Planning and Specification Problems for Multi-Robot Systems, Powered by Formal Methods

Thesis approved by
the Department of Computer Science
Technische Universität Kaiserslautern
for the award of the Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Ivan Gavran

Date of Defense:	24.6.2022.
Dean:	Prof. Dr. Jens Schmitt
Reviewer:	Prof. Dr. Eva Darulova
Reviewer:	Prof. Dr. Nils Jansen
Reviewer:	Prof. Dr. Rupak Majumdar
Reviewer:	Prof. Dr. Daniel Neider

DE-386

Summary

Robotic systems are entering the stage. Enabled by advances in both hardware components and software techniques, robots are increasingly able to operate outside of factories, assist humans, and work alongside them. The limiting factor of robots' expansion remains the programming of robotic systems. Due to the many diverse skills necessary to build a multi-robot system, only the biggest organizations are able to innovate in the space of services provided by robots.

To make developing new robotic services easier, in this dissertation I propose a programming model in which users (programmers) give a *declarative specification* of what needs to be accomplished, and then a *backend system* makes sure that the specification is safely and reliably executed. I present `Antlab`, one such backend system. `Antlab` accepts Linear Temporal Logic (LTL) specifications from multiple users and executes them using a set of robots of different capabilities.

Building on the experience acquired implementing `Antlab`, I identify problems arising from the proposed programming model. These problems fall into two broad categories, specification and planning.

In the category of specification problems, I solve the problem of inferring an LTL formula from sets of positive and negative example traces, as well as from a set of positive examples only. Building on top of these solutions, I develop a method to help users transfer their intent into a formal specification. The approach taken in this dissertation is combining the intent signals from a single demonstration and a natural language description given by a user. A set of candidate specifications is inferred by encoding the problem as a satisfiability problem for propositional logic. This set is narrowed down to a single specification through interaction with the user; the user approves or declines generated simulations of the robot's behavior in different situations.

In the category of planning problems, I first solve the problem of planning for robots that are currently executing their tasks. In such a situation, it is unclear what to take as the initial state for planning. I solve the problem by considering multiple, speculative initial states. The paths from those states are explored based on a quality function that repeatedly estimates the planning time. The second problem is a problem of reinforcement learning when the reward function is non-Markovian. The proposed solution consists of iteratively learning an automaton representing the reward function and using it to guide the exploration.

Zusammenfassung

Robotiksysteme sind auf dem Vormarsch. Fortschritte bei Hardware-Komponenten und Software-Technologie führen dazu, dass Roboter zunehmend Menschen bei Aufgaben assistieren und in deren Umfeld eigene Aufgaben bewältigen können. Das Programmieren und Konfigurieren von Robotiksystemen erfordert viele verschiedenene Fertigkeiten, was ein großes Hindernis für deren weitere Verbreitung darstellt. Diese Anforderungen führen ebenfalls dazu, dass Innovationen im Bereich robotik-gestützter Dienstleistungen nur großen Unternehmen und Organisationen vorbehalten sind.

In dieser Dissertation stelle ich ein Progammiermodell vor, dass das Entwickeln solcher Robotiksysteme vereinfachen soll. Das Modell erlaubt es dem/der NutzerIn (Software-EntwicklerIn) das Ziel des Systems *deklarativ zu spezifizieren*, während ein *Backend-System* garantiert, dass die Spezifikation sicher und zuverlässig ausgeführt wird. Als mögliches Backend-System stelle ich `Antlab` vor, welches mehrere Roboter mit unterschiedlichen Fähigkeiten steuern kann, sodass diese Spezifikationen in linearer temporaler Logik (LTL) erfüllen. Aufbauend auf die Erfahrung die ich durch Implementieren von `Antlab` erhalten habe, habe ich Probleme in Bezug auf das o.g. Programmiermodell identifiziert. Diese Probleme lassen sich grob in zwei Kategorien einteilen: Spezifikation und Planung.

