback system is challenging due to multiple discrete computations. First, the controller π has a discrete internal state due to use of the mode predictor, which creates additional modes in the hybrid automaton given to Flow*. In addition, if a given LiDAR ray can reach multiple walls in a given reachable set of states, then each case needs to be encoded as a different mode of the hybrid automaton. During verification, a reachable set can get split into multiple reachable sets due to case analysis, generating multiple *branches* each of which is a verification instance of its own. The number of such branches can be exponential in the number of modes since branching occurs dynamically as time progresses. Thus, it is essential to keep the uncertainty as small as possible as reachable sets are propagated through time. However, closed-loop verification tools such as Verisig rely on overapproximating the system's reachable set, and this approximation error can grow quickly over time. A standard strategy is to partition the initial set and verify each subset separately. This process can also suffer from exponential blowup, but it alleviates the compounding uncertainty issue. Another benefit of this partitioning is that we can parallelize verification.

Mode	# instances	# branches	Verification time (hours)	
			Composed system	NN only
90-degree right	1000	8.22	5.80	0.31
90-degree left	1000	6.77	7.08	0.35
120-degree right	1000	12.17	12.22	0.580
120-degree left	1000	14.62	11.41	0.531
90-degree right*	20	2.30	2.18	0.110
90-degree left*	20	1.95	2.38	0.110
120-degree right*	20	3.45	2.97	0.150
120-degree left*	20	2.15	2.87	0.130
straight initial	20	1.35	0.93	0.043
straight inductive	1	1.00	0.04	0.002

Table 7.1: Verification results for LiDAR observations. Verification times and number of branches are averaged across all the instances for that mode. The cases labeled $*$ do not handle discrete sampling of the controller. "Composed system" is the (average) time for fully verifying a single instance, and "NN only" is the time spent propagating reachable sets through the NNs during closed loop verification.

An additional verification challenge is that for a real system, we need to sample the controller at discrete points in time. Thus, we cannot switch modes at the exact point in time after the mode transition happens. We can account for this error by enlarging the pre-region—e.g., for a controller sampled at 0.1s intervals, we need to enlarge the pre-region by 0.25m in the y-direction. We report results both with and without this modification, in order to illustrate the challenge introduced by an extra dimension of uncertainty.

Verification of turns. The results are summarized in Table 7.1. We use slightly larger pre/post-regions than those computed by the synthesis algorithm to account for overapproximation errors introduced in verification. We split the initial set by increments of 0.005 along the x-dimension and 0.005 along the y-dimension, resulting in 1000 verification instances per turn. We verify them in parallel on an 80-core machine running at 1.2GHz. Although the left and right turns are symmetric, we need to verify them separately since the full compositional controller may not be symmetric.

As shown in Table 7.1, most instances took a few hours to verify on average, depending mostly on the number of branches (of reachable sets) through the hybrid automaton. Note that verification

requires significantly less computation for the case when no y uncertainty due to the discrete control sampling is considered. Note also that the 120-degree turn verification is much more challenging because of the larger open space in the turn, resulting in more branching due to LiDAR rays reaching different walls. Furthermore, the controller needs to take a more drastic action to make the turn, which makes the reachable set computation harder since the NN is sensitive to small changes to its input, which amplifies approximation errors. In particular, there were some instances with more that 70 branches, taking more than 60 hours to finish.

Verification of straight segments. The straight segment verification is different since straights can be of arbitrary length (above some minimum). Thus, we need an inductive argument to perform verification. Ideally, we would establish an inductive invariant $\mathcal{X}_{\text{inv}}$ such that if the car starts in $\mathcal{X}_{\text{inv}}$ at step k, then it remains in $\mathcal{X}_{\text{inv}}$ until step $k+1$ while making progress along the track (i.e., in the y-direction).

For a typical choice of such a region, the car might leave but then return after multiple steps. For example, if $\vartheta = \frac{\pi}{2} - 0.005$ and $x = 0.85$, then the car is facing to the right and will reach a value of x greater than 0.85 as soon as it moves; however, our NN controller eventually steers the car back to a smaller x value. We find it is significantly easier to identify a *recurrent* set such that the system returns to this set periodically. Let[22] $\tilde{\mathcal{X}}_{\text{post}} = \{(x, \vartheta, v) \mid \exists y \,.\, (x, y, \vartheta, v) \in \mathcal{X}_{\text{post}}\}$. Then, we compute a subset $\tilde{\mathcal{X}}_{\text{rec}} \subseteq \tilde{\mathcal{X}}_{\text{post}}$, and prove (i) the car reaches $\tilde{\mathcal{X}}_{\text{rec}}$ from $\mathcal{X}_{\text{pre}}$ in i steps, and (ii) if the car starts in $\tilde{\mathcal{X}}_{\text{rec}}$, then it returns to $\tilde{\mathcal{X}}_{\text{rec}}$ in j steps for some $j \in \mathbb{N}$; during this time, it always stays in $\tilde{\mathcal{X}}_{\text{post}}$ and makes progress in the y-direction. Intuitively, (i) is the base case and (ii) is the inductive case of a safety proof by induction. We do not need to consider y-uncertainty for this case; thus, the number of instances is just 20.

