Learning Probabilistic and Real-Time Task Specifications from Demonstrations for Strategy Synthesis

by

Kandai Watanabe

B.A., Keio University, 2017

M.S., Keio University, 2019

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2024

Committee Members:

Morteza Lahijanian, Chair

Dr. Sriram Sankaranarayanan

Dr. Ashutosh Trivedi

Dr. Christoffer Heckman

Dr. Georgios Fainekos

Dr. Fabio Somenzi

Watanabe, Kandai (Ph.D., Computer Science)

Learning Probabilistic and Real-Time Task Specifications from Demonstrations for Strategy Synthesis

Thesis directed by Dr. Morteza Lahijanian

Recent technological advancements have enabled robots to become highly autonomous in various domains, including deep-sea and space exploration, human-shared assembly lines, and surgical robotics. In these domains, robots must perform complex tasks that demand specific requirements that constrain the task orders and timings. They are often unavailable as in a formal specification or too cumbersome for non-experts to specify. To address this challenge, this thesis introduces the concept of "Specification Learning from Demonstrations" as a key step towards the ultimate goal of Learning from Demonstrations (LfD). This new LfD concept aims to equip robots with the ability to learn task specifications in the form of formal languages from human demonstrations or data collected from past operations, facilitating autonomous task execution across diverse environments.

To achieve this, the thesis proposes a methodology for learning task specifications as state machine representations. This approach has multiple advantages over learning logical formulas. We demonstrate how automata-based learning can easily incorporate safety requirements, enhance interpretability, and improve efficiency in planning. Building on these strengths, the research further refines the state machine concept by introducing two models: Probabilistic Deterministic Finite Automata (PDFA) and Timed Partial Orders (TPO). They can capture a wide range of tasks, ensuring domain independence and the ability to represent non-Markovian tasks. The thesis also presents algorithmic efficiency by employing heuristic methods and mixed integer linear programming for learning and planning. Importantly, these algorithms are theoretically sound and guaranteed to provide correct and optimal solutions. The thesis further discusses case studies, including complex robotic scenarios (e.g., aircraft turnaround, Overcooked! game, and manipulators) to demonstrate the practicality and robustness of the proposed approach.

Ultimately, this thesis marks a significant step forward in enabling autonomous robots to adapt to varying task environments, enhancing their usability, and expanding their potential applications across different domains.

Dedication

I would like to thank my parents, Yoshihisa Watanabe and Eiko Watanabe, for supporting me at all times and giving me opportunities. I could not have achieved this far without their support. Thank you to my sisters, Shoko and Yoko, for always being a good conversational partner. Lastly, I thank Yurie Aiura for offering encouragement, overcoming the problems together, and supporting me over the five years of my PhD.

Acknowledgements

First and foremost, I would like to thank my advisors, Morteza Lahjianian and Sriram Sankaranarayanan, for their continuous support and invaluable guidance throughout my PhD. I was truly inspired by your knowledge, dedication to science and passion in research. Thank you to my committee members Georgios Fainkeos, Ashutosh Trivedi, Christoffer Heckman, and Fabio Somenzi for your support and guidance through this process. I am grateful for my previous advisor, Alessandro Roncone, for fruitful conversations. I also thank my advisor in my Master's, Masaki Takahashi, for teaching me the joy of conducting research and for inspiring me to pursue a career in research.

I would also like to thank everyone in my labs (AUT, CUPLV) for fruitful discussions and for providing insightful feedback. In particular, I'd like to thank Qi Heng, Rayan Mazouz, Karan Muvvala, Zakariya Laouar, Anne Theurkauf, Hunter Ray, Himanshu Gupta, Akash Ratheesh, Ibon Garcia, Peter Amorese, Ben Kraske, and Monal Narasimhamurthy. I enjoyed sharing time with you all, making lattes, eating meals together, skiing and talking about life. I truly appreciate the mountains in Colorado. Colorado has treated me tremendously well. We enjoyed skiing, hiking, and camping in the mountains.

I am grateful to have had the opportunity to do an internship at Amazon and Toyota Motors North America under Georgios Fainekos (thank you for taking part in my committee) and Bardh Hoxha. I was fortunate enough to have the freedom to work on my research which became a chapter of this thesis, and for their continuous support, guidance, and thoughtful conversations.

Contents

Figures

Figure

Notations

PDFA Learning

Π	Atomic Propositions $\Pi = \{p_1, ..., p_k\}$
Σ	Alphabet $\Sigma = 2^{\Pi}$
ω	Trace $\omega = \omega_1\omega_2 \dots \omega_n$, where $\omega_i \in \Sigma$ for all $1 \leq i \leq n$
Ω	Traces $\Omega = [\omega_i]_{i=1}^{n_\Omega}$
$\mathcal{A}$	Automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$
z	Run of a DFA $\mathcal{A}$, $z = z_0 z_1 \dots z_n$, where $z_0 = q_0$ and $z_i = \delta(z_{i-1}, \omega_i)$ for $i = 1, \dots, n$
$\mathcal{L}(\mathcal{A})$	Language of $\mathcal{A}$
φ_{task}	Task Specification
φ_{safe}	Safety Specification
$\mathcal{X}$	Next Operator
$\mathcal{U}$	Until Operator
$\mathcal{F}$	Eventually Operator
$\mathcal{G}$	globally operator
$\mathcal{A}_{\text{safe}}$	Safe Automaton Tuple $\mathcal{A}_{\text{safe}} = (Q^s, \Sigma, q_0^s, \delta^s, F^s)$
$\mathbb{P}$	Probabilistic DFA $\mathcal{A}^{\mathbb{P}} = (\mathcal{A}, \delta_{\mathbb{P}}, F_{\mathbb{P}})$
$P(\omega)$	Probability of a trace ω, $P(\omega) = \prod_{i=1}^{n} \delta_{\mathbb{P}}(z_{i-1}, \omega_i, z_i) \cdot F_{\mathbb{P}}(z_n)$
$\mathcal{T}$	DTS Tuple $\mathcal{T} = (X, A, x_0, \delta_{\mathcal{T}}, \Pi, L)$
γ	plan $\gamma = \gamma_0\gamma_1 \dots \gamma_{n-1}$ is a sequence of actions, where $\gamma_i \in A$ for all $0 \leq i \leq n-1$
Γ	Set of plans

s Trajectory $s = s_0 s_1 \dots s_n$, where $s_0 = x_0$ and $s_{i+1} = \delta_{\mathcal{T}}(s_i, \gamma_i)$

ρ_γ Observation $\rho_\gamma = L(s_0)L(s_1)\dots L(s_n)$

$\mathcal{L}(\mathcal{T})$ Language of $\mathcal{T}$

$\mathcal{P}$ Product Automaton $\mathcal{P} = (Q^{\mathcal{P}}, q_0^{\mathcal{P}}, E^{\mathcal{P}}, W^{\mathcal{P}})$

λ Path over $\mathcal{P}$, $\lambda = (q_0, \bar{x}_0)(q_1, x_0)\dots(q_n, x_{n-1})q_t^{\mathcal{P}}$ be a path over $\mathcal{P}$

Λ Set of paths of $\mathcal{P}$

Multi-Objective Reactive Synthesis

G Two-Player Game $G = (S, A, s_0^G, \delta^G, \Pi, L^G, W^G)$ where $S = S_R \cup S_E$ and $A = A_R \cup A_E$

$\cdot_R, \cdot_E$ Subscripts for the System and Environment players

$\mathcal{S}$ A Play $\mathcal{S} = s_0 s_1 \dots s_n$

$\mathcal{S}(i)$ The prefix of play $\mathcal{S}$ at position i

Play(G) Set of all plays in G

$\text{Pref}_{i\in\{R,E\}}(\mathcal{S}(k))$ Set of prefixes that belongs to the robot R or environment E

$\sigma_{i\in\{R,E\}}$ Deterministic Strategy $\sigma_i : S^* S_i \to A_i$ that chooses the next action given a (finite) play that ends in a state in S_i.

TP$(\mathcal{S})$ Total Payoff of a play $\text{TP}(\mathcal{S}) = \sum_{k=0}^{n-1} W^G(s_k, s_{k+1})$.

$\mathcal{P}^G$ Game Product Automaton $\mathcal{P}^G = (S^{\mathcal{P}}, A, s_0^{\mathcal{P}}, s_t^{\mathcal{P}}, E_G^{\mathcal{P}}, W_G^{\mathcal{P}})$

F Greatest Fixed Operator

$U(s^{\mathcal{P}})$ Set of total payoffs $U(s^{\mathcal{P}})$

$\mathcal{T}$ Tree

p Pareto Point

$\mathcal{P}$ Pareto Point Set

TPO Mining

e Event e

$\mathcal{E}$ Events $\mathcal{E} = \{e_1, \dots, e_n\}$

τ Timed Trace $\tau : \mathcal{E} \to \mathbb{R}_{\geq 0}$, i.e., $\tau = \{e_1 \mapsto t_1, e_2 \mapsto t_2, \dots, e_n \mapsto t_n\}$ wherein $t_i \geq 0$ denotes the timestamp for event $e_i \in \mathcal{E}$

$\prec$ Partial Order

S Timed Partial Order Graph $S : (\mathcal{E}, \prec)$

C Set of clocks $C = \{c_1, \ldots, c_m\}$

g A guard map that maps each event e_i to a guard condition, which is a conjunction of the form $g(e_i) : \bigwedge_{j=1}^{n_i} c_j \bowtie a_j$, wherein $c_j \in C$ denotes a clock, $\bowtie \in \{\leq, \geq\}$, and $a_j \in \mathbb{R}_{\geq 0}$ is a non-negative constant

R A reset map $R : \mathcal{E} \to 2^C$ that associates each event e_i with a subset of clocks $R(e_i) \subseteq C$ that are to be reset to 0 whenever event e_i is encountered

ν Valuation of the clock $C \to \mathbb{R}$

$l_{i,j}, u_{i,j}$ Lower Bound and Upper Bound time differences $t_j - t_i \in [l_{i,j}, u_{i,j}]$

a_i, b_i Lower Bound and Upper Bound on timestamp t_i

φ^{TPO} Time Constraints $\varphi^{TPO} : \bigwedge_{i,j} \bigwedge_k (t_j - t_i) \bowtie l_{i,j,k}$, wherein $\bowtie \in \{\leq, \geq\}$

G_c Clock Allocation Graph $G_c = (C, E_i, L_c)$

$\Delta(G_c)$ The maximum number of neighbors for any vertex in G_c

Optimal Planning for TPO

$\mathcal{T}^S$ A single-robot deterministic transition system (DTS) $\mathcal{T}^S = (X, A, x_0, \delta_\mathcal{T}, \Delta_\mathcal{T}, \mathcal{E}, L^S)$

$\Delta_\mathcal{T}$ Transition duration function $\Delta_\mathcal{T} : X \times A \to \mathbb{R}$

L^S Labeling function that maps each state to an event or an empty set $L^S : X \to \mathcal{E} \cup \{\emptyset\}$

s^γ Trajectory generated by a plan $s^\gamma = s_0^\gamma s_1^\gamma \ldots s_n^\gamma$

ρ^γ Observation Trace of the planned trajectory $\rho^\gamma = L(s_0^\gamma)L(s_1^\gamma)\ldots L(s_n^\gamma)$

$D(s^\gamma)$ Duration of the planned trajectory $D(s^\gamma) = \sum_{i=0}^{n-1} \Delta_T(s_i^\gamma, \gamma_i)$

τ^γ Timed Trace induced by a plan $\tau^\gamma = (L(s_0^\gamma), t_1)\ldots(L(s_n^\gamma), t_n)$

n_R Number of robots

X_I Initial States of the robot

$\mathcal{T}^M$ A multi-robot deterministic transition system (DTS) $\mathcal{T}^M = (X, A, X_I, \delta_\mathcal{T}, \Delta_\mathcal{T}, \mathcal{E}, L^S)$

G^{TSP} TSP Graph $G^{TSP} = (V^{TSP}, E^{TSP})$

V^{TSP} Set of vertices in TSP with verex cost d_i

E^{TSP}	Set of edges in TSP with edge cost d_{ij}
${V^{TSP}}_i$	The disjoint subsets
$y_{i,j}$	Integer variable indicating the active edge $i \rightarrow j$
I	Initial state indices $I = 0_1, ..., 0_{n_R}$
$\delta(\cdot)$	All possible timing variations of d_i or $d_{i,j}$

Chapter 1

Introduction

1.1 Motivation

Recent developments in technology (e.g., computational power, sensors, big data, machine learning, and cloud computing) have enabled robots to progressively become autonomous. As a result, they operate in environments shared with humans such as deep-sea and space exploration, assistive and service tasks (e.g., human-shared assembly lines), and surgical procedures (Figure 1.1). Robots can perform simple actions with high speed, precision, and robustness, such as walking over bumpy terrain from point A to B while grasping objects. As they gain more access to various domains, robots are increasingly expected to perform more complex tasks.

For example, consider an underwater robot in a deep-sea science mission as schematically illustrated in Figure 1.2a. The scientists want the robot to explore a shipwreck on the sea-floor,

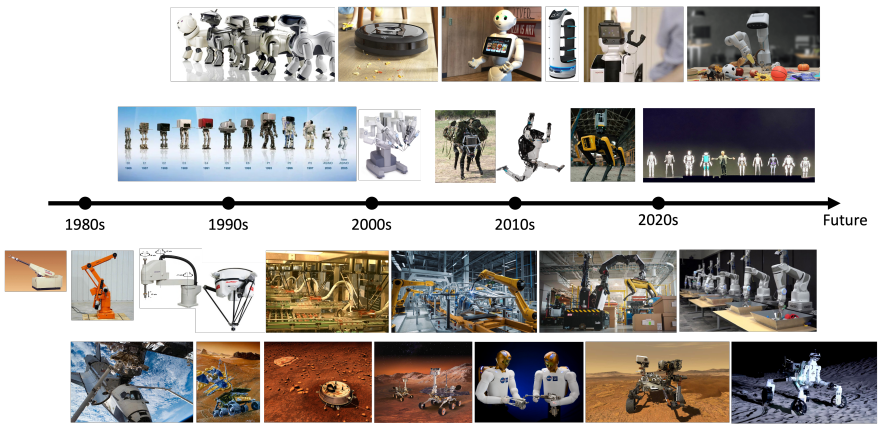
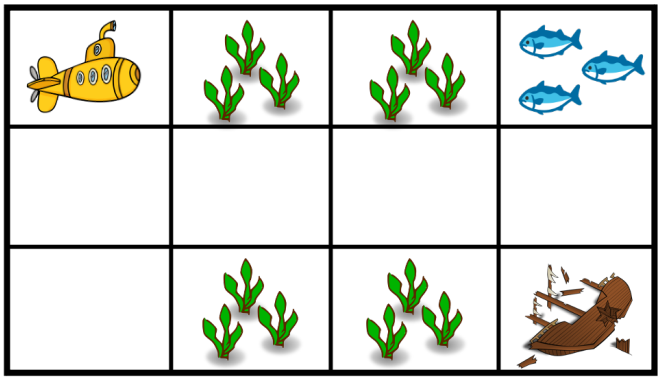


Figure 1.1: Recent Developments in Robotics

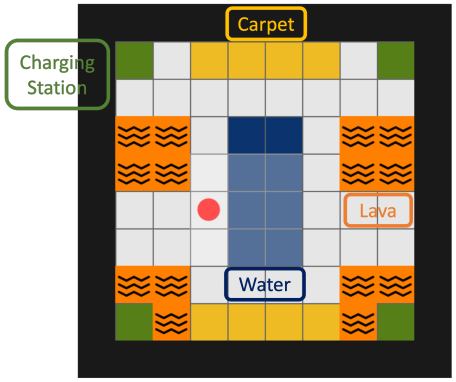
study the behavior of a school of fish, while keeping away from the dangerous coral reefs. Consider another example of indoor robots (e.g., quadruped walking robots) navigating to a charging station as shown in Figure 1.2b. They have to reach one of the charging stations nearby. However, if they get wet, they must be dried out before reaching the charging station.

To be able to execute these tasks, the robots must have knowledge of the *specifications* (requirements). The conventional approach is to implement the specification as a hard-coded program (e.g., if-else), use an optimization-based controller by specifying objective/cost/reward function, or specify the task as in a special high-level language (e.g., PDDL[39]). However, programming the task can be too cumbersome, induce mistakes, and require expert knowledge. Designing objective functions can be unintuitive and involve gain parameters that are hard to tune. Finally, despite the clarity of high-level languages, they have a high learning curve and require expert knowledge. In this thesis, we assume that tasks can be easily demonstrated through various means – demonstrations through human operation or data collected from past manual operations (e.g., event logs). Given such demonstrations, robots need to infer the task specification and execute the task autonomously. We call this problem *Specification Learning from Demonstrations*, which can be regarded as a new form of *Learning from Demonstrations (LfD)*[73].

Learning specification has a few advantages over common LfD approaches such as learning policy or reward functions. It is domain-independent, meaning that the model is not trained in a specific environment, but can be transferred to other environments. Thus LfD approaches are robust to changes in the environment. More importantly, they handle non-Markovian tasks. For example, the robot task in Figure 1.2a, which requires a visit to both the shipwreck and school of fish in any order, is non-Markovian. To achieve it, the robot has to maintain a memory of previous locations to decide the next location to visit. Our approach to LfD involves learning a specification from demonstrations. The learned specification is interpretable unlike other representations and can easily be deleted or composed with additional requirements. Finally, once the specification is learned, e.g., in the form of Linear Temporal Logic (LTL) [8], there are many existing tools and



(a) Schematic of an autonomous deep-sea science mission.



(b) Schematic of autonomous vehicle reaching a charging station

Figure 1.2: Examples of Complex robotic tasks

techniques for developing algorithms for reachability analysis, model-checking, monitoring, and strategy synthesis [8]. Given a new environment domain, these tools can synthesize a strategy that satisfies the learned specifications for static, reactive, and uncertain environments [12].

There are three main challenges in learning specifications from demonstrations: correctness, explainability, and efficiency. Correct specifications must be learned that respect the demonstrations. It should not learn anything that violates the behavior of the system. Safety constraints given before the learning process must be satisfied by the learned strategies. Furthermore, explainability is needed for users to understand the intention of the robot and to validate the correctness of the learned task. Lastly, efficiency is key to deploying robots in real-world applications. The users need to know computation times in order to make a plan. To overcome these problems, we learn specifications in the form of a state machine.

Related works [92, 91, 100, 50, 80] have investigated learning an LTL formula directly from data, however, they either require a set of predefined formula templates or infer over all combinations of sub-formulas. Another important issue with formula learning methods for the purpose of verification and planning is that they typically need to be translated into an automaton. This translation is exponential in the number of sub-formulas (operators) both in space and time and is known as a **state-explosion** problem [8, 53]. Automata learning does not require such prior

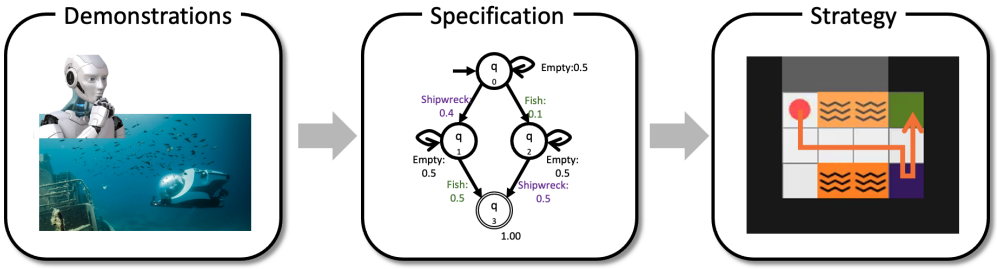


Figure 1.3: The abstract framework of the proposing works

knowledge or translation, leading to its efficiency while keeping the correctness guarantees and its explainability. Note that the complexity of the original problem itself has not changed. Rather, automata learning handles this exponentiality during the learning process, which makes the approach truly efficient.

1.2 Thesis Statement

> **Thesis Statement**: In Learning from Demonstrations, inferring formal specifications facilitates the learning of complex (non-Markovian) tasks and enhances the explainability of the outcome. Among the specification representations, learning automata (1) allows for safe inference, (2) improves efficiency in learning and planning, and (3) extends to multi-robot planning.

This thesis proposes to learn task specifications from data (e.g., human demonstrations and log data) and synthesize plans for the robots to complete the task in various environments. The overall framework is depicted in Figure 1.3. This thesis proposes two explainable and easily-interpretable specification models for learning tasks with their learning and planning algorithms. The algorithms are applied to a variety of realistic applications and demonstrated on real-world robots.

1.3 Contributions

The main contributions of this thesis are divided into three categories: modeling, algorithmic efficiency, and performance guarantees. We consider two specification models: Probabilistic Deterministic Finite Automata (PDFA) and Timed Partial Order (TPO).

1.3.0.1 Modeling

This thesis proposes a method to learn domain-independent specification models in the form of PDFAs and TPOs. This allows us to reuse the learned specification model even if the environment changes. We show that PDFAs can explain many types of robotic tasks, and are a better representation than policy or reward functions.

The PDFAs can sometimes be too expressive and hence too expensive for our purposes, specifically for manufacturing processes. They do not capture timing constraints in a succinct manner. Thus, in this thesis, we also propose a specification model, called Timed Partial Order, that is especially suitable for modeling manufacturing processes (e.g., factories), also known as workflows. These workflows are conventionally structured around tasks that have to be completed subject to precedence and timing constraints. We demonstrate that the complex tasks can be modeled (and learned) as TPOs through two examples inspired by real-life applications: a model of events involved in the workflow for commercial aircraft turnaround and an analysis of the multiplayer computer game Overcooked!, as played by beginner and expert players.

1.3.0.2 Algorithmic Efficiency

In this thesis, we present efficient algorithms and analyze their complexity. For learning PDFAs, we employ an off-the-shelf heuristic algorithm called the Evidence-Driven State-Merging (EDSM) algorithm [31]. The algorithm can estimate a generalized task in the form of PDFA from a finite set of demonstrations and exhibits the time complexity that is cubic in the number of predicates (event label) but is reported to run in linear time in practice. For learning TPOs, we

proposed an algorithm that iteratively solves Linear Programming (LP) problems, that runs in polynomial time in the number of event labels. This helps the user to easily feed data and obtain an estimate of the task specifications. Furthermore, we introduced efficient planning algorithms for PDFAs to find valid plans and strategies even under environmental changes. We formulate the planning problem for TPOs as a Traveling Salesman Problem (TSP) [28] and solve the problem optimally by formulating it as a Mixed Integer Linear Program (MILP). MILP runs in exponential time relative in the number of event labels. TSP is a well-explored problem and there are many existing heuristics to solve the problem suboptimally, which we can leverage in the future.

1.3.0.3 Guarantees

Another contribution is that the proposed methods provide guarantees for the solutions we obtained. Many *efficient* algorithms rely on heuristics that lack guarantees for finding correct/optimal solutions. In this thesis, we only propose or employ theoretically sound algorithms (EDSM algorithms, graph search, LP, and TSP) to learn the specification models and plan for these specification models. Such guarantees are important in risk-aware environments, such as surgery and rescues. As an example, we demonstrate that our algorithms can solve complex problems such as planning for multiple robots (scaling to 40 agents) with up to 80 different task locations [96].

1.4 Structure

The remainder of the thesis is structured as follows.

- Preliminaries: Chapter 2
- Part I. Probabilistic Specifications
  (1) Mining: Chapter 3
  (2) Strategy Synthesis: Part of Chapter 3 and Chapter 4
- Part II. Real-Time Specifications

(1) Mining: Chapter 5

(2) Strategy Synthesis: Chapter 6

- Conclusion Chapter 7

Chapter 2 provides the necessary background on the fundamentals of the Formal Language Theory and definitions of transition systems. Chapters 3 to 6 are the core of this thesis.

Chapter 2

Preliminaries

This section provides background on the fundamentals of formal language theory, and the modeling of deterministic discrete transition systems.

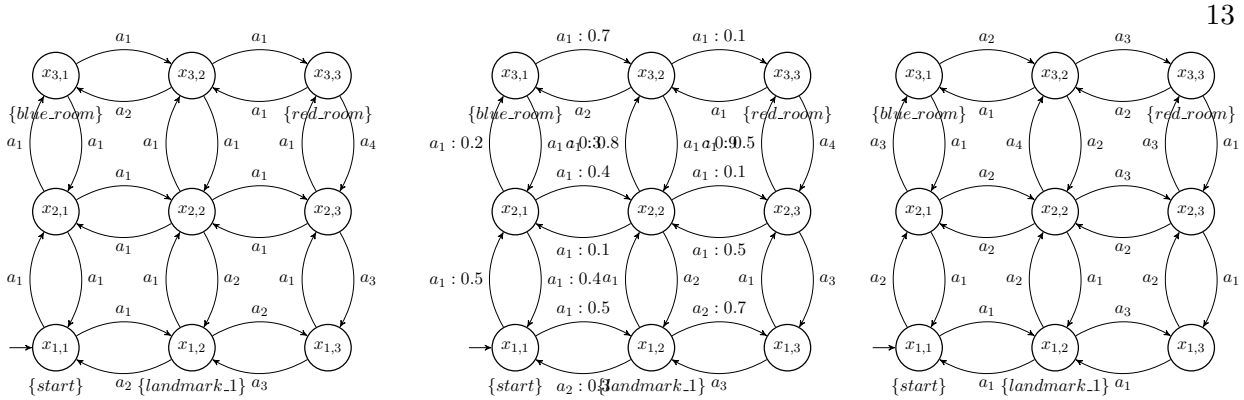
2.1 Transition System

Transition systems are often used in various fields of computer science (e.g., computing systems, control, robotics) and as models to describe the behavior of systems. Many of the robotic systems can also be described as transition systems. In this chapter, we start by introducing the general definition of the transition systems and subsequently define *Deterministic* Transition Systems (DTS) that will be used throughout this thesis.

Definition 1 (Transition System)**.** *A Transition System (TS) is a tuple* $(X, A, X_0, \delta_{\mathcal{T}}, \Pi, L)$*, where*

- X *is a set of states,*
- A *is a set of controls or actions,*
- $X_0 \subseteq X$ *is a set of initial states,*
- $\delta_{\mathcal{T}} : X \times A \to 2^X$ *is the transition function,*
- Π *is a set of atomic propositions, and*
- $L : X \to 2^{\Pi}$ *is a labeling function that maps each state to the set of atomic propositions that are true at that state.*

TS is said to be *finite* if X, A, and Π are finite.



(a) Non-determinsitic (b) Stochastic (c) Deterministic

Figure 2.1: Examples of different transition systems

The transition system starts in some initial state $x \in X_0$ and evolves according to the transition function, $x' = \delta_{\mathcal{T}}(x, a)$. If the x is the current state, and an action a is performed, the transition system evolves from state x to x' nondeterministically. This procedure is repeated forever, or until it reaches a state that has no outgoing transitions. The sequence of states starting from some initial state is called a *path*. We say a path is *finite* if the transitions terminate at some state, and a path is *infinite* if the transitions are performed indefinitely.

Example 1. *Figure 2.1a depicts an example of a TS. The system starts from the initial state* $x_{1,1}$ *(Label: start) and the goal is to reach* $x_{3,3}$*. At* $x_{1,1}$*, performing the action* a_1 *transitions to either* $x_{1,2}$ *or* $x_{2,1}$ *nondeterministically. Depending on the non-determinism, the system may or may not observe* $\{landmark\_1\}$*.*

Figure 2.1 compares three transition systems: non-deterministic, stochastic, and deterministic transition systems.

A non-deterministic transition system allows for each state x, zero, one or multiple possible states $x' = \delta(x, a)$ for action a. The choice of next state is assumed to be "arbitrary". Non-determinism can capture and model system failures or uncertainties such as state-transition uncertainties due to wind or ocean current. However, it makes the model-checking, verification, and strategy synthesis complicated. In practice, we distinguish deterministic and non-deterministic

systems and model them differently. Later, we introduce how we model non-determinism as a game (in Definition 29) between the system and the environment.

A stochastic transition system restricts the transition probability to a certain probability distribution, denoted as $x' \sim P(x'|x, a)$. A stochastic transition system with a reward function is called Markov Decision Process (MDP) and is often employed in robotics. It can model a wide variety of robotic systems, especially dynamical systems that contain noise.

A deterministic transition system, as the name suggests, limits to deterministic transitions where the cardinality of the transition system under action a at state x is 1, i.e., $|\delta_{\mathcal{T}}(x, a)| \leq 1$. In this work, we model the robotic systems as a deterministic system, since we are interested in reasoning over the abstraction of the robotic systems rather than considering the low-level dynamics of the robotic system. For example, we assume that the robots complete the action *go_to_location_A* with high accuracy. Such an abstraction is commonly used and constructed in formal approaches to both mobile robotics [53, 52, 55] and robotic manipulators [45, 46].

Definition 2 (DTS)**.** *A deterministic transition system (DTS) is a tuple* $\mathcal{T} = (X, A, x_0, \delta_{\mathcal{T}}, \Pi, L)$*, where*

- X *is a finite set of states,*
- A *is a finite set of controls or actions,*
- $X_0 \in X$ *is the initial state,*
- $\delta_{\mathcal{T}} : X \times A \to X$ *is the (partial) transition function where* $|\delta_{\mathcal{T}}(x, a)| \leq 1$*,*
- Π *is a finite set of atomic propositions (predicates), and*
- $L : X \to 2^{\Pi}$ *is a labeling function that maps each state to the set of predicates that are true at that state.*

Next, we define terms associated with DTS, e.g., plan, trajectory, and trace. Some terms are commonly used in robotics but they may have different definitions from other communities, e.g., motion planning, task scheduling, decision making, and machine learning communities.

Definition 3 (Plan)**.** *A plan* $\gamma = \gamma_0\gamma_1 \dots \gamma_{n-1}$ *is a sequence of actions, where* $\gamma_i \in A$ *for all*

$0 \leq i \leq n-1$.

Definition 4 (Trajectory)**.** *By executing γ, the robot generates a trajectory $s = s_0 s_1 \dots s_n$, where $s_0 = x_0$ and $s_{i+1} = \delta_{\mathcal{T}}(s_i, \gamma_i)$.*

Definition 5 (Valid Plan)**.** *A valid plan is plan γ that respects the transition function $\delta_{\mathcal{T}}$, i.e, $\delta_{\mathcal{T}}(s_i, \gamma_i)$ exists for all $0 \leq i \leq n-1$. We denote the set of all valid plans by Γ.*

Definition 6 (Trace)**.** *The trace of the above trajectory is the sequence of observed labels, i.e., $\rho_\gamma = L(s_0)L(s_1)\dots L(s_n)$.*

Definition 7 (Language)**.** *We refer to the set of all observation traces that a robot can generate as the language of $\mathcal{T}$, i.e., $\mathcal{L}(\mathcal{T}) = \{\rho_\gamma \mid \gamma \in \Gamma\}$.*

Example 2. *Figure 2.1c depicts a DTS. It is a deterministic form of Figure 2.1a. Notice that there are no duplicate actions (outgoing edges with the same label) at each state. Given a plan $\gamma = a_1 a_2 a_3 a_3$ and a current state $x = x_{1,1}$, the transition system generates a trajectory $s = x_{1,1}x_{1,2}x_{2,2}x_{2,3}, x_{3,3}$. This run induces trace $\rho_\gamma = \{\texttt{start}\}\{\texttt{landmark\_1}\}\{\}\{\}\{\texttt{red\_room}\}$.*

2.2 Formal Specification Formalisms

Now we want to reason over the behavior of a transition system. To achieve this, we use formal language to define the behavior of the system to formally specify the desired behavior to avoid any ambiguity that may commonly arise when using Natural Language. For example, the command "Visit Room Red and Room Blue" leaves ambiguity in the order of the execution. Perhaps "Room Blue" must be visited before "Room Red". If so, this order can easily be specified using temporal logic. For example, in Linear Temporal Logic (LTL), we can express using the formula $\mathcal{F}\big(\texttt{Room\_Blue} \wedge \mathcal{F}(\texttt{Room\_Red})\big)$ which reads "Eventually reach Room Blue and then next eventually reach Room Red". We will explain it more in detail later.

In this thesis, we use atomic propositions to express the states of the surrounding world. Let AP be a set of atomic propositions. An example of an atomic proposition is *red_room* to encode

whether the robot entered the red room or not. Multiple atomic propositions can be true at the same time. E.g., *red_room* and *landmark_1*. The set of all subsets of the atomic propositions is called *alphabet*.

Definition 8 (Alphabet)**.** *An alphabet is a finite, non-empty set of symbols* $\Sigma = 2^{AP}$.