Um gewünschtes Verhalten spezifizieren zu können, erläutere ich zuerst, wie eine LTL-Formel aus einer Menge von positiven und negativen Beispielen sowie ausschließlich aus positiven Beispielen hergeleitet werden kann. Auf dieser Lösung aufbauend, entwickle ich eine Methode um NutzerInnen zu helfen, ihre Intuition als LTL-Formel zu formalisieren. Hierzu wird in dieser Dissertation eine einzelne Demonstration des beabsichtigten Verhaltens mit einer Beschreibung in natürlicher Sprache durch den/die NutzerIn kombiniert. Durch eine Kodierung des Problemes als Erfüllbarkeitsproblem in propositioneller Logik werden mögliche Spezifikationen generiert. Diese Menge an Spezifikationen wird durch Interaktion mit dem/der NutzerIn auf eine einzige eingeschränkt, indem dieser/diese verschiedene Simulationen als korrekt oder falsch bewertet.

Zur Planung von Ausführungen beschäftige ich mich zuerst damit, wie Pläne für Roboter erstellt werden, die bereits eine Aufgabe erfüllen. Hierbei ist es unklar, in welchem initialen Zustand die Planung beginnen soll, weshalb wir mehrere spekulative Zustände in Betracht ziehen. Die möglichen Wege von diesen Zuständen werden mithilfe einer Qualitäts-Funktion erkundet, die die Planungszeit wiederholt evaluiert. Das zweite Problem beschäftigt sich mit “Reinforcement Learning” (zu deutsch: Bestärkendes Lernen), wobei die Nutzenfunktion von einer längeren Historie von Zuständen und Aktionen abhängt (sogenannte “non-Markovian” Nutzenfunktion). Die vorgeschlagene Lösung lernt iterative einen Automaten, der die Nutzenfunktion darstellt, und nutzt diesen um den Erkundungsprozess zu steuern.

Acknowledgements

This Ph.D. thesis comes at the end of a wonderful time, in which I encountered great scientific collaborators and even greater people. I feel a need to convey my appreciation and gratitude to those who were a part of this journey, privately or scientifically.

I spent five wonderful years in Kaiserslautern. This city (a *Großstadt*, indeed) is very often unfairly described as not exciting and lacking charm. I strongly disagree! In my years spent here, I got to appreciate its beautiful nature, passionate football fans, and the unlikely mix of natives, international students, immigrants, and personnel of the nearby military base.

In Kaiserslautern, my wife Mia and I had our first home. It is also the birthplace of our daughter Klara. I am grateful to Mia for her love, encouragement, and support; we both are grateful to Klara for helping us see the world anew.

I believe that my interest in science in general stems from the intellectually encouraging environment in which I grew up. Setting up a beautiful playground of ideas was one of many great things done by my parents, Marijana Gavran and Petar Gavran. Many thanks also to the playmates, my sisters Lidija Okroglič, Zrinka Bohr, and Kristina Gavran.

Mine and Mia's family also made it so that we never felt far away from our home country Croatia: their regular visits and partaking in our new experiences made adapting to a new city and country a smooth experience. We are also grateful to many friends who came to visit us: in particular, our quite regular visitors Marija Sertić and Filip Lavriv. Thanks for not letting the distance be an obstacle.

Max Planck Institute for Software Systems (MPI-SWS) was a perfect place for research: it is largely due to great work by people who made sure we got all the support we needed. Vera Schreiber, Susanne Girard, Corinna Kopke, Roslyn Stricker, Mouna Litz, Tobias Kaufmann, Pascal Briehl, Andreas Ries, Mary-Lou Albrecht, Torsten Koch, Frank Zimmer, Geraldine Andreson, and Christian Mickler made sure that everything non-scientific seemed extremely easy and thus enabled me to focus fully on research. Their positive attitude towards us students and willingness to assist even in private matters made us all feel at home.

In my five years here I had the opportunity to meet amazing researchers: they came as guests, interns, or students; they stayed longer or shorter, but all of them were people to learn from. Many good people were my friends and colleagues. We shared discussions, laughs, problems, cakes, runs, travels, coffees, wines, and beers.