More formally, we want to verify straight segments of all lengths $\ell \geq \ell_{\min}$ where $\ell_{\min}$ is a bound on the shortest straight segment. We first choose $\mathcal{Z}^q_{\text{post}}$ to be of the form $\mathcal{Z}^q_{\text{post}} = \{(q, (x, y, \vartheta, v)) \mid (x, \vartheta, v) \in \tilde{\mathcal{X}}_{\text{post}}, y \in [y^\ell_{\text{inf}}, y^\ell_{\text{sup}}]\}$, where $\tilde{\mathcal{X}}_{\text{post}}$[23] $\subseteq \mathbb{R}^3$ is a set of possible values for x, ϑ and v. Next, we identify a *recurrent* set $\tilde{\mathcal{X}}_{\text{rec}} \subseteq \tilde{\mathcal{X}}_{\text{post}}$ and define $\mathcal{Z}^q_{\text{rec}} = \{(q, (x, y, \vartheta, v)) \mid (x, \vartheta, v) \in \tilde{\mathcal{X}}_{\text{rec}}\}$. Then

[22] We only consider x, ϑ, and v since y does not affect the observations.
[23] The superscript q is omitted since it is clear from context.

we split VC 1 into two VCs as follows.

Definition 7.18 (VC 1.1)**.** *For any $z \in \mathcal{Z}^q_{pre}$, there is a $t \in [0, \ell_{\min}/v_{\max}]$ such that $f(z, \pi, t) \in \mathcal{Z}^q_{rec}$ and $F(z, \pi, t) \subseteq \mathcal{Z}_{\texttt{safe}}$.*

This VC says that the car safely reaches $\mathcal{Z}^q_{\text{rec}}$ from any state in $\mathcal{Z}^q_{\text{pre}}$ within time $\ell_{\min}/v_{\max}$ where $v_{\max}$ is the maximum speed of the car. The time bound guarantees that the car does not reach a state in $\mathcal{Z}_F$ before reaching $\mathcal{Z}^q_{\text{rec}}$. For the next VC, we define the progress region with respect to a state $z = (q, (x, y, \vartheta, v)) \in \mathcal{Z}^q_{\text{rec}}$ as $\mathcal{Z}^q_{>z} = \{(q, (x', y', \vartheta', v')) \mid (x', \vartheta', v') \in \tilde{\mathcal{X}}_{\text{rec}},\ y' \geq y + \varepsilon\}$ where $\varepsilon \in \mathbb{R}_{>0}$ is a lower bound on the increase in y-position[24].

Definition 7.19 (VC 1.2)**.** *For any $z \in \mathcal{Z}^q_{rec}$, there is a $t \in \mathbb{R}_{>0}$ such that $f(z, \pi, t) \in \mathcal{Z}^q_{>z}$ and $F(z, \pi, t) \subseteq \tilde{\mathcal{Z}}^q_{post}$ where $\tilde{\mathcal{Z}}^q_{post} = \{(q, (x, y, \vartheta, v)) \mid (x, \vartheta, v) \in \tilde{\mathcal{X}}_{post}\}$.*

This VC says that the car stays in $\tilde{\mathcal{Z}}^q_{\text{post}}$ while making progress in the y-direction. Since the observation in a straight segment is independent of the y position, it is enough to verify VC 1.2 for a fixed starting value of y. Furthermore, the progress criterion ensures that the car will safely reach the post-region of any straight segment of length $\ell \geq \ell_{\min}$ when started in its pre-region.

7.5.4. Verification Results for State-Feedback System

In this section, we provide verification results for the F1/10 system with a state-feedback controller. In this setting, it is sufficient to verify the 90-degree right, 120-degree right and the straight segments since the left turns are symmetric. This is not the case with the LiDAR-feedback system since the mode predictor is not symmetric.