In natural (spoken) language an alphabet might be a set of words. In robotics applications, an alphabet will have a very different meaning as we will see subsequently. For example, an alphabet would be the power set of atomic propositions such that $\Sigma = 2^{\{\texttt{landmark\_1},\texttt{red\_room},\texttt{blue\_room}\}} = \Big\{\lambda, \{\texttt{landmark\_1}\}, \{\texttt{red\_room}\}, \{\texttt{blue\_room}\}, \{\texttt{landmark\_1}, \texttt{red\_room}\}, \{\texttt{landmark\_1}, \texttt{blue\_room}\}, \{\texttt{red\_room}, \texttt{blue\_room}\}\ \{\texttt{landmark\_1}, \texttt{red\_room}, \texttt{blue\_room}\}\Big\}$. We often describe an alphabet over propositions in logical formulas such that we can then reason about these propositions.

Definition 9 (String / Word)**.** A *(finite) string or word* $w = a_1 \dots a_n$ *for* $a \in \Sigma$ *is a possibly empty, finite, ordered list of symbols. We write* λ *for the unique string of length 0 (called the empty string) and* $|w|$ *for the length of* w*. Thus* $n = |w|$*.* $\Sigma^\star$ *denotes the set of all finite strings over* Σ*, while* Σ^ω *denotes the set of all infinite strings over* Σ *[47].*

Example 3. *A trace induced by a run in Example 2 is word* $\omega = \{\texttt{start}\}\ \{\texttt{landmark\_1}\}\ \{\}\ \{\}\ \{\texttt{red\_room}\}$.

Definition 10 (*Language*)**.** *A language* $\mathcal{L}$ *is a possibly infinite set of strings:* $\mathcal{L} \subseteq \Sigma^\star$*. Let* $\mathbb{N}$ *denote the set of non-negative integers. For all* $k \in \mathbb{N}$*, let* $\Sigma \leq k = \{w \in \Sigma^\star : |w| \leq k\}$ *and* $\Sigma^{>k} = \{w \in \Sigma^\star : |w| > k\}$ *[47].*

For any finite set of strings $\mathcal{L}$, let $\|\mathcal{L}\|$ be the sum of the lengths of the strings in $\mathcal{L}$. We will write $|\mathcal{L}|$ for the cardinality of $\mathcal{L}$. The Kleene star of a language $\mathcal{L}$ is another language written $\mathcal{L}^\star$ and is defined recursively [47] as the smallest set $\mathcal{L}^*$ that satisfies:

$$\begin{cases} \lambda \in \mathcal{L}^\star \text{ and } \forall w \in \mathcal{L}, w \in \mathcal{L}^\star & \text{(base cases)} \\ w, v \in \mathcal{L}^\star \Rightarrow wv \in \mathcal{L}^\star & \text{(recursive case)} \end{cases} \tag{2.1}$$

In the formal descriptions of languages, symbols in Σ and strings from $\Sigma^\star$ are simply formal concepts. Depending on what symbols are available in Σ and hence can be used in $\Sigma^\star$, people might assign specific meaning to these symbols, strings, and languages. However, from a formal perspective, these definitions do not require the assignment of a particular meaning.

2.3 Deterministic Finite Automaton (DFA)

To reason over a system, we need to consider the language of the system. This means that we may require infinite memory size. To encode the language, we can utilize a state machine called finite automaton. Finite automata are good models for computers with a finite memory. Automata can model a wide variety of behaviors of a system while it is easy to interpret, verify, and model-check. Automata are widely investigated in computer science and they have interesting properties that we will leverage in this thesis.

Definition 11 (DFA)**.** *A deterministic finite automaton (DFA) is a tuple* $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$*, where*

- Q *is a finite set of states,*
- Σ *is a finite set of input symbols,*
- $q_0 \in Q$ *is the initial state,*
- $\delta : Q \times \Sigma \to Q$ *is the transition function, and*
- $F \subseteq Q$ *is the set of final or accepting states.*

The transition function δ can be also viewed as a relation $\delta \subseteq Q \times \Sigma \times Q$, where every transition is a tuple $(q, \sigma, q') \in \delta$ iff $q' = \delta(q, \sigma)$, where $\sigma \in \Sigma$.

Definition 12 (Run)**.** *A run of a DFA* $\mathcal{A}$ *on trace* $\omega = \omega_1\omega_2 \dots \omega_n$ *is a sequence of states* $z = z_0 z_1 \dots z_n$*, where* $z_0 = q_0$ *and* $z_i = \delta(z_{i-1}, \omega_i)$ *for* $i = 1, \dots, n$*.*

Definition 13 (Accepting Run)**.** *A run* z *is called accepting if* $z_n \in F$*. A trace* ω *is accepted by* $\mathcal{A}$ *if it induces an accepting run.*

Definition 14 (Language)**.** *The set of all traces that are accepted by DFA $\mathcal{A}$ is called the language of $\mathcal{A}$ and is denoted by $\mathcal{L}(\mathcal{A})$.*

We can now see that the automata encode a language. Here, we also introduce non-deterministic finite automaton (NFA) as this becomes helpful to show the relationship between finite automata and logic such as regular expression and linear temporal logic.

Definition 15 (NFA)**.** *A non-deterministic finite automaton (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, where*

- *Q is a finite set of states,*
- *Σ is a finite set of input symbols,*
- *$q_0 \in Q$ is the initial state,*
- *$\delta : Q \times \Sigma \to 2^Q$ is the transition function, and*
- *$F \subseteq Q$ is the set of final or accepting states.*

Note that the definition of the transition function is different from that of DFA. The transition maps to the collection of all subsets of Q, i.e., the power set of Q. Due to this non-determinism, the definition of the accepting run also varies. In NFA, the non-deterministic transition under the same action generates two copies of the trace. Any one of the copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

Example 4. *Given a run $z = \beta\beta$ over the NFA in Figure 2.2, it generates three sequences of states $z1 = q_0, q_0, q_0$, $z2 = q_0, q_0, q_1$ and $z3 = q_0, q_1, q_2$. As $z3$ ends in an accepting state, the run is accepting.*

Importantly, the following lemma holds between DFA and NFA.

Theorem 1. *Every nondeterministic finite automaton has an equivalent deterministic finite automaton.*

Proof. See Theorem 1.39 in [81]. □

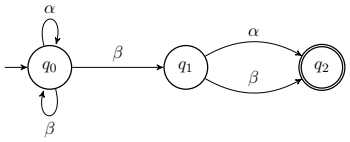


Figure 2.2: NFA representation of the Regular Expression Example 5

2.4 Regular Language

Definition 16 (Regular Language)**.** *A language is called a regular language if some finite automaton recognizes it.*

Definition 17 (Regular Operations)**.** *Let $\mathcal{L}_A$ and $\mathcal{L}_B$ be languages. We define the regular operations union, concatenation, and star as follows:*

- *Union:* $\mathcal{L}_A \cup \mathcal{L}_B = \{x \mid x \in \mathcal{L}_A \text{ or } x \in \mathcal{L}_B\}$.
- *Concatenation:* $\mathcal{L}_A \circ \mathcal{L}_B = \{xy \mid x \in \mathcal{L}_A \text{ and } y \in \mathcal{L}_B\}$.
- *Star:* $\mathcal{L}_A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in \mathcal{L}_A\}$

Lemma 1. *The collection of regular languages is closed under all three of the regular operations: union, concatenation, and star.*

Proof. See Theorem 1.45, Theorem 1.47, and Theorem 1.49 in [81]. □

Definition 18 (Regular Expression)**.** *Say that R is a regular expression if R is*

(1) empty $\emptyset$,

(2) A empty string λ,

(3) A symbol p where $p \in \Sigma$,

(4) $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,

(5) $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions,

(6) R_1^**, where* R_1 *is a regular expression.*

In other words, regular expression E *is recursively defined as:*

$$E := \emptyset \mid \lambda \mid p \mid E + E \mid E.E \mid E^*$$

where the add operator $+$ *corresponds to the union operation and the dot operator* $.$ *corresponds to the concatenation operation.*

The concatenation operation may sometimes be hidden. For example $E_1.E_2 = E_1E_2$.

Theorem 2. *Any regular expression can be converted into a finite automaton (NFA) that recognizes the language it describes, and vice versa*

Proof. See Theorem 1.54, Lemma 1.55, Lemma 1.60 in [81] □

Example 5. *A regular expression* $(\alpha \cup \beta)^*\beta(\alpha \cup \beta)$ *can be translated into a finite automaton shown in Figure 2.2.*

Because the language is called regular language if some finite automaton recognizes it, and any regular expression can be converted to a finite automaton, they all represent the same language.

2.5 Linear Temporal Logic

Linear Temporal Logic is commonly used as a temporal specification of the course of actions of an agent or a system of agents over time. LTL reasons about how properties of a system change over time [11]. LTL's syntax contains path formulae such that each logical expression describes a specification that can be validated over a trajectory of any system, thus, LTL formulas define *path* formula only. LTL has the following grammatical syntax given a set of *atomic propositions AP* [11]:

$$\varphi := \top \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathcal{X}\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

To obtain the full expressivity of propositional logic, additional operators are defined as [11]:

$$\varphi_1 \rightarrow \varphi_2 := \neg\varphi_1 \vee \varphi_2$$

$$\varphi_1 \vee \varphi_2 := \neg\left(\neg\varphi_1 \wedge \neg\varphi_2\right)$$

$$\varphi_1 \leftrightarrow \varphi_2 := (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$$

In addition, the temporal operators $\mathcal{F}$ ("eventually") and $\mathcal{G}$ ("always" or "globally") are defined as follows [11]:

$$\mathcal{F}\varphi := \top\mathcal{U}\varphi \tag{2.2}$$

$$\mathcal{G}\varphi := \neg\mathcal{F}\neg\varphi \tag{2.3}$$

For example, we might task an autonomous system to "first go to the red room through landmark 1 and then go to the blue room." This statement in natural language, while mostly understandable, is more clearly defined by a temporal logical formula (Linear Temporal Logic (LTL) in this case):

$$\mathcal{F}\left(\texttt{landmark\_1} \wedge \mathcal{F}(\texttt{red\_room} \wedge \mathcal{F}\,\texttt{blue\_room})\right) \tag{2.4}$$

Given the syntax, we formally define the semantics of LTL to define the acceptance condition of a word.

Definition 19. *LTL Semantics: The satisfaction of formula φ over a set of observations (the alphabet) $\Sigma = 2^{AP}$ at position k by infinite word (see Def. 9) $w = w(1)w(2)w(3)\ldots \in O^{\omega}$, denoted by $w(k) \vDash \varphi$, is defined as [11]:*

- $w(k) \vDash \top$
- $w(k) \vDash o$ *for some* $o \in O$ *if* $w(k) = o$
- $w(k) \vDash \neg\varphi$ *if* $w(k) \neq \varphi$
- $w(k) \vDash \varphi_1 \wedge \varphi_2$ *if* $w(k) \vDash \varphi_1$ *and* $w(k) \vDash \varphi_2$

- $w(k) \vDash \mathcal{X}\varphi$ *if* $w(k+1) \vDash \varphi$
- $w(k) \vDash \varphi_1 \mathcal{U} \varphi_2$ *if there exist* $j \geq k$ *such that* $w(j) \vDash \varphi_2$ *and, for all* $k \leq i < j$ *we have* $w(i) \vDash \varphi_1$

Given the semantics, we can now define the language of an LTL formula. A word w satisfies an LTL formula φ, written as $w \vDash \varphi$, if $w(1) \vDash \varphi$. The language of infinite words that satisfy formula φ is thus L_φ [11]. Once we have languages defined by a formula, we can begin to classify the properties of the languages we define:

Definition 20 (*Safety Property*)**.** *A safety property* $L(\phi)$ *is a language over* 2^{AP} *such that for any word* w *that violates* ϕ *(and thus is not in the language defined by* ϕ*),* w *has a prefix for which all extensions of which also violate* ϕ *[11].*

Definition 21 (Safe Syntax)**.** *A syntactically safe LTL[54] formula over* Π *is recursively defined as*

$$\phi := p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathcal{X}\phi \mid \mathcal{G}\phi$$

where $p \in \Pi$*,* $\neg$ *(negation),* $\vee$ *(disjunction), and* $\wedge$ *(conjunction) are boolean operators, and* $\mathcal{X}$ *("next") and* $\mathcal{G}$ *("globally") are temporal operators.*

Safe LTL formulas reason over infinite traces, but finite traces are sufficient to violate them [54]. We denote the set of finite traces that violate safety formula ϕ_{safe} by $\mathcal{L}(\neg\phi_{\text{safe}})$.

Definition 22 (*Co-Safety Property*)**.** *A co-safety property is a language that complements the safety property over* 2^{AP} *such that* $L(\varphi)$ *is a co-safety property* $\iff$ $L(\phi) = 2^{AP} \backslash L(\varphi)$ *is a safety property [11].*

Using Defs. 20 & 22, an important fragment of LTL formula, co-safe LTL (scLTL) formula can be defined. A scLTL formula do not allow for the negation of formula, only APs (Observations), and thus does not have the important *globally* operator ($\mathcal{G}$) defined in Eq. 2.3.

Definition 23 (*Co-Safety (scLTL) Syntax*)**.** *A syntactically co-safe linear temporal logic (scLTL) formula φ over a set of atomic propositions AP is defined as [11]:*

$$\varphi = \top \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathcal{X}\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

where $p \in AP$ is an atomic proposition and φ, φ_1 and φ_2 are scLTL formulas.

Even though scLTL formulas are interpreted over infinite words $w \in \Sigma^\omega$, their satisfaction is guaranteed in finite time. This is because any (possibly infinite) word in satisfying a scLTL formula *must* have a finite prefix that satisfies the formula. An even more useful class of LTL formula is known as a linear temporal logic on finite traces (LTL_f) formula, which shares syntax with a normal LTL formula. The difference is in the languages of an LTL_f formula φ, which is defined over finite traces instead of infinite traces. We are not going to reason over infinite traces (although safety must reason over infinite traces) in this thesis. It is sufficient for the robotics systems to consider finite traces.

The syntax of LTL_f is the same as LTL, but the semantics are different [30].

Definition 24. *LTL_f Semantics: The satisfaction of formula φ over a set of observations (the alphabet) $\Sigma = 2^{AP}$ at position k by finite word (see Def. 9) $w = w(1)w(2)w(3)\ldots \in O^*$ with $k \leq |w|$, denoted by $w(k) \vDash \varphi$, is defined as [30]:*

- $w(k) \vDash \top$
- $w(k) \vDash o$ *for some* $o \in O$ *if* $w(k) = o$
- $w(k) \vDash \neg\varphi$ *if* $w(k) \neq \varphi$
- $w(k) \vDash \varphi_1 \wedge \varphi_2$ *if* $w(k) \vDash \varphi_1$ *and* $w(k) \vDash \varphi_2$
- $w(k) \vDash \mathcal{X}\varphi$ *if* $k < |w|$ *and* $w(k+1) \vDash \varphi$
- $w(k) \vDash \varphi_1 \mathcal{U} \varphi_2$ *if there exist* $k \leq j \leq |w|$ *such that* $w(j) \vDash \varphi_2$ *and, for all* $k \leq i < j$ *we have* $w(i) \vDash \varphi_1$

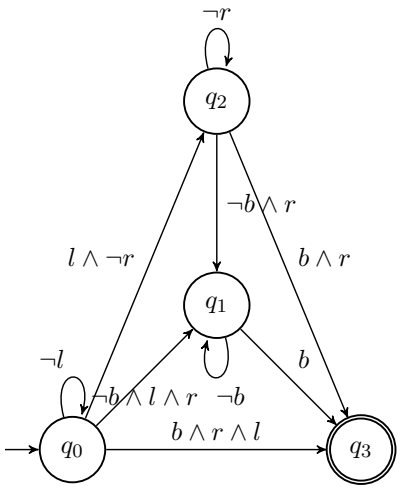


Figure 2.3: DFA representation of Equation (2.4). The letters l, r, b represent the atomic propositions $landmark\_1, red\_room, blue\_room$.

LTL_f can be translated into a DFA. For example, the formula Equation (2.4) that reasons over a finite trace can be translated into a DFA depicted in Figure 2.3. The letters l, r, b represent the atomic propositions $landmark\_1, red\_room, blue\_room$.

Given that the formula follows the scLTL syntax, finite traces are enough to satisfy the formula. Thus, if there exists a finite run that reaches the accepting state q_3 in Figure 2.3, then we say the run is accepting.

We have looked at a few different languages: Regular Languages and the language of LTL_f. This poses the question of how they are related to each other. Are they equivalent? It turns out that LTL_f is less expressive than Regular Expression.

Lemma 2. *Regular expressions are strictly more expressive than LTL_f.*

Proof. See Theorem 6 in [30] □

Definition 25. *Start free regular expressions are defined as by the following:*

$$E := \varrho \mid E + E \mid E.E \mid \overline{E} \tag{2.5}$$

where ϱ is a propositional formula that is an abbreviation for the union of all the propositional interpretations that satisfy ϱ, that is $\varrho = \sum_{AP \vDash \varrho} AP$ ($\emptyset$ is now abbreviated by false). Note, however,

that several regular expressions involving * can be rewritten by using complementation instead, including, it turns out, all the ones that correspond to LTL$_f$ properties.

- $(2^{AP})^* = true^*$ *is in fact star-free, as it can be expressed as* $\overline{false}$
- $true^*.\varrho.true^*$ *is star-free as* $true^*$ *is star-free,*
- ϱ^* *star free, as it is equivalent to* $true^*.\neg\varrho.true^*$

Lemma 3. *LTL$_f$ has exactly the same expressive power of star-free RE (over finite traces) .*

Proof. See Theorem 8. in [30]. □

Here is the list of expression that the regular expression can express but not LTL$_f$.

- "Alternating sequence": $(\psi.\varrho)^*$ that means that ψ and ϱ, not necessarily disjoint, alternate for the whole sequence starting with ψ and ending with ϱ.
- "Pair sequence": $(true.true)^*$ that means that the sequence is of even length.
- "Eventually on an even path ϱ": $(true.true)^*.\varrho.true^*$, i.e., we can constrain the path fulfilling the eventuality to satisfy some structural (regular) properties, in particular in this case that of $(true.true)^*$.

Lemma 4. *LTL$_f$ is less expressive than DFA.*

Proof. DFA has the same expressive power as Regular Expression. From Lemma 2, LTL$_f$ is less expressive than DFA. □

Chapter 3

Probabilistic DFA Learning with Safety Guarantees

This chapter proposes a framework for learning task specifications from demonstrations, while ensuring that the learned specifications do not violate safety constraints. We formulate the specification learning problem as a grammatical inference problem, using probabilistic automata (PDFA) to represent specifications. The edge probabilities of the resulting automata represent the demonstrator's preferences. The main novelty in our approach is to incorporate the safety property during the learning process. We prove that the resulting automaton always respects a pre-specified safety property, and furthermore, the proposed method can easily be included in any Evidence-Driven State Merging (EDSM)-based automaton learning scheme.

Furthermore, we show how these specifications can be used in a planning problem to control the robot under environments that can be different from those encountered during the learning phase. We introduce a planning algorithm that produces the most desirable plan by maximizing the probability of an accepting trace of the automaton. Case studies show that our algorithm learns the true probability distribution most accurately while maintaining safety. Since, specification is detached from the robot's environment model, a satisfying plan can be synthesized for a variety of different robots and environments including both mobile robots and manipulators.

3.1 Introduction

Humans perform tasks in various orders that may differ from time to time or from human to human based on their preferences. Preferences may be important when collaborating with another

human or they may lead to efficient or safe executions. Our goal is to mine the underlying specification with human preferences from such demonstrations and autonomously execute the specification even against varying environments. In this regard, we identify four key challenges: (a) identifying a rich specification formalism that permits efficient and precise learning algorithms from the given demonstrations; (b) ensuring that key safety properties are satisfied by the resulting specification; (c) capturing "operator preferences" or "hidden costs" from the demonstrations; and (d) using the specifications to guide autonomous behavior under environments that may be different from those under which the demonstrations were provided. In this work, we propose solutions to all four problems using the framework of probabilistic automata learned using grammatical inference. We focus on **safety-property constrained learning**: wherein the final learned specification must satisfy the safety properties. Our proposed learning approach ensures that the safety properties constrain the intermediate steps of the learning algorithm as opposed to an "after-the-fact" approach wherein the safety properties are intersected with the final specifications to rule out any unsafe behavior. We show that our safety-enabled learning approach clearly outperforms the after-the-fact approach.

Many LfD studies focus on either learning a policy or a reward structure [73, 49]. For policy learning, techniques such as *reinforcement learning* (RL) [84] and Dynamic Movement Primitives [77, 67] are typically used to learn a function that maps agent states to actions. In reward learning, a scalar reward function that maps agent states to rewards is learned via, e.g., *inverse reinforcement learning* (IRL) [63, 101, 99, 72], to later train a policy on an agent. While very powerful, these methods learn a function that is specific to the environment (and robot) model used during training and hence are fragile to the changes to those models. More importantly, those methods are restricted to Markovian tasks. For example, the robot task in Figure 1.2, which requires a visit to both the shipwreck and school of fish in any order, is non-Markovian. To achieve it, the robot has to maintain a memory of previous locations to decide the next location to visit. Hence, it is important to enable LfD for non-Markovian tasks, but such an extension is nontrivial for the existing LfD methods.

An alternative approach to expressing tasks is to use formal languages such as *linear temporal logic* (LTL) [8], which is widely used in formal verification and increasingly employed in robotics

in recent years, e.g., [53]. Such languages enable formal expression of rich missions, including non-Markovian tasks [92] as well as *liveness* ("something good eventually happens") and *safety* ("something bad never happens") requirements. Other important benefits of formal languages is in their ease of interpretability and flexibility to compose multiple specifications. Such benefits have even led to their use in RL, e.g., [18, 56, 57]. Nevertheless, writing correct formal specifications requires domain knowledge.

In recent years, a new line of research has emerged with a focus on learning formal specifications from data [92, 91, 100, 50, 80]. Most work has been concerned with learning temporal logic formulas with the purpose of classification and prediction from user data (in the supervised learning sense) [100, 50] or interpretation and planning for tasks [80]. Those studies restrict the exploration problem to a set of formula templates provided *a priori*. Recent work [92] overcomes this restriction by iterating over all combinations of formulas. The method is based on maximum a posterior learning and can account for noisy samples. It however is slow due to the large space of exploration for formulas. Another important issue with formula learning methods for the purpose of planning is that they typically need to be translated to an automaton, which could lead to the **state-explosion** problem [8, 53]. In detail, Theorem 5.41 of [8] proves that translation of a formula to a language-equivalent automaton is exponential $2^{O(|\varphi|)}$ in time and space, where $|\varphi|$ denotes the length of LTL formula φ in terms of the number of operators in φ. This translation is unavoidable when learning a formula from demonstrations. Work [6] overcomes this issue by directly learning a *Deterministic Finite Automaton* (DFA). They however assume the structure of the DFA is known and only learn the transitions between the DFA states while an oracle labels each sample with DFA states.

In this work, we propose a new approach to task specification learning where we infer formal task specifications as probabilistic automata. Our approach used ideas from the field of **grammatical inference** (GI) [31], by modifying existing "evidence-driven state merging" algorithms in GI to learn the specification as a *Probabilistic Deterministic Finite Automaton* (PDFA). We further extend this method to incorporate safety properties during the learning process, so that all runs of

the final PDFA satisfy the safety properties. Furthermore, we propose a planning algorithm with the inferred PDFA that generates the most preferred plan to achieve the task for any robot that can be abstracted to a deterministic transition system.

To the best of our knowledge, this is the first work that employs PDFA as a learning model for task specifications in robotics. The key technical contributions include: (a) the derivation of the safety guaranteed PDFA learning algorithm that is compatible with any EDSM techniques; (b) a planning algorithm with a learned PDFA that can handle varying environments; and (c) a set of experiments that show the efficacy of the proposed algorithms in both mobile and manipulator robots. We also provide a comparison case study against [92] to highlight the similarities and differences between the two approaches.

3.2 Preliminaries

3.2.1 Task Specifications and demonstrations

In this section, we describe and formalize our problem of task specification learning with preferences and safety constraints. Our aim is to use an interpretable learning model that we can use for planning for a robot to perform the task according to the user's preferences.

We assume the demonstrator has a task in mind that needs to be achieved in finite time. Our goal is to learn this task in the form of a deterministic finite state automaton (DFA) that reflects the overall goals of the demonstrator. Such automata accommodate a large class of tasks, including the ones that can be specified using co-safe LTL [54] and *LTL over finite traces* [30] formulas. Nevertheless, to avoid a trivial solution to the problem (e.g., a trivial specification that accepts all possible behaviors), we consider two important caveats.

- We specify, on the side, a safety property specification that the robot must not violate. The safety property characterizes a potentially infinite set of negative examples for our learner consisting of those specifications that violate the safety property.

- We want to encode the demonstrator's **preferences** between various ways of satisfying an

intended specification. For instance, a demonstrator may prefer to avoid a collision with an obstacle by **steering left** preferentially since it may position the vehicle to avoid a future collision with less effort.

Recall the definitions from Chapter 2. We assume that demonstrations Ω are sampled from the underlying task φ_{task} define over a set of atomic propositions Π and our goal is to learn a temporal logic formula in the form of a DFA $\mathcal{A}$ from these demonstrations. We consider valid demonstration ω that achieves the task φ_{task}.

Definition 26 (Valid Demonstration)**.** *A valid demonstration for task φ_{task} is a finite trace $\omega \in \Sigma^*$ such that $\omega \in \mathcal{L}(\mathcal{A}_{\varphi_{task}})$, where $\mathcal{A}_{\varphi_{task}}$ is the DFA that represents φ_{task}.*

To express the safety constraints, we use safe LTL [54] as defined in Chapter 2. Hence, given valid demonstrations and safety formula φ_{safe}, we want to learn a DFA $\tilde{\mathcal{A}}_{\varphi_{\text{task}}}$ such that it does not accept any trace that violates safety, i.e, $\mathcal{L}(\tilde{\mathcal{A}}_{\varphi_{\text{task}}}) \cap \mathcal{L}(\neg\varphi_{\text{safe}}) = \emptyset$.

On the learned DFA, we also want to encode the preferences of the demonstrator. We assume that the preferences are correlated with the number of demonstrations of the same trace. Hence, preferences can be quantified as weights over traces and normalized over the entire language, which can be viewed as a probability distribution over $\mathcal{L}(\mathcal{A}_{\varphi_{\text{task}}})$. The higher the probability of an accepting trace is, the more preferred it is to the demonstrator. Hence, our aim is to learn a probabilistic DFA (PDFA), which captures both the accepting traces and their corresponding probabilities.

Definition 27 (PDFA)**.** *A probabilistic DFA (PDFA) is a tuple $\mathcal{A}^{\mathbb{P}} = (\mathcal{A}, \delta_{\mathbb{P}}, F_{\mathbb{P}})$, where $\mathcal{A}$ is a DFA, and $\delta_{\mathbb{P}} : \delta \to [0, 1]$ assigns a probability to every transition in δ such that $\sum_{\sigma\in\Sigma} \delta_{\mathbb{P}}(q, \sigma, \delta(q, \sigma)) = 1$ for every $q \in Q$, and $F_{\mathbb{P}} : Q \to [0, 1]$ assigns a probability of terminating at each state, where $F_{\mathbb{P}}(q) = 0$ if $q \notin F$.*

Consider trace $\omega = \omega_1\omega_2 \dots \omega_n$ and its induced run $z = z_0 z_1 \dots z_n$ on PDFA $\mathcal{A}^{\mathbb{P}}$. The

probability of ω is given by

$$P(\omega) = \prod_{i=1}^{n} \delta_{\mathbb{P}}(z_{i-1}, \omega_i, z_i) \cdot F_{\mathbb{P}}(z_n).$$

We say $\mathcal{A}^{\mathbb{P}}$ accepts ω iff $P(\omega) > 0$, and the demonstrator prefers ω over $\omega' \in (2^{\Pi})^*$ iff $P(\omega) > P(\omega')$. The language of $\mathcal{A}^{\mathbb{P}}$ is the set of traces with non-zero probabilities, i.e.,

$$\mathcal{L}(\mathcal{A}^{\mathbb{P}}) = \{\omega \in (2^{\Pi})^* \mid P(\omega) > 0\}.$$

Therefore, our goal becomes to learn a $\tilde{\mathcal{A}}^{\mathbb{P}}{}_{\varphi_{\text{task}}}$ that captures the task φ_{task}, the demonstrator's preferences (probability distribution over traces), and never violates φ_{safe}.

3.2.2 Robot Model and Plans

Once a specification is learned, we want to synthesize a plan for a robot that can realize the specification. To do so, we assume that we are given an abstraction model of the robot as a deterministic transition system $\mathcal{T}$.

In the planning problem, we are interested in a plan γ that generates an observation trace ρ_γ that achieves task φ_{task} and is the most preferred behavior, i.e.,

$$\rho_\gamma = \arg \max_{\omega \in \mathcal{L}(\varphi_{\text{task}}) \cap \mathcal{L}(\mathcal{T})} P(\omega).$$

3.3 Problem Formulation

We are now able to formally define our problem.

Problem 1 (PDFA Learning & Planning Problem)**.** *Given demonstrations $\Omega = [\omega_i]_{i=1}^{n_\Omega}$ that are sampled from a hidden task specification DFA $\mathcal{A}_{\varphi_{task}}$ according to some preferences (probability distribution) of the demonstrator and a safety specification φ_{safe},:*

(1) learn φ_{task} and the demonstrator's preferences as a PDFA $\tilde{\mathcal{A}}^{\mathbb{P}}{}_{\varphi_{task}}$ such that safety is never violated, i.e, $\mathcal{L}(\tilde{\mathcal{A}}^{\mathbb{P}}{}_{\varphi_{task}}) \cap \mathcal{L}(\neg\varphi_{safe}) = \emptyset$;

(2) furthermore, given a DTS robot model $\mathcal{T}$, compute a plan for the robot $\mathcal{T}$ that generates the most preferred behavior that satisfies φ_{task}, i.e, generates the trace with the highest probability in $\mathcal{L}(\tilde{\mathcal{A}}^{\mathbb{P}}{}_{\varphi_{task}}) \cap \mathcal{L}(\mathcal{T})$.

For Problem 1, we use grammatical inference [31] while incorporating the safety property during the learning process.

3.4 Approach

In this section, we explain how a PDFA can be learned from demonstrations and present our method that can embed safety specification to guarantee safety on the outcome. We first show a general PDFA learning algorithm, and then we describe our algorithms to incorporate safety.

3.4.1 Grammatical Inference: PDFA Learning

PDFA learning has been extensively studied as part of **grammatical inference** (GI) with existing algorithms such as ALERGIA, DSAI, and MDI, that can learn PDFAs from unlabeled demonstrations[31]. These algorithms are all based on a principle called **evidence-driven state-merging** (EDSM). At a high level, EDSM approaches find an appropriate structure for an automaton $\mathcal{A}^{\mathbb{P}}$ and simultaneously estimate the probability distribution parameters $F_{\mathbb{P}}$ and $\delta_{\mathbb{P}}$ given a set of sample traces. This is achieved by first constructing a large (prefix) tree from the samples, and repeatedly merging the states of the tree in order to form an automaton that is as simple as possible while continuing to accept the sample traces from the demonstration. The various algorithms (e.g., ALERGIA and MDI) differ on what states are merged.

Figure 3.1 shows a general scheme for an EDSM-based algorithm for learning a PDFA. The initial step is to construct a *frequency prefix tree acceptor* (FPTA) from the traces in ω (Fig. 3.1-Left) and then incrementally merge states of the FPTA, two at a time, based on a **compatibility** criterion that varies depending on the actual algorithm. As two states are merged, so are their subtrees in the FPTA (Fig. 3.1-Middle). The nodes of the intermediate automata are variously colored red/blue using a coloring scheme to influence how states are selected for merging. Furthermore, algorithms

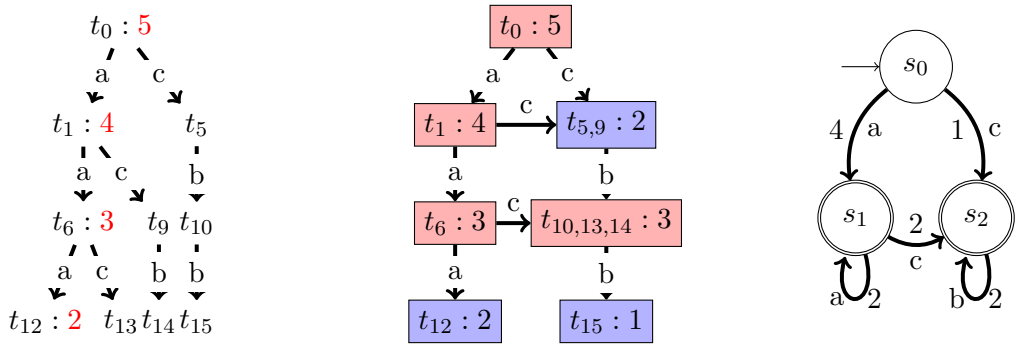


Figure 3.1: Schematic illustration of evidence-driven state merging (EDSM) algorithm. **(Left)** A frequency prefix tree acceptor (FPTA) is constructed from the given demonstrations with frequencies greater than 1 shown in red; **(Middle)** intermediate automaton as states are merged according to criteria that differ across various algorithms with frequencies shown at each node; and **(Right)** the final frequency DFA (FDFA) that is learned is shown in red.

also maintain frequencies alongside the nodes based on the number of demonstration traces that reach a particular node. These frequencies are also combined during the state merging process. The final result is a *frequency DFA* (FDFA) wherein frequencies along edges indicate how often they are taken by a demonstration (Fig. 3.1-Right). The frequencies of all outgoing edges are normalized to yield a distribution.