I joined MPI-SWS after hearing a great talk given by Filip Nikšić. There, I was warmly welcomed by him and Marko Doko, Ori Lahav, Dmitry Chistikov, Marko Horvat, Isabel Valera, Johannes Kloos, Utkarsh Upadhyay, Sadegh Soudjani, Soham Chakraborty, and my great first officemate, Rayna Dimitrova. With this group, I started enjoying playing board games, quizzes, puzzles, and hours-long, free-ranging discussions.

With me and shortly thereafter, many new people joined who were forming a new generation of people at MPI-SWS. I am thanking all of them for the wonderful days we spent together. They are Burcu Kulahcioglu Ozkan, Murat Ozkan, Azalea Raad, Samira Farahani, Heiko Becker, James Robb, Kata Einarsdottir, Rosa Abbasi, Amir Mashaddi, Michalis Kokologiannakis, Aman Mathur, Lovro Rožić, Laura Stegner, Hasan Eniser, Ezgi Eniser, Felix Stutz, Tobias Blass, Clothilde Jeangodoux, Anne Kathrin Schmuck, Mehrdad Zareian, Mahmoud Salamati, Ivan Fedotov, Numair Mansour, Stanly Samuel, Marcus Pirron, Ramanathan Thinniyam, Pascal Baumann, Mitra Nasri, Nastaran Okati, Arpan Gujarati, Ralf Jung, Juraj Držić, Asia Biega, and Manohar Vanga.

Kaushik Malik and I went through the Ph.D. journey together and graduated together, on the same day: it was great to have him by my side, as a colleague and a friend. I am thankful to my officemates, Simin Oraee and Rajarshi Roy, for gossips, jokes, and discussions. Damien Zufferey introduced all of us to bouldering, while Manuel Gomez Rodriguez made sure we always went a little beyond our limits in running and skiing.

Our institute is split between two locations, Kaiserslautern and Saarbrücken. I am grateful to colleagues from Saarbrücken for always welcoming me warmly during visit days.

The third generation of students, who joined well after me, showed us how to properly have fun and create even better research and friendly connections. I was proud to see them and occasionally take part in the activities they organized. These great people are Stratis Tsirtsis, Marco Maida, Marco Perronet, Ashwani Anand, Satya Prakash Nayak, Xuan Xie, Iason Marmanis, Ana Mainhardt, Eiren Vlassi Pandi, Jie An, Jiarui Gan, Jaroslav Bendik, Julian Haas, Eleni Straitouri, Nina Corvelo Benz, and Irmak Saglam.

I would like to thank Felix Stutz and Heiko Becker for their help with translating the abstract of this thesis into German. Thanks to reviewers of the thesis, Eva Darulova, Nils Jansen, Daniel Neider, and Rupak Majumdar: their comments improved the thesis. I would like to thank the thesis defense committee chair Anthony Lin for presiding over an enjoyable and lively discussion.

My development as a researcher was greatly influenced by my collaborators. I am grateful to Damien Zufferey, Eva Darulova, Indranil Saha, Rüdiger Ehlers, Bo Wu, Jörg Hoffmann, Zhe Xu, Vinayak Prabhu, Sadegh Soudjani, Rayna Dimitrova, Ivan Fedotov, Heiko Becker, Ufuk Topcu, Filip Nikšić, Brendon Boldt, Viktor Vafeiadis, Richard Peifer, Aditya Kanade, Jean-Raphael Gaglione, Nathaniel Bos, Wheeler Ruml, Elena Glassman, and Max Fickert. I learned from them about different topics of computer science and about being a good scientist.

I interned at Microsoft Resarch with Shaz Qadeer and stayed a couple of weeks at UT Austin, in the group of Ufuk Topcu: thanks to all people there for joint work and inspiring discussions. I too had interns, with whom I enjoyed working and who made significant contributions to the projects in this thesis: Ivan Fedotov, Brendon Boldt, Jan Corazza, Eman Eman, Chuntong Gao, Akshal Aniche, Basavaraj Hampiholi, and Aaron Miller.