Similar to the LiDAR-feedback system, pre-region is the same for all modes, given by $x \in [0.6, 0.9]$, $y \in [0, 0.24]$, $\vartheta \in [-0.005, 0.005]$ and $v \in [2.4, 2.4]$; the post-regions are similar. To reduce the overapproximation error introduced during verification, we split the initial set by increments of 0.05 along the x-dimension and 0.06 along the y-dimension, thus ending up with 24 verification instances per turn. For the initial straight segment, we split the initial set by increments of 0.01 along the

[24]Here y and y' denote y-distances from the start of the straight segment.

Mode	# instances	# branches	Verification time (seconds)	
			Composed system	NN only
90-degree right	24	1.50	258	40
120-degree right	24	1.75	285	38
straight initial	30	1.00	71	17
straight inductive	1	1.00	11	3

Table 7.2: Verification results for state-feedback system. Verification times and number of branches are averaged across all the instances for that mode. "Composed system" is the (average) time for fully verifying a single instance, and "NN only" is the time spent propagating reachable sets through the NNs during closed loop verification. All times are in seconds.

x-dimension and there is no uncertainty in the y-dimension.

Finally, in order to apply inductive reasoning in straights, we clip the y-distances to a maximum value of 5 (before feeding the state to the NN controller) which makes the observations independent of y in the straights. The verification results are summarized in Table 7.2.

7.6. Related Work

Verifying control & hybrid systems. There has been significant interest in verifying controllers for hybrid systems. Traditional techniques rely on inferring invariant such as Lyapunov functions [38, 143] or control barrier functions [128, 10]. More recent techniques have been proposed for checking safety and reachability properties in hybrid systems [36, 95]. Compositional reasoning principles for hybrid systems have also been developed [109, 9], but their focus is mainly on decomposing the problem of reasoning about concurrent composition of hybrid automata into reasoning about individual components. Finally, there has also been work on compositional control synthesis through control verification [143]; however, these techniques are designed for state-feedback systems.

Invariant synthesis. There has been work on automatically inferring program invariants from tests [48, 44]. Recent work has leveraged ideas similar to counterexample-guided inductive synthesis (CEGIS) [138], that alternate between synthesizing an invariant that satisfies the current counterexamples and using testing and verification to identify new counterexamples [136, 135, 122]. A particular challenge is handling implication examples [54], which connect different parts of the

invariant. We designed a novel pre/post-region synthesis algorithm based on these ideas.

CHAPTER 8

Future Work

This thesis shows that logical specifications and formal methods can be successfully used to improve the sample-efficiency and usability of reinforcement learning as well as the reliability and interpretability of policies trained using RL. This motivates the long-term research agenda of discovering ways to incorporate formal reasoning in machine learning tools and techniques to enable the building of reliable, interpretable, and intelligent systems. This can potentially lead to a future where AI systems leverage the benefits of both symbolic reasoning and machine learning while minimizing their drawbacks. A few concrete research directions that follow as natural next steps to the work presented in this thesis are discussed below.

8.1. Formal Reasoning in Reinforcement Learning

The hardness results in Chapter 3 imply that existing specification languages such as LTL are not well-suited for RL. One potential direction is to develop specification languages that are expressive and user-friendly while simultaneously admitting RL algorithms with strong theoretical guarantees. There is already some preliminary work on analyzing different ways of defining time-discounted semantics for LTL in this context [108, 14]. On the other hand, one can also look at assumptions on the underlying MDP that are often satisfied by realistic systems which make PAC learning possible. On the practical front, motivated by the promise shown by compositional approaches, one could explore algorithms for decomposing a given task into simpler subtasks that represent logical steps required to complete the whole task. This will involve considering a wide range of ways of specifying the overall objective and handling the lack of an exact model of the environment.

One benefit of formal specifications is that they have precise semantics and are more interpretable when compared to reward functions. In this thesis, we showed that we can achieve a logical decomposition of the policy into subtask policies when learning from Spectrl specifications. A potential direction for future work is to study the use of formal specifications in interpretable reinforcement learning–e.g., synthesizing programmatic policies to perform complex tasks.

There is growing interest among researchers in studying offline reinforcement learning algorithms which use a dataset of past interactions with the environment to reduce the number of additional samples needed to learn to perform new tasks. The use of temporal specifications in offline RL has not been explored and offers a lot of potential for future research. For example, one could study the problem of learning temporal specifications from expert demonstrations or the problem of identifying subtasks from the offline dataset to enable compositional learning.

Another interesting direction is to develop statistical verification algorithms to provide guarantees about neural network policies in the model-free setting; such approaches are more widely applicable to real-life scenarios. Such methods enable obtaining probabilistic guarantees without needing access to a model of the environment. A concrete idea is to explore whether the compositional verification framework from Chapter 7 can be adapted to this setting.