The various PDFA learning algorithms such as ALERGIA or MDI differ on how they implement the **compatibility** check for whether two given nodes can be merged. For instance, the ALERGIA algorithm implements a statistical test based on frequencies to compare if two states are compatible, whereas the MDI approach first temporarily merges two states and their subtrees, while accepting the merge if a metric computed on automaton after the merge is smaller than that before the merge. We assume that the basic PDFA learning algorithm as a given, and our goal is to learn while respecting a safety property.

3.4.2 Learning with Safety Specification

We now consider two different approaches for learning with a safety specification. The first method is a **post-processing** technique that simply runs the PDFA learning algorithm on the given demonstration traces and then subsequently intersects the resulting PDFA with the automaton for

the safety property. The second method incorporates the safety specification during the learning process by modifying the EDSM algorithm. In particular, the merges are defined so that the result continues to satisfy the safety specifications.

3.4.3 Post-process Algorithm

From φ_{safe}, we first construct a DFA $\mathcal{A}_{\neg\varphi_{\text{safe}}}$ that accepts precisely all those traces that violate the safety property [54]. Then, by complementing $\mathcal{A}_{\neg\varphi_{\text{safe}}}$, we obtain $\mathcal{A}_{\text{safe}} = (Q^s, \Sigma, q_0^s, \delta^s, F^s)$ that accepts all the traces that do not violate φ_{safe}. Let $\mathcal{A}^{\mathbb{P}} = (Q, \Sigma, q_0, \delta, F, \delta_{\mathbb{P}}, F_{\mathbb{P}})$ be the PDFA learned from the given demonstration traces without considering the safety property. We intersect the languages of $\mathcal{A}^{\mathbb{P}}$ and $\mathcal{A}_{\text{safe}}$ by constructing a product automaton $\mathcal{A}^{\mathbb{P}}_{\text{safe}} = \mathcal{A}^{\mathbb{P}} \otimes \mathcal{A}_{\text{safe}} = (Q_{\text{safe}}, \Sigma, q_{0,\text{safe}}, \delta_{\text{safe}}, F_{\text{safe}}, \delta_{\mathbb{P},\text{safe}}, F_{\mathbb{P},\text{safe}})$, where

- $Q_{\text{safe}} = Q \times Q^s$,
- $q_{0,\text{safe}} = (q_0, q_0^s)$,
- $F_{\text{safe}} = F \times F^s$,
- $\delta_{\text{safe}}((q, q^s), \sigma) = (q', q^{s\prime}) \quad$ if $q' = \delta(q, \sigma) \wedge q^{s\prime} = \delta^s(q^s, \sigma)$,
- $F_{\mathbb{P},\text{safe}}((q, q^s)) = F_{\mathbb{P}}(q)$, and

$$\delta_{\mathbb{P},\text{safe}}((q, q^s), \sigma, (q', q^{s\prime})) = \begin{cases} \frac{\delta_{\mathbb{P}}(q,\sigma,q')}{N(q,q^s)} & \text{if } (q', q^{s\prime}) = \delta_{\text{safe}}((q, q^s), \sigma) \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

where $N(q, q^s)$ is the normalizing function such that

$$\sum_{(\sigma, q'_{\text{safe}}) \in (\Sigma \times Q_{\text{safe}})} \delta_{\mathbb{P},\text{safe}}((q, q^s), \sigma, q'_{\text{safe}}) = 1$$

.

The resulting PDFA is guaranteed to be safe due to the intersection of languages. However, this method of pruning (imposing safety) as a post-process step alters the probability distributions

over the next-state transitions, since we remove the transitions that violate safety and renormalize the probability distribution at each state, as shown in (3.1). This overrides the probability distributions constructed by the original PDFA learning algorithm in an unpredictable manner. Therefore, while this method of imposing safety always succeeds, its probability distributions may not reflect the preferences embedded in the demonstrations accurately.

3.4.4 Safety-Incorporated Learning Algorithm using "Pre-Processing"

Whereas the **post-processing** approach enforces safety after the PDFA is learned, the pre-processing approach guarantees that the intermediate results also preserve safety, hence preventing alterations to the probability distributions due to unsafety. The main idea is to build the PDFA that generalizes the demonstrated traces but carries along with it information about how the generalization satisfies the safety property φ_{safe} at the same time in the form of a simulation relation with $\mathcal{A}_{\text{safe}}$.

Definition 28 (Simulation Relation)**.** *A simulation relation R between two automata $\mathcal{A}$ and $\mathcal{B}$ is a relation between their states, $R \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$*

(a) Initial states of $\mathcal{A}$ relate to the initial states of $\mathcal{B}$;

(b) If pair $(s,t) \in R$, where $s \in Q_{\mathcal{A}}$ and $t \in Q_{\mathcal{B}}$, and automaton $\mathcal{A}$ can transition from s to $s' \in Q_{\mathcal{A}}$ on symbol σ, then there must exist a state $t' \in Q_{\mathcal{B}}$ such that automaton $\mathcal{B}$ transitions from t to t' on the same symbol σ and $(s',t') \in R$;

(c) For each $(s,t) \in R$, if s is final in $\mathcal{A}$ then t must be final in $\mathcal{B}$.

Theorem 3 (Language Inclusion)**.** *Let R be a simulation relation between automata $\mathcal{A}$ and $\mathcal{B}$. It follows that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.*

The proof simply shows by induction that for any accepting run corresponding to an input trace ω in automaton $\mathcal{A}$ from the initial state to a final state, there exists an accepting run in $\mathcal{B}$ for the same trace ω from its initial state to the final state. The relation R allows us to construct such a run.

The key idea behind the **pre-process** approach is to maintain a simulation relation between the FDFA and safety automaton $\mathcal{A}_{\text{safe}}$ at all intermediate states. The key is to restrict the merging of states so that we can guarantee that a simulation relation between the original automaton and $\mathcal{A}_{\text{safe}}$ before merging can be modified to yield a simulation relation between the merged automaton and $\mathcal{A}_{\text{safe}}$ afterwards.

Formally, we build the safety FPTA by augmenting the initial FPTA so that each state is now a tuple of the form (t_j, s_k) wherein t_j is a node in the original FPTA and s_k is the state in $\mathcal{A}_{\text{safe}}$ reached when the prefix that leads upto the state t_j is run through $\mathcal{A}_{\text{safe}}$. Thus, we ensure that every branch not only corresponds to a demonstration but also to a valid trace in $\mathcal{A}_{\text{safe}}$.

Let R be a relation between states of the FPTA and $\mathcal{A}_{\text{safe}}$ that contains all nodes (t_j, s_k) in the safety FPTA.

Lemma 5. *Assuming no demonstration trace violates the safety property φ_{safe}, then R is a simulation relation between the initial FPTA and the automaton $\mathcal{A}_{safe}$.*

We can represent any intermediate FDFA state in the form (T, s) wherein T is a set of states from the initial FPTA, and s is state in $\mathcal{A}_{\text{safe}}$. Next, we modify the EDSM approach to allow a merge between two states (T_i, s_k) and (T_j, s_l) only if $s_k = s_l$. The result of the merge creates a state $(T_i \cup T_j, s_k)$.

Lemma 6. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the automata before and after an EDSM merge that is compatible with respect to the $\mathcal{A}_{safe}$ states. Let R_1 be the relation between the states of $\mathcal{A}_1$ and those of $\mathcal{A}_{safe}$ that is a simulation relation. We can construct a simulation relation R_2 between the states of $\mathcal{A}_2$ and $\mathcal{A}_{safe}$.*

Combining Lemmas 5 and 6, we conclude by induction on the number of merging steps that the final resulting PDFA must have a simulation relation to the safety automaton $\mathcal{A}_{\text{safe}}$. Since we have a simulation relation, we conclude that the language of the final resulting FDFA and PDFA are contained in the that of $\mathcal{A}_{\text{safe}}$, i.e., the resulting PDFA does not accept a trace that violates φ_{safe}.

Once a task is learned as a PDFA, we are interested in synthesizing a plan that not only achieves the task but also respects the demonstrator's preferences (Problem 2). Here, we introduce a method to do this for any robot with a DTS (abstract) model. We first construct a (product) graph that captures all the ways the robot can achieve the task by composing the DTS with the learned PDFA. Then, we reduce the optimal planning problem to a search on this graph.

The plans are computed over the product graph $\mathcal{P} = \mathcal{A}^{\mathbb{P}} \times \bar{\mathcal{T}}$, where $\bar{\mathcal{T}}$ is a DTS obtained by augmenting $\mathcal{T}$ with a new initial state $\bar{x}_0$ with a transition to x_0. Formally, $\bar{\mathcal{T}} = (\bar{X}, A_R, x_0^{\bar{\mathcal{T}}}, \delta^{\bar{\mathcal{T}}}, \Pi, L^{\bar{\mathcal{T}}})$, where $\bar{X} = X \cup \{\bar{x}_0\}$, and $\delta^{\bar{\mathcal{T}}}(x, a) = x_0$ if $x = \bar{x}_0$, otherwise $\delta^{\bar{\mathcal{T}}}(x, a) = \delta_{\mathcal{T}}(x, a)\ \forall a \in A$. This augmentation allows to correctly observe the label of x_0 and assign the edge weights in $\mathcal{P}$ according to the probabilities in $\mathcal{A}^{\mathbb{P}}$ through the product rule below. Given learned $\mathcal{A}^{\mathbb{P}} = (Q, \Sigma, q_0, \delta, F, \delta_{\mathbb{P}}, F_{\mathbb{P}})$ and $\bar{\mathcal{T}}$, we construct *weighted product graph* $\mathcal{P} = (Q^{\mathcal{P}}, q_0^{\mathcal{P}}, E^{\mathcal{P}}, W^{\mathcal{P}})$, where

- $Q^{\mathcal{P}} = (Q \times \bar{X}) \cup \{q_t^{\mathcal{P}}\}$ is a set of states, where $q_t^{\mathcal{P}}$ is the terminal state,
- $q_0^{\mathcal{P}} = (q_0, \bar{x}_0)$ is the initial state,
- $E^{\mathcal{P}} \subseteq Q^{\mathcal{P}} \times Q^{\mathcal{P}}$ is a set of edges, and
- $W^{\mathcal{P}} : E^{\mathcal{P}} \to \mathbb{R}^{<0}$ is a weight function that assigns to each edge $e \in E^{\mathcal{P}}$ a weight according to its probability in $\mathcal{A}^{\mathbb{P}}$.

The constructions of $E^{\mathcal{P}}$ and $W^{\mathcal{P}}$ are as follows.

- Edge $e = ((q, x), (q', x')) \in E$ if $q' = \delta(q, L(x'))$ and $x' = \delta^{\bar{\mathcal{T}}}(x, a)$ for some $a \in A$. Then, $W^{\mathcal{P}}((q, x), (q', x')) = \log(\delta_{\mathbb{P}}(q, L(x'), q'))$.
- Edge $e = ((q, x), q_t^{\mathcal{P}})) \in E^{\mathcal{P}}$ if $F_{\mathbb{P}}(q) > 0$. Then, its weight $W^{\mathcal{P}}((q, x), q_t^{\mathcal{P}}) = \log(F_{\mathbb{P}}(q))$.

Product graph $\mathcal{P}$ captures the constraints of both the robot and task along with the demonstrator's preferences. Let $\lambda = (q_0, \bar{x}_0)(q_1, x_0) \dots (q_n, x_{n-1}) q_t^{\mathcal{P}}$ be a path over $\mathcal{P}$. The projection of this path (with the deletion of $q_t^{\mathcal{P}}$) onto $\mathcal{A}^{\mathbb{P}}$ is an accepting run with the trace $\omega = \omega_1 \dots \omega_n$. The projection of λ on $\mathcal{T}$ is the robot trajectory that generates the accepting trace ω. The probability of this trace is in fact the inverse logarithm of the total weight of λ, i.e.,

$$\sum_{i=0}^{n} W^{\mathcal{P}}(\lambda_i, \lambda_{i+1}) = \sum_{i=0}^{n-1} \log(\delta_{\mathbb{P}}(q_i, q_{i+1})) + \log(F_{\mathbb{P}}(q_n))$$
$$= \log\Big(\prod_{i=0}^{n-1} \delta_{\mathbb{P}}(q_i, q_{i+1}) \cdot F_{\mathbb{P}}(q_n)\Big)$$
$$= \log(P(\omega)).$$

Therefore, to compute a robot plan that produces an accepting trace with the highest probability in $\mathcal{L}(\mathcal{A}^{\mathbb{P}}) \cap \mathcal{L}(\mathcal{T})$, it is enough to find a path on $\mathcal{P}$ that reaches the terminal state $q_t^{\mathcal{P}}$ with the maximum total weight, i.e.,

$$\arg\max_{\omega \in \mathcal{L}(\mathcal{A}^{\mathbb{P}}) \cap \mathcal{L}(\mathcal{T})} P(\omega) = \textsc{proj}\Big(\arg\max_{\lambda \in \Lambda} \sum_{i=0}^{|\lambda|-1} W^{\mathcal{P}}(\lambda_i, \lambda_{i+1})\Big),$$

where Λ is the set of paths of $\mathcal{P}$ that terminate in $q_t^{\mathcal{P}}$, and PROJ is the projection operator that maps λ to its corresponding trace ω. Note that the all the edge weights of $\mathcal{P}$ are negative, hence, it is a simple graph search problem that can be performed using algorithms such as Dijkstra's.

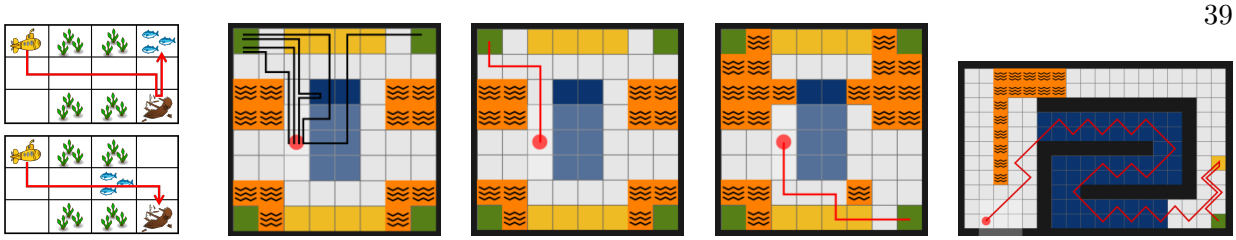
3.5 Case Studies and Evaluations

We illustrate the performance of the proposed algorithms in five case studies. Our implementation of the EDSM algorithm is based on the MDI method that is used in the *flexfringe* library [93]. We call the basic algorithm the *Vanilla* algorithm. All the case studies were run on a MacBook Pro with 2.3 GHz Dual-Core Intel Core i5 and 16 GB RAM. Videos of all case studies are available to view[1] .

3.5.1 Learning and Planning for Non-Markovian Tasks

In this case study, we consider the robotic scenario in Figure 1.2. The task is to visit both the school of fish and the shipwreck in any order and always avoid coral reefs. The preference is to visit the shipwreck first. A PDFA representation of this specification is shown in Figure 3.3a.

[1] `https://youtu.be/TU8MhPBDBBs`



(a) Non-Markovian Task (b) 5 demos from [92] (c) Synthesized Plan (d) Synthesized Plan (e) Synthesized Plan for a diagonal-moving robot

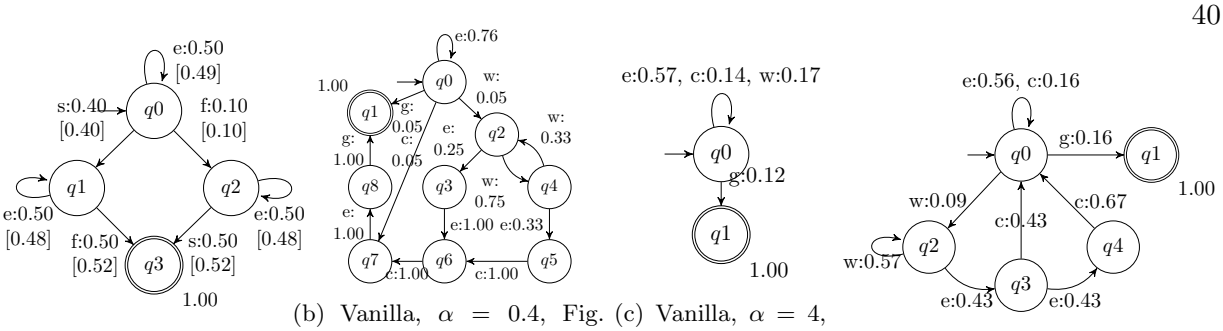
Figure 3.2: Various environments and robots considered for the case studies. (a) Learning and planning for the non-Markovian task. (b) Environment and demonstrations from [92]. (c)-(e) Synthesized plans (shown in red) based on the learned task from (b).

To learn this task, we sampled 1000 traces from this PDFA on the gridworld environment in Figure 1.2. From these demonstrations, the Vanilla algorithm learned a PDFA with the same exact structure as the true PDFA and probabilities within 0.02 of the true values (in square brackets in Fig. 3.3a).

As the PDFA shows, our method correctly learned the non-Markovian task of visiting both the shipwreck and the school of fish in both orders and favors going to the shipwreck first. Using this PDFA, our planner generated the robot trajectory shown in Figure 3.2a (top), which correctly visits the shipwreck first and then the school of fish. Next, we changed the environment by moving the location of the fish to be on the robot's way to the shipwreck as shown in Figure 3.2a (bottom). This figure also shows the synthesized plan in this environment using the same learned PDFA. Notice that the robot is not visiting the shipwreck first due to environmental constraint. Instead, it visits the fish and then the shipwreck, which is also a correct behavior. This generality is the strength of learning the specification rather than learning a policy that is strongly dependent on the environment.

3.5.2 Learning from Small Number of Samples with Safety

In this case study, we consider the environment and five demonstrations depicted in Figure 3.2b taken from [92] to learn the specification in a form of a PDFA as a comparison to the approach in [92], which is based on learning specification formulas. In this gridworld, each color



(a) PDFA for Fig. 3.2a (b) Vanilla, $\alpha = 0.4$, Fig. 3.2b. (c) Vanilla, $\alpha = 4$, Fig. 3.2b. (d) Pre-process, $\alpha = 4$, Fig. 3.2b

Figure 3.3: The task specification and the learned PDFAs for the scenarios in Fig. 3.2a and 3.2b. Each letter represents a symbol with a single atomic proposition s=$\{ship\}$, f=$\{fish\}$, b=$\{blue\}$, c=$\{carpet\}$, g=$\{green\}$, p=$\{purple\}$, and $e = \emptyset$. The termination probability $F_{\mathbb{P}}$ of double-edged states is 1 and 0 at all other states.

represents an object, where orange is **lava**, blue is **water**, yellow is a **drying carpet**, white is an **empty** space, and green is a **charging station**. The task is to reach a charging station. However, the robot should not charge while it is wet. That is, once it gets wet (goes to water), the robot has to dry at the **drying carpet**.

3.5.3 Small number of samples

We first used the Vanilla algorithm with the five demonstrations, which learned the PDFA in Figure 3.3b. Note, in the learned PDFA, region green (**charging station**) must always be observed to reach the final state. This shows that the task of reaching the **charging station** is learned correctly. Next, on the right most branch of the PDFA, **carpet** is always observed when the robot gets wet. Again, the algorithm succeeded in learning the task of visiting **carpet** once the robots gets wet before reaching the **charging station**. One interesting observation is that the PDFA also learned that the robot has to go to the **charging station** in one step after leaving the **carpet**. This is in fact a bias in the samples since every shown demonstration that includes **carpet** has this property. If that is the intention of the demonstrator, then it is a correct behavior. If it is not, then it can be resolved by providing more samples.

Such one-step bias is not apparent in [92] because the "next" operator is not allowed in the syntax of the *language* they consider. In contrary, our method infers over regular language, which

includes the next operator. Furthermore, in [92], it took 95 seconds to learn the specification from just 5 demonstrations whereas ours took less than 0.01 seconds.

3.5.4 Hyperparamter choice and safety

The PDFA in Figure 3.3b is the result of the Vanilla algorithm when the hyperparameter of α is set to 0.4. It is a knob of how aggressive we allow the merges. Higher the value of α is, the smaller the PDFA becomes. If we can tune the hyperparameter correctly, we can get a desirable result as described above. But, if we increase α too much, some merges could induce unsafe behavior. Unwanted merges occur because the algorithm is simply trying to **minimize** the size of the structure. In fact, the question of how to choose a correct value for α is an open problem. For $\alpha = 4$, the learned PDFA from the same demonstrations is shown in Figure 3.3c. This PDFA has no regards for safety and only requires to reach the **charging station**. We can mitigate this problem by embedding safety specification. We define the following safety formula:

$$\varphi_{\text{safe}} = \mathcal{G}\neg\text{lava} \wedge \mathcal{G}(\text{water} \rightarrow \mathcal{X}(\varphi(\neg\text{charge}, \text{carpet}, \text{k}))$$

where φ is a formula recursively defined as: $\varphi(a, b, k) = a \wedge (b \vee \mathcal{X}(\varphi(a, b, k-1)))$ and $\varphi(a, b, 0) = a$, and is read, "visit a for k steps unless b is visited". This formulas requires never going to **lava** and, if the robot enters **water**, it cannot **charge** unless it visits **carpet** or stays in **empty** for k consecutive steps to get dry. We set $k = 10$ in all experiments.

From the same five demonstrations, we now learn PDFAs using the Post-process and Pre-process algorithms with $\alpha = 4$ subject to φ_{safe}. The Post-process algorithm generates a large PDFA with 13 nodes and 36 edges since the safety DFA itself is large (12 nodes and 34 edges). Despite the size, it always guarantees no violation to φ_{safe}. The PDFA generated by the Pre-process algorithm is shown in Figure 3.3d. It is small and correctly embeds both safety and liveness. Further, all the demonstrations are accepted by both learned PDFA. As for probabilities, the average L1 norm error was 1.65×10^{-3} for the Post-process PDFA and 7.42×10^{-5} for the Pre-process PDFA, indicating better performance by the Pre-process algorithm. The larger error in the probabilities of the Post-

process PDFA is due to the composition with the safety DFA, which prunes away the unsafe traces in the learned PDFA, corrupting the learned probability distributions.

Next, we perform a thorough comparison of the learning methods by increasing the number of samples.

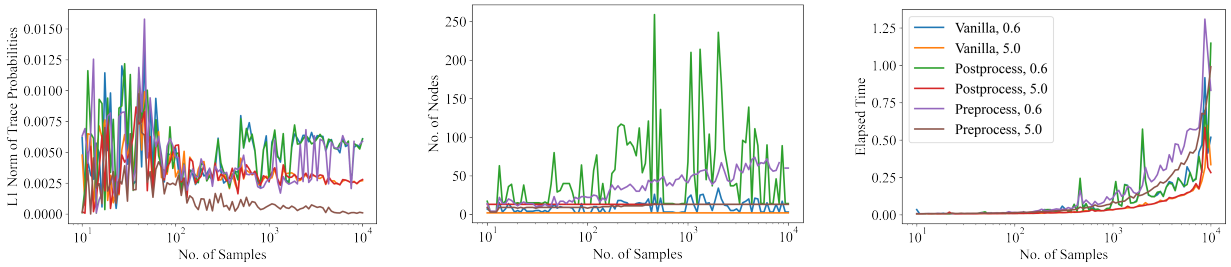
3.5.5 Post-process versus Pre-process Algorithm

Here, the task is similar to the one above, but the goal is to quantitatively analyze and compare the performances of the proposed algorithms as the number of samples increases. We sampled demonstrations randomly from the true PDFA and used Post-process and Pre-process algorithms to learn PDFAs with hyperparameter values of $\alpha = 0.6$ (less aggressive merge) and $\alpha = 5.0$ (aggressive merge) to show the extreme results. We evaluated the resulting PDFAs with respect to the following metrics: L1 norm of the trace probability errors, number of states, and computation times. The results are shown in Figure 3.4 (all the plots share the same legend).

With respect to the error metric, the Pre-process algorithm (purple line for $\alpha = 0.6$ and brown line for $\alpha = 5$) consistently performs the best, followed by the Post-process and then the Vanilla algorithms. It indicates that the Pre-process algorithm is learning the correct distribution over the language of the target PDFA. As for the number of states and computation time metrics, the results indicate that Pre-process algorithm again performs better and faster than the others. From these results, we can say that the Pre-process algorithm is the best performing algorithm. Moreover, its output PDFA does not violate the safety across all the trials (checked but not shown in the figures).

3.5.6 Planning for Various Robots in different Environments

From the learned PDFAs above, we picked one with a small L1 error norm. Then, using this PDFA, we planned for various robots and environments that are different from the one the demonstrations were shown in (see Figure 3.2b). In all the cases, the computed plans correctly meet the requirements and preferences. In the environment in Figure 3.2d, the **lava** forces the robot to



(a) L1 norm error (log scale) (b) Number of nodes (c) Computation time [s]

Figure 3.4: Performance analysis for the proposed algorithm. Plots in (a)-(b) use the same legend as (c).

go to the bottom-right **charging station**. Note that the robot avoids **water** by going through **carpet**, which is the preferred behavior. In Figure 3.2e, we modified the robot's dynamics to only allow diagonal moves. The algorithm is again successful in generating a satisfying plan without violating the specification. Because the specification is independent from any robotic systems and any environments, our framework is robust against the changes in the environment and robot dynamics.

3.5.6.1 Learning and Planning for Manipulation tasks

To show that our method is not limited to mobile robots, we considered a manipulation example in Figure 3.5 (left). The robot is the Franka Emika Panda manipulator with 7 DoF, and the latent task is to build an arch with two cylindrical objects as columns and a rectangular box on top. The abstraction of the robot to a DTS was done according to [46, 45], which ended up with around 20,000 states. The robot was given nine demonstrations: five most preferred (fastest), three mid-length (1.4 times as many actions), and one very bad demonstration (3 times as many actions). A PDFA was learned with $\alpha = 1.8$. The learned PDFA has four states, and planning took 0.036 seconds. The execution of the plan by the robot is shown in Figure 3.5 (middle and right), which shows that robot successfully learned and executed the most preferred method of completing the task.

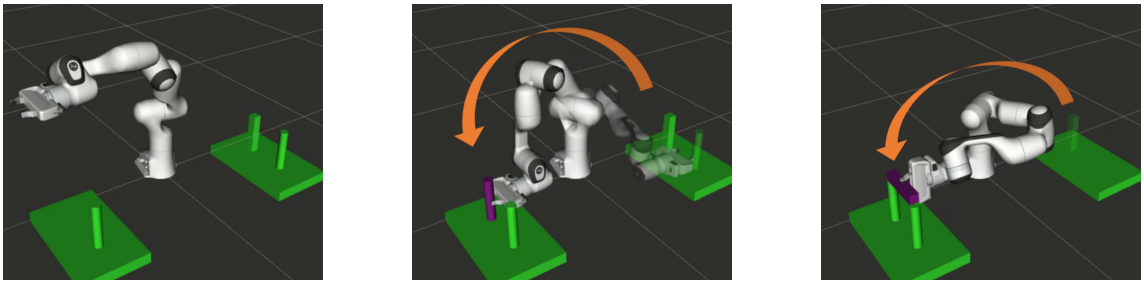


Figure 3.5: Manipulator completing the learned task of building an arch.

3.6 Conclusion

In this work, we presented a new approach to learning specifications from demonstrations in a form of a PDFA. Unlike existing work, this method does not require prior knowledge, is fast, and captures preferences. We showed how safety constraints can be incorporated in the learning algorithm, which significantly improves performance. We also introduced a planning algorithm for specifications learned in this form, which generates the most preferred (most probable) plan. Extensive evaluations illustrate the framework's flexible and capability of robust knowledge transfer to various environments and robots.

Chapter 4

Multi-Objective Reactive Synthesis with PDFAs

This chapter addresses the problem of planning for task completion under uncertainty in dynamic and unpredictable environments, which is crucial for safety-critical applications. Given an inferred Probabilistic Deterministic Finite Automaton (PDFA), a reactive synthesis algorithm is proposed to generate strategies that achieve the task while maximizing the demonstrator's preferences and minimizing the robot's cost. As these objectives often compete, resulting in trade-offs, a method is introduced to compute the set of optimal solutions and generate strategies for each solution. We run several robotic experiments to demonstrate that the algorithm computes all optimal solutions and synthesizes a strategy for each solution. Extensive evaluations illustrate the framework's flexibility and capability of robust knowledge transfer to various environments and robots.

4.1 Introduction

The previous chapter assumed that the environment is static and deterministic. We can model various problems in such an approach, however, in many cases, we encounter dynamically moving objects and unpredictable events. For example, consider an underwater robot in a deep-sea science mission from the previous section. The school of fish may swim from one point to another, or some paths may be blocked due to a strong current. It is important that we can still guarantee the completion of the task under such uncertainties in a safety-critical environment. In the underwater robot example, the strong current may drag the robot onto a rocky shore and may damage the

robot.

This work proposes a reactive synthesis algorithm that generates strategies to achieve the learned PDFA while maximizing the demonstrator's preferences and minimizing the cost of the robot in an environment. These objectives, however, are often competing, which results in a trade-off between the objectives. There could be many optimal solutions, known as the Pareto front. A point in the front can have a higher preference than another but at the same time, the cost could be high. In this work, we introduce a method to compute the Pareto front and generate strategies for each Pareto point. The problem is formulated as a two-player game between the robot (system) and the environment, modeled as an adversarial agent to account for environmental changes. The proposed approach ensures task completion while balancing the demonstrator's preferences and the robot's cost in uncertain and dynamic environments, making it suitable for safety-critical applications.

4.2 Related Work

Planning algorithms with LTL are widely explored [53] and this work also leverages the developed approaches. The common approaches are automata-based approach for discrete states [53], sampling-based motion planning [15] and reinforcement learning for continuous states [17]. They have been extended to synthesize in a reactive environment [35]. In our work, we consider a reactive environment and formulate it as a game between the player and the environment with uncertainties. The difference is that our formulation turns into a multi-objective game rather than just reachability objective. Chen et al. proposed a synthesis algorithm that computes a stochastic strategy for multi-objective stochastic game (for stopping and non-stopping game) [24, 10, 23]. Chatterjee et al. proposed another synthesis algorithm that computes a deterministic strategy for multi-objective (multi-energy, mean-payoff, and parity) non-stopping game [22]. Our solution, in contrast, synthesizes a deterministic strategy for multi-objective (stopping) game using value iteration, which is most similar to [24]. Sastry et al. solves the same problem but they convert the multi-objectives into a single-objective by applying the cost transformation and reduce the problem

to a shortest-path problem, but it is far from optimizing for the exact Pareto front.

4.3 Preliminaries

In this work, we consider a robot that has to interact with a dynamic environment to achieve a task. For example, the robot in Figure 4.1, to fullfill its goal, has to interact with a school of fish that can freely move around. This interaction can be viewed as a game between the robot and the environment (fish), where each player has their own objectives and set of actions. While in reality, this game takes place in a continuous domain, abstractions can be made to represent it as a discrete two-player game. Such an abstraction is commonly used and constructed in formal approaches to both mobile robotics [53, 52, 55] and robotic manipulators [45, 46].