I was lucky to have co-authored many papers with Daniel Neider. With his precision and attention to detail, he served as a great role model for me, and he never hesitated to spend hours discussing, until we reached a perfect understanding of a problem.

Finally, I would like to mention my advisor, Rupak Majumdar. He approaches work lightly but deeply, focusing always on the essence of the problem at hand. I thoroughly enjoyed working with Rupak and learning from him. I am grateful to him for creating a perfect environment for my scientific development, opening many opportunities for me, and guiding me in my research.

There still remain many unmentioned people (and their work) who inspired me, helped me, and upgraded my understanding. Those are the people that I met at conferences, seminars, or summer schools; whose talks I attended, or whose papers I discovered by chance. I thank them, and hope that my work will touch future researchers in a similar way.

Contents

Summary **iii**

Zusammenfassung **v**

Acknowledgements **vii**

1 Introduction **1**

2 Antlab: a Multi-Robot Task Server **7**

2.1 The Programming Model 8

2.2 Antlab Implementation 15

2.3 Task Assignment and Path Planning 16

2.4 Evaluation 20

2.5 Related work 25

2.6 Conclusion 27

3 Inferring Specifications from Examples **29**

3.1 Inferring Specifications from Positive and Negative Examples 30

3.1.1 Preliminaries 32

3.1.2 Inferring a Minimal formula with a SAT-based Learning Algorithm . 34

3.1.3 A Decision Tree Based Learning Algorithm 42

3.1.4 Evaluation 45

3.1.5 Related Work 49

3.2 Inferring Specifications from Positive Examples Only 50

3.2.1 Preliminaries 51

3.2.2 Learning Universal Very-Weak Automata 53

3.2.3 Evaluation 58

3.2.4 Related Work 62

3.3 Interactive Specification Inference for Robotic Systems 63

3.3.1 Overview and Motivating Example 65

3.3.2 Formal Models for Tasks and the World 68

3.3.3 Interactive Specification Synthesis 70

3.3.4 Grammar-based Generalization of Learnt Specifications 73

3.3.5 Evaluation 76

3.3.6 Related work 84

3.4 Conclusion 86

4 Planning with Multiple Speculative Initial States 87
4.1 Introduction . . . 87
4.2 Problem Definition . . . 88
4.3 The Multiple Initial State Technique (MIST) . . . 90
4.4 MIST for Recoverable Tasks . . . 93
4.5 Evaluation . . . 95
4.6 Related Work . . . 99
4.7 Conclusion . . . 100

5 Reinforcement Learning with Non-Markovian Rewards 101
5.1 Introduction . . . 101
5.2 Preliminaries . . . 102
5.3 Joint Inference of Reward Machines and Policies (JIRP) . . . 105
5.4 JIRP Case Studies . . . 109
5.5 RL in non-Markovian Environments with Advice (JIRP^{Adv}) . . . 113
5.6 Optimal Convergence . . . 117
5.7 JIRP^{Adv} Case Studies . . . 125
5.8 Related Work . . . 128
5.9 Conclusion . . . 129

6 Conclusion 131

Bibliography 133

Curriculum Vitae 153

In memory of my teachers, Senka Sedmak and Tomislav Lipić

Chapter 1

Introduction

Robots impress any observer by demonstrations of their abilities. A flawless back-flip by the Atlas[1], versatile driving skills by cars participating in the DARPA grand challenge[2], or an efficient coordination by warehouse logistics robots[3] are some breathtaking examples. We humans are left in awe by such demonstrations for two reasons: they make us feel closer to the future as pre-imagined by artists, and we appreciate the unfathomable skill and effort by the creators of such robotic behaviors.