8.2. Specification-Guided Machine Learning

Traditional synthesis has always been associated with formal specifications such as LTL, Hoare triples, and input-output examples. Viewing ML as a method for program synthesis (where the programs are ML models), it is often possible to write (partial) formal specifications for the model being trained. For instance, requiring adversarial robustness of a vision model at some input is an example of such a specification. Existing work, including the work presented in this thesis, provides evidence that one can leverage such specifications to improve the learning algorithm.

Looking beyond RL, one broad direction is to develop learning algorithms that are guided by formal specifications in other domains. One such domain is large-language models for code generation where the user can often provide formal requirements for the generated code in the form of assert statements, in addition to a natural language prompt. This research direction can potentially lead to new domain-specific learning algorithms that require fewer data and generate models that are interpretable and verifiable.

BIBLIOGRAPHY

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*, pages 1–8, 2004.

[2] David Abel, Nate Umbanhowar, Khimya Khetarpal, Dilip Arumugam, Doina Precup, and Michael Littman. Value preserving state-action abstractions. In *International Conference on Artificial Intelligence and Statistics*, pages 1639–1650. PMLR, 2020.

[3] David Abel, Will Dabney, Anna Harutyunyan, Mark K Ho, Michael Littman, Doina Precup, and Satinder Singh. On the expressivity of markov reward. *Advances in Neural Information Processing Systems*, 34, 2021.

[4] Jinane Abounadi, Dimitrib Bertsekas, and Vivek S Borkar. Learning algorithms for Markov decision processes with average cost. *SIAM Journal on Control and Optimization*, 40(3): 681–698, 2001.

[5] Anayo K Akametalu, Jaime F Fisac, Jeremy H Gillula, Shahab Kaynama, Melanie N Zeilinger, and Claire J Tomlin. Reachability-based safe learning with gaussian processes. In *Conference on Decision and Control*, pages 1424–1431. IEEE, 2014.

[6] Natalia Akchurina. Multi-agent reinforcement learning algorithm with variable optimistic-pessimistic criterion. In *ECAI*, volume 178, pages 433–437, 2008.

[7] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[8] Derya Aksaray, Austin Jones, Zhaodan Kong, Mac Schwager, and Calin Belta. Q-learning for robust satisfaction of signal temporal logic specifications. In *Conference on Decision and Control (CDC)*, pages 6565–6570. IEEE, 2016.

[9] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR '97: Eighth International Conference on Concurrency Theory*, LNCS 1243, pages 74–88. Springer-Verlag, 1997.

[10] Rajeev Alur. Formal verification of hybrid systems. In *International Conference on Embedded Software*, pages 273–278, 2011.

[11] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229. Springer, Berlin, Heidelberg, 1992.

[12] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, Pei-Hsin Ho,

Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[13] Rajeev Alur, Suguman Bansal, Osbert Bastani, and Kishor Jothimurugan. A Framework for Transforming Specifications in Reinforcement Learning. *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, 2021.

[14] Rajeev Alur, Osbert Bastani, Kishor Jothimurugan, Mateo Perez, Fabio Somenzi, and Ashutosh Trivedi. Policy synthesis and reinforcement learning for discounted ltl. *In submission*, 2023.

[15] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

[16] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. Optimization and abstraction: A synergistic approach for analyzing neural network robustness. In *Programming Language Design and Implementation*, pages 731–744, 2019.

[17] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 39–48, 2016.

[18] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, pages 166–175, 2017.

[19] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

[20] Yu Bai and Chi Jin. Provable self-play algorithms for competitive reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

[21] Christel Baier, Luca de Alfaro, Vojtech Forejt, and Marta Kwiatkowska. Model checking probabilistic systems. In *Handbook of Model Checking*, pages 963–999. Springer, 2018.

[22] Anand Balakrishnan and Jyotirmoy V Deshmukh. Structured reward shaping using signal temporal logic specifications. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3481–3486. IEEE, 2019.

[23] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2016.

[24] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via

policy extraction. In *Advances in Neural Information Processing Systems*, 2018.

[25] Marc G Bellemare, Salvatore Candido, Pablo Samuel Castro, Jun Gong, Marlos C Machado, Subhodeep Moitra, Sameera S Ponda, and Ziyu Wang. Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature*, 588(7836):77–82, 2020.

[26] Felix Berkenkamp, Matteo Turchetta, Angela Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. In *Advances in Neural Information Processing Systems*, pages 908–918, 2017.