Definition 29 (Two-player Game)**.** *A two-player game is a tuple* $G = (S, A, s_0^G, \delta^G, \Pi, L^G, W^G)$*, where*

- $S = S_R \cup S_E$ *is a finite set of states, where* S_R *and* S_E *are the set of robot and environment states, respectively, and* $S_R \cap S_E = \emptyset$*,*
- $A = A_R \cup A_E$ *is a finite set of controls or actions, where* A_R *and* A_E *are the set of robot and environment actions, respectively,*
- $s_0^G \in S$ *is the initial state,*
- $\delta^G : S \times A \to S$ *is the transition function,*
- Π *is a finite set of atomic propositions (predicates), and*
- $L^G : S \to 2^\Pi$ *is a labeling function that maps each state to the set of predicates that are true at that state.*
- $W^G : S \times S \to \mathbb{R}^m_{\geq 0}$ *is a weighting function that assigns to edge* $(s, s') \in S \times S$ *an* m*-dimensional vector of non-negative weights* $W^G(s, s')$*.*

Game G is also referred to as *multi-objective* two-player game since it allows the encoding of multiple weights to each edge via W^G, i.e., m weights and hence m objectives.

Example 6 (2-player game)**.** *The game abstraction of Figure 4.1 is defined as follows. A state is*

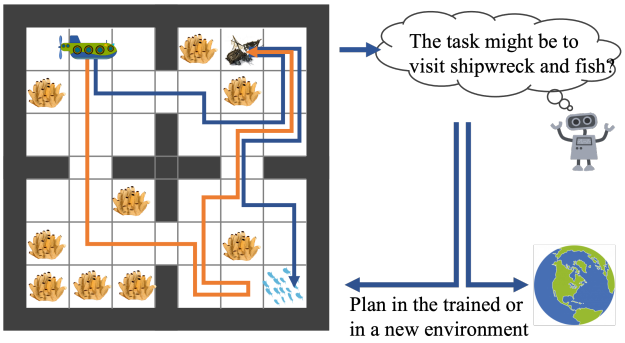


Figure 4.1: Schematic of an autonomous deep-sea science mission. The task for the green underwater vehicle is to visit a school of fish (blue) and shipwreck (brown) while avoiding coral reefs (yellow). Our goal is to infer the underlying task from the demonstrations (the colored path) and synthesize a controller in the trained/new environment that achieves the learned task.

a tuple of vehicle location, fish location and player's turn, i.e., $s = (l_R, l_E, i)$ *where* $l_j = (x_j, y_j)$ *for* $x_j, y_j \in \{1, \ldots, 7\}$ *is the location (coordinate) of agent* $j \in \{R, L\}$*, and* $i \in \{R, L\}$ *indicates whose turn it is. Starting at the initial state* $s_0^G = ((2,1),(7,7),R)$*, the vehicle can take action N (north), S (south), E (east), or W (west) and transition to an adjacent cell at distance of 1 by consuming energy cost of 2, i.e.,* $W^G(s, s') = (1,2)^T$*. Fish can take an action likewise. The set of all possible atomic predicates that can be observed in this environment is* $\Pi = \{\texttt{shipwreck}, \texttt{fish}, \texttt{coral\_reefs}\}$*. When the vehicle and fish are both at location* $l_R = l_E = (7,7)$*, then the observation is* $L^G((l_R, l_E, i)) = \{fish\}$ *where* $i \in \{R, E\}$*.*

The evolution of the game is as follows. At state $s \in S_i$, where $i \in \{R, E\}$, player i picks an action $a \in A_i$, and then, the state of the game evolves to $s' = \delta^G(s, a) \in S_j$, where $j \in \{R, E\}$ (note that j could be equal to i). Next, it is player j's turn to take an action, and the process repeats. This evolution results in a sequence of states called a play, which in turn gives rise to an output trace.

Definition 30 (Play & Trace)**.** *A play* $\mathcal{S} = s_0 s_1 \ldots s_n$ *is a sequence of states such that* $s_0 = s_0^G$*, and for all* $0 \leq k < n$*, there exists* $a_k \in A$ *such that* $s_{k+1} = \delta^G(s_k, a_k)$*. The set of all plays in G is denoted by* Play(G)*. The output trace of* $\mathcal{S}$ *is* $\omega = L^G(\mathcal{S}) = L^G(s_0)L^G(s_1)\ldots L^G(s_n)$ *the sequence of state labels.*

The prefix of play $\mathcal{S}$ at position k is a sequence of states $\mathcal{S}(k) = s_0 s_1 \dots s_k$ from the initial state to the k-th state where $k \leq n$. We say a prefix $\mathcal{S}(k)$ belongs to the robot player if $s_k \in S_R$; otherwise, it belongs to the environment player. The set of prefixes that belongs to the robot and environment player is denoted by $\text{Pref}_R(\mathcal{S}(k))$ and $\text{Pref}_E(\mathcal{S}(k))$, respectively.

Example 7 (Play and Observation Trace). *Starting at $s_0^G = ((2,1),(7,7))$, if the robot takes actions S and W and fish takes action S in turn, then the resulting play is $\mathcal{S} = ((2,1),(7,7),R)$ $((2,2),(7,7),E)$ $((2,2),(7,7),R)$ $((1,2),(7,7),E)$. This induces the observation trace $\rho_{\mathcal{S}} = \lambda\ \lambda\ \lambda$ $\{$`coral-reefs`$\}$.*

Recall that each edge of the game is associated with an m-dimensional weight vector, where each dimension is related to an objective. For instance, the weights could represent energy and distance, and the corresponding objectives could be to minimize the total time and energy required to execute a task. Hence, each play results in a multi-objective total payoff.

Definition 31 (Multi-objective total payoff). *The multi-objective total payoff of play $\mathcal{S} = s_0 \dots s_n$ is the sum of the weight vectors along $\mathcal{S}$, i.e,*

$$\text{TP}(\mathcal{S}) = \sum_{k=0}^{n-1} W^G(s_k, s_{k+1}).$$

In game G, each player picks an action according to a strategy. This choice of action can generally be deterministic or stochastic. In this work, we focus on deterministic strategies since we are interested in the robot behavior in one deployment instead of the expected behavior over multiple deployments.

Definition 32 (Strategy). *A (deterministic) strategy for player $i \in \{R, E\}$ is a function $\sigma_i : S^* S_i \to A_i$ that chooses the next action given a (finite) play that ends in a state in S_i.*

Under robot and environment strategies σ_R and σ_E, the game results in a single play denoted by $\text{Play}(G, \sigma_R, \sigma_E)$. However, σ_E is usually unknown; hence, our goal is to choose a σ_R that achieves robot's objectives against all possible environment strategies, i.e., all possible plays under σ_R, denoted by $\text{Play}(G, \sigma_R, \cdot)$, accomplish the robot's task. This is known as a *winning* strategy.

4.4 Problem Formulation

Problem 2 (Reactive Synthesis Problem)**.** *Given robot-environment model as a two-player game G and a set of demonstrations $\Omega = \{\omega^i\}_{i=1}^{n_\Omega}$ of a (latent) task φ_{task} according to some preference (probability distribution) and a safe LTL formula φ_{safe}, compute a (deterministic) strategy σ_R^* such that every play under σ_R^* achieves φ_{task} and never violates φ_{safe} while minimizing the worst total cost and maximizing the lowest preference, i.e.,*

$$\sigma_R^* = \arg\min_{\sigma_R} \max_{\sigma_E} \Big(\mathrm{TP}\big(\mathrm{Play}(G, \sigma_R, \sigma_E)\big) - P\Big(L^G\big(\mathrm{Play}(G, \sigma_R, \sigma_E)\big)\Big)\Big) \tag{4.1}$$

subject to

$$L^G\big(\mathrm{Play}(G, \sigma_R, \sigma_E)\big) \in \mathcal{L}(\tilde{\mathcal{A}}^{\mathbb{P}}{}_{\varphi_{task}}) \qquad \textit{for all } \sigma_E \tag{4.2}$$

$$L^G\big(\mathrm{Play}(G, \sigma_R, \sigma_E)\big) \models \varphi_{safe} \qquad \textit{for all } \sigma_E \tag{4.3}$$

where $P(\cdot)$ is the probability measure over PDFA $\tilde{\mathcal{A}}^{\mathbb{P}}{}_{\varphi_{task}}$, and the vector optimization in (4.1) *is element-wise.*

Note that minimizing cost and maximizing preferences are often competing objectives, i.e., by optimizing for one, the other becomes suboptimal. In such cases, the interest is in the trade-offs of the objectives. Hence, our goal is to **optimize** for the trade-off between the objectives, which is known as **Pareto optimal**. Since there could be multiple optimal trade-offs, we aim to compute all Pareto optimal points possible under deterministic strategies. Then, given a choice of one, we synthesize the corresponding strategy.

4.5 Approach

4.5.1 Reactive Strategy Synthesis with PDFA

We are interested in synthesizing a strategy to accomplish the learned task in a reactive environment. At first, we reduce the reactive synthesis problem to graph-search. To do so, we take a product between the dynamical environment G and a PDFA $\mathcal{A}^{\mathbb{P}}$ to construct a Game Product

graph that accommodates all possible plays that can achieve the task in G. The problem now turns into finding optimal winning strategies on this graph. We show an algorithm to find the Pareto front on this graph, and additionally present an algorithm to synthesize a strategy for each Pareto solution.

4.5.2 Product Construction

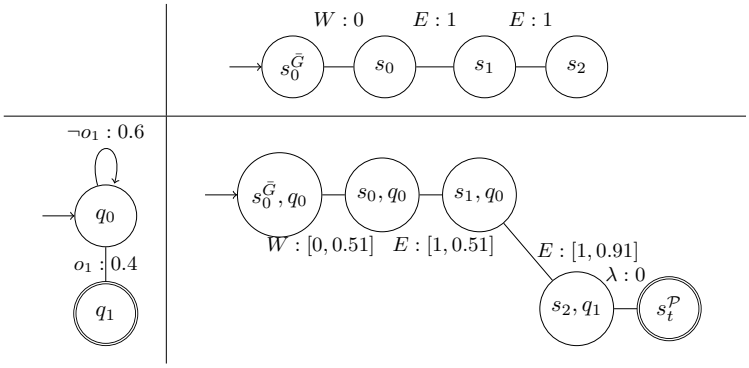


Figure 4.2: Schematic of Product Graph Construction. The left automaton represents a PDFA $\mathcal{A}^{\mathbb{P}}$ and the top automaton represents a game graph G. A product graph (in center) is constructed by taking a product of these two automata. E represents an action "East", λ is a random action, and $o1$ is an observation at $s_2^{\mathcal{P}}$ which satisfies the guard between q_0 and q_1 in $\mathcal{A}^{\mathbb{P}}$.

Our goal is to find winning strategies on the Game Product graph to accomplish the learned task. The problem of finding winning strategies is then boiled down to a reachability problem from the initial state to the accepting state in the Game Product graph.

In Figure 4.2, we show a schematic of the Game Product construction procedure. First, we take a product between a PDFA $\mathcal{A}^{\mathbb{P}}$ (automaton on the left column) and a DTG $\bar{G}$ (automaton on the top row) where $\bar{G}$ is a DTG obtained by augmenting G with a new initial state $s_0^{\bar{G}}$ with a transition to s_0^G by taking any action. Formally, $\bar{G} = (\bar{S}, A, s_0^{\bar{G}}, \delta^{\bar{G}}, \Pi, L^{\bar{G}})$, where $\bar{S} = S \cup \{s_0^{\bar{G}}\}$, and $\delta^{\bar{G}}(s, a) = s_0^G$ if $s = s_0^{\bar{G}}$, otherwise $\delta^{\bar{G}}(s, a) = \delta^G(s, a)\ \forall a \in A$. This augmentation allows to correctly observe the label of s_0^G and assign the edge weights in $\mathcal{P}^G$ according to the probabilities in $\mathcal{A}^{\mathbb{P}}$ through the product rule below. Given learned $\mathcal{A}^{\mathbb{P}} = (Q, \Sigma, q_0, \delta, F, \delta_{\mathbb{P}}, F_{\mathbb{P}})$ and $\bar{G}$, we construct *multi-weighted game product graph* $\mathcal{P}^G = (S^{\mathcal{P}}, A, s_0^{\mathcal{P}}, s_t^{\mathcal{P}}, E_G^{\mathcal{P}}, W_G^{\mathcal{P}})$,

Definition 33 (Game Product)**.** *A game product is a tuple* $\mathcal{P}^G = (S^{\mathcal{P}}, A, s_0^{\mathcal{P}}, s_t^{\mathcal{P}}, E_G^{\mathcal{P}}, W_G^{\mathcal{P}})$*, where*

- $S^{\mathcal{P}} = (\bar{S} \times Q) \cup \{s_t^{\mathcal{P}}\}$ *is a set of states,*
- A *is a finite set of controls or actions,*
- $s_0^{\mathcal{P}} \in S^{\mathcal{P}}$ *is the initial state,*
- $s_t^{\mathcal{P}}$ *is the terminal state,*
- $E_G^{\mathcal{P}} \subseteq S^{\mathcal{P}} \times S^{\mathcal{P}}$ *is a set of edges,*
- $W_G^{\mathcal{P}} : E_G^{\mathcal{P}} \to \mathbb{R}^m_{\geq 0}$ *is a weighting function over edges*

Game Product graph $\mathcal{P}^G$ captures the constraints of both the robot and task along with the environment cost and demonstrator's preference cost. Let $\Lambda^{\mathcal{P}} = s_0^{\mathcal{P}} s_1^{\mathcal{P}} \dots s_t^{\mathcal{P}} = (s_0^G, q_0)(s_0^G, q_1) \dots (s_{n-1}, q_n) s_t^{\mathcal{P}}$ be a path over $\mathcal{P}^G$. The projection of this path (with the deletion of $s_t^{\mathcal{P}}$) onto $\mathcal{A}^{\mathbb{P}}$ is an accepting play with the observation $\omega = \rho_{\mathbb{S}} = L^G(s_0^{\mathcal{P}}) L^G(s_1^{\mathcal{P}}) \dots L^G(s_{n-1}^{\mathcal{P}})$. The projection of $\Lambda^{\mathcal{P}}$ on G is the play that generates the accepting observation ω. The edge weight is a pair of the environment cost and the logarithm of the probabilities of $\mathcal{A}^{\mathbb{P}}$, i.e.,

- Edge $e = ((s, q), (s', q')) \in E_G^{\mathcal{P}}$ if $q' = \delta(q, L^G(s'))$ and $s' = \delta^{\bar{G}}(s, a)$ for some $a \in A$. Then, the multi-objective edge weight is $W_G^{\mathcal{P}}((s, q), (s', q')) = [W^G(s, s'), -\log(\delta_{\mathbb{P}}(q, L^G(s'), q'))]$.
- Edge $e = ((s, q), s_t^{\mathcal{P}})) \in E_G^{\mathcal{P}}$ if $F_{\mathbb{P}}(q) > 0$. Then, its weight $W_G^{\mathcal{P}}((q, s), s_t^{\mathcal{P}}) = [\vec{0}, -\log(F_{\mathbb{P}}(q))]$.

Therefore, to compute a robot strategy that produces an accepting trace in $\mathcal{L}(\mathcal{A}^{\mathbb{P}}) \cap \mathcal{L}(G)$, we need to find a winning strategy on $\mathcal{P}^G$ that reaches the terminal state $s_t^{\mathcal{P}}$.

4.5.3 Pareto Points Computation

In this section, we present a polynomial algorithm (Algorithm 1) to compute the Pareto points on this game using value iteration approach. We introduce an operator F (also known as a greatest fixed operator) and iteratively apply the operator on $\mathcal{P}^G$ until the solution converges, i.e.,

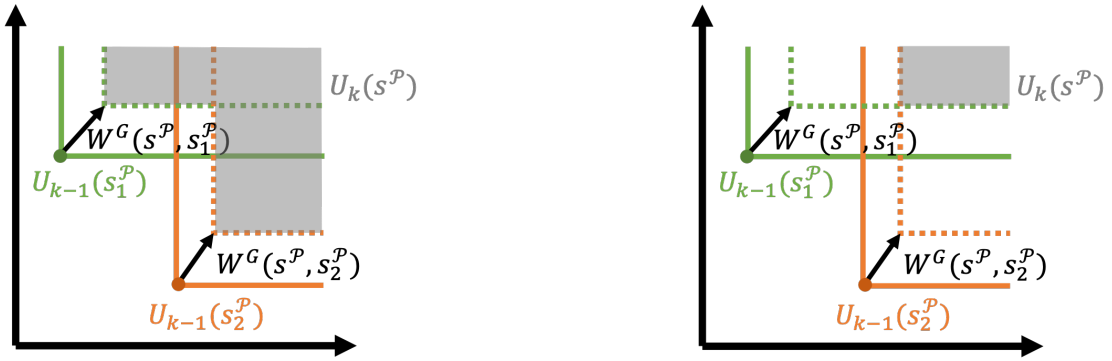
Algorithm 1: Pareto Points Computation

Input : A Game Product Graph $\mathcal{P}^G = (S^{\mathcal{P}}, A, s_0^{\mathcal{P}}, s_t^{\mathcal{P}}, E_G^{\mathcal{P}}, W_G^{\mathcal{P}})$, Convergence margin ϵ
Output: Pareto Points at $s_0^{\mathcal{P}}$
// Initialize Pareto Fronts
$U(s^{\mathcal{P}}) = uwc(\{+\vec{\tilde{\infty}}\})$ where $+\tilde{\infty} \in \mathbb{R}^{\mathrm{m}}$, $\forall \mathrm{s}^{\mathcal{P}} \in \mathrm{S}^{\mathcal{P}}$;
$U(s_t^{\mathcal{P}}) = uwc(\{\vec{0}\})$ where $\tilde{0} \in \mathbb{R}^{\mathrm{m}}$;
while U **and** D **are not converged within some** ϵ **do**
  $U' = copy(U)$;
  $D' = copy(D)$;
  for $s^{\mathcal{P}} \in S^{\mathcal{P}}$ **do**
    if $s^{\mathcal{P}}$ **belongs to the** $Player_R$ **then**
      $U(s^{\mathcal{P}}) = \bigcup_{s^{\mathcal{P}\prime} = E_G^{\mathcal{P}}(s^{\mathcal{P}})} \Big[U'(s^{\mathcal{P}\prime}) + W_G^{\mathcal{P}}(s^{\mathcal{P}}, s^{\mathcal{P}\prime})\Big]$;
    end
    else
      $U(s^{\mathcal{P}}) = \bigcap_{s^{\mathcal{P}\prime} = E_G^{\mathcal{P}}(s^{\mathcal{P}})} \Big[U'(s^{\mathcal{P}\prime}) + W_G^{\mathcal{P}}(s^{\mathcal{P}}, s^{\mathcal{P}\prime})\Big]$;
    end
  end
end
return $Cnr(U(s_0^{\mathcal{P}}))$

$$F(U(s^{\mathcal{P}})) = \begin{cases} \bigcup\limits_{s^{\mathcal{P}\prime} = E_G^{\mathcal{P}}(s^{\mathcal{P}})} \Big[U'(s^{\mathcal{P}\prime}) + W_G^{\mathcal{P}}(s^{\mathcal{P}}, s^{\mathcal{P}\prime})\Big] & (\text{if } s \in S_R \text{ where } (s,q) = s^{\mathcal{P}}) \\ \\ \bigcap\limits_{s^{\mathcal{P}\prime} = E_G^{\mathcal{P}}(s^{\mathcal{P}})} \Big[U'(s^{\mathcal{P}\prime}) + W_G^{\mathcal{P}}(s^{\mathcal{P}}, s^{\mathcal{P}\prime})\Big] & (\text{if } s \in S_E \text{ where } (s,q) = s^{\mathcal{P}}) \end{cases} \tag{4.4}$$

where we initialize the pareto point at the accepting state $s_t^{\mathcal{P}}$ to be zeros and at other states to be infinities (Line 1 of Algorithm 1). We define the *upward closure* of X as $uwc(X) = \{y \mid \exists x, x \in X \text{ and } y \geq x\}$. Formally, $U(s^{\mathcal{P}}) = uwc(\{+\vec{\infty}\})$ where $+\vec{\infty} \in \mathbb{R}^m$ for $s^{\mathcal{P}} \in S^{\mathcal{P}} \backslash \{s_t^{\mathcal{P}}\}$ and $U(s_t^{\mathcal{P}}) = uwc(\{\vec{0}\})$ where $\vec{0} \in \mathbb{R}^m$ where m is the dimension of the edge weight. The operator $\bigcup$ represents the union of the sets and $\bigcap$ represents the intersection of the sets.

The operator F computes the *worst-case minimum* total payoff that the system (robot) can guarantee to achieve from each node. Note that the system player has an option to visit any successor nodes. Therefore, we apply the union operation on set of successor's region at system's nodes to include all possible solutions. On the other hand, we assume that the environment may



(a) Union of two sets

(b) Intersection of two sets

Figure 4.3: The depiction of the two different set operations at node $s_1^{\mathcal{P}}$ at iteration k where $s_1^{\mathcal{P}}$ and $s_2^{\mathcal{P}}$ are its children.

behave adversarial. Therefore, we take the intersection of the successor's region at the environment's node to take the worst case scenario into account, so that the system can always guarantee to maintain the total costs lower than these values. The visualization of the above operation is shown in Figure 4.3. The orange and green regions represent the set of all possible solutions at the successor's nodes. The regions are then expanded by adding the edge weights. The left figure shows the union operation and the right figure shows the intersection operation. Notice that the union operation maintains the largest possible set whereas the intersection operation shrinks the region. Pareto points are the corners of each region that is the minimal set among all solutions.

Example 8. *Figure 4.4 illustrates an example of the product game graph. The values in a square bracket represent edge weights. Those that do not have any edge weights are assumed to be zeros. Initially, pareto point at each node is initialized to be inifinities and zeros at the accepting state* $s_t^{\mathcal{P}}$*. The computation starts from the accepting state and is propagated backwards to the initial state* $s_0^{\bar{G}}, q_0$*. As values propagate, the pareto points at nodes between* $s_4^{\mathcal{P}}$ *and* $s_8^{\mathcal{P}}$ *turn into zeros due to the zero edge weights. At* $s_2^{\mathcal{P}}, q_0$*, the system can only choose either* $s_5^{\mathcal{P}}, q_0$ *or* $s_6^{\mathcal{P}}, q_0$ *which result in the pareto point of (5,5). This is because* $s_4^{\mathcal{P}}$ *belongs to the losing region (infinite loop by the environment). Similarly, at* $s_3^{\mathcal{P}}, q_0$*, the robot can choose either of the successor states. Thus, the pareto points are (1,10) and (10,1). Lastly, at* $s_1^{\mathcal{P}}, q_0$*, the environment can be adversarial at worst*

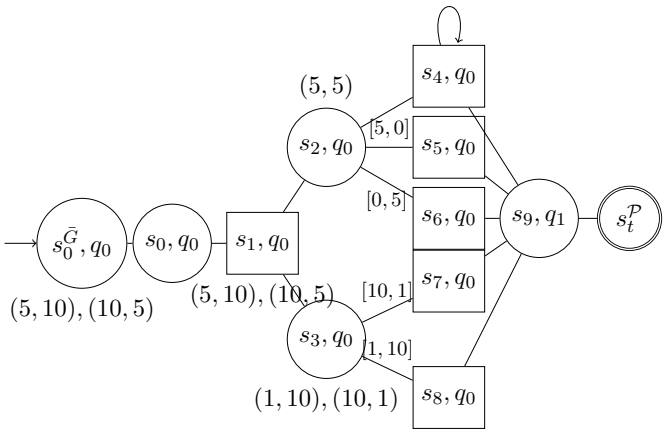


Figure 4.4: Schematic of Pareto Points Computation. Edge weights are shown with square brackets [] and pareto points at nodes are colored in red. Edges that do not have weights shown are assumed to be all zeros.

scenario so we take the intersection of (5,5) and (1,10), (10,1) which is (5, 10) and (10,5).

Next, we show its completeness and complexity of this algorithm in the following sections. Before we show that our algorithm can compute a set of all pareto points in a polynomial time, let us define a set of all solutions at a state $s^{\mathcal{P}}$ after applying the operator F for i times as U_i.

Lemma 7. *Given a game product $\mathcal{P}^G$, applying the operator F on $\mathcal{P}^G$ monotonically decreases the set of total payoffs in U at all nodes, i.e., $U_i(s^{\mathcal{P}}) = F(U_{i-1}(s^{\mathcal{P}})) \preceq U_{i-1}(s^{\mathcal{P}}), \quad \forall s^{\mathcal{P}} \in S^{\mathcal{P}}$*

Proof. This can easily be shown for the system's node. Every operation F on $s^{\mathcal{P}} \in S_R$ expands the set $U(s^{\mathcal{P}})$ if it finds a smaller total payoff path. Thus, we can derive $U_i(s^{\mathcal{P}}) = F(U_{i-1}(s^{\mathcal{P}})) \leq U_{i-1}(s^{\mathcal{P}})$ for the system's node. The set of solutions for the environment's node get initialized only after all of its successor nodes get initialized. Otherwise, the environment's node can always choose to transition to a node with the total payoff of $+\infty$. Because its successors (system's nodes) can only monotonically decrease, the set of all total payoffs at the environment nodes also decreases monotonoically. This is true even if there exists a (non-negative) cycle. In the winning region, there must be a path that breaks the loop and reaches the accepting state. This can only be enforced by the system's nodes. Let $s_k^{\mathcal{P}}$ be a node that has an option to exit the loop, $s_{k+1}^{\mathcal{P}}$ be its successor node that leads to the accepting state and $\mathcal{S}_{\text{loop}} = \{s_k^{\mathcal{P}}, \ldots, s_k^{\mathcal{P}}\}$ be a sequence of states in the loop. Let us assume the smallest total payoff at $s_k^{\mathcal{P}}$ is $U(s_k^{\mathcal{P}}) = U(s_{k+1}^{\mathcal{P}}) + W_G^{\mathcal{P}}(s_k^{\mathcal{P}}, s_{k+1}^{\mathcal{P}})$. Taking a loop only

increases the total payoff of a path $U(s_k^{\mathcal{P}}) + \sum_{s_i \in \mathcal{S}_{\text{loop}}} W_G^{\mathcal{P}}(s_i, s_{i+1}) \geq U(s_k^{\mathcal{P}})$. Since the system's node takes the union of the successor's set, the total payoff stays as it is. □

Proposition 1. *Given the F operator applied on $\mathcal{P}^G$ for i times, its result U_i represents a set of minimum costs that the system can guarantee to perform in i steps from state $s^{\mathcal{P}} \in S^{\mathcal{P}}$.*

Proof. We unroll the game to construct a tree and show that $U_i(s^{\mathcal{P}})$ is the set of costs in a game played in a tree of depth i. Let us consider a node $s^{\mathcal{P}} \in S^{\mathcal{P}}$. Given a root $s^{\mathcal{P}} \in S^{\mathcal{P}}$, the tree can be defined as, $(s^{\mathcal{P}}, \{\mathcal{T}_1, \ldots, \mathcal{T}_{|E_G^{\mathcal{P}}(s^{\mathcal{P}})|}\})$. The tree can be constructed iteratively as,

$$\mathcal{T}_0(s^{\mathcal{P}}) = (s^{\mathcal{P}})$$

$$\mathcal{T}_i(s^{\mathcal{P}}) = (s^{\mathcal{P}}, \{\mathcal{T}_{i-1}(s^{\mathcal{P}'}) \mid (s^{\mathcal{P}}, s^{\mathcal{P}'}) \in E_G^{\mathcal{P}}\}) \quad \forall i \geq 1$$

$\mathcal{T}_i(s^{\mathcal{P}})$ represents all paths that can be played within i steps. If we apply the operator F for i times, all paths in this tree will be explored along with its costs, that is,

$$U_i(s^{\mathcal{P}}) = \begin{cases} \bigcup\limits_{s^{\mathcal{P}'} = E_G^{\mathcal{P}}(s^{\mathcal{P}})} \left[U_{i-1}(s^{\mathcal{P}'}) + W_G^{\mathcal{P}}(s^{\mathcal{P}}, s^{\mathcal{P}'}) \right] & (\text{if } s \in S_R \text{ where } (s,q) = s^{\mathcal{P}} \\ \\ \bigcap\limits_{s^{\mathcal{P}'} = E_G^{\mathcal{P}}(s^{\mathcal{P}})} \left[U_{i-1}(s^{\mathcal{P}'}) + W_G^{\mathcal{P}}(s^{\mathcal{P}}, s^{\mathcal{P}'}) \right] & (\text{if } s \in S_E \text{ where } (s,q) = s^{\mathcal{P}} \end{cases} \tag{4.5}$$

At each step, the system's node takes the union of the successor's sets to update its minimum possible costs whereas the environment's node takes the intersection of the set to compute the maximum costs assuming that they might act adversarial. Through this iteration, it finds the worst minimum costs among all possible paths that the system can guarantee in this game. In case if there exists a loop by the system's agent, the union operation always takes the smaller value that leads to the accepting state opposed to the additional weights enforced by the loop as described in Lemma 7. □

Now the questions are what are the upper bound of the greatest fixed point and is there a convergence guarantee for this algorithm.

Lemma 8. *The maximum number of steps from the initial state $s_0^{\mathcal{P}}$ to the accepting state $s_t^{\mathcal{P}}$ is $|S^{\mathcal{P}}| - 1$.*

Proof. If the initial state is in the winning region, there always exists a path from the initial state to accepting state. Since winning deterministic strategies will never take the same node again, it can visit each node only once. Therefore, the maximum number of steps (edges) the strategy can take is $|S^{\mathcal{P}}| - 1$. □

Proposition 2. *Let the maximum edge weights be $|W^{\mathcal{P}}_{G\,max}|$ in the graph, the maximum costs can be bounded by $U(s^{\mathcal{P}}) \leq (|S^{\mathcal{P}}| - 1) * |W^{\mathcal{P}}_{G\,max}|$.*

Proof. The maximum cost at one step is $|W^{\mathcal{P}}_{G\,max}|$ and the maximum number of steps is $|S^{\mathcal{P}}| - 1$. Therefore, the maximum cost of plays in $\mathcal{P}^G$ is $(|S^{\mathcal{P}}| - 1) * |W^{\mathcal{P}}_{G\,max}|$. □

Proposition 3. *The maximum cost set $U_k(s^{\mathcal{P}})$ converges in $\mathcal{O}(|S^{\mathcal{P}}|(|S^{\mathcal{P}}| + |E^{\mathcal{P}}_G|))$.*

Proof. From Lemma 8, all paths must be reached in $|S^{\mathcal{P}}| - 1$ steps at each node. Therefore, after $|S^{\mathcal{P}}| - 1$ iterations, all paths must be explored and the cost of all paths are taken account. Thus, the maximum cost sets $U_k(s^{\mathcal{P}})$ converge $(|S^{\mathcal{P}}| - 1)^2$. □

Theorem 4 (Pareto Points Computation Complexity)**.** *Given a game product $\mathcal{P}^G$, the pareto points of the game can be computed in polynomial time (within $\mathcal{O}(|S^{\mathcal{P}}|(|S^{\mathcal{P}}| + |E^{\mathcal{P}}_G|))$ steps) using Algorithm 1.*

Proof. of Theorem 4. Let us first suppose the game $\mathcal{P}^G$ has a finite number of nodes and finite weights. By applying Proposition 3, all of the pareto points are computed correctly within $\mathcal{O}(|S^{\mathcal{P}}|(|S^{\mathcal{P}}| + |E^{\mathcal{P}}_G|))$ number of steps. □

In [71], the polygon clipping algorithm is $\mathcal{O}(n \cdot m)$ where n and m are the number of vertices of the polygons, or $\mathcal{O}((k + n) \log n)$ where k is a number of intersections if in 2D.

Algorithm 2: Strategy Synthesis For a pareto point.