Indeed, programmers of an autonomous robot that operates in an unknown environment must take care of diverse challenges: the robot's dynamics, task planning under uncertainty, reacting to changes in the environment, and recovering from failures. The problems multiply for programmers of a multi-robot system. Alongside challenges of robotics come the ones of concurrent distributed systems: provisioning robots, messaging, or local and global coordination, to name a few. Underlying all these tasks is a requirement for the perfect safety of the robots' behavior, because they physically interact with humans. This complexity makes the development of each new robotic system a heroic expedition.

Not a hero, but an average person is a typical developer of any mainstream, ubiquitous technology. In a somewhat self-contradictory way, the day of true success for robot programming shall be the day when it does not impress anyone. The question is this: what is a programming model that can abstract away the shared complexity of robotic programming and let the developers focus only on the application logic?

The utility of suitable programming abstractions is apparent in many historical examples. Consider, for instance, the task of storing and accessing data. There, the job of programmers is made simple by turning them into users of distributed database systems. These systems solve the common difficult problems (such as indexing or ensuring consistency between replicas while providing high throughput), freeing the programmers to focus on the application-specific logic [184]. A more recent example are deep learning frameworks (e.g., TensorFlow [2], PyCharm [186], or Theano [213]), which take care of efficient multi-dimensional array operations, allowing the programmer to focus on the conceptual part of a problem at hand.

In this thesis, I first describe and implement `Antlab` (Chapter 2), an end-to-end robotic backend system that accepts declarative specifications from its users and executes those specifications on a fleet of robots. The effect of such a programming model is that the

[1] https://www.bostondynamics.com/atlas
[2] https://www.darpa.mil/about-us/timeline/-grand-challenge-for-autonomous-vehicles
[3] https://www.youtube.com/watch?v=ULswQgd73Tc

programmer (the user) does not have to be aware of the internals of the robots' execution. How many robots are there and what are their abilities, how to coordinate them and what to do in case a hardware component fails, or how to handle requests from independent users is all left to the backend system to deal with.

`Antlab` is one concrete implementation of the desired system, with particular design choices and tradeoffs. In the second, main part of the thesis, we isolate the individual problems that are a prerequisite for building any such multi-robot backend system. Those fall into broad categories of *specification* and *planning* problems. While they are inspired by a multi-robot backend, they are relevant as standalone problems (and are presented in full generality).

Specification: Making a Formal Specification More Natural

As its specification language, `Antlab` uses Linear Temporal Logic (LTL) [189]. This logic is a popular choice in research on *correct-by-construction* robotic systems [228, 230]. Despite its many advantages, it has a practical flaw: it is not commonly used among engineers. There are many reasons for that, but an important one is how easy it is to make a mistake when creating an LTL specification [114, 71].

We bring LTL closer to non-experts by an interactive algorithm for inferring a temporal specification. The algorithm uses a natural language description of the command and a single example by the user, given through a visual interface. As users interact with the system, the grammar of the formal language is expanded so as to generalize from the natural commands seen thus far. The language of the system keeps expanding until it converges to the jargon of the user group, while keeping the advantages of a formal language.

We implement the whole process in the system named LTLTALK. The visual interface of LTLTALK features a robot in a simulation robotic world. Users interact with the system, gradually expanding on LTL, the original specification language: each time the user gives an unparsable command, the LTLTALK asks for an example in the visual interface. Using this example, LTLTALK derives the desired specification and adds new rules to the underlying specification grammar.

The key step of our interactive algorithm is deriving a set of LTL specifications consistent with a sample (where the sample consists of positive and negative examples). We aim to learn a formula that is satisfied by all positive and none of the negative examples. Additionally, we require that the formula is minimal in size. We encode the problem as an instance of satisfiability (SAT) problem for propositional logic, and improve its scalability by using the decision tree algorithm.

We also study the problem in which only positive examples are available (e.g., the execution traces of a system). In that case, finding a minimal formula that accepts all positive examples is insufficient. (Indeed, the formula *true* that accepts all examples is a trivial solution. The same holds if we were looking for a minimal deterministic finite automaton.) Therefore, we identify a suitable class of automata, *universal very weak automata* [157], for which a meaningful definition of learning can be given, and we devise a learning algorithm based on enumerating *simple chains*.