[27] Patricia Bouyer, Romain Brenguier, and Nicolas Markey. Nash equilibria for reachability objectives in multi-player timed games. In *International Conference on Concurrency Theory*, pages 192–206. Springer, 2010.

[28] Alper Kamil Bozkurt, Yu Wang, Michael M Zavlanos, and Miroslav Pajic. Control synthesis from linear temporal logic specifications using model-free reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10349–10355. IEEE, 2020.

[29] Ronen Brafman, Giuseppe De Giacomo, and Fabio Patrizi. Ltlf/ldlf non-markovian rewards. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[30] Ronen I. Brafman and Moshe Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3, 2003.

[31] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[32] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *International Joint Conference on Artificial Intelligence*, pages 6065–6073, 7 2019.

[33] Krishnendu Chatterjee. Two-player nonzero-sum ω-regular games. In *International Conference on Concurrency Theory*, pages 413–427. Springer, 2005.

[34] Krishnendu Chatterjee, Rupak Majumdar, and Marcin Jurdziński. On nash equilibria in stochastic games. In *International workshop on computer science logic*, pages 26–40. Springer, 2004.

[35] Yevgen Chebotar, Karol Hausman, Marvin Zhang, Gaurav Sukhatme, Stefan Schaal, and Sergey Levine. Combining model-based and model-free updates for trajectory-centric reinforcement learning. In *International conference on machine learning*, pages 703–711. PMLR, 2017.

[36] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.

[37] Artur Czumaj, Michail Fasoulakis, and Marcin Jurdzinski. Approximate nash equilibria with near optimal social welfare. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[38] Jamal Daafouz, Pierre Riedinger, and Claude Iung. Stability analysis and control synthesis for switched systems: a switched lyapunov function approach. *IEEE Transactions on Automatic Control*, 47(11):1883–1887, 2002.

[39] Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. Foundations for restraining bolts: Reinforcement learning with ltlf/ldlf restraining specifications. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 128–136, 2019.

[40] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51(1):5–30, 2017.

[41] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.

[42] S. Dutta, X. Chen, and S. Sankaranarayanan. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *HSCC*, pages 157–168. ACM, 2019.

[43] R. Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.

[44] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[45] Benjamin Eysenbach, Ruslan Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. In *NeurIPS*, 2019.

[46] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, September 2009. ISSN 0304-3975. doi: 10.1016/j.tcs.2009.06.021. URL http://dx.doi.org/10.1016/j.tcs.2009.06.021.

[47] Mahyar Fazlyab, Alexander Robey, Hamed Hassani, Manfred Morari, and George Pappas.

Efficient and accurate estimation of lipschitz constants for deep neural networks. In *Advances in Neural Information Processing Systems*, pages 11427–11438, 2019.

[48] Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for esc/java. In *International Symposium of Formal Methods Europe*, pages 500–517. Springer, 2001.

[49] Jie Fu and Ufuk Topcu. Probably approximately correct MDP learning and control with temporal logic constraints. In *Robotics: Science and Systems*, 2014.

[50] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596, 2018.

[51] Briti Gangopadhyay, Harshit Soora, and Pallab Dasgupta. Hierarchical program-triggered reinforcement learning agents for automated driving. *arXiv preprint arXiv:2103.13861*, 2021.

[52] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[53] Kunal Garg and Dimitra Panagou. Control-lyapunov and control-barrier functions based quadratic program for spatio-temporal specifications. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 1422–1429. IEEE, 2019.

[54] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, 2014.

[55] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *IEEE Symposium on Security and Privacy*, 2018.

[56] Claire Glanois, Paul Weng, Matthieu Zimmer, Dong Li, Tianpei Yang, Jianye Hao, and Wulong Liu. A survey on interpretable reinforcement learning. *arXiv preprint arXiv:2112.13112*, 2021.

[57] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[58] Amy Greenwald, Keith Hall, and Roberto Serrano. Correlated q-learning. In *ICML*, volume 3, pages 242–249, 2003.

[59] Shangding Gu, Long Yang, Yali Du, Guang Chen, Florian Walter, Jun Wang, Yaodong Yang, and Alois Knoll. A review of safe reinforcement learning: Methods, theory and applications. *arXiv preprint arXiv:2205.10330*, 2022.

[60] Iman Haghighi, Noushin Mehdipour, Ezio Bartocci, and Calin Belta. Control from signal

temporal logic specifications with smooth cumulative quantitative semantics. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 4361–4366. IEEE, 2019.

[61] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Omega-regular objectives in model-free reinforcement learning. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 395–412, 2019.