Input : A Game Product Graph $\mathcal{P}^G = (S^{\mathcal{P}}, A, s_0^{\mathcal{P}}, s_t^{\mathcal{P}}, E_G^{\mathcal{P}}, W_G^{\mathcal{P}})$, A Pareto Point at the initial state p, and Cost Set U

Output: A Deterministic Strategy σ

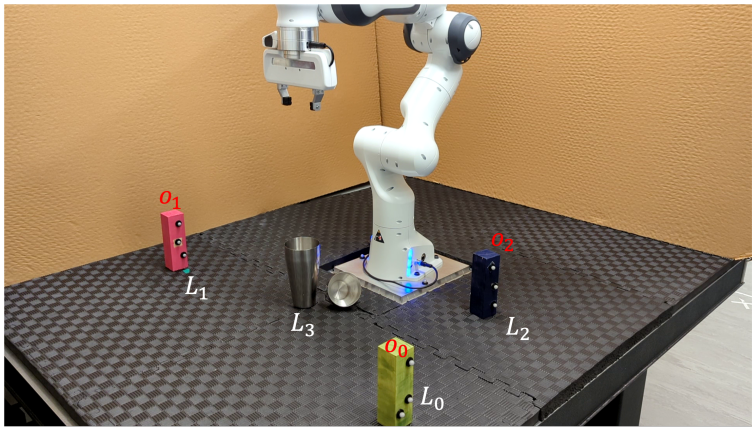
$Q = Queue(s_0^{\mathcal{P}}, p)$;
$Visited = Set(s_0^{\mathcal{P}})$;
$E = List()$;
while Q **is not empty do**
  $s^{\mathcal{P}},\ p \leftarrow Q.pop()$;
  for $p' \in Cnr(U(s_i^{\mathcal{P}}))$ **do**
    if $p - W_G^{\mathcal{P}}(s^{\mathcal{P}}, s_i^{\mathcal{P}}) >= p'$ **then**
      $\sigma(s^{\mathcal{P}}) = s^{\mathcal{P}'}$;
      if $s^{\mathcal{P}'}$ **not in** $Visited$ **then**
        $Q.add(s^{\mathcal{P}'}, p')$;
        $Visited.add(s^{\mathcal{P}'})$;
        $E.append(s^{\mathcal{P}}, s^{\mathcal{P}'})$;
      end
    end
  end
end
return A Deterministic Strategy from E**.**

4.5.4 Strategy Synthesis

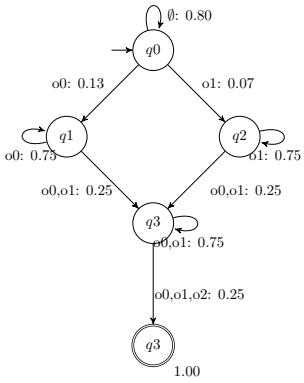
In this section, we show that the strategy for a selected pareto point can be computed in linear time in the number of nodes. The algorithm is shown in Algorithm 2. Note that there must be at least a strategy for each pareto point. Otherwise, the pareto point could not be computed at first place. To find a strategy, we need to progressively find transitions such that the remaining cost (the initial cost minus the transition cost) can be maintained greater or equal to 0.

On Line 2, we start by adding the initial state and the selected pareto point $p \in \mathcal{P}$ to a queue. On Line 2, we choose a successor node $s^{\mathcal{P}}$ such that the successor's pareto point p' remains inside the current node's cost set, $p - W_G^{\mathcal{P}}(s^{\mathcal{P}}, s_i^{\mathcal{P}})$. Once the strategy graph is obtained, we then run a backward reachability analysis and only keep the states that are reachable from the accepting state. This prevents from cycling in a loop in the strategy graph.

Proposition 4. *Line 2 chooses the successor that leads to the accepting state.*



(a) Experimental Setup



(b) Learned PDFA

Figure 4.5: Cocktail Making Experiment

Lemma 9. *The successor in the loop and the successor node that leads to the accepting state have the same pareto point only if the loop is a zero weight cycle.*

Proof. From the proof of Lemma 7, taking a loop can only increase the cost of a path $U(s_k^{\mathcal{P}}) + \sum_{s_i \in \mathcal{S}_{\text{loop}}} W_G^{\mathcal{P}}(s_i, s_{i+1}) \geq U(s_k^{\mathcal{P}})$. The pareto point will be exactly the same if sum of the weights is equal to zero. □

Theorem 5 (Strategy Synthesis Complexity)**.** $\mathcal{P}'$ *computes a winning strategy that always maintains its path cost within the selected pareto point* $p \in \mathcal{P}$ *can be computed in linear time.*

Proof. From Line 2, the resulting strategy chooses a successor that always keep the pareto point within the predecessor's cost set. Since, it only visits each node once, it can be computed in $|S^{\mathcal{P}}|$. □

4.6 Case Studies and Evaluations

In this section, we illustrate the performance of the proposed algorithms in two case studies. Through the case studies, we confirm that our approach answers all the problems formulated in Problem 2. Specifically, we are going to answer these questions in each case study.

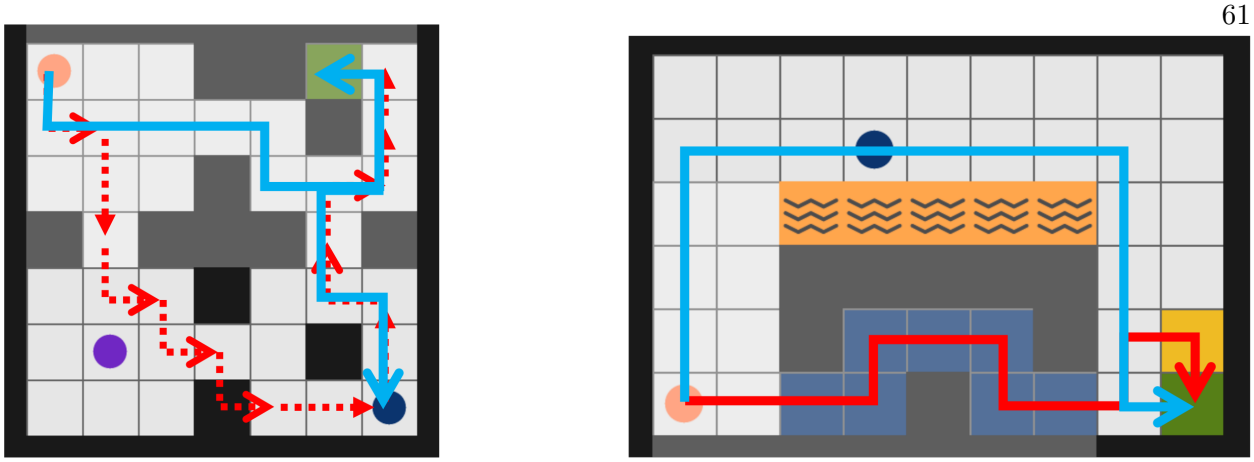
(1) Can the algorithm guarantee the completion of a task in a reactive environment, maximize preferences, and minimize the game cost at the same time?

(2) Can the algorithm be applied to various robots including the real-world robots and achieve all the things above?

4.6.1 Planning in Dynamic Environments

We can also apply the learned PDFA to synthesize a strategy in a dynamic environment. Let us recall the task of visiting both the school of fish and the shipwreck. We now assume the school of fish can dynamically move around freely. The new example is shown in Figure 4.6a. The red underwater vehicle has to catch the blue fish and find the green shipwreck while avoiding purple dynamic vehicle that can only move within the left bottom space. Moreover, we added an option for the robot to take one or two steps at a time. We set the example such that the higher number of steps cost more energy to make the example more interesting. Taking two-steps at time will guarantee to catch the fish in less number of steps but will cost more energy. Our algorithm reveals that there exists six pareto optimal costs that the robot can guarantee. One of the pareto optimal strategies keeps the total payoff within 40.0 for the energy cost and 40.72 for the preference cost. Its path is drawn as a red path in Figure 4.6a. Another optimal strategy can keep the total cost within $(59.0, 20.01)$ and its play is drawn as a blue path in Figure 4.6a. We simulated 1000 plays on this game. All plays successfully completed the task and their total payoffs are bounded by either of the pareto points $\forall \mathcal{S} \in \text{Play}(G)$, $\exists \mathcal{S}$, $TP(\mathcal{S}) \leq p$, $p \in \mathcal{P}$. Regardless of the environment's behavior, the strategy computed by our algorithm can guarantee a completion of the task and maintain the total payoff within the bounds.

We show that our algorithm can avoid another agent and complete the task in various ways. Recall the example of robot reaching the charging station. We extended the environment as in Figure 4.6b such that the system agent can choose two different paths for reaching the green charging station. Imagine the robot being an autonomous driving car and the blue agent as a



(a) Fish and Shipwreck

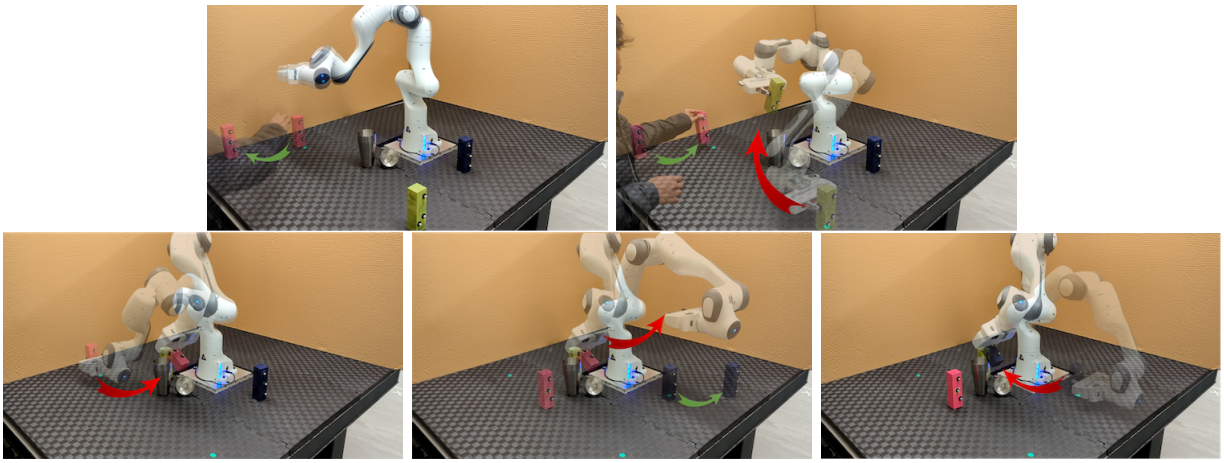
(b) Charging Station

Figure 4.6: Dynamic MiniGrid Environments.

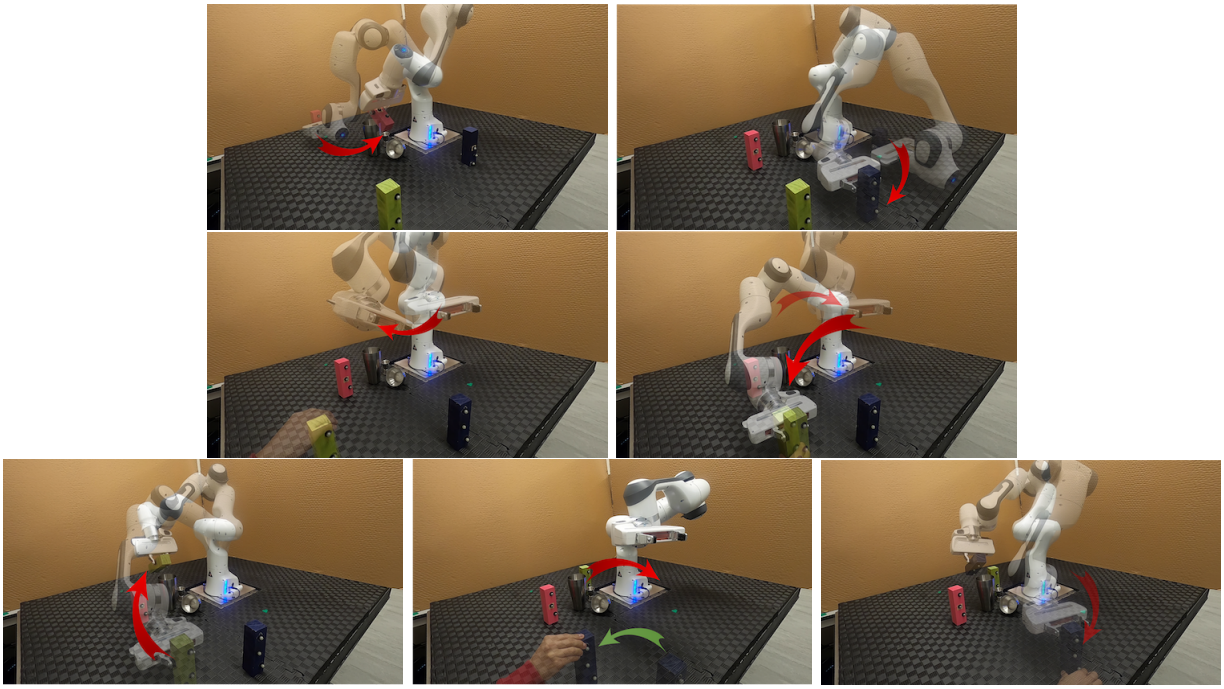
pedestrian. The car has to avoid the pedestrian and reach the charging station or it can go through **water** and get dried at **carpet** to avoid the pedestrian. We assume that the pedestrian can only walk around in the top two rows and the robot can take one or three steps to avoid conflicts with the pedestrian. Our algorithm found eight pareto optimals. We show two distinct plays that can be simulated by the strategies in Figure 4.6b. As describe above, the red vehicle successfully chooses the path above for minimizing the preference cost (47, 5.24) or the path below for minimizing the energy cost (12, 15.39). Paths for other pareto points are similar to these two but its behavior changes based on how many steps to take. We presented a reactive strategy in a dynamic environment while respecting the learned preferences.

4.6.2 Cocktail Making Example

To demonstrate the power of our reactive algorithm, we show a cocktail making example in a human-robot collaboration setting. Imagine a robot and a human making cocktail individually next to each other in a bar kitchen. The setup of the experiment is shown in Figure 4.5a. The blocks represent liquors and they can be moved between predefined locations L_0, L_1, L_2, L_3 by taking actions "transitGrasp", "transferRelease". The human operator can either intervene and move other object at any time or wait until the robot takes an action, however, it cannot intervene



(a) A play with the worst distance cost of [**4.41**, 7.88] by Strategy 0



(b) A play with the worst preference cost of [4.35, **12.65**] by Strategy 0

Figure 4.7: Plays by Strategy σ_0 with the maximum total payoffs.

while the object is in the robot's hand nor intervene the same object twice. Once the human intervenes an object, the object must be returned after two robot action steps. The edge weights are automatically assigned based on the distance between each location.

The robot's task is to pour liquor $o_0, \ldots, o_2$ into a shaker while human may intervene to borrow some of the liquors one at a time. The underlying task is to first pour o_0 and o_1 in any order and o_2 at the end. The most preferred way of picking objects is $o_0 \rightarrow o_1 \rightarrow o_2$. We sampled 3 demonstrations and learned a PDFA given a safety requirement of "never observe o_2 before o_0 and o_1", i.e., $\mathcal{G}(o2 \rightarrow \mathcal{X}(\neg o1 \wedge \neg o2)$. The learned PDFA is presented in Figure 4.5b.

Our algorithm computed two pareto points [distance cost, preference cost] = (4.41, 12.65) and (4.68, 10.88) along with strategies for each pareto point (σ_0 and σ_1 respectively). Without any human interventions, the strategies generate plays that keep the total payoffs of the plays within the bound of each pareto point. For example, strategy σ_0 pours o_1, o_0, o_2 in order and performs the cost of [3.44, 7.83], whereas strategy σ_1 pours o_0, o_1, o_2 and performs the cost of [3.45 7.21]. Since o_1 is closer (less cost distance) and less preferred to o_0, the total payoff of strategy σ_0 achieves smaller distance cost and higher preference cost compared to that of strategy σ_1. Notice that the total payoffs are clearly within the bound of the pareto points and that the robot never tries to pick up o_2 until others are picked up due to the safety requirement.

In the case of human interventions, these strategies can still ensure the worst-case total payoffs to be within the bound of the pareto points. To achieve the worst-case cost, human can maximize either of the total payoffs for reaching the maximum value of a pareto point. In Figure 4.7, we show all the plays by strategy σ_0 that performed the worst total payoffs. In this work, we only show for σ_0 because, the plays by strategy σ_1 is very similar to that of strategy σ_0 except the order of o_0 and o_1 is flipped. All the total payoffs are guaranteed to remain in the bound of the pareto points.

In both plays by strategy σ_0, the robot first moves to L_1 to pick up o_1, but in Figure 4.7a, the human intervenes before the robot grasp o_1. Next, because human cannot intervene other object while intervening one of the objects, the robot knows it won't be intervened so it picks up o_0 and pours it into the shaker at L_3. Once human returns the object, the robot then goes back and

pour o_1 into the shaker. Lastly, the robot tries to pick up o_2 but human intervenes again. Robot waits until o_2 is returned and pour the last liquor into the shaker. In Figure 4.7b, the robot first pours o_1 into the shaker and then, interestingly, it transits to o_2 to relocate it close to L_0. It seems surprising, but this strategy prepares for minimizing the cost of traveling between L_0 and L_2 back and forth in the case of human intervention at o_0. In fact, when the robot tried to pick up o_0, human interevened and the robot had to move back to L_2 until o_0 was returned. Lastly, the robot pours o_0 and o_2 in the shaker in order.

As seen in these plays, with our algorithm, the demonstrators/users can always ensure the completion of the task and choose the behavior they want depending on their objective.

4.7 Conclusion

In this work, we presented a new approach to learning specifications from demonstrations in a form of a PDFA. Unlike existing work, this method does not require prior knowledge, is fast, and captures preferences. We showed how safety constraints can be incorporated in the learning algorithm, which significantly improves performance. We also introduced a planning algorithm for specifications learned in this form, which generates the most preferred (most probable) plan. Extensive evaluations illustrate the framework's flexibility and capability of robust knowledge transfer to various environments and robots.

Chapter 5

Timed Partial Order (TPO) Mining

This chapter proposes the model of timed partial orders (TPOs) for specifying workflow schedules, especially for modeling manufacturing processes. TPOs integrate partial orders over events in a workflow, specifying "happens-before" relations, with timing constraints specified using guards and resets on clocks – an idea borrowed from timed-automata specifications. TPOs naturally allow us to capture event ordering, along with a restricted but useful class of timing relationships.

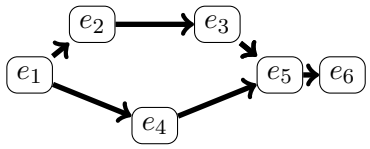
Next, we consider the problem of mining TPO schedules from workflow logs, which include events along with their time stamps. We demonstrate a relationship between formulating TPOs and the graph-coloring problem, and present an algorithm for learning TPOs with correctness guarantees. The approach first mines partial order specification and timing constraints from the data, translating these constraints into TPO specifications. In the TPO model, we allow setting/resetting clocks to enforce the timing constraints just as in the timed-automaton model [2]. To this end, it is desirable to minimize the number of clocks used in order to control for model complexity. However, finding the minimum number of clocks is NP-hard. This work examines heuristic algorithms for solving this problem.

We demonstrate our approach on synthetic datasets, including two datasets inspired by real-life applications of aircraft turnaround and gameplay videos of the Overcooked computer game. Our TPO mining algorithm can infer TPOs involving hundreds of events from thousands of data-points within a few seconds. We show that the resulting TPOs provide useful insights into the dependencies and timing constraints for workflows.

5.1 Introduction

Workflows appear in diverse areas, including business processes [1, 29], software engineering [26], and factory pipelines [25]. The individual events in a workflow, such as the start/end of a particular task or the achievement of an intermediate sub-goal, are ordered according to a strict partial order that specifies that some event e_i always **happens before** another event e_j. Such partial orders have been used to represent plans in classic AI planning algorithms [75]. Beyond partial orders, we often have timing constraints between events in a workflow that place bounds on the time when an individual event occurs or the time elapsed between two events in the workflow. In this work, we tackle the key problems of specifying these timing constraints in a succinct manner and mining such schedules from data that consists of sequences of time-stamped events, each representing an execution of the workflow. Such a specification enables us to implement workflow monitoring algorithms, understand sources of timing uncertainties in workflows and optimize the workflow in order to realize cost savings. Beyond monitoring, timing models enable optimal scheduling in real-time by observing the timings of in-progress and completed tasks in order to plan future tasks on-the-fly.

In this work, we introduce a model called Timed Partial Order (TPO), and propose an algorithm to infer a TPO from timestamped event sequences. The TPO model integrates partial orders with timing constraints that are expressed with clocks, whose idea comes from timed automata specifications [2]. The clocks act as timers that express bounds on the time intervals between pairs of events. We first introduce the TPO model and present an analysis of its expressivity. We show that TPOs can succinctly capture a complex set of timing constraints by checking assertions over clocks and selectively resetting these clocks when certain events happen. In particular, we identify structural constraints that yield a restricted class called **race-free** TPOs. We show that race-free TPOs correspond precisely to difference constraints involving time intervals between events. We demonstrate an algorithm that translates a system of explicit constraints on the timing between events into a race-free TPO specification. Next, we solve the problem of mining race-free TPOs



e_1	Car Arrived
e_2	Add Glue Primer
e_3	Grip/Lift Window
e_4	Clean Dust
e_5	Place Window
e_6	Release Grippers

Figure 5.1: Partial Order for a windshield installation task in an automobile manufacturing facility. Events $e_1, \ldots, e_5$ represent events such as "car arrived" (e_1) or the commencement of various tasks such as "clean dust" (e_4).

from timestamped event sequences. Our approach first mines partial order specification and timing constraints from the data, translating these constraints into race-free TPO specifications. In our TPO model, we allow setting/resetting clocks to enforce the timing constraints just as in the timed-automaton model [2]. This allows a procedural approach to monitoring or enforcing these constraints during task execution. To this end, it is desirable to minimize the number of clocks used in order to control for model complexity. However, finding the minimum number of clocks is NP-hard. We examine heuristic algorithms for solving this problem.

We present an evaluation of our approach on a combination of synthetic benchmarks to show that our algorithm can process large numbers of event sequences in a matter of seconds, and provides succinct TPO specifications in terms of the number of clocks needed. Next, we demonstrate our approach on two examples inspired by real-life applications: a model of events involved in the workflow for commercial aircraft turnaround and an analysis of the multiplayer computer game Overcooked, as played by beginner and expert players. In both of these examples, we use TPO mining to produce specifications that can yield useful insights about the nature of the workflows in question.

Example 9. *Figure 5.1 shows the set of events that define the (simplified) process of placing a windshield on a car in an automobile manufacturing facility. The events are described along with a directed graph that represents the partial order between events. For instance, the edge from* e_2 *to* e_3 *specifies that the event* e_2 *must always precede* e_3 *for a (successful) windshield installation.*

However, like many tasks in an assembly line, there are timing constraints that must be respected. Some of the timing constraints for the windshield installation task are summarized in the table below:

ID	*Interval*	*Cons.*	*Remark*
C_1 :	$e_2 \to e_5$	$\leq 40s$	*Glue appl. to place window*
C_2 :	$e_5 \to e_6$	$\geq 30s$	*Min. glue setting time*
C_3 :	$e_1 \to e_4$	$\leq 5s$	*Max. time to start cleaning*
C_4 :	$e_1 \to e_6$	$\leq 100s$	*Max. end-to-end time.*

C_1 enforces that once the glue primer is applied to the window (e_2), the window must be placed on the car (e_5) within 40 *seconds. Similarly, C_2 states that once the window is placed on the car, a wait of at least* 30 *seconds is required for the glue to set before releasing the grippers.*

In this work, we first describe the model of **timed partial orders**, which combines partial orders between events with ideas from timed automata to represent timing constraints. Next, we show how, given a dataset of time-stamped events, we can mine the timed partial order specification.

5.2 Related Work

Learning a task from log data is known as *Workflow Mining* or *Process Mining*. Most approaches to process mining focus on modeling the order in which tasks are performed, but do not capture timing constraints. As such, process mining has been widely used in the industry [88]. It is supported by commercial tools such as Disco, Celonis, and Process Gold; and open source tools such as ProM, Apromore and pm4py [14]. Under the hood, these tools implement algorithms such as the α-algorithm that outputs a Petri Net [69], $\alpha+$ algorithm that can handle loops [87], and an algorithm that can handle duplicate events [48]. Various types of α-algorithm were introduced [90] to overcome some of its limitations. Other approaches are also introduced such as region-based approaches [89, 19] that can express more complex control-flow structures and heuristic mining [97], fuzzy mining [42], query-based mining [34] that can handle incomplete data and genetic

process mining [32] that can handle noise. The problem of learning timing constraints has been studied, as well. Berlingerio et al. [13] focuses on inferring "typical transition time" between two events by counting the number of steps between them. Sciavicco et al. [78] mine **Conditional Simple Temporal Network with Uncertainty and Decisions** (CSTNUD) models from log data. CSTNUD are temporal networks (Cf. [33]), wherein timing differences between events are represented as "durations". These durations can also be viewed as a single clock that resets at every transition. This representation is frequently used in works such as Verwer et al. [94]. However, their expression is limited to a single clock, whereas our approach uses multiple clocks that need not be reset at each transition. Multiple clocks are necessary in order to capture more complex timing constraints that are frequently seen in our examples and case-studies. Moreover, their method assumes that dependencies between events are manually provided in the log, e.g., event E happens 5 seconds after A, and only depends on **one** event. In contrast, our method can mine the structure at the same time and allows events to depend on multiple events. Moreover, we enforce timing constraints with "clocks" for easier interpretations and faster computation when planning.

Automata such as timed-automata [2] can be used to model workflow schedules. Researchers have extended automata learning techniques to learn timed automata that can capture timing constraints. For instance, Verwer et al. [94] extended the Evidence-Driven State Merging (EDSM)-based algorithms originally proposed by Gold [40] to learn from data with timestamps and estimate a real-time timed automaton whose edges are labeled with time duration. Although the algorithm is fast, it can only infer simple time constraints. An et al. [4] also extended the L*-based approach of Angluin [5], and Tappler et al. [86] formulated the problem such that it can be solved by Satisfiability Modulo Theories (SMT). However, the former assumes a perfect oracle, and the latter is often very slow due to the nature of the SMT solver, which often leads to exponential time. The genetic algorithm-based approach [85] is fast and gives a good solution if it finds one, but has no optimality guarantees. All these approaches can estimate some types of Timed Automata, which are very expressive models, but are disadvantageous due to the fundamental difficulty of learning timed automata from trace data.

5.3 Time Partial Order

We model workflows as a timed sequence involving a fixed number of **events**. These events may include the start/finish of a given task, or the achievement of a certain physical condition, e.g., the temperature of the water has exceeded 100°C. We assume that the set of events are fixed **a priori**. Furthermore, we assume that repetitions of events are **disambiguated** by giving them unique labels.

Let $\mathcal{E} = \{e_1, \ldots, e_n\}$ be the set of events. A given **run** of the workflow is defined by a **timed trace** $\tau : \mathcal{E} \to \mathbb{R}_{\geq 0}$ that maps each event with a non-negative timestamp, i.e.,

$$\tau = \{e_1 \mapsto t_1, e_2 \mapsto t_2, \ldots, e_n \mapsto t_n\},$$

wherein $t_i \geq 0$ denotes the timestamp for event $e_i \in \mathcal{E}$. With an abuse of notation, we also use τ as a set.

A timed-trace τ induces an ordering over the events according to increasing time stamps, and thus may also be viewed as a sequence: $(\omega_1, t^{(1)}), \ldots, (\omega_n, t^{(n)})$, wherein each $\omega_i \in \mathcal{E}$ denotes a unique i^{th} event in the trace with corresponding time stamp $t^{(i)}$ and furthermore, $t^{(1)} < t^{(2)} < \cdots < t^{(n)}$. Below, we formulate models for the possible timed traces corresponding to runs of a workflow.

A *(strict) Partial Order* (PO) P is a relation $\prec$ on a set $\mathcal{E}$ that is irreflexive, asymmetric, and transitive. We write $e_i \preceq e_j$ if $e_i \prec e_j$ or $i = j$. If $e_i \prec e_j$ holds, then for any timed trace τ we will require that $\tau(e_i) \leq \tau(e_j)$. We now describe the model of timed POs.

Definition 34 (Timed Partial Order)**.** *A timed partial order (TPO) is specified by a directed-acyclic graph (DAG) $S : (\mathcal{E}, \prec)$ describing a strict partial order over $\mathcal{E}$ augmented with the following:*

*(1) A finite set of **clocks** $C = \{c_1, \ldots, c_m\}$,*

*(2) A **guard map** g that maps each event e_i to a guard condition, which is a conjunction of the form $g(e_i) : \bigwedge_{j=1}^{n_i} c_j \bowtie a_j$ wherein $c_j \in C$ denotes a clock, $\bowtie \in \{\leq, \geq\}$, and $a_j \in \mathbb{R}_{\geq 0}$ is a non-negative constant,*

(3) *A* ***reset map*** $R : \mathcal{E} \to 2^C$ *that associates each event* e_i *with a subset of clocks* $R(e_i) \subseteq C$ *that are to be reset to* 0 *whenever event* e_i *is encountered.*

A valuation $\nu : C \to \mathbb{R}_{\geq 0}$ assigns each clock $c_i \in C$ to a non-negative number $\nu(c_i)$. A given valuation ν can be advanced in time by a fixed $\delta \geq 0$ to yield a new valuation $\nu' := \nu \oplus \delta$ such that $\nu'(c_j) = \nu(c_j) + \delta$ for all $c_j \in C$. Likewise, given a valuation ν and a subset of clocks $\hat{C} \subseteq C$, we denote the valuation $\nu' := \mathsf{reset}(\nu, \hat{C})$ as that obtained by setting each clock $c \in \hat{C}$ to be 0:

$$\nu'(c) = \begin{cases} 0 & c \in \hat{C} \\ \nu(c) & c \notin \hat{C} \end{cases}$$

. Let ν_0 represent a fixed special initial valuation wherein $\nu_0(c_j) = 0$ for all clocks and let $t^{(0)} = 0$. Timestamps represent global time since the inception of the process whereas clocks measure the time elapsed since their last reset.

Definition 35 (Semantics of Timed Partial Orders)**.** *A* ***run*** *of a timed-partial order is a sequence of triples*

$$\rho : (\sigma_1, t^{(1)}, \nu_1), \ldots, (\sigma_n, t^{(n)}, \nu_n)\,,$$

wherein, each $\sigma_i \in \mathcal{E}$, $\sigma_i \neq \sigma_j$ *for* $i \neq j$, *and*

each ν_j *is a valuation of clocks* C.

(1) *Time stamps are non-decreasing:* $t^{(1)} \leq t^{(2)} \leq \cdots \leq t^{(n)}$. *Let* $\Delta t^{(j)}$ *denote the difference* $t^{(j)} - t^{(j-1)}$ *for* $j \in \{1, \ldots, n\}$ *(note that* $t^{(0)} = 0$*).*

(2) *The sequence* $\sigma_1, \ldots, \sigma_n$ *is a linearization of the partial order* $\prec$*: if* $\sigma_a \prec \sigma_b$ *holds then* $a < b$.

(3) *For each* $j \in \{1, \ldots, n\}$, *the valuation given by* $\nu_{j-1} \oplus \Delta t^{(j)}$ *satisfies the guard condition* $g(\sigma_j)$.

(4) *The valuation* ν_j *must equal* $\mathsf{reset}(\nu_{j-1} + \Delta t^{(j)}, R(\sigma_j))$, *i.e, we allow time* $\Delta t^{(j)}$ *to elapse and then reset the clocks in* $R(\sigma_j)$.

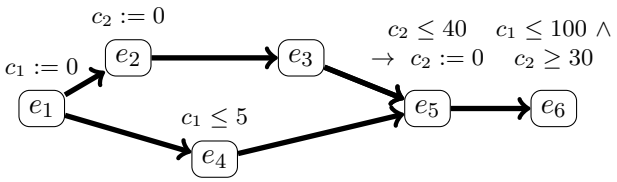


Figure 5.2: TPO for the car windshield installation workflow.

A timed trace τ viewed as a sequence $(\sigma_1, t^{(1)}), \ldots, (\sigma_n, t^{(n)})$ is **compatible** with a timed partial order specification iff there exists a run of the form $(\sigma_1, t^{(1)}, \nu_1), \ldots, (\sigma_n, t^{(n)}, \nu_n)$.

Example 10. *Figure 5.2 shows the TPO specification for the windshield installation task in Example 9. Two clocks c_1, c_2 are used to enforce the constraints $C_1, \ldots, C_4$ from that example. First, we use clock c_1 which is reset to zero at the very beginning when event e_1 occurs. The guard $c_1 \leq 5$ at event e_4 checks that the timing interval between e_1 and e_4 is at most 5 time units (C_3). Likewise, the guard $c_1 \leq 100$ at event e_6 ensures constraint C_4. The clock c_2 is reset first when event e_2 occurs. The guard $c_2 \leq 40$ associated with e_5 enforces constraint C_1. The clock is then reset to zero as part of the same event and then further the guard $c_2 \geq 30$ associated with e_6 ensures constraint C_2.*

5.3.1 Expressivity of TPOs

We will now examine the expressivity of TPOs. In general, TPOs enforce timing constraints using clocks. We derive key insights into the nature of these timing constraints. Furthermore, we define useful structural restrictions to TPOs that make the problem of reasoning about their behavior easier. This will pave the way for the TPO mining algorithm that will be presented in the subsequent section.

Example 11 (Limits on Expressivity)**.** *Suppose, for the windshield installation task in Example 9, we wish to add a constraint – the time taken to clean the windshield (time elapsed between events e_4 and e_5) must be less than the time taken to add the glue primer to the window (time elapsed between events e_2 and e_3). Such a constraint compares the intervals between two sets of events. As*

such, this will not be expressible in the formalism of TPOs. The reason (in part) is that we disallow the guards to compare the values of clocks.