These specification problems are described in Chapter 3 of the thesis.

Planning: Assigning Tasks to Robots and Choosing the Initial Planning State

Given an LTL specification, the backend system needs to do two things: 1) decide which robots should execute the task, and 2) provide individual plans for the recruited robots so that the specification is satisfied. While these two problems (task assignment and planning) are often considered separately [203, 154], there is a benefit in tackling them simultaneously, optimizing for the same desired metric. Our approach to the problem is based on translating the LTL specification into a propositional formula that is satisfiable exactly when there is a plan of a particular length for the original temporal specification. Using a SAT solver for propositional formulas associated with different lengths of the action-sequence, we guarantee to find an optimal plan if one exists.

The main building blocks in our solution are primitive actions available to each individual robot. Each primitive action is defined by its preconditions and postconditions, and these conditions are used in the SAT encoding. The generality of this approach enables us to support a fleet of robots with different capabilities. The joint assignment and planning is implemented in `Antlab` and is therefore presented together with the whole system in Chapter 2.

The envisioned multi-robot backend programming model presents another planning challenge: it is supposed to serve multiple independent users who can send their requests in an asynchronous manner. What kind of problems can this cause? Imagine that a request r_A was received, and the system successfully planned and recruited a group of robots to execute it. As soon as the execution has started, another request, r_B, arrives. This is new information that potentially renders all the previous planning suboptimal.

More generally, if an agent (robot, or any other entity) is executing a plan, and a need for replanning arises, an engaging problem appears: how to choose the initial state for the new planning process while the agent's state is constantly changing? This small detail is often hidden in formulations of (re-)planning problems [93].

With execution underway, this initial state must be one far enough along the current plan that the agent will not encounter it until the replanning process has finished, to allow transitioning to the new plan. But choosing an initial state that is too far along the old plan risks inefficiency: the agent's actions will not reflect the new information until the state is reached, possibly causing it to miss opportunities.

The most common solution in current systems appears to be choosing a transition point that is a fixed time ahead in the future, either as part of system design [97, 164] or through an estimate for the replanning time [149, 200]. But this is at odds with the purpose of domain-independent automated planning (which is to enable an agent to handle a variety of problems and situations with a single algorithm). In Chapter 4, I present a search algorithm that simultaneously considers multiple speculative initial states and judges how promising they are during the search itself.

Planning: Maximizing Total Reward

In some cases, the task specification is given only implicitly, through a reward that needs to be maximized by an autonomous agent (the fleet of robots, in our case). In such setup, various algorithms for the reinforcement learning (RL) problem can be used to devise the agent's policy [210].

Reinforcement learning assumes the agent's environment to be modeled as a Markov Decision Process (MDP): the states of the MDP capture the relevant information about the environment, while state-action pairs are equipped with rewards that either reinforce desired or penalize undesired behaviors. In many tasks, however, the agent receives its reward sparsely, for complex actions over a long period. This is also true for the tasks that can be described by temporal logic specifications. The setting in which the reward depends on the history of the agent's actions, and not only on the immediate state of the environment and the chosen action, does not map naturally to an MDP. In other words, the reward function is *non-Markovian*.

Clearly, whether a reward function is Markovian or non-Markovian is a modeling question: one can augment the states of any MDP with (relevant parts of the) history to obtain an equivalent problem with a Markovian reward function. However, the exact augmentation is crucial: if done naively, the augmented state space becomes too large to be computationally tractable.

To overcome this problem, algorithms that use different kinds of automata have been proposed to concisely capture the temporal nature of non-Markovian reward functions and make the RL task feasible [19, 127, 122, 40, 46]. All that work assumed that an appropriate automaton is provided to the algorithm. But this assumption is often unrealistic: reward functions (and, thus, corresponding automata) may not be fully known.