[62] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Faithful and effective reward schemes for model-free reinforcement learning of omega-regular objectives. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 108–124, Cham, 2020. Springer International Publishing. ISBN 978-3-030-59152-6.

[63] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Model-free reinforcement learning for stochastic parity games. In *31st International Conference on Concurrency Theory (CONCUR 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[64] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Model-free reinforcement learning for lexicographic omega-regular objectives. In *International Symposium on Formal Methods*, pages 142–159. Springer, 2021.

[65] Lewis Hammond, Alessandro Abate, Julian Gutierrez, and Michael Wooldridge. Multi-agent reinforcement learning with temporal logic specifications. In *International Conference on Autonomous Agents and MultiAgent Systems*, page 583–592, 2021.

[66] M. Hasanbeig, Y. Kantaros, A. Abate, D. Kroening, G. J. Pappas, and I. Lee. Reinforcement learning for temporal logic control synthesis with probabilistic satisfaction guarantees. In *Conference on Decision and Control (CDC)*, pages 5338–5343, 2019.

[67] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-constrained reinforcement learning. *arXiv preprint arXiv:1801.08099*, 2018.

[68] Elad Hazan and Robert Krauthgamer. How hard is it to approximate the best nash equilibrium? In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, page 720–727. Society for Industrial and Applied Mathematics, 2009.

[69] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, pages 4565–4573, 2016.

[70] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[71] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[72] Junling Hu and Michael P Wellman. Nash q-learning for general-sum stochastic games. *Journal of machine learning research*, 4(Nov):1039–1069, 2003.

[73] Junling Hu, Michael P Wellman, et al. Multiagent reinforcement learning: theoretical framework and an algorithm. In *ICML*, volume 98, pages 242–250. Citeseer, 1998.

[74] Chao Huang, Jiameng Fan, Wenchao Li, Xin Chen, and Qi Zhu. Reachnn: Reachability analysis of neural-network controlled systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–22, 2019.

[75] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.

[76] Rodrigo Toro Icarte, Toryn Klassen, Richard Valenzano, and Sheila McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, pages 2107–2116. PMLR, 2018.

[77] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

[78] León Illanes, Xi Yan, Rodrigo Toro Icarte, and Sheila A. McIlraith. Symbolic plans as high-level instructions for reinforcement learning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1):540–550, Jun. 2020.

[79] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *International Conference on Hybrid Systems: Computation and Control*, 2019.

[80] Radoslav Ivanov, Taylor J Carpenter, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. Case study: verifying the safety of an autonomous racing car with a neural network controller. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–7, 2020.

[81] Radoslav Ivanov, Kishor Jothimurugan, Steve Hsu, Shaan Vaidya, Rajeev Alur, and Osbert Bastani. Compositional Learning and Verification of Neural Network Controllers. *ACM Transactions on Embedded Computing Systems*, 2021.

[82] Yuqian Jiang, Sudarshanan Bharadwaj, Bo Wu, Rishi Shah, Ufuk Topcu, and Peter Stone. Temporal-logic-based reward shaping for continuing learning tasks, 2020.

[83] Chi Jin, Akshay Krishnamurthy, Max Simchowitz, and Tiancheng Yu. Reward-free exploration for reinforcement learning. In *International Conference on Machine Learning*, pages 4870–4879. PMLR, 2020.

[84] Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. A Composable Specification Language for Reinforcement Learning Tasks. In *Advances in Neural Information Processing Systems*, 2019.

[85] Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. Compositional Reinforcement Learning from Logical Specifications. In *Advances in Neural Information Processing Systems*, 2021.

[86] Kishor Jothimurugan, Osbert Bastani, and Rajeev Alur. Abstract value iteration for hierarchical reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 1162–1170. PMLR, 2021.

[87] Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. Specification-Guided Learning of Nash Equilibria with High Social Welfare. In *Computer Aided Verification*, 2022.

[88] Kishor Jothimurugan, Steve Hsu, Osbert Bastani, and Rajeev Alur. Robust subtask learning for compositional generalization. *arXiv preprint arXiv:2302.02984*, 2023.

[89] Sham Machandranath Kakade. *On the sample complexity of reinforcement learning.* University of London, University College London (United Kingdom), 2003.

[90] Parv Kapoor, Anand Balakrishnan, and Jyotirmoy V Deshmukh. Model-based reinforcement learning from signal temporal logic specifications. *arXiv preprint arXiv:2011.04950*, 2020.

[91] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[92] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine learning*, 49(2):209–232, 2002.

[93] Michael Kearns, Yishay Mansour, and Satinder Singh. Fast planning in stochastic games. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 309–316, 2000.