For the remainder of this section, let us fix a TPO with events $\mathcal{E} = \{e_1, \ldots, e_n\}$ and partial order relation $\prec$, clocks $C = \{c_1, \ldots, c_m\}$, guard map g and resets R. We will represent a "generic" timed trace $\tau : \{e_1 \mapsto t_1, \ldots, e_n \mapsto t_n\}$ as a vector $(t_1, \ldots, t_n) \in \mathbb{R}^n$, wherein t_i denotes the time at which the event e_i occurs.

Definition 36 (Constraints representing a TPO)**.** *An assertion $\varphi^{TPO}[t_1, \ldots, t_n]$ involving $t_1, \ldots, t_n$ represents a TPO iff every timed trace $\tau : \{e_1 \mapsto t_1, \ldots, e_n \mapsto t_n\}$ compatible with the TPO satisfies φ^{TPO}, and conversely, every timed trace satisfying φ^{TPO} is compatible with the TPO.*

For the partial order $\prec$, let ψ_{POR} denote the assertion

$$\psi_{POR} : \bigwedge_{e_i \prec e_j} t_j - t_i \geq 0 \ \wedge \ \bigwedge_{j=1}^{n} t_j \geq 0 \tag{5.1}$$

expressing the timing constraints of $\prec$.

Example 12. *The TPO in Ex. 10 is represented by constraints:*

$$\left(\begin{array}{c} t_1 \leq t_2 \leq t_3 \leq t_5 \ \wedge \ t_1 \leq t_4 \leq t_5 \leq t_6 \ \wedge \ t_5 - t_2 \leq 40 \\ \wedge \ t_6 - t_5 \geq 30 \ \wedge t_4 - t_1 \leq 5 \ \wedge \ t_6 - t_1 \leq 100 \end{array} \right)$$

The first row represents the relation $\prec$ from the partial order whereas the last row represents the timing constraints $C_1, \ldots, C_4$ discussed in Example 9. Note that (a) TPOs provide a more succinct representation of the timing constraints; and (b) they also specify a procedure to monitor the timed-trace by maintaining some clocks, checking guards on them upon events and resetting the clocks as specified by the TPO. Timing constraints on the other hand require us to potentially store the times of each and every event in the trace.

Let us consider another example below.

Example 13. *Consider a TPO over 3 events e_1, e_2, e_3 with the $\prec = \emptyset$. In other words, there is no fixed "happens-before" order between these events.*

$$c_1 \leq 1 \underset{\boxed{e_1}}{\rightarrow} c_1 := 0 \qquad c_1 \leq 1 \underset{\boxed{e_2}}{\rightarrow} c_1 := 0 \qquad c_1 \leq 1 \underset{\boxed{e_3}}{\rightarrow} c_1 := 0$$

The TPO has a single clock c_1*, with each event having a guard* $c_1 \leq 1$ *and resetting the clock* c_1 *to* 0*. The set of admissible timed traces ensure that: (a) the events* $e_1, \ldots, e_3$ *may happen in arbitrary order; (b) the first event must appear within* 1 *time unit of the start of the process; (c) each subsequent event must happen within* 1 *time unit of the previous event. The following constraints represent this TPO:*

$$(t_1 \leq t_2 \leq t_3 \;\Rightarrow\; t_1 \leq 1 \;\wedge\; t_2 - t_1 \leq 1 \;\wedge\; t_3 - t_2 \leq 1) \wedge$$

$$(t_1 \leq t_3 \leq t_2 \;\Rightarrow\; t_1 \leq 1 \;\wedge\; t_3 - t_1 \leq 1 \;\wedge\; t_2 - t_3 \leq 1) \wedge$$

$$\ldots$$

$$(t_3 \leq t_2 \leq t_1 \;\Rightarrow\; t_3 \leq 1 \;\wedge\; t_2 - t_3 \leq 1 \;\wedge\; t_1 - t_2 \leq 1)$$

Each of the 3! clauses correspond to a linearization of the partial order which leads to different constraints between intervals over successive events as dictated by the TPO.

To avoid the need to reason over linearization of the underlying partial order, we define a structurally restricted class of TPOs that we call "race-free TPOs". The race-freedom here denotes that the timing constraints remain independent of the actual order in which the events occur. We say that an event e_i in a TPO is **dependent** on a clock c_j iff the guard for e_i refers to c_j or c_j is reset by e_i.

Definition 37 (Race-Free TPOs)**.** *A TPO is said to be race-free iff for every clock* $c \in C$ *and two different events* e_i, e_j *that are dependent on* c*, either* $e_i \prec e_j$ *or* $e_j \prec e_i$*. In other words, events that are independent according to the partial order (parallel events) do not refer to the same clock.*

Note that the TPO in Example 10 is clearly race-free. For instance, events e_2, e_5 refer to clock c_2 but $e_2 \prec e_5$. This can be checked for every clock and every pair of events that are dependent on that clock. However, the TPO in Example 13 is not race-free. Events e_1, e_2, e_3 all refer to clock c_1 but we do not have any precedence relationship between any of them.

We will now establish the key theorem that characterizes the timing constraints corresponding to race-free TPOs.

Theorem 6. *A (race-free) TPO can be represented by a conjunction of inequalities of the form:*

$$\varphi : \bigwedge_{i,j \text{ s.t. } e_i \prec e_j} (t_j - t_i) \in [\ell_{j,i}, u_{j,i}] \ \wedge \ \bigwedge_{j=1}^{n} t_j \in [a_j, b_j], \tag{5.2}$$

wherein $\ell_{i,j} \geq 0, a_j \geq 0$ *form lower bounds and* $u_{j,i}, b_j \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ *are upper bounds that can be non-negative real numbers as well as* $+\infty$.

Full proofs are provided in the appendix. Briefly, the theorem holds because in a race-free TPO, a clock c that is reset at some event e_i and subsequently referred to at event e_k (without intervening reset) ensures that $e_i \prec e_k$. Therefore, the clock c at any point refers to the time difference $t_k - t_i$. Therefore, guards on clocks translate into constraints involving differences $t_k - t_i$. However, if a clock is not reset, it measures the time elapsed since the start of the process. A guard on such a clock is simply a constraint on t_k.

The "race-free" assumption limits our ability to compare timings of independent events. For instance, even if the wheels on a car can be mounted in parallel to each other, we may have a constraint that the left front and left rear wheels are attached no more than a minute apart from each other.

5.3.2 Constructing Timed Partial Orders From Constraints

The main insight behind our work lies in proving the converse of Theorem 6. Let us fix a set of events $\mathcal{E} = \{e_1, \ldots, e_n\}$ and a partial order $\prec$ between them.

Theorem 7. *Given timing constraints of the form*

$$\varphi^{TPO} : \ \bigwedge_{j=1}^{n} t_j \in [a_j, b_j] \ \wedge \ \bigwedge_{e_i \prec e_j} (t_j - t_i) \in [\ell_{i,j}, u_{i,j}], \tag{5.3}$$

wherein $a_j, \ell_{i,j} \geq 0$ *and* $b_j, u_{i,j} \in \mathbb{R}_{\geq 0} \cup \{\infty\}$, *there is a race-free TPO that represents the timing constraints* φ^{TPO}.

The proof of this theorem lies in the procedure we will now present to synthesize such a TPO involving three major steps: (a) Remove **redundant** timing constraints from φ^{TPO} to obtain

an irredundant representation $\tilde{\varphi}^{TPO}$; (b) Construct a clock allocation graph G from $\tilde{\varphi}^{TPO}$; (c) solve a graph coloring problem on G and (d) translate the graph coloring results into clocks, clock guards and resets for the TPO.

Example 14. *Consider once again the windshield installation task from Example 9. Ignoring $C_1, \ldots, C_4$, let us instead consider the following timing constraints:*

$$\varphi^{TPO}: \left(\begin{array}{c} t_3 - t_1 \in [10, \infty] \ \wedge \ t_5 - t_1 \in [0, 15] \ \wedge \\ t_5 - t_3 \in [0, 5] \ \wedge \ t_6 - t_5 \in [0, 8] \ \wedge \\ t_5 - t_4 \in [5, \infty] \ \wedge \ t_6 - t_4 \in [4, 10] \end{array} \right) \tag{5.4}$$

Redundancy Elimination and Simplification First, we introduce a fictitious initial event e_0 which always happens at fixed time $t_0 = 0$ such that $e_0 \prec e_j$ for all $j \in \{1, \ldots, n\}$. The constraints in Eq. (5.3) are now written as:

$$\bigwedge_{e_i \prec e_j} \ell_{i,j} \leq (t_j - t_i) \wedge (t_j - t_i) \leq u_{i,j} \,. \tag{5.5}$$

Next, we eliminate redundant constraints of two types: (a) any constraint of the form $t_j - t_i \bowtie a$ that is implied by the conjunction of the remaining constraints; (b) trivial constraints with $\ell_{i,j} = 0$ or $u_{i,j} = \infty$;. The result of redundancy elimination may be witten as:

$$\tilde{\varphi}^{TPO}: \bigwedge_{i,j} \bigwedge_{k} (t_j - t_i) \bowtie l_{i,j,k}, \text{ wherein } \bowtie \in \{\leq, \geq\} \,. \tag{5.6}$$

where k iterates over all inequalities that involve $t_j - t_i$. Additionally, for each inequality $t_j - t_i \bowtie l_{i,j,k}$, the corresponding events must satisfy $e_i \prec e_j$. We provide further details on redundancy elimination in the subsequent section.

Example 15. *Consider the constraint* (5.4) *in Example 14. The constraint $t_5 - t_3 \leq 5$ is redundant since we can infer it from the two constraints $t_3 - t_1 \geq 10$ and $t_5 - t_1 \leq 15$. Removing the trival and redundant constraints yields*

$$\tilde{\varphi}^{TPO}: \left(\begin{array}{c} t_3 - t_1 \geq 10 \ \wedge \ t_5 - t_1 \leq 15 \ \wedge \ t_6 - t_5 \leq 8 \\ t_5 - t_4 \geq 5 \ \wedge \ t_6 - t_4 \in [4, 10] \end{array} \right) . \tag{5.7}$$

Allocating Clocks to Enforce Constraints In order to enforce a constraint of the form $t_j - t_i \bowtie l_{i,j,k}$ using clocks:

(1) Reset a "dedicated" clock c_i at the same instant when event e_i occurs;

(2) Add the guard $c_i \bowtie l_{i,j,k}$ for event e_j.

In effect, c_i measures time elapsed since event e_i. When event e_j happens, its value equals $t_j - t_i$. In fact, the clock c_i can be used to enforce multiple conjunctions of the form $t_{j_1} - t_i \bowtie a_1 \ \wedge \ \cdots \ \wedge \ t_{j_l} - t_i \bowtie a_l$ since the structure of $\tilde{\varphi}^{TPO}$ (Eq. (5.6)) guarantees that $e_i \prec e_{j_1}, \ldots, e_i \prec e_{j_l}$. Thus, the modified strategy is as follows:

(1) Write $\tilde{\varphi}^{TPO}$ as $\tilde{\varphi}_0^{TPO} \ \wedge \ \cdots \ \tilde{\varphi}_n^{TPO}$, wherein $\tilde{\varphi}_i^{TPO}$ collects all inequalities in $\tilde{\varphi}^{TPO}$ of the form $(t_k - t_i) \bowtie a_{k,i}$.

(2) If $\tilde{\varphi}_i^{TPO}$ is not empty, then allocate a dedicated clock c_i that is reset at event e_i. In special case, since event e_0 is fictitious, we allocate the clock c_0 but do not reset it.

(3) For the inequality $(t_k - t_i) \bowtie a_{k,i}$ in $\tilde{\varphi}^{TPO}$, add the conjunction $c_i \bowtie a_{k,i}$ to the guard $g(e_k)$ for event e_k.

Thus, the scheme so far constructs a TPO with at most $n + 1$ clocks. Since n can be quite large (~ 500) for some manufacturing workflows, we wish to minimize the number of clocks to reduce the complexity of the overall TPO.

Example 16. *Continuing from Example 15. Following the technique presented thus far, we split the constraint $\tilde{\varphi}^{TPO}$ (Cf. (5.7)) into three parts (underlining is for emphasis) given by $\tilde{\varphi}_1^{TPO} = \ (t_3 - \underline{t_1} \geq 10 \ \wedge \ t_5 - \underline{t_1} \leq 15)$, $\tilde{\varphi}_4^{TPO} = (t_5 - \underline{t_4} \geq 5 \ \wedge \ t_6 - \underline{t_4} \geq 4 \ \wedge \ t_6 - \underline{t_4} \leq 10)$ and $\tilde{\varphi}_5^{TPO} = (t_6 - \underline{t_5} \leq 8)$. We allocate three clocks c_1, c_4, c_5 to track these three sets of constraints, respectively. Clock c_1 is reset*

at event e_1, c_4 at event e_4 and c_5 at event e_5. The following guards are added:

Constraint	*Guard*
$t_3 - t_1 \geq 10 \ \wedge \ t_5 - t_1 \leq 15$	$c_1 \geq 10@e_3, \ c_1 \leq 15@e_5$
$t_5 - t_4 \geq 5 \ \wedge \ t_6 - t_4 \in [4, 10]$	$c_4 \geq 5@e_5, (c_4 \in [4, 10])@e_6$
$t_6 - t_5 \leq 8$	$c_5 \leq 8@e_6$

Minimizing Clocks in the TPO In order to reduce the number of clocks, we ask the following question: under what conditions can we "reuse" the clock c_i corresponding to event e_i for a different event e_j?

For any given event e_i, let e_k be the event that is maximal according to the precedence relation $\prec$ such that a timing constraint of the form $(t_k - t_i) \bowtie a_{k,i}$ exists in $\tilde{\varphi}^{TPO}$. If no such inequality exists in the first place, then clock c_i would not exist in the first place. We can reuse the clock c_i for any "later" event e_j such that $e_k \preceq e_j$, since the last time clock c_i is used is at event e_k. Recall, from TPO semantics in Def. 35, that clocks are reset only **after** the guards are checked.

We construct a **clock allocation graph** G_c, an undirected graph whose vertices are the clocks considered so far.

(1) Corresponding each clock c_i, we compute its latest guarded event $L(i)$ as follows:

   (a) Let $E_i = \{e_k \mid \text{inequality} \ \ t_k - t_i \bowtie a \text{ in } \tilde{\varphi}^{TPO}\}$.

   (b) Set $\mathrm{L}_c(i) = \mathsf{sup}_{\prec}(E_i)$, the supremum in E_i according to the $\prec$ order. We observe that clock c_i can be reused after event $L_c(i)$ has occurred.

(2) Add an undirected edge (c_i, c_j) whenever $L_c(i) \not\preceq e_j$ and $L_c(j) \not\preceq e_i$ enforcing that c_i and c_j be kept distinct.

Recall that the graph coloring problem seeks to assign one of m colors to each vertex of an undirected graph so that no two vertices connected by an edge have the same color. The main idea behind minimizing clock usage is to examine the optimal coloring of the graph and whenever two nodes c_i, c_j are the same color, we can substitute the use of clock c_j by c_i. This process ensures that we use as many clocks as the number of colors used in graph coloring.

Theorem 8. *If the clock allocation graph G_c can be colored using m colors, then we can construct a TPO with at most m clocks to represent the timing constraints in $\tilde{\varphi}^{TPO}$.*

Example 17. *Continuing with Example 16, we compute the clock allocation graph G_c with vertices $\{c_1, c_4, c_5\}$. Note that $L_c(1) = e_5$, $L_c(4) = e_6$, and $L_c(5) = e_6$ as defined above. Thus, according to the construction above, the clock allocation graph has two edges $\{(c_1, c_4), (c_4, c_5)\}$. This can be colored with two colors and in particular nodes c_1 and c_5 have the same color. This denotes that the clock c_5 can be replaced with c_1 everywhere.*

Note that the problem of checking if a graph G may be colored using $m \geq 3$ colors is known to be NP-complete [38]. Nevertheless, graph coloring has been studied for numerous applications including notably register allocation for compilers and scheduling problems [20, 59]. We may employ a simple greedy algorithm for graph coloring [16] that guarantees that the number of colors is bounded by $1 + \Delta(G_c)$, wherein $\Delta(G_c)$ denotes the maximum number of neighbours for any vertex in G_c.

5.3.3 Relationship between TPOs and Timed Automata (TA)

An important question still remains: How are they related to TA? In this section, we show that the TPO is a subset class of TA and can always be translated to a TA, specifically, a directed acyclic TA, but may result in an exponential blowup in size. The downside of TPOs is that they cannot encode history-dependent timing constraints, e.g., $c >= 10$ if e_1e_2 is visited or else $c <= 10$. This makes the model less expressive and we show that not all TAs can be translated to TPOs.

At a high level, we prove that the TPOs and the translated TA are equivalent by showing the equivalence relation between the languages they generate. We show that their languages, i.e. the runs, of the model can always be mapped to each other by maintaining a timed bisimulation relation. Let us first define Timed Languages and later define Timed Automata.

Recall from Chapter 3, we extend the definition of trace ω to include dense-time. A timed trace τ over an alphabet Σ is a sequence of pairs (ω_i, t_i) where $t_i > 0$ is a time that monotonically

increases at every step i. A *timed language* $\mathcal{L}(\mathcal{A})$ is a set of timed traces. Now we define the formal definition of Timed Automata.

Definition 38 (Timed Automata (TA)). *A Timed Automaton* $\mathcal{A} = (\Sigma, Q, Q_0, C, F, E)$ *where*

- Σ *is a finite alphabet,*
- Q *is a set of states,*
- $Q_0 \subseteq Q$ *is a set of initial states,*
- C *is a finite set of clocks,*
- F *is a set of final states, and*
- $E \subseteq Q \times \Sigma \times Q \times 2^C \times \Phi(C)$ *is a set of transitions where an edge* $(q, a, q', \hat{C}, g)$ *represents a transition from state* q *to state* q' *on input symbol* a*. The set* $\hat{C} \subseteq C$ *gives the clocks to be reset with this transition, and* g *is a clock guard over* C*, i.e.,* $\bigwedge_{j=1}^{n_i} c_j \bowtie a_j$*.*

Given a timed trace τ, the transition starts from $q_i \in Q_0$ with time 0 with all clocks and valuations initialized to 0. A transition is made at time t_i by observing a symbol ω_i if the time satisfies the clock guard g. With this transition, the clocks in $\hat{C}$ are reset to 0. We say that the timed trace is *accepting* if the transitions end at the final state F. At each state s, local times are calculated to help evaluate the guards. They are called a *valuation* of clocks. A valuation $\nu : C \to \mathbb{R}_{\geq 0}$ assigns each clock $c_i \in C$ to a non-negative number $\nu(c_i)$. A given valuation ν can be advanced in time by a fixed $\delta \geq 0$ to yield a new valuation $\nu' := \nu \oplus \delta$ such that $\nu'(c_j) = \nu(c_j) + \delta$ for all $c_j \in C$. Likewise, given a valuation ν and a subset of clocks $\hat{C} \subseteq C$, we denote the valuation $\nu' := \mathsf{reset}(\nu, \hat{C})$ as that obtained by setting each clock $c \in \hat{C}$ to be 0: $\nu'(c) = \begin{cases} 0 & c \in \hat{C} \\ \nu(c) & c \notin \hat{C} \end{cases}$. Let ν_0 represent a fixed special initial valuation wherein $\nu_0(c_j) = 0$ for all clocks and let $t^{(0)} = 0$. Timestamps represent global time since the inception of the process whereas clocks measure the time elapsed since their last reset.

Definition 39 (Semantics of TA). *A run of a timed automaton is a sequence of the form over a*

timed trace τ*,*

$$r : (q_0, \nu_0) \xrightarrow[t_1]{\omega_1} (q_1, \nu_1) \cdots (q_{n-1}, \nu_{n-1}) \xrightarrow[t_n]{\omega_n} (q_n, \nu_n)$$

where $q_i \in Q$*,* ν_i *is a valuation of clocks.*

In this work, we restrict the Σ to be a set of events $\mathcal{E}$. Consequently, the formulation of the timed trace aligns with the definition employed in TPO's framework $\tau : \{e_1 \mapsto t_1, \ldots, e_n \mapsto t_n\}$.

Lemma 10. *For any run of a TPO* S*, there exists a mapping to a path in a TA* $\mathcal{A}$

Proof. Recall the definition of a run of a TPO to be a sequence of triples of symbol (event), timestamp, and valuations.

$$\rho : (\sigma_1, t^{(1)}, \nu_1), \ldots, (\sigma_n, t^{(n)}, \nu_n)\,,$$

The sequence of symbols and timestamps is equivalent to the definition of a timed trace τ which induces a run in a TA that generates the same sequences of valuations.

$$r : (q_0, \nu_0) \xrightarrow[t^(1)]{\sigma_1} (q_1, \nu_1) \cdots (q_{n-1}, \nu_{n-1}) \xrightarrow[t^(n)]{\sigma_n} (q_n, \nu_n)$$

where q_i represents a node in TA. Starting from a root node q_0, each run generates a new path on TA along with its corresponding guards from the TPO. Therefore, any run in a TPO can be mapped to a run in a TA. □

However, the other direction is not true. Not all TA can be translated to a TPO. We show this by simply providing counter-examples. One easy-to-understand example is a Timed Automaton with a loop. Assume a transition $(q, \sigma, q, \hat{C}, g)$, it can generate a timed trace $(\sigma, t^{(0)}, \nu_0)$, $(\sigma, t^{(1)}, \nu_1)$, …, $(\sigma, t^{(k)}, \nu_k)$. TPO cannot have an arbitrary length of timed trace and requires the constraints to be linear. What if we only consider Acyclic Timed Automata? Can they all be translated to TPO? The answer is, no.

Example 18. *A timed automaton in Figure 5.3 has two paths.* $a \to b \to c \to d$ *and* $a \to c \to b \to d$*. The first path is taken if the clock* $c1 \leq 10$ *and the second path is taken if the clock* $c1 > 10$*. Such*

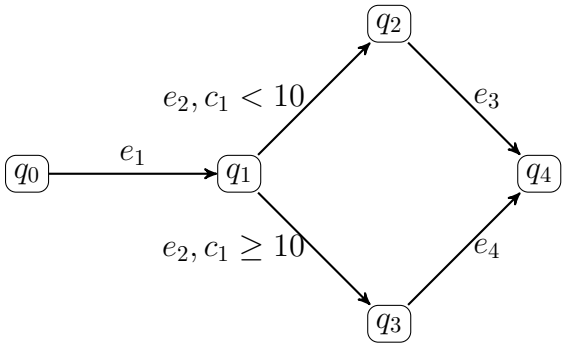


Figure 5.3: An example of an Acyclic Timed Automaton

conditional transitions ("OR" operation) cannot be expressed in a TPO as every symbol must have the same guard.

Translation Algorithm The key idea of the algorithm is to ensure all linearizations of the TPO exist in the TA. For convenience, we define functions $Succ(e) : \mathcal{E} \to 2^{\mathcal{E}}$, $Pred(e) : \mathcal{E} \to 2^{\mathcal{E}}$ and $Succ(H) = \bigcup_{e \in H} \{e' \mid e' \in Succ(e), Pred(e') \subseteq H\}$ to denote the successors and predecessors of an event, and successors of all observed events (history H).

At every state $q = (H, N)$, the algorithm maintains a history H of which states were visited and a set of possible next events N. The algorithm starts by creating an artificial initial state q_0 with an empty history and a set of initial events in TPO. At every state q, the algorithm creates a transition $(q, e, q', g(e), R(e))$ from a state $q = (H, N)$ to $q' = (H', N')$ by observing a symbol $e \in N(q)$, its guard $g(e)$, and the reset clocks $R(e)$. We create a new state q' and update the history $H' = H \cup \{e\}$ and a set of possible next events N' by sequentially adding e's predecessors, i.e., $N(q') = N(q) \cup \{e' \in Succe(e) | Pred(e') \subseteq H\} \setminus \{e\}$. One must be careful if the successor node is a merging node since a merging node can only be visited if its all predecessors are visited. Keeping a set of all possible next events and creating transition for all events ensures that any order of events in a partial order also exists in the TA. In order words, this creates every linearization of the TPO in the TA. This process continues until it exhaustively visits all events in the TPO. Since the clock guards are directly carried to the TA, there always exists a sequence of valuations that is satisfied by the translated TA. This keeps the timed bisimulation relation between the two, and thus they

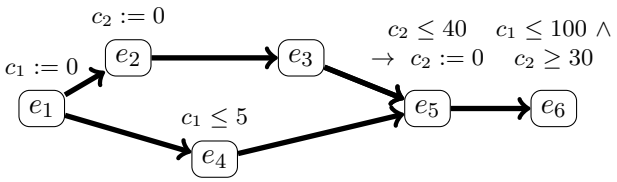


Figure 5.4: TPO for the car windshield installation workflow.

are equivalent.

Example 19. *In Figure 5.4, we show the original TPO and its translated TA in Figure 5.5. The initial event e_1 at time t_1 creates a transition in a TA from the artificial initial state q_0 to q_1 by observing an event e_1 with the clock guard $g(e_1)$ and the reset clocks $\hat{C} = R(e_1)$. When creating a new state q_1 in the TA, it keeps a set of all possible successors of e_1 in the TPO. In Figure 5.4, e_1 has successors e_2 and e_4. The state q_1 can either take a transition $(q_1, e_2, q_2, g(e_2), R(e_2))$ or $(q_1, e_4, q_3, g(e_4), R(e_4))$ where q_2 is a tuple of $\{e_4, e_3\}$ and q_3 is a tuple of $\{e_2\}$. Note that e_5 is not appended to q_3 because e_5 has an unvisited predecessor e_3. This process continues until it visits all the events in the TPO.*

Lemma 11. *Given a TPO of size n, the number of states in the equivalent TA can be in the order of 2^n.*

Proof. A TPO can model up to n number of parallel events, with $n-1$ events for a race-free TPO, where one initial (or last) node connects to the rest. The events can be executed in any sequence to generate a run. There could be 2^n such combinations of runs. Consequently, this would result in mapping each run to a Timed Automata, leading to $O(2^n)$ distinct states. □

5.4 Mining TPO from Timed Traces

Given timed traces $\tau_1, \ldots, \tau_n$ by observing some workflow with a fixed set of events $\mathcal{E} = \{e_1, \ldots, e_n\}$, we wish to synthesize a TPO such that all the timed traces in the given data D are compatible with the TPO. To do so, requires identifying the partial order $\prec$, the clocks, guards and resets.

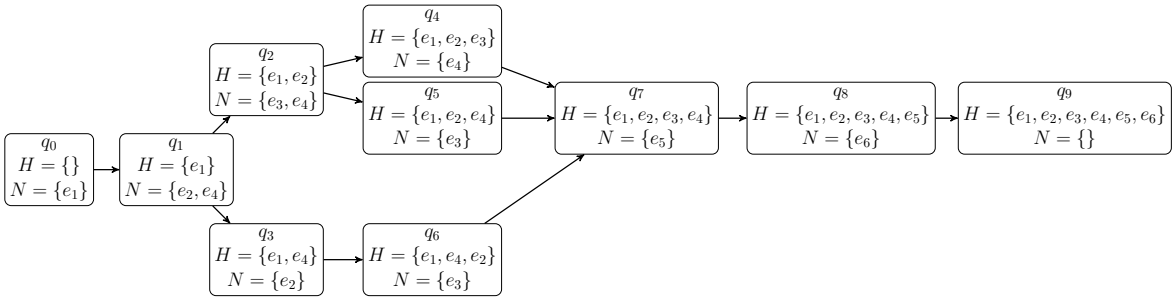


Figure 5.5: The translated TA from Figure 5.4

Our proposed approach proceeds in three steps: (a) identify the partial order information from the timed traces; (b) compute the tightest possible timing constraints of the form (5.3) that includes all the data; and (c) mine a TPO from the timing constraints in step (b) using the algorithm described in the previous section. Note that steps (a) and (b) are based on well-known techniques that will be briefly recalled in this section. We briefly describe these steps, concluding with a description of our implementation.

Partial Order Identification In order to identify the partial order $\prec$, we set $e_i \prec e_j$ for events $e_i, e_j \in \mathcal{E}$ iff in all the timed traces the event e_i precedes e_j.

Formulating Timing Constraints In order to formulate timing constraints, we translate each timed trace in the data into a vector $(t_1, \ldots, t_n)$. Next, we consider bounds of two types: (a) Upper/lower bounds on each event time t_i by itself to yield intervals $t_i \in [a_i, b_i]$, and (b) Upper/lower bounds on the time difference $t_j - t_i$ whenever $e_i \prec e_j$ holds.

Whereas the interval bounds are simply the maximum and minimum values in the data, there are two drawbacks: (a) The process assumes that all upper bounds are finite since we can never infer a constraint of the form $t_j - t_i \in [a, \infty)$ from the data. However, there are statistical tests from **extreme value theory** that can identify whether a distribution has an infinite support [44]. The application of these techniques relies on having a large volume of data. (b) The bounds themselves depend intimately on the amount of data and the sampling method. To mitigate this, we refer the reader to ideas from conformal prediction that allow us to bloat the intervals obtained from data appropriately to achieve a prediction with some associated confidence [9].

Redundancy Elimination To eliminate redundancies, we formulate a series of optimization problems involving the constraints in Equation (5.3). We iterate through each inequality $(t_j - t_i) \leq u_{i,j}$ (alternatively, $t_j - t_i \geq l_{i,j}$) from the system in some order and carry out the following steps:

(1) Remove the selected inequality and set the objective to maximize (alternatively, minimize) $t_j - t_i$ subject to the remaining constraints.

(2) If the resulting optimal value is $> u_{i,j}$ (alternatively, $< l_{i,j}$) then the constraint is irredundant, and needs to be added back to the problem.

(3) Otherwise, the constraint is removed once and for all.

The optimization problem in question is a linear programming problem that can be solved quite efficiently for the special class of difference constraints encountered here. However, the result of redundancy elimination varies, depending on the order of constraints in which we process the inequalities. For example, the constraint $t_5 - t_1 \leq 15$ in Example 15 can be removed instead of $t_5 - t_3 \leq 5$. The problem of finding the set of irredundant constraints of the least cardinality is known to be NP-hard following a reduction from the minimum equivalent graph problem [38]. Therefore, we consider various heuristics for deciding the order in which the constraints are to be considered.

(1) NEAREST: Consider constraints $t_j - t_i$ in increasing order of the number of intermediate events between e_i and e_j: i.e, $|\{e_k \mid e_i \prec e_k \prec e_j\}|$.

(2) DISTANT: Consider constraints $t_j - t_i$ in decreasing order of the number of intermediate events between e_i and e_j.

(3) RANDOM: Consider constraints in a randomized order.

(4) SOUND: The SOUND algorithm starts from the last node, checks for all time constraints $t_j - t_i \bowtie a$ that are dependent on t_i. If ALL of them are redundant, then we remove all

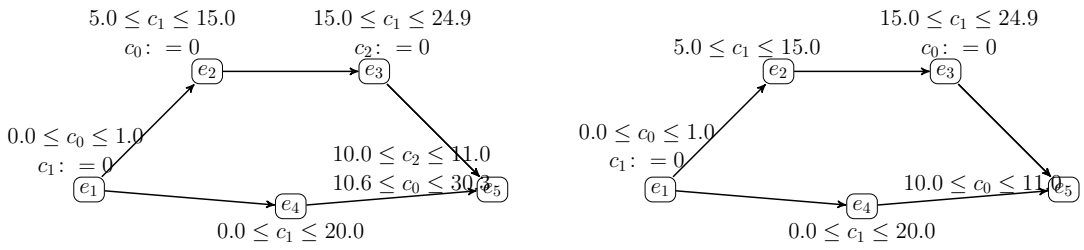


Figure 5.6: Mined TPOs using the NEAREST and SOUND heuristics.

the constraints, or else, if ANY of them are required, we keep all the constraints, because the clock at node i is going to be required anyways.

We remark that there is no good approach to choose between these heuristics, other than empirically testing each of them and selecting the best one.