In Chapter 5, I describe an algorithm that does away with this unrealistic assumption. The algorithm combines automata learning techniques and standard reinforcement learning techniques. Furthermore, the user can provide advice to the algorithm. The advice guides the agent's exploration. At the same time, the algorithm is robust to incorrect advice.

Prior Publications

The material in the thesis has been published in the following papers:

1. Ivan Gavran, Rupak Majumdar, and Indranil Saha. "Antlab: A Multi-Robot Task Server". In: *ACM Trans. Embedded Comput. Syst.* 16.5s (2017), 190:1–190:19

2. Daniel Neider and Ivan Gavran. "Learning Linear Temporal Properties". In: *FMCAD*. IEEE, 2018, pp. 1–10

3. Rüdiger Ehlers, Ivan Gavran, and Daniel Neider. "Learning Properties in LTL $\cap$ ACTL from Positive Examples Only". In: *FMCAD*. IEEE, 2020, pp. 104–112

4. Ivan Gavran, Eva Darulova, and Rupak Majumdar. "Interactive synthesis of temporal specifications from examples and natural language". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 201:1–201:26

5. Maximilian Fickert, Ivan Gavran, Ivan Fedotov, Jörg Hoffmann, Rupak Majumdar, and Wheeler Ruml. "Choosing the Initial State for Online Replanning". In: *AAAI*. AAAI Press, 2021, pp. 12311–12319

6. Zhe Xu, Ivan Gavran, Yousef Ahmad, Rupak Majumdar, Daniel Neider, Ufuk Topcu, and Bo Wu. “Joint Inference of Reward Machines and Policies for Reinforcement Learning”. In: *ICAPS*. AAAI Press, 2020, pp. 590–598

7. Daniel Neider, Jean-Raphaël Gaglione, Ivan Gavran, Ufuk Topcu, Bo Wu, and Zhe Xu. “Advice-Guided Reinforcement Learning in a non-Markovian Environment”. In: *AAAI*. AAAI Press, 2021, pp. 9073–9080

These papers are the result of collaborations with many great researchers. This is why I will often in the text use the first person plural to describe the (indeed, *our*) results.

Chapter 2

Antlab: a Multi-Robot Task Server

In this chapter, I present `Antlab`, a system that abstracts away low-level details of multi-robot programming. As we have seen in the introduction, developing a multi-robot application successfully takes many diverse skills and the ability to integrate different modules without introducing subtle errors. `Antlab` offers a different programming model: developers specify *what needs to be done*, and it is the responsibility of `Antlab`, the backend server, to first determine *how to do it*, then *do it while monitoring the execution*, and finally to provide a summary of *what has been done* back to the developer.

`Antlab` provides an abstract programming model and a declarative *task specification* language based on linear temporal logic (LTL). Our abstract model represents the underlying world as an *occupancy map* and provides an abstraction for the set of available robots. The occupancy map is a standard data structure in robotics and represents a discrete abstraction of physical space using a set of predicates. In the programming abstraction, the user does not program individual robots or even know how many robots are there; instead, the user knows a set of *action primitives* the robots can perform, and declaratively specifies a desired temporal sequence of actions.

The propositions in a task can range over spatial locations ("reach location ℓ") as well as action primitives ("pick up," "drop"), and the temporal connectives allow expressing application-level behaviors over time. The quantification over robots allows us to specify a task without referring to individual robots (just as query languages allow expressing the intent without specifying specific servers) but also helps express coordinated behaviors ("two robots follow each other"). Specifically, the user does not need to know about the current states of the underlying robots; it is `Antlab`'s responsibility to decide which robots to assign to a task, how to schedule and plan the task, and how to ensure the system has high throughput.