[94] Arbaaz Khan, Ekaterina Tolstaya, Alejandro Ribeiro, and Vijay Kumar. Graph policy gradients for large scale robot control. In *Conference on Robot Learning*, 2020.

[95] Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. dreach: δ-reachability analysis for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 200–205. Springer, 2015.

[96] Yen-Ling Kuo, B. Katz, and A. Barbu. Encoding formulas as deep networks: Reinforcement learning for zero-shot execution of ltl formulas. *IEEE/RSJ International Conference on*

Intelligent Robots and Systems (IROS), pages 5604–5610, 2020.

[97] Marta Kwiatkowska, Gethin Norman, David Parker, and Gabriel Santos. Equilibria-based probabilistic model checking for concurrent stochastic games. In *International Symposium on Formal Methods*, pages 298–315. Springer, 2019.

[98] Marta Kwiatkowska, Gethin Norman, David Parker, and Gabriel Santos. Prism-games 3.0: Stochastic game verification with concurrency, equilibria and time. In *International Conference on Computer Aided Verification*, pages 475–487. Springer, 2020.

[99] Morteza Lahijanian, Joseph Wasniewski, Sean B Andersson, and Calin Belta. Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In *2010 IEEE International Conference on Robotics and Automation*, pages 3227–3232. IEEE, 2010.

[100] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[101] Shuo Li and Osbert Bastani. Robust model predictive shielding for safe reinforcement learning with stochastic dynamics. In *International Conference on Robotics and Automation*, pages 7166–7172. IEEE, 2020.

[102] Xiao Li, Cristian-Ioan Vasile, and Calin Belta. Reinforcement learning with temporal logic rewards. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3834–3839. IEEE, 2017.

[103] Xiao Li, Yao Ma, and Calin Belta. A policy search method for temporal logic specified reinforcement learning tasks. In *2018 Annual American Control Conference (ACC)*, pages 240–245. IEEE, 2018.

[104] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[105] Lars Lindemann and Dimos V Dimarogonas. Control barrier functions for signal temporal logic tasks. *IEEE control systems letters*, 3(1):96–101, 2018.

[106] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

[107] Michael L Littman. Friend-or-foe q-learning in general-sum games. In *ICML*, volume 1, pages 322–328, 2001.

[108] Michael L Littman, Ufuk Topcu, Jie Fu, Charles Isbell, Min Wen, and James MacGlashan. Environment-independent task specifications via GLTL. *arXiv preprint arXiv:1704.04341*, 2017.

[109] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg. Hybrid I/O automata. In *Hybrid Systems III: Verification and Control*, LNCS 1066, pages 496–510, 1996.

[110] Oded Maler and Dejan Nivcković. Monitoring properties of analog and mixed-signal circuits. *International Journal on Software Tools for Technology Transfer*, 15(3):247–268, 2013.

[111] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1800–1809, 2018.

[112] Richard D. McKelvey, Andrew M. McLennan, and Theodore L. Turocy. Gambit: Software tools for game theory, 2014. URL http://www.gambit-project.org.

[113] Salar Moarref and Hadas Kress-Gazit. Automated synthesis of decentralized controllers for robot swarms from high-level temporal logic specifications. *Autonomous Robots*, 44:585–600, 2020.

[114] Teodor Mihai Moldovan and Pieter Abbeel. Safe exploration in markov decision processes. In *International Conference on Machine Learning*, 2012.

[115] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3303–3313, 2018.

[116] Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Near-optimal representation learning for hierarchical reinforcement learning. In *ICLR*, 2019.

[117] Nikhil Naik and Pierluigi Nuzzo. Robustness contracts for scalable verification of neural network-enabled cyber-physical systems. In *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–12. IEEE, 2020.

[118] Cyrus Neary, Zhe Xu, Bo Wu, and Ufuk Topcu. Reward machines for cooperative multi-agent reinforcement learning, 2021.

[119] Andrew Y. Ng and Stuart J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 663–670, 2000.

[120] Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems*, pages 149–178. Springer, Berlin, Heidelberg, 1992.

[121] Matthew O'Kelly, Hongrui Zheng, Dhruv Karthik, and Rahul Mangharam. F1tenth: An open-source evaluation environment for continuous control and reinforcement learning. In *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, volume 123 of *Proceedings of Machine Learning Research*, pages 77–89. PMLR, 08–14 Dec 2020.

[122] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *PLDI*, 2016.

[123] Yash Vardhan Pant, Houssam Abbas, Rhudii A Quaye, and Rahul Mangharam. Fly-by-logic: Control of multi-drone fleets with temporal logic objectives. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 186–197. IEEE, 2018.