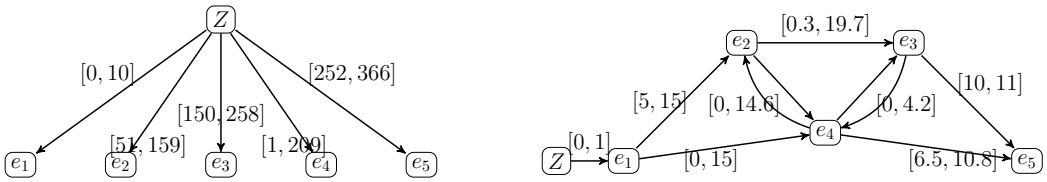
Example 20. *Consider Example 9 with the new time constraints* $t_1 \in [0, 1]$, $t_2 - t_1 \in [5, 15]$, $t_3 - t_1 \in [15, 25]$, $t_4 - t_1 \in [0, 20]$, *and* $t_5 - t_3 \in [10, 11]$. *We randomly sampled 1000 timed traces that satisfy these constraints and used the procedure described in this section to construct TPOs, as shown in Figure 5.6. The TPO to the left uses the* NEAREST *heuristic for redundancy elimination requiring three clocks, whereas the TPO to the right uses the* SOUND *heuristic using just two clocks to explain the same data.*

5.4.1 Time Complexity

Let $n = |\mathcal{E}|$. The algorithm solves a linear program (LP) at each step whose time complexity is bounded by a polynomial over n. In the worst case, there are $O(n^2)$ pairs of time constraints (LP problems to solve).

5.5 Experiments

In this section, we evaluate our approach for mining TPOs from timed trace data. First, we compare our method against the closest work, the CSTNUD mining algorithm. Next, we



(a) Global (b) Local

Figure 5.7: Outputs by the CSTNUD mining algorithm assuming (Left) all the events depend on the global clock $Z = 0$ and (Right) each event depends on the previous event. The constraints $e_4 \to e_3 : [0, 18.2]$ and $e_2 \to e_4 : [0, 13.2]$ are not visualized due to the limited space.

run an ablative study on a benchmark to evaluate the clock allocation performance against a few heuristics. Subsequently, we show interesting results on two datasets inspired from real-life applications: aircraft turnaround and Overcooked game. Our evaluation focuses on how well the resulting model can provide human interpretable insights into the dependencies between the various tasks in the workflow. In particular, we do not classify between different outcomes such as success/failure due to the lack of such labels in our dataset.

Comparison against a CSTNUD mining algorithm [1] As a comparison to Example 9, we ran the algorithm proposed by [78]. Their method requires human annotations on event relationships, so we prepared two different results that can easily be derived from the raw data: 1) all events depending on the global clock (called "global") and each event depending on the previous event (called "local") and showed the resulting graphs in Figure 5.7. The globally-dependent graph entirely depends on the global clock and cannot mine the relationships between events. As a result, the timings constraints (durations) between events tend to become large. Whereas the locally-dependent graph is shown to mine the event relationships well but with unnecessary edges such as $e_2 - e_4$ and $e_3 - e_4$. In other words, they cannot handle parallelized tasks like (e_2, e_3) pair and e_4. Moreover, the CTSNUD mining algorithm can only mine relationships between neighboring edges that result in a one-clock model whereas our model can be viewed as a generalization to multiple-clock graphs.

[1] `https://github.com/matteozavatteri/cstnud-miner`

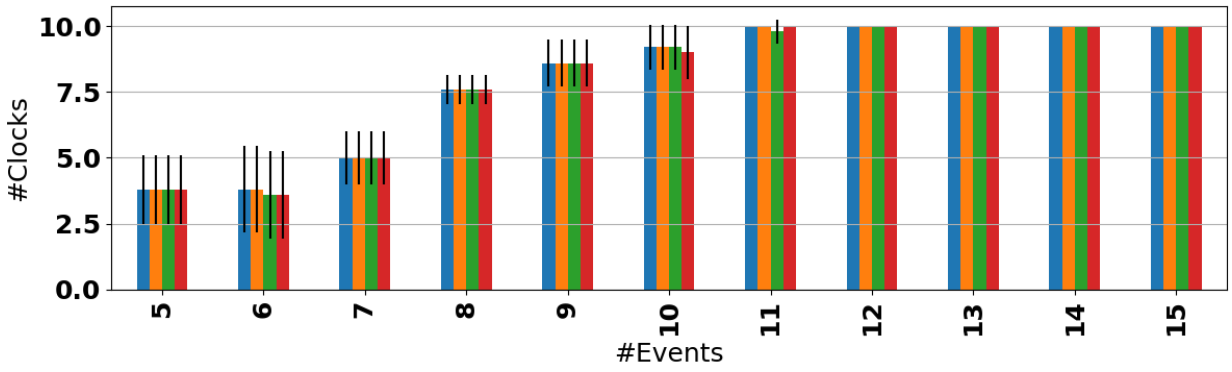


Figure 5.8: The number of clocks identified by the different heuristic algorithms: DISTANCE (blue), NEAR (orange), RANDOM (green), SOUND (red).

Analysis on Synthetic TPO Data In this experiment, we seek to understand (a) how the running time of our procedure scales with increasing number of events/traces; and (b) the number of clocks generated by the various redundancy elimination heuristics. We generated a bunch of random TPOs according to a method described in the appendix. The TPOs vary according to the number of events n. For each TPO, we randomly sampled 1000 traces.

Figure 5.8 shows the average number of clocks identified over ten runs of this procedure. The number of clocks increases as the number of events increases. The heuristics yield a similar number of clocks. Computation time did not depend on the choice of the heuristics, but it varied on how many times the LP optimizations were called and how fast they found the solution. The more eliminations happen at the earlier stage, the fewer constraints remain in later LP. Often, however, the extra cost of upfront elimination does not provide enough of a payoff in the later stages. This is very much dependent on the nature of the data and constraints.

Aircraft Turnaround Example Next, we evaluate our algorithm on the processes involved in the turnaround of an aircraft at a gate. Aircraft turnaround is a critically important process that affects the operating costs of airlines. It involves a series of tasks such as deboarding, cleaning, refueling and boarding with happens-before orders. For instance, cleaning must be performed after deboarding. However, refueling can be performed in parallel with cleaning. We

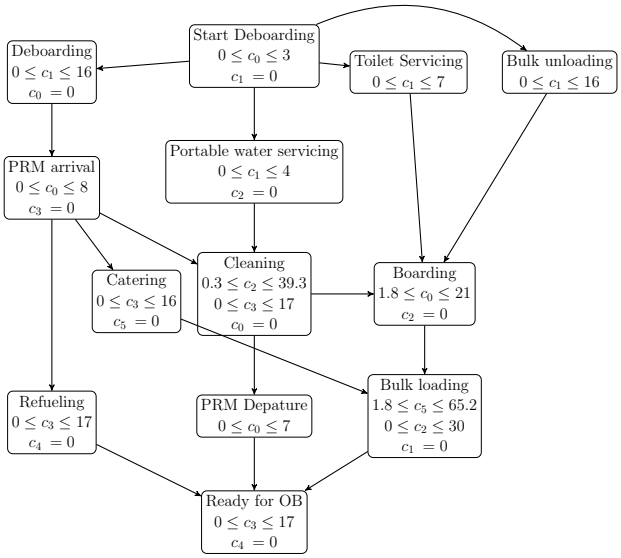


Figure 5.9: Mined TPO for aircraft turnaround.

defined a TPO of the aircraft turnaround operations using data synthesized from the information presented in Nosedal Sánchez and Piera Eroles [65]. In particular, we use *average time that led to delays* as the maximum time for each operation. Timestamps were sampled from truncated normal distributions, as specified by Nosedal Sánchez and Piera Eroles [65]. We synthesized 1000 timed traces and evaluated our algorithm against the SMT-based timed automata inference algorithm [86]. The SMT-based algorithm did not terminate over the original data set (timed out due to expensive calls to SMT solvers). We had to reduce the data size to just 20 timed traces in order to get the algorithm to run. However, the result fails to capture the timing constraints present in the original problem. The RTI+ algorithm [94] also fails to run on the reduced data.

In contrast, the TPO mined using our algorithm shown in Figure 5.9 respects all original precedence orders, along with the maximum time duration for each operation. Furthermore, notice that the algorithm identified new precedence orders such as "Toilet servicing" happens-before "Boarding" due to the relation between the two time constraints. The TPO just requires 6 clocks in all. This example shows that our algorithm can be applied to realistic scenarios.

Overcooked Example Overcooked is a multiplayer game that simulates a busy restaurant kitchen, requiring players to collaborate on producing numerous plates of food according to a fixed

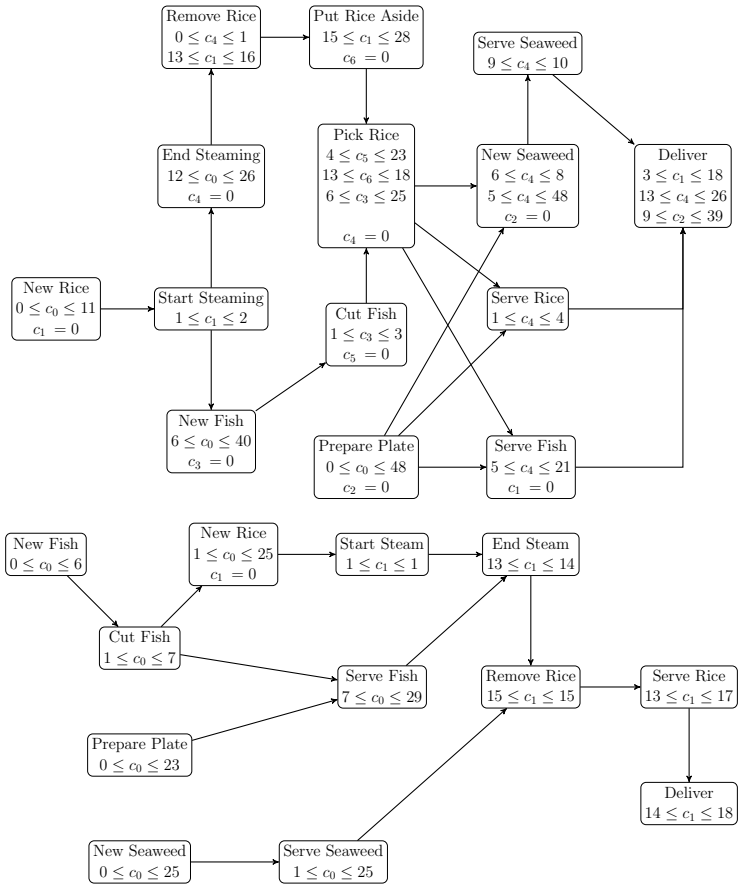


Figure 5.10: Overcooked gameplay analysis: (Top) TPO specification for a beginner player, (Bottom) TPO specification of a professional player.

recipe against timing constraints. We tested our algorithm on publicly available gameplay videos on YouTube. We analyzed the difference between the beginner[2] versus professional[3] gamers on Overcooked 2. The task of the game is to repeatedly make Sushi plates from seaweed, sliced fish, and cooked rice, and serve them to customers in a target area. There is no serving order between the three ingredients, but there are strict orders on how each ingredient is prepared. We manually annotated the video with event labels describing the actions of the players. We were able to identify eight timed traces in the beginner's play and collected the same number of traces from the professional's play. The mined TPOs are shown in Figure 5.10.

In both cases, strict orders are correctly identified. For example, a fish must be cut before serving on a plate. The difference between the two is the order of the parallelizable tasks. Interestingly, the beginners start with *cooking rice* and then *cutting fish*, whereas the professionals start with *cutting fish* and then *cooking rice*. This is because beginners prepare dishes one by one, and hence they must start with rice, which takes the longest time to prepare, whereas the professionals cook in a batch and the fish comes first in this strategy.

In terms of time constraints, strict constraints are imposed, such as (1) fish/seaweed must be cut/served immediately after a new object is taken out of the box (2) rice must be steamed for about 13-16 seconds in both figures. In contrast, a new object (plate, rice, and fish) can be taken out of the box at any time in the scene (with a large bound in c_0). Furthermore, a time constraint between serving fish and rice (Serve Rice $4 \leq c_1 \leq 14$) in Figure 5.10 (bottom) specifies that parallel tasks must take similar times, so the dish can be delivered immediately after (Deliver $1 \leq c_4 \leq 2$). This experiment shows that our TPO mining algorithm provides an interpretable representation for a task solely based on a small amount of data.

[2] `https://www.youtube.com/watch?v=jTrenjjZDtA&t=668s`
[3] `https://www.youtube.com/watch?v=YcnpWo4Y01M&t=60s`

5.6 Discussion

5.6.1 Pipeline Workflow: Repetitive Events

In our problem setting, we assumed that the data log contains a neatly classified set of traces and each trace contains only a unique set of events. However, in reality, a log consists of a set of mixed traces sorted by timestamps and there could be multiple occurrences of the same event in a trace. For example, an automotive assembly line manufactures multiple cars concurrently and the same events (e.g., install a door) appear multiple times in each trace. To split the log into a set of traces, we need to identify the counts of each event appearing in a trace $x \in \mathbb{Z}^n_{>0}$ and split the log accordingly. To do so, we formulate the problem as an integer programming. Given a number of products being manufactured k and a log, minimize x, such that, $k \cdot I \cdot x + I \cdot y = b$, $x > 0, y >= 0$ and $y < k$, where $y \in \mathbb{Z}^n_{\geq 0}$ is a vector of variables representing the remaining events in a trace and $b \in \mathbb{Z}^n_{>0}$ is the counts of events in the log. Then, we greedily split the log from the top with each trace containing x counts of events.

5.6.2 Loops

In our formulation, we cannot model loops in TPOs. However, in reality, a data log can come from a workflow that requires certain events to loop. For, example, cracking three eggs can be represented as repetitions of **NewEgg** and **Crack** events. To learn a TPO from such data, the naive way is to relabel the repetitive events with unique events, e.g., **NewEgg1**, **NewEgg2**. Once the partial among other events are identified, then the repeated events can be folded. For example, **Crack1** can be relabeled back to **Crack** and add an edge to **NewEgg** to form a loop.

5.7 Conclusions

We have introduced and analyzed the expressivity of TPOs, leading to a procedure for mining TPOs from data. Experiments demonstrate how mining TPOs from process data can yield useful insights for important workflows inspired by real-life manufacturing processes.

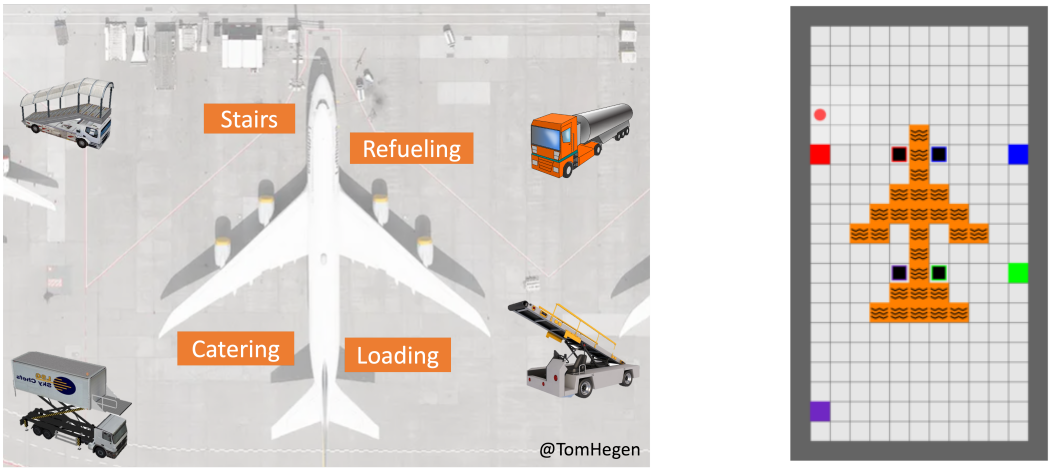
Chapter 6

Optimal Planning for Timed Partial Order

This chapter addresses the challenge of planning a sequence of tasks to be performed while minimizing the overall completion time subject to timing and precedence constraints. Our approach uses the Timed Partial Orders (TPO) model to specify these constraints. We translate this problem into a Traveling Salesman Problem (TSP) variant with timing and precedent constraints, and we solve it as a Mixed Integer Linear Programming (MILP) problem. We show that the formulation for the single-agent can easily be extended to a multi-agent scenario by adding the same number of initial nodes as the number of robots to the TSP graph. Our contributions include a general planning framework for TPO specifications, a MILP formulation accommodating time windows and precedent constraints, its extension to multi-robot scenarios, and a method to quantify plan robustness. We demonstrate our framework on several case studies, including an aircraft turnaround task involving three Jackal robots, highlighting the approach's potential applicability to important real-world problems. Our benchmark results show that our MILP method outperforms state-of-the-art open-source TSP solvers OR-Tools.

6.1 Introduction

Workflow analysis and optimization techniques are of crucial importance in increasing efficiency across many domains, from manufacturing to administrative processes. These workflows are conventionally structured around tasks that have to be completed subject to precedence and timing constraints. Such constraints are often defined by the user or inferred from demonstrations [95, 79].



(a) Top-down view

(b) Gridworld

Figure 6.1: Aircraft Turnaround Example

A recently-proposed model, **Timed Partial Order** (TPO) [95], provides a succinct, understandable, and easily analyzable representation of these timing constraints. It allows for precedence constraints using partial orders, and the timing constraints are specified using clocks that can be reset when events in the workflow happen. However, the problem of planning task sequences given environmental constraints and TPO specifications has not been solved. The main challenge lies in the combination of environmental constraints that specify how the events in the workflow can be achieved, timing costs for various events, and the TPO. All of these are to be taken into account while planning for such agents. In this work, we develop a planning framework with TPO specifications against environments modeled as deterministic transition systems.

Consider the agent operating in an aircraft turnaround task in Figure 6.1. Many tasks such as bulk loading/unloading and refueling can be completed in parallel. Additionally, there are precedence and timing constraints. For example, deplaning can only be done after the stair truck is placed, and deplaning takes at least 15 minutes. Catering can only begin after all passengers have deplaned. To this end, we need a plan synthesis algorithm that can find the minimum-makespan plan under the timing and partial-order constraints.

In this work, we propose a new framework to solve the time-constrained plan synthesis

problem for a single as well as multiple robots. We employ TPOs as task specifications and use Deterministic Transition Systems (DTS) as abstraction models of the robots in their environments. We show that the planning problem on DTS with TPO specifications reduces to a type of Traveling Salesman Problem (TSP), which asks, given a map of cities, to find the lowest cost path to visit every city once. TSP is well-studied and known to be NP-hard [38], but there exist tools that use highly-effective heuristics, allowing fast computations [27]. We formulate our DTS with TPO planning problem as an instance of TSP with the addition of timing and precedent constraints [68]. These constraints introduce the difficulty of applying off-the-shelf heuristics to our problem. Instead, we solve the problem as a Mixed Integer Linear Program (MILP), which finds an optimal solution rapidly for some of our benchmarks. We also show that for the multi-robot setting, a slight modification of the MILP can be employed. Furthermore, we provide an efficient approach to robustness analysis of the synthesized plans.

The contributions of this work are fourfold: we (i) introduce a general planning framework for TPO specifications that can be applied to one or more robots, (ii) formulate a MILP with time windows (global time with respect to the start event) and precedence constraints (local time between sub-tasks), and extend it to multiple robots, (iii) propose a method based on LP to quantify the robustness of the synthesized plans to capture the lower and upper bounds on the delays that the plan can tolerate w.r.t. the given TPO, and (iv) provide a set of illustrative case studies and benchmarks that empirically show that TPO constraints actually narrow down the search space, speed up the computation time, and enable scaling up the algorithm to 160 nodes and 40 robots. We perform a physical experiment for an aircraft turnaround task with three Jackal robots that demonstrates the ability of our approach to plan for practically relevant problems.

6.2 Related Work

Simple Temporal Networks (STNs) [33] and Timed Automata (TA) [3] allow us to specify complicated timing specifications. A comparison of TPOs with related formalisms is provided by Watanabe et al [95].

Model Checking Problem [3]: The problem focuses solely on properties such as consistency of the specification model, neglecting the operating environment. The focus of this work involves plan synthesis which combines the timing specification and the model of the operating environment.

Plan Synthesis Problem: The problem is to find a plan in the operating environment that satisfies the specification. This is a common problem in robotics, and many studies have been conducted for TAs [7] and Signal Temporal Logic (STL) [60] which can be translated into TAs [61]. In fact, the synthesis algorithm is known to be exponential in the number of clocks (Lemma 4.5 and Theorem 7.8 of [3]) and blows up very quickly in the number of tasks and environmental states. This makes the algorithm difficult to apply to larger instances and multi-agent systems [82]. In this work, we focus on TPOs that can be turned into linear inequality constraints, which reduce the complexity of the problem.

Task & Motion Planning Problem: The problem is to assign (heterogeneous) tasks to robots, involving the problem of coalition formation [43]. The work in [83] tackles the task-allocation with timing and precedent constraints by forming coalitions of robots to accomplish tasks more efficiently. Similarly, [41] tackles the coalition formation problem to assign tasks to robots via a min-cost network flow approach. Our focus is not on allocating (heterogeneous) tasks to (heterogeneous) robots via forming coalitions, but rather it is on scheduling tasks that satisfy given formal specifications (e.g., precedence order, timing constraints, etc.). Others have also formulated the task allocation problem as a Traveling Salesman Problem (see [21] for a short survey), but without precedence and timing constraints. As highlighted in survey [66], some works have considered timing and precedent constraints along with others (multi-agent, hard/soft constraints, deterministic/stochastic, etc.). However, no work has looked into local timing dependencies, and more importantly, into tasks that can be performed at multiple different locations. The latter creates a choice of not only which robot should perform the task, but also in which location (for which we need to formulate a Generalized TSP).

6.3 Problem Formulation

In this section, we formally define the problem of single/multi-robot plan synthesis for TPOs.

6.3.1 Single-Robot Setting

For the single-robot setting, we consider the robotic environment abstracted as a DTS as defined in Definition 2.

Problem 3. *(Single Robot Plan Synthesis) Given a robotic system as a DTS $\mathcal{T}$ with a specification as a TPO φ (recall Definition 34), synthesize a valid plan $\gamma^* \in \Gamma$ for the robot that satisfies the TPO in a minimum time duration, i.e.,*

$$\gamma^* = \arg\min_{\gamma \in \Gamma} D(s^\gamma) \quad s.t. \quad \tau^\gamma \models \varphi$$

where τ^γ represents a timed trace induced by the plan. We further extend the problem to a multi-agent system.

6.3.2 Multi-Robot Setting

We consider the same environment as above but now with $n_R \in \mathbb{N}_{>1}$ robots, each with its own initial state. We define a multi-robot DTS (mDTS) by extending $\mathcal{T}^S$ to have a set of initial states $X_I = \{x_0^1, x_0^2, \ldots, x_0^{n_R}\} \subseteq X$, i.e., $\mathcal{T}^M = (X, A, X_I, \delta_\mathcal{T}, \Delta_\mathcal{T}, \mathcal{E}, L^S)$, where $X, A, \delta_\mathcal{T}, \mathcal{E}$, and L^S are as in Def. 2. The notion of valid plan $\gamma^i \in \Gamma^i$ for robot i is adopted from the single case, and the set of all valid plans for all robots is denoted by $\Gamma = \bigcup_{i=1}^{n_R} \Gamma^i$.

Similar to the single robot case, from initial state $x_0^i \in X_I$, plan γ^i induces a trajectory s^{γ^i}, and a timed trace τ^{γ^i}, and timetamps t^{γ^i}. We assume that when a robot executes action $a \in A$ at state $x \in X$, it remains in x for the entire duration $\Delta_\mathcal{T}(x, a)$ before transitioning to $x' = \delta_\mathcal{T}(x, a)$. Then, the induced trajectory can be viewed as a piecewise function of time. With an abuse of notation, we use $s^{\gamma^i} : \mathbb{R}_{\geq 0} \to X$ to denote this function, where $s^{\gamma^i}(t)$ is the state visited by trajectory s^{γ^i} at time t. We say two trajectories s^{γ^1} and s^{γ^2} are *non-colliding* if, for

all $t \leq \max\{D(s^{\gamma 1}), D(s^{\gamma 2})\}$, $s^{\gamma 1}(t) \neq s^{\gamma 2}(t)$. We define a timed trace $\tau^{\gamma^1 \cdots \gamma^{n_R}}$ of the multi-robot system to be the union of the individual robot's timed traces in the time-increasing order[1] .

Then, the multi-robot problem is to find plans for the p robots that generate non-colliding trajectories with the timed trace $\tau^{\gamma^1,\ldots,\gamma^p}$ that satisfies the TPO specification with a minimum time duration.

Problem 4 (Multi-Robot Plan Synthesis)**.** *Given a system of p robots as a mDTS* $\mathcal{T}^M$ *and a TPO specification* φ*, synthesize plans* $\gamma^{1*}, \ldots, \gamma^{p*} \in \Gamma$ *under which the multi-robot system satisfies the TPO in a minimum time duration:*

$$\gamma^{1*}, \ldots, \gamma^{p*} = \arg\min_{\gamma^1 \cdots \gamma^p \in \Gamma} \max\{D(s^{\gamma^1}), \ldots, D(s^{\gamma^p})\}$$

subject to

$$\tau^{\gamma^1 \cdots \gamma^p} \models \varphi$$

$$s^{\gamma^j} \text{ and } s^{\gamma^j} \text{ are non-colliding} \quad \forall \gamma^i, \gamma^j \in \Gamma.$$

Note that Problems 3 and 4 ask for a satisfying plan that minimizes the maximum duration of the induced trajectories, which is also known as the *makespan* of the plan.

6.4 Approach

In this section, we explain how a Problem 3 can be formulated as an instance of the Generalized Traveling Salesman Problem (GTSP), but with timing and precedent constraints. These additional constraints introduce difficulty in applying existing heuristics to our problem out of the box. In this work, we first focus on the Mixed Integer Linear Program (MILP) formulation. We first introduce the problem of GTSP and later discuss how the problem can be translated into the graph representation of GTSP.

[1] Recall that a timed trace is also viewed as a set $\{(\sigma_k, t^{(k)})\}_{k=0}^m$.

6.4.1 Problem 3 as Generalized Traveling Salesman Problem

The Generalized Traveling Salesman Problem [64] is the problem of finding the minimum cost path that visits exactly one city from given subsets of cities. GTSP is known to be an NP-hard problem, and it is a well-studied problem in combinatorial optimization research. There are many existing methods to obtain either exact or approximate solutions to this problem [70].

We want to translate Problem 3 into a GTSP, more specifically, GTSP with Time-Windows and Precedence Relations (GTSP-TWPR) [62] so that we can utilize the existing approaches and extend the problem formulation. Formally, GTSP-TWPR is defined as follows.

Definition 40 (GTSP-TWPR)**.** *Let $G = (V, E)$ be a weighted directed graph with vertices V and edges $E = V \times V$. Node $v_i \in V$ is associated with a vertex cost d_i, which represents the time delay at that vertex. Edge $(v_i, v_j) \in E$ is assigned a time cost d_{ij}, which is the time required to move from v_i to v_j. Node $v_0 \in V$ is designated as the depot with $d_0 = 0$.*

*The problem seeks a tour that visits some of the vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_m}$ with starting and ending at the depot, $v_{i_0} = v_{i_m} = v_0$, while minimizing the total time of the tour: $\sum_{j=0}^{m} d_{i_j} + d_{i_j, i_{j+1}}$. Note that we can set $d_{i,0} = 0$ for all i if return to the depot is not required for the problem. We refer to this total time as the **makespan** of the tour. The tour is subject to the following additional constraints: (a) We partition the set $V \cup \{v_0\}$ into disjoint subsets $V_1, V_2, \ldots, V_k$. The TSP tour is required to visit exactly one node from each subset V_i. Let t_i be the time at which the node in the set V_i is visited by our tour. (b) We require that $l_i \leq t_i \leq u_i$ for a time interval $[l_i, u_i]$ provided as input. (c) We specify qualitative precedence constraints of the form $V_i \prec V_j$ that specifies that the tour must visit a node in V_i before it visits some node in V_j. (d) Whenever we have $V_i \prec V_j$, we require $t_i \leq t_j$. Additionally, we may also specify quantitative relative time window $[l_{ij}, u_{ij}]$ requiring that $l_{ij} \leq t_j - t_i \leq u_{ij}$.*

We show how Problem 3 is mapped to a GTSP-TWPR.

Definition 41 (Translation of Problem 3 to GTSP-TWPC)**.** *We abstract DTS $\mathcal{T}$ and the TPO φ to a GTSP-TWPR graph, by defining the set of node $V = \{x_0\} \cup \{x \in X \mid L(x) \neq \emptyset\}$ to be the*

set of states with non-empty labels as well as x_0. Then, the depot node $v_0 = x_0$, and subset V_i is a set of states whose label is event e_i, i.e., $V_i = \{x \in X | L(x) = e_i\}$. A directed edge (x_i, x_j) is added whenever $L(x_i) \preceq L(x_j)$ or the events $L(x_i), L(x_j)$ can happen in parallel. The edge cost d_{ij} between two states x_i and x_j is defined by the trajectory duration $D(s, \gamma)$ where s is the shortest path between x_i and x_j on $\mathcal{T}$, i.e., $s = s_0 s_1 \dots s_n$ that induces a trace $L(x_i)\emptyset ... \emptyset L(x_j)$. Likewise, if state x_i in T has a self-transition under action a with time cost $\Delta_T(x_i, a)$, then we set $d_i = \Delta_T(x_i, a)$ as the vertex cost for node x_i; otherwise, we set $d_i = 0$.

Edge costs are calculated by running an all-shortest path algorithms such as Floyd-Warshall algorithm [36]. We note that our method can be extend to a continuous space kinodynamical robots by running Stable Sparse RRT (SST) [58] to obtain the shortest paths between states. Generally, this is run once for environments where locations are fixed, e.g., manufacturing factories.

6.4.2 MILP Formulation

6.4.2.1 Single Robot

We solve Problem 3 on graph G^{TSP} exactly using a Mixed Integer Linear Programming (MILP) formulation. The formulation is shown in Figure 6.2. Recall that $n = |\mathcal{E}|$ is the number of events, and per our construction of G^{TSP}, it is also the number of the subsets $V_1^{TSP}, \dots, V_n^{TSP}$. We define the square bracket $[k]$ to represent the set $\{1, \dots, k\}$. Let N be the number of nodes $N = |V^{TSP}|$, $y_{i,j}$ be an integer variable indicating the active edge $i \to j$, and τ_i be the continuous time variable that represents the completion of the ith event, where τ_{N+1} represents the time coming back to the depot. We additionally introduce the set $V_0^{TSP} = \{v_0\}$ with the depot node and the indices $I = \{0\}$ to simplify the formulation.

Note that the continuous variables include $N + 1^{\text{th}}$ variable τ_{N+1} that represents the time at the end of the tour. Constraint (6.1a) expresses that the number of incoming and outgoing edges must be equal at every node. Constraints (6.1b) and (6.1c) represent that there is only one incoming and one outgoing edge for each subset. Together with the first constraint, we ensure

$$
\begin{aligned}
&\min\ \tau_{N+1} \quad \text{s.t.} \\
&\sum_i y_{i,j} - \sum_k y_{j,k} = 0, && j \in I \cup [N] && \text{(6.1a)}\\
&\sum_{v_j \in V_l^{TSP}} \sum_i y_{i,j} = 1, && l \in I \cup [n] && \text{(6.1b)}\\
&\sum_{v_j \in V_l^{TSP}} \sum_k y_{j,k} = 1, && l \in I \cup [n] && \text{(6.1c)}\\
&\tau_j - \tau_i \bowtie a_{i,j}, && t_j, t_i, a_{i,j} \in \varphi && \text{(6.1d)}\\
&y_{i,j} = 1 \implies \tau_j - \tau_i \geq d_{i,j} + d_j, && i \in I \cup [N], && \\
& && j \in [N] && \text{(6.1e)}\\
&y_{i,j} = 1 \implies \tau_{N+1} - \tau_i \geq d_{i,j}, && i \in [N], j \in I && \text{(6.1f)}\\
&y_{i,j} \in \{0, 1\}, && i, j \in I \cup [N] && \text{(6.1g)}\\
&\tau_i \geq 0, && i \in I \cup [N+1] && \text{(6.1h)}
\end{aligned}
$$

Figure 6.2: MILP formulation of the GTSP-TWPR problem. Notation $[N] = \{1, \ldots, N\}$.

that the incoming/outgoing edge to a particular subset V_i^{TSP} must involve the same node $v_i \in V_i^{TSP}$. Constraint (6.1d) represents all the TPO constraints, and (6.1e) delays the j^{th} event by $d_{i,j}+d_j$ from i^{th} event only if the edge is activated ($y_{i,j} = 1$). Constraint (6.1f) delays $N + 1^{\text{th}}$ visit (makespan) by the edge cost between $[N]$ nodes back to the initial nodes I. $\implies$ represents "implies" and can be expressed by using the Big-M method [98].

A tour can be obtained by following the enabled edges $y_{i,j} = 1$ from the depot node.