[124] Corina S Păsăreanu, Divya Gopinath, and Huafeng Yu. Compositional verification for autonomous systems with deep learning components. In *Safe, Autonomous and Intelligent Vehicles*, pages 187–197. Springer, 2019.

[125] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 54(5):1–35, 2021.

[126] Julien Perolat, Florian Strub, Bilal Piot, and Olivier Pietquin. Learning Nash Equilibrium for General-Sum Markov Games from Batch Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, 2017.

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

[128] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *International Workshop on Hybrid Systems: Computation and Control*, pages 477–492. Springer, 2004.

[129] HL Prasad, Prashanth LA, and Shalabh Bhatnagar. Two-timescale algorithms for learning nash equilibria in general-sum stochastic games. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1371–1379, 2015.

[130] Aditi Raghunathan, Jacob Steinhardt, and Percy S Liang. Semidefinite relaxations for certifying robustness to adversarial examples. *Advances in neural information processing systems*, 31, 2018.

[131] Rajesh Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[132] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[133] Dorsa Sadigh and Ashish Kapoor. Safe control under uncertainty with probabilistic signal temporal logic. In *Proceedings of Robotics: Science and Systems XII*, 2016.

[134] Lloyd S Shapley. Stochastic games. *Proceedings of the national academy of sciences*, 39(10): 1095–1100, 1953.

[135] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, 48(3):235–256, 2016.

[136] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. Verification as learning geometric concepts. In *International Static Analysis Symposium*, pages 388–411. Springer, 2013.

[137] Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Křetínskỳ. Limit-deterministic büchi automata for linear temporal logic. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 312–332. Springer, 2016.

[138] Armando Solar-Lezama and Rastislav Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[139] Alexander L Strehl, Lihong Li, Eric Wiewiora, John Langford, and Michael L Littman. PAC model-free reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 881–888, 2006.

[140] Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. Program guided agent. In *International Conference on Learning Representations*, 2020.

[141] X. Sun, H. Khedr, and Y. Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 147–156. ACM, 2019.

[142] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.

[143] Russ Tedrake, Ian R Manchester, Mark Tobenkin, and John W Roberts. Lqr-trees: Feedback motion planning via sums-of-squares verification. *The International Journal of Robotics Research*, 29(8):1038–1052, 2010.

[144] H. Tran, F. Cai, D. M. Lopez, P. Musau, T. T. Johnson, and X. Koutsoukos. Safety verification of cyber-physical systems with reinforcement learning control. *ACM Transactions on Embedded Computing Systems*, 18(5s):105, 2019.

[145] Pashootan Vaezipoor, Andrew C Li, Rodrigo A Toro Icarte, and Sheila A Mcilraith. Ltl2action: Generalizing ltl instructions for multi-task rl. In *International Conference on Machine Learning*, pages 10497–10508. PMLR, 2021.

[146] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[147] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaud-

huri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[148] Abhinav Verma, Hoang M Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Advances in Neural Information Processing Systems*, 2019.

[149] Kim P Wabersich and Melanie N Zeilinger. Linear model predictive safety certification for learning-based control. In *Conference on Decision and Control (CDC)*, pages 7130–7135. IEEE, 2018.

[150] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*, pages 6367–6377, 2018.

[151] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.

[152] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[153] Chen-Yu Wei, Yi-Te Hong, and Chi-Jen Lu. Online reinforcement learning in stochastic games. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 4994–5004, 2017.

[154] Min Wen and Ufuk Topcu. Constrained cross-entropy method for safe reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 7450–7460, 2018.

[155] T. Weng, H. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. Boning, and I. Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, pages 5273–5282, 2018.

[156] Zhe Xu and Ufuk Topcu. Transfer of temporal logic formulas in reinforcement learning. In *International Joint Conference on Artificial Intelligence*, pages 4010–4018, 7 2019.

[157] Cambridge Yang, Michael Littman, and Michael Carbin. Reinforcement learning for general ltl objectives is intractable. *arXiv preprint arXiv:2111.12679*, 2021.

[158] Lim Zun Yuan, Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Modular deep reinforcement learning with temporal logic specifications. *arXiv preprint arXiv:1909.11591*, 2019.

[159] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis

framework for verifiable reinforcement learning. In *Programming Language Design and Implementation*, pages 686–701, 2019.

[160] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1433–1438, 2008.

[161] Martin Zinkevich, Amy Greenwald, and Michael Littman. Cyclic equilibria in markov games. *Advances in Neural Information Processing Systems*, 18:1641, 2006.