6.4.2.2 Multiple Robots

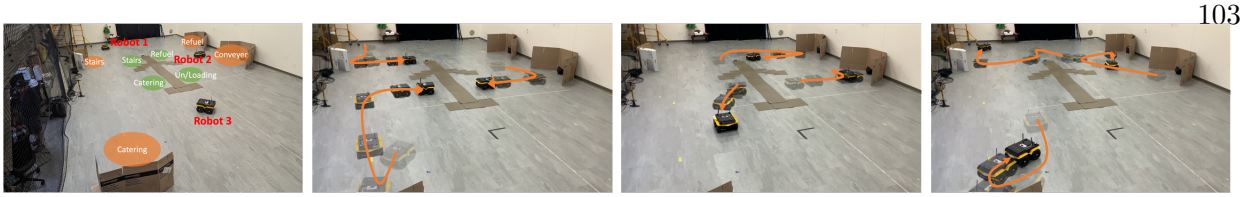
For the multi-robot setting, we construct graph G^{TSP} in a similar manner as the single-robot case, except that for each initial state $x_0^i \in X_0$, we add a depot node v_0^i and its associated subset $V^{TSP}{}_0^i$ to G^{TSP}. For simplicity, we redefine the indices $I = 0_1, ..., 0_{n_R}$ to denote the depot nodes and their subsets. The MILP formulation then becomes exactly the same as that of the single-agent case in Figure 6.2. In practice, we avoided introducing new depots, but instead, we changed the number of incoming and outgoing edges at the initial node v_0 to prevent the increase in the number of integer variables. The right-hand sides of (6.1b) and (6.1c) equate to the number of robots

starting at V_0.

The resulting tours minimize the makespan, and their induced timed trace satisfies the TPO specification. However, when the tours are mapped to plans on $\mathcal{T}^M$, they are not guaranteed to produce non-colliding trajectories. For instance, two robots may collide at the intersection of two crossing trajectories. Hence, they need to be checked for collisions as a post-process. If a collision is detected, then we repair the plans by introducing delays to the individual plans as in [51]. Acceptable limits on these delays can be calculated using a robustness analysis. Another approach is to resolve conflicts via the Conflict-Based Search (CBS) method for multi-agent as in [74]. This method first computes K best solutions to the TSP problem. For each tour (tours), it constructs a search tree to find non-conflicting trajectories as in the standard CBS algorithm. While expanding the tree, it may move to another tour's tree node if the cost of the previous node becomes larger than the new node in the different tree. In this way, the method is guaranteed to find an optimal solution and is a complete algorithm. We can replace the TSP with GTSP-TWPC to find non-colliding trajectories.

6.4.3 Robustness Analysis

Once a tour is returned by our MILP formalism, it is now possible to understand how robust the tour is to variations in the edge and node costs or in other words, variations in the delays associated with a given node or an edge between two locations. Such delays are common during plan execution. Consider a single-agent tour with edge costs: $v_0 \xrightarrow{d_{01}} v_1 \xrightarrow{d_{12}} v_2 \cdots v_{m-1} \xrightarrow{d_{m-1,0}} v_0$. Let d_i be the node cost of v_i with $d_0 = 0$. Our goal is to characterize all possible timing variations $\delta(d_{ij})$ to the edge costs and $\delta(d_j)$ to the node costs such that the TPO constraints will continue to hold. To this end, note that the nominal visit time for node v_i in the tour is given by $t_i = \sum_{j=0}^{i-1} d_j + d_{j,j+1}$ for $i \in [1, m-1]$ while the visit time taking into account the unknown variations in d_j, $d_{i,j}$ will be $t_i = \sum_{j=0}^{i-1} d_j + d_{j,j+1} + \delta(d_j) + \delta(d_{j,j+1})$. Robustness analysis seeks a uniform bound ϵ such that whenever each $|\delta(d_j)| \leq \epsilon$ and $|\delta(d_{i,j})| \leq \epsilon$, the TPO constraints are guaranteed to hold. To



(a) Experimental Setup (b) Trajectory 1/3 (c) Trajectory 2/3 (d) Trajectory 3/3

Figure 6.3: (see attached video: `https://youtu.be/WUuWFlOoKW8`)

compute such a limit ϵ, we formulate the LP:

$$\begin{aligned}
\max \quad & \epsilon \\
\text{s.t.} \quad & t_i = \textstyle\sum_{j=0}^{i-1} \left(d_j + d_{j,j+1} + \delta(d_j) + \delta(d_{j,j+1})\right) \\
& \qquad\qquad \text{for } i = 1, \ldots, m-1 \\
& t_i - t_j \bowtie a_{i,j}, \quad \text{TPO constraints } t_i, d_j, a_{i,j} \\
& \epsilon \;\leq\; \delta(d_j) \\
& \epsilon \;\leq\; \delta(d_{i,j})
\end{aligned}$$

The LP above is always feasible and its optimal solution ϵ denotes a uniform bound on the timing variations $\delta(d_j), \delta(d_{i,j})$ such that as long as $|\delta(d_j)| \leq \epsilon$ and $|\delta(d_{i,j})| \leq \epsilon$ for each node and edge delay, the tour continues to be a feasible solution that satisfies the TPO constraints. The formulation for a multi-robot tour is almost identical except that the times t_i are computed differently for each robot. Unfortunately, trying to incorporate the robustness analysis as part of the solution to the TSP itself results in a robust optimization problem which often requires more expensive approaches to solve. Our future work will consider incrementally eliminating tours that fail a robustness criterion by adding a "blocking" constraint to force the MILP solver to return back with a different tour.

6.5 Experiments

In this section, we evaluate our algorithm for planning with TPO specification for single and multiple robots on various case studies. First, we demonstrate how different timing constraints of TPO cause different robot behaviors. Then, we illustrate the scalability of the algorithm on a set of

(a) Blue/Orange=With/out φ^{TPO} (b) Multi-robot trajectories

Figure 6.4: Gridworlds with synthesized plans

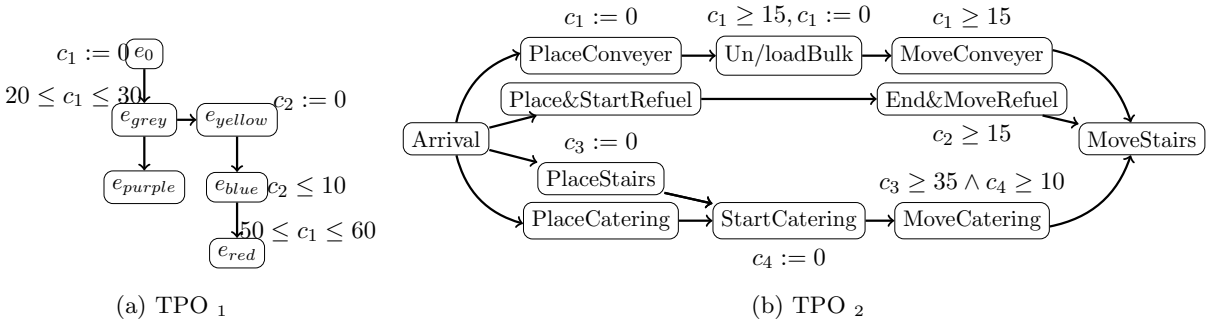
benchmarks. Lastly, we return to the aircraft turnaround example and show a physical experiment to demonstrate the applicability of the approach to a more realistic scenario.

6.5.1 Illustrative Case Studies

We demonstrate our method on a gridworld environment in 6.4 with multiple colored locations and two TPO tasks: with and without timing constraints. For the simple task, the robot must visit every color in any order. It has the option of visiting any location of the same color but must find a plan with a minimum makespan time. We ran the algorithm without any constraints and its trajectory is shown in orange in Figure 6.4a. The robot first visits red, green, purple, yellow, blue, and grey in order. The second task is TPO $_1$ in Figure 6.5a. The robot visited grey, green, purple, yellow, blue, and red in order. Observe that the robot now visits the grey first and the red last due to the constraints. In Figure 6.4b, we extend to two robots the same TPO $_1$ task. The result was two trajectories for the two robots that satisfy TPO $_1$. The trajectories are long since red has to be visited between 50 to 60 time units.

6.5.2 Benchmarks

Here, we explore how our algorithm scales with the increasing number of states with non-empty labels, timing constraints, and number of robots. We generated a random set of events varying in number from 5 to 80 in a 30-by-30 gridworld environment. We incrementally add timing



(a) TPO $_1$ (b) TPO $_2$

Figure 6.5: TPO Specifications for the case studies.

constraints of the form $d_{i,j} + d_j - \Delta t \le t_j - t_i \le d_{i,j} + d_j + \Delta t$ where $\Delta t > 0$ is a constant padding to see how the overall solution time depends on the number of constraints we add and the "tightness" of these constraints. We ran benchmarks with varying the number of timing constraints (constraints involving $p = 25, 50, 75, 100\%$ of all TPO edges), $\Delta t = 10, 30, 50$, and the number of robots of $1, 10, 20, 30$, and 40. We compared our method to heuristic approaches implemented in OR-Tools Routing Libraray [37]. The time padding Δt and the percentage of the number of edges p did not have significant effects on the results. Figure 6.6 shows the result at when $p = 25\%$ and $\Delta t = 30$. As the number of nodes increases, the problem gets more difficult to solve, taking more time to find the optimal solution. Also, the computation time mostly stays the same as we increase the number of robots. Interestingly, the OR-Tools did not perform well compared to MILP. This is because the heuristics are disabled when unexpected additional constraints (e.g., local timing constraints) are added. Instead, they use constraint programming to solve the problem, which is slower than the Branch and Bound method employed in the MILP solver. Also, the algorithm easily scaled up to 80 nonempty-label states as shown in the figure. We further ran the stretch test with a timeout of 30 minutes, and MILP was able to solve up to 160 nodes within 182 ± 102 seconds excluding a case (out of 60 runs) when it hit the timeout. It took 210 ± 232 seconds including the timeout.

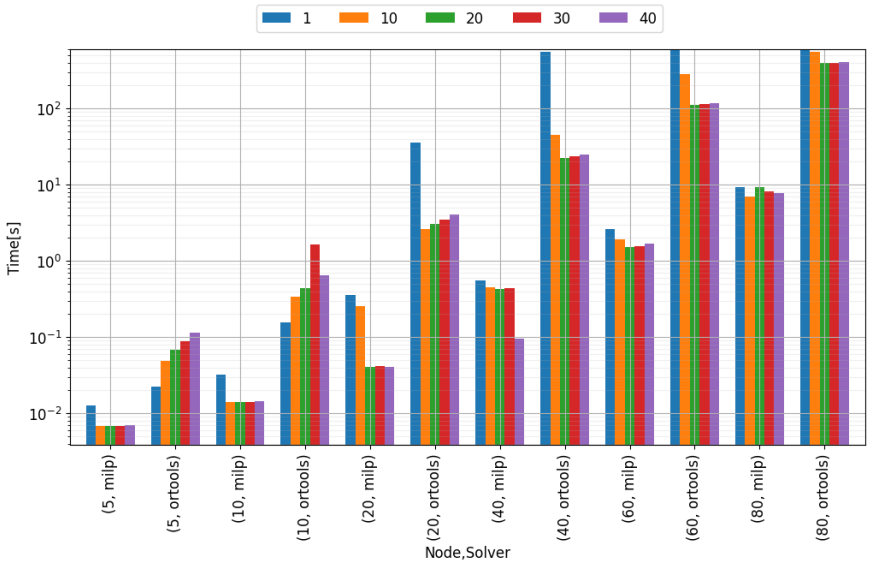


Figure 6.6: Benchmark results. The x-axis is (#nodes, solver) and the y-axis is computation time in seconds in log scale. The timeout is set to 600 seconds. The colors of the bars indicate the number of robots (see legend).

6.5.3 Aircraft Turnaround

We consider the aircraft turnaround example in Figure 6.1 with robots as ground staff. The task is TPO $_2$ in Figure 6.5b, specifies moving vehicles around, refueling, and bulk loading/unloading. The goal is to find the most efficient plans for completing the task according to TPO $_2$. In Figure 6.3, we show the plans of all robots. Robot 1 places the stair truck, refueling vehicle, and moves out the stair truck. Robot 2 performs bulk unloading/loading and then moves out of the refueling vehicle. Robot 3 performs the catering services. Almost all robots finish their assigned events at the same time. The overall makespan was 59 time units. We also repeated the same case study for a single robot from various initial states and obtained makespans $152, 155, 163$ time units. A video of this experiment accompanies the submission.

6.6 Conclusion

We introduced a general framework for planning under Timed Partial Order (TPO) specifications for multiple robots. Our solution maps the task allocation problem to the Generalized

Traveling Salesman Problem (GTSP) with time windows and precedence constraints which we solve by a Mixed-Integer Linear Program (MILP). Our evaluations of the algorithm on various case studies demonstrate the time-effectiveness of our plans for up to 40 robots with 160 nodes.

For future work, we plan to investigate variants of the dynamic task assignment problem under TPO specifications and to consider robustness and contingencies in the MILP formulation.

Chapter 7

Conclusion

7.1 Summary

This thesis proposes the concept of specification learning from demonstrations as automata. Learning automata allows for safe inference, improves efficiency in learning and planning, and extends to multi-agent planning in various domains. To model various specifications, the thesis introduces two models: PDFA for probability specifications and TPO for real-time specifications. PDFA are suitable for modeling general robotic tasks and for encoding probabilistic human behaviors, while TPOs are suitable for modeling workflows such as manufacturing or business processes that involve precedent and timing constraints. We introduce the mining and synthesis algorithms for each model that enable users to easily teach robots through demonstrations.

The mining algorithm for PDFA leveraged the EDSM algorithm introduced in the grammatical inference community to efficiently find a *correct* model as in the algorithm is ensured to generate a generalization of the demonstrations. We introduce a preprocessing algorithm that embeds the safety specification during the learning process to ensure the safety of the outcome. Experimental validation confirms the efficacy of this approach in learning safe and generalized tasks that are then used to synthesize strategies in different domains.

TPOs are newly introduced in this thesis. They offer concise representation of precedence orders, timing constraints, and concurrences compared to TAs. TPOs can be translated into a conjunction of inequality constraints, enabling an efficient mining and synthesis algorithm. The relationship between TPOs and TAs is explored to understand their differences in expressivity.

Experiments mine TPOs from expert and non-expert demonstrations, deriving insights from their differences. Case studies demonstrate the scalability of the synthesis algorithm, assigning 160 tasks to 40 robots within 3 minutes while adhering to learned task specifications, showcasing the safety, efficiency, and extensibility of the automata inference approach.

7.2 Future Extensions

This work has shown the power of automata learning that were less focused in robotics. In many cases, the theory of control and reinforcement learning (RL) are worshiped in robotics, however, we should focus more on automata learning to mine and execute complex tasks that RL has difficulty solving. Automata learning could be a solution to their problems and I believe this thesis leaves various interesting questions to the robotics community, such as:

- How can we learn long-horizon/hierarchical policies along that can accomplish the task specifications?
- How can we learn the policies and task specifications together?
- How can we automatically extract important features or symbols from a sequence of state-action pairs?
- How can we apply deep learning methods to specification learning?

Incorporating the low-level observations controls can be challenging, however, it would be a big step forward in robotics if we can automatically extract important symbols that shape the specifications, while learning policies/rewards at the same time. We also need to consider extending classic automata learning techniques to:

- online learning,
- optimization problem (e.g., maximizing the language of the PDFA s.t. negative samples),
- iterative learning through sampling

- interactive learning via human feedback with good visualization method of the learned specification, and
- probabilistic real-time specifications.

In recent years, LLMs (e.g., ChatGPT) have been very successful in translating, answer questions, summarizing texts, and generating new ideas. We also need to consider leveraging the power of Large Language Models (LLM) to possibly estimate the most likely specification or safety specification to foster the specification learning. The thesis also raised questions in the planning domain. For example,

- How can we efficiently assign LTL tasks and plan conflict-free paths to multiple agents?
- How can we define a formal specification language for a multi-agent system?
- How can we plan under uncertain events?
- How can we plan for contingencies? What if the robot is hindered or stopped due to unexpected events?

I believe these questions give rise to future research in this community.

Bibliography

[1] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In EDBT, pages 467–483. Springer, 1998.

[2] R. Alur and D. L. Dill. A theory of timed automata. TCS, 126(2):183–235, 1994.

[3] R. Alur and D. L. Dill. A theory of timed automata. Theoretical computer science, 126(2): 183–235, 1994.

[4] J. An, M. Chen, B. Zhan, N. Zhan, and M. Zhang. Learning one-clock timed automata. In TACAS, pages 444–462. Springer, 2020.

[5] D. Angluin. Learning regular sets from queries and counterexamples. Information and computation, 75(2):87–106, 1987.

[6] B. Araki, K. Vodrahalli, T. Leech, C. I. Vasile, M. Donahue, and D. Rus. Learning to Plan with Logical Automata. In Robotics: Science and Systems Conference (RSS), pages 1–9, Messe Freiburg, Germany, June 2019. link.

[7] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. IFAC Proceedings Volumes, 31(18):447–452, 1998.

[8] C. Baier and J.-P. Katoen. Principles of Model Checking. The MIT Press, Cambridge, MA, 2008.

[9] V. N. Balasubramanian, S.-S. Ho, and V. Vovk. Conformal Prediction for Reliable Machine Learning. Morgan Kaufmann, 2014.

[10] N. Basset, M. Kwiatkowska, U. Topcu, and C. Wiltsche. Strategy synthesis for stochastic games with multiple long-run objectives. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 256–271. Springer, 2015.

[11] C. Belta, B. Yordanov, and E. Gol. Temporal Logics and Automata, volume 89, chapter 2. Springer International Publishing AG 2017, 01 2017. doi: 10.1007/978-3-319-50763-7.

[12] C. Belta, B. Yordanov, and E. A. Gol. Formal methods for discrete-time dynamical systems, volume 89. Springer, 2017.

[13] M. Berlingerio, F. Pinelli, M. Nanni, and F. Giannotti. Temporal mining for interactive workflow data analysis. In KDM, pages 109–118, 2009.

[14] A. Berti, S. J. Van Zelst, and W. van der Aalst. Process mining for python (pm4py): bridging the gap between process-and data science. arXiv preprint arXiv:1905.06169, 2019.

[15] A. Bhatia, L. E. Kavraki, and M. Y. Vardi. Sampling-based motion planning with temporal goals. In 2010 IEEE International Conference on Robotics and Automation, pages 2689–2696. IEEE, 2010.

[16] D. Brélaz. New methods to color the vertices of a graph. Communications of the ACM, 22 (4):251–256, 1979.

[17] A. Camacho, R. T. Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith. Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In IJCAI, volume 19, pages 6065–6073, 2019.

[18] A. Camacho, R. Toro Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith. LTL and beyond: Formal languages for reward function specification in reinforcement learning. In Int'l Joint Conference on Artificial Intelligence, pages 6065–6073, 7 2019.

[19] J. Carmona, J. Cortadella, and M. Kishinevsky. A region-based algorithm for discovering petri nets from event logs. In BPM, pages 358–373. Springer, 2008.

[20] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. Computer Languages, 6(1):47–57, January 1981.

[21] H. Chakraa, F. Guérin, E. Leclercq, and D. Lefebvre. Optimization techniques for multi-robot task allocation problems: Review on the state-of-the-art. Robotics and Autonomous Systems, page 104492, 2023.

[22] K. Chatterjee, M. Randour, and J.-F. Raskin. Strategy synthesis for multi-dimensional quantitative objectives. In International Conference on Concurrency Theory, pages 115–131. Springer, 2012.

[23] T. Chen, V. Forejt, M. Kwiatkowska, A. Simaitis, and C. Wiltsche. On stochastic games with multiple objectives. In International Symposium on Mathematical Foundations of Computer Science, pages 266–277. Springer, 2013.

[24] T. Chen, M. Kwiatkowska, A. Simaitis, and C. Wiltsche. Synthesis for multi-objective stochastic games: An application to autonomous urban driving. In International Conference on Quantitative Evaluation of Systems, pages 322–337. Springer, 2013.

[25] A. K. Choudhary, J. A. Harding, and M. K. Tiwari. Data mining in manufacturing: a review based on the kind of knowledge. Journal of Intelligent Manufacturing, 20(5):501–521, 2009.

[26] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. ACM Trans. on Soft. Engg. (TOSEM), 7(3):215–249, 1998.

[27] W. J. Cook. In Pursuit of the Traveling Salesman. Princeton University Press, Princeton, NJ, USA, Sept. 2014. ISBN 978-0-69116352-9. URL `https://press.princeton.edu/books/paperback/9780691163529/in-pursuit-of-the-traveling-salesman`.

[28] W. J. Cook. In pursuit of the traveling salesman: mathematics at the limits of computation. Princeton University Press, 2015.

[29] A. Datta. Automating the discovery of as-is business process models: Probabilistic and algorithmic approaches. Information Systems Research, 9(3):275–301, 1998.

[30] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In Int. Joint Conf. on Artificial Intelligence, volume 13, pages 854–860, 2013.

[31] C. De la Higuera. Grammatical inference: learning automata and grammars. Cambridge University Press, 2010.

[32] A. K. A. de Medeiros, A. J. Weijters, and W. M. van der Aalst. Genetic process mining: an experimental evaluation. Data mining and knowledge discovery, 14(2):245–304, 2007.

[33] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. Artificial intelligence, 49 (1-3):61–95, 1991.

[34] J. Esparza, M. Leucker, and M. Schlund. Learning workflow petri nets. In Intl. Conf. Appl. and Theory of Petri Nets, pages 206–225. Springer, 2010.

[35] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas. Temporal logic motion planning for mobile robots. In Proceedings of the 2005 IEEE International Conference on Robotics and Automation, pages 2020–2025. IEEE, 2005.

[36] R. W. Floyd. Algorithm 97: shortest path. Communications of the ACM, 5(6):345, 1962.

[37] V. Furnon and L. Perron. Or-tools routing library, 2023. URL `https://developers.google.com/optimization/routing/`.

[38] M. R. Garey and D. S. Johnson. Computers and Intractability: A guide to the theory of NP-Completeness. W.H.Freeman, 1979.

[39] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—The Planning Domain Definition Language, 1998. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212`.

[40] E. M. Gold. Complexity of automaton identification from given data. Information and control, 37(3):302–320, 1978.

[41] W. Gosrich, S. Mayya, S. Narayan, M. Malencia, S. Agarwal, and V. Kumar. Multi-robot coordination and cooperation with task precedence relationships. In 2023 IEEE International Conference on Robotics and Automation (ICRA), pages 5800–5806. IEEE, 2023.

[42] C. W. Günther and W. M. Van Der Aalst. Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In BPM, pages 328–343. Springer, 2007.

[43] H. Guo, F. Wu, Y. Qin, R. Li, K. Li, and K. Li. Recent trends in task and motion planning for robotics: A survey. ACM Computing Surveys, 2023.

[44] L. Haan and A. Ferreira. Extreme Value Theory. Springer, 2006.

[45] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi. Towards manipulation planning with temporal logic specifications. In Int. Conf. Robotics and Automation, pages 346–352. IEEE, May 2015.

[46] K. He, M. Lahijanian, E. Kavraki, Lydia, and Y. Vardi, Moshe. Automated abstraction of manipulation domains for cost-based reactive synthesis. IEEE Robotics and Automation Letters, 4(2):285–292, Apr. 2019.

[47] J. Heinz, C. De La Higuera, and M. Van Zaanen. Formal Preliminaries, chapter 1. Morgan and Claypool Publishers, 2016.

[48] J. Herbst. A machine learning approach to workflow management. In European conference on machine learning, pages 183–194. Springer, 2000.

[49] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne. Imitation learning: A survey of learning methods. ACM Computing Surveys (CSUR), 50(2):1–35, 2017.

[50] S. Jha, A. Tiwari, S. A. Seshia, T. Sahai, and N. Shankar. Telex: Passive stl learning using only positive examples. In International Conference on Runtime Verification, pages 208–224. Springer, 2017.

[51] J. Kottinger, S. Almagor, O. Salzman, and M. Lahijanian. Introducing delays in multi-agent path finding. arXiv preprint arXiv:2307.11252, 2023.

[52] H. Kress-Gazit, G. Fainekos, and G. J. Pappas. Where's Waldo? sensor-based temporal logic motion planning. In Int. Conf. on Robotics and Automation, pages 3116–3121, Rome, Italy, 2007. IEEE.

[53] H. Kress-Gazit, M. Lahijanian, and V. Raman. Synthesis for robots: Guarantees and feedback for robot behavior. Annual Review of Control, Robotics, and Autonomous Systems, 1:211–236, May 2018. doi: 10.1146/annurev-control-060117-104838.

[54] O. Kupferman and M. Y. Vardi. Model checking of safety properties. Formal Methods in System Design, 19:291–314, 2001.

[55] M. Lahijanian, M. Kloetzer, S. Itani, C. Belta, and S. Andersson. Automatic deployment of autonomous cars in a robotic urban-like environment (RULE). In Int. Conf. on Robotics and Automation, pages 2055–2060, Kobe, Japan, 2009. IEEE.

[56] X. Li, C.-I. Vasile, and C. Belta. Reinforcement learning with temporal logic rewards. In IROS, pages 3834–3839. IEEE, 2017.

[57] X. Li, Z. Serlin, G. Yang, and C. Belta. A formal methods approach to interpretable reinforcement learning for robotic planning. Science Robotics, 4(37), 2019.

[58] Y. Li, Z. Littlefield, and K. E. Bekris. Asymptotically optimal sampling-based kinodynamic planning. The International Journal of Robotics Research, 35(5):528–564, 2016.

[59] V. Lotfi and S. Sarin. A graph coloring algorithm for large scale scheduling problems. Computers & Operations Research, 13(1):27–32, Jan. 1986. ISSN 0305-0548. doi: 10.1016/0305-0548(86)90061-4.

[60] O. Maler and D. Nickovic. Monitoring Temporal Properties of Continuous Signals. In Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems: Joint International Conferences on Formal Modeling and Analysis of Timed Systmes, pages 152–166. Springer, 2004. ISBN 978-3-540-23167-7. doi: 10.1007/978-3-540-30206-3_12.

[61] O. Maler, D. Nickovic, and A. Pnueli. From MITL to Timed Automata. In Proceedings of FORMATS, volume 4202 of LNCS, pages 274–289. Springer, 2006.

[62] A. Mingozzi, L. Bianco, and S. Ricciardelli. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. Operations research, 45(3): 365–377, 1997.

[63] A. Y. Ng, S. J. Russell, et al. Algorithms for inverse reinforcement learning. In ICML, volume 1, page 2, 2000.

[64] C. E. Noon. The generalized traveling salesman problem. University of Michigan, 1988.

[65] J. Nosedal Sánchez and M. A. Piera Eroles. Causal analysis of aircraft turnaround time for process reliability evaluation and disruptions' identification. Transportmetrica B: Transport Dynamics, 6(2):115–128, 2018.

[66] E. Nunes, M. Manner, H. Mitiche, and M. Gini. A taxonomy for task allocation problems with temporal and ordering constraints. Robotics and Autonomous Systems, 90:55–70, 2017.

[67] A. Paraschos, C. Daniel, J. Peters, G. Neumann, et al. Probabilistic movement primitives. Neurips, 2013.

[68] S. N. Parragh, K. F. Doerner, and R. F. Hartl. A survey on pickup and delivery problems. Part II: Transportation between pickup and delivery locations, to appear: Journal für Betriebswirtschaft, 2007.

[69] C. Petri. Kommunikation mit automaten (phd thesis). Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[70] P. C. Pop, O. Cosma, C. Sabo, and C. P. Sitar. A comprehensive survey on the generalized traveling salesman problem. European Journal of Operational Research, 2023.

[71] S. Puri and S. K. Prasad. Efficient parallel and distributed algorithms for gis polygonal overlay processing. In 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, pages 2238–2241. IEEE, 2013.

[72] D. Ramachandran and E. Amir. Bayesian inverse reinforcement learning. In IJCAI, volume 7, pages 2586–2591, 2007.

[73] H. Ravichandar, A. S. Polydoros, S. Chernova, and A. Billard. Recent advances in robot learning from demonstration. Annual Review of Control, Robotics, and Autonomous Systems, 3:297–330, 2020.

[74] Z. Ren, S. Rathinam, and H. Choset. Conflict-based steiner search for multi-agent combinatorial path finding. In Proceedings of Robotics: Science and Systems, 2022.

[75] S. Russell and P. Norvig. Artificial intelligence: a modern approach. Prentice Hall series in artificial intelligence. Prentice Hall, 3rd edition, 2010.

[76] V. Sastry, T. Janakiraman, and S. I. Mohideen. New polynomial time algorithms to compute a set of pareto optimal paths for multi-objective shortest path problems. International Journal of Computer Mathematics, 82(3):289–300, 2005.

[77] S. Schaal. Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In Adaptive motion of animals and machines, pages 261–280. Springer, 2006.

[78] G. Sciavicco, M. Zavatteri, and T. Villa. Mining cstnuds significant for a set of traces is polynomial. Information and Computation, 281:104773, 2021.

[79] A. Senderovich, K. E. Booth, and J. C. Beck. Learning scheduling models from event data. In Proceedings of the International Conference on Automated Planning and Scheduling, volume 29, pages 401–409, 2019.

[80] A. J. Shah, P. Kamath, S. Li, and J. A. Shah. Bayesian inference of temporal task specifications from demonstrations. Advances in Neural Information Processing Systems, 31, 2018.

[81] M. Sipser. Introduction to the theory of computation. ACM Sigact News, 27(1):27–29, 1996.

[82] D. Sun, J. Chen, S. Mitra, and C. Fan. Multi-agent motion planning from signal temporal logic specifications. IEEE Robotics and Automation Letters, 7(2):3451–3458, 2022.

[83] E. Suslova and P. Fazli. Multi-robot task allocation with time window and ordering constraints. In 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 6909–6916. IEEE, 2020.

[84] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[85] M. Tappler, B. K. Aichernig, K. G. Larsen, and F. Lorber. Time to learn–learning timed automata from tests. In FORMATS, pages 216–235. Springer, 2019.

[86] M. Tappler, B. K. Aichernig, and F. Lorber. Timed automata learning via smt solving. In NASA Formal Methods Symposium, pages 489–507. Springer, 2022.

[87] W. Van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. IEEE Trans. KDM, 16(9):1128–1142, 2004.

[88] W. M. Van der Aalst. Process mining in the large: a tutorial. European Business Intelligence Summer School, pages 33–76, 2013.

[89] W. M. Van der Aalst, V. Rubin, H. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. Software & Systems Modeling, 9(1):87–111, 2010.

[90] B. F. Van Dongen, A. Alves de Medeiros, and L. Wen. Process mining: Overview and outlook of petri net discovery algorithms. Petri Nets and Other Models of Concurrency II, pages 225–242, 2009.

[91] M. Vazquez-Chanlatte, J. V. Deshmukh, X. Jin, and S. A. Seshia. Logical clustering and learning for time-series data. In Computer Aided Verification, pages 305–325. Springer, 2017.

[92] M. Vazquez-Chanlatte, S. Jha, A. Tiwari, M. K. Ho, and S. Seshia. Learning task specifications from demonstrations. In NeurIPS, volume 31, 2018.

[93] S. Verwer and C. A. Hammerschmidt. Flexfringe: a passive automaton learning package. In Intl. Conf. Software Maintenance and Evolution (ICSME), pages 638–642. IEEE, 2017.

[94] S. Verwer, M. de Weerdt, and C. Witteveen. Efficiently identifying deterministic real-time automata from labeled data. Machine learning, 86(3):295–333, 2012.

[95] K. Watanabe, G. Fainekos, B. Hoxha, M. Lahijanian, D. Prokhorov, S. Sankaranarayanan, and T. Yamaguchi. Timed partial order inference algorithm. Proceedings of the International Conference on Automated Planning and Scheduling, 33(1):639–647, Jul. 2023. doi: 10.1609/icaps.v33i1.27246. URL https://ojs.aaai.org/index.php/ICAPS/article/view/27246.

[96] K. Watanabe, G. Fainekos, B. Hoxha, M. Lahijanian, H. Okamoto, and S. Sankaranarayanan. Optimal planning for timed partial order specifications, 2024.

[97] A. Weijters and W. M. Van der Aalst. Rediscovering workflow models from event-based data using little thumb. Integrated Computer Aided Engineering, 10(2):151–162, 2003.

[98] H. P. Williams. Model building in mathematical programming. John Wiley & Sons, 2013.

[99] M. Wulfmeier, P. Ondruska, and I. Posner. Maximum entropy deep inverse reinforcement learning. arXiv:1507.04888, 2015.

[100] Z. Xu, S. Saha, B. Hu, S. Mishra, and A. A. Julius. Advisory temporal logic inference and controller design for semiautonomous robots. IEEE Trans. on Autom. Sci. and Engg., 16(1): 459–477, 2018.

[101] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In AAAI, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.