When implementing `Antlab`, we had to make some core algorithmic and systems decisions. First, we describe a constraint-based combined task and path planner which produces optimal paths for a group of robots and a collection of temporal logic specifications over the occupancy grid. The planning algorithm can be implemented using an SMT solver (Z3 [173] in our implementation) or an AI planner (Metric-FF [111] in our implementation)

In practice, one must consider the dynamic and uncertain nature of the robotic environment; for example, there can be dynamic obstacles from other robots fulfilling other tasks in the system, or sensor noise and actuator imprecision. Unfortunately, most synthesis algorithms from specifications do not consider the dynamic nature of the environment [79, 102, 220, 219, 29, 202, 203] or require a complex and a priori specification of all environment events as assumptions to the synthesis procedure [136, 229, 64]. In our experience, the "ideal world" assumption leads to unexpected crashes at runtime as the abstract view does not match the real world, and the "model everything" view does not scale.

Thus, in Antlab, we implement a separation of concerns. We synthesize a path plan for the robots based on the ideal world assumption ignoring all dynamic obstacles and represent the strategy as waypoints. Then, we implement the strategy on a real robot using an off-the-shelf navigation stack [78] that is able to react to dynamic obstacles, and we augment it with a dynamic communication protocol, used by robots to resolve possible collisions between them. We track the compatibility between the ideal and the actual path, potentially re-synthesizing a strategy if possible or triggering an error to the higher layers.

Finally, we implement a distributed systems layer between the task management and the robots. This layer provides standard systems primitives such as monitoring the robots' status, provisioning robots for task execution, and tracking failures (which get increasingly frequent with growing numbers of robots).

We have evaluated Antlab on a group of TurtleBot2 robots implementing a warehouse scenario where the system has to respond to a stream of "collection" requests which require the robots to visit certain positions, gather objects, and deposit them at other positions while remaining safe and collision-free. Through our experiments, both on actual robots and in simulations, we show the potential of an end-to-end system like Antlab to scalably implement distributed robotic systems with many robots without individual reference to robots by the user.

In summary, in this chapter we will see a programming model and a runtime system for programming multi-robot applications by integrating the following components.

1. A declarative programming model based on linear temporal logic for multiple mobile robots serving requests in a workspace.

2. An algorithm for combined task and path planning for multiple robots on top of dynamic robot motion planners.

3. A runtime system to support *planning*, *robot management*, and *task execution*. Specifically, the runtime system considers real-world deployment issues such as resource management and provisioning, as well as dynamic task failures or robot failures.

2.1 The Programming Model

We will use a multi-robot warehouse management system scenario [100] as a running example for Antlab. We consider robots moving on a warehouse floor. Parts of the workspace contain objects of interest. Other parts may be blocked by obstacles, such as boxes or walls.

A task in this setting consists of a user requesting a set of objects; the task is fulfilled when a set of robots traverses the workspace to collect the requested set of objects and brings all the objects to a special part of the workspace called the *workstation*. The specific formalism for describing tasks is discussed below; informally, each task requires one or more robots to traverse a path in the workspace and carry out certain actions so that (a) the robots fulfill the request (liveness), (b) the robots remain safe, i.e., do not collide with obstacles such as walls, shelves, or other robots.

The task assignment and planning problem is to assign each task to one or more robots, and to synthesize and execute trajectories for each of these robots, such that the safety and liveness goals are met.

The System State

We now provide a formal description of the problem.

Occupancy Grids Robots move in a 2-dimensional or 3-dimensional physical space. However, the *configuration* of the robot may require specifying more dimensions, for example, to provide their velocity and orientation in the space. Thus, in general, we assume that a robot's configuration is given as a point in some compact subset of the n-dimensional Euclidean space. We represent this configuration space in a discrete way, using an n-dimensional *occupancy grid* [77]. An occupancy grid partitions the continuous space into discrete blocks using a uniform grid along each dimension and annotates each block with valuations for a set of predicates.

We assume a predicate that tells, for each block b, whether it is *occupied* or *free*. Each block is assigned a unique identifier by providing the coordinates of its center in any fixed coordinate system on $\mathbb{R}^n$. By suitably scaling the distance, we can assume that the unit of distance is one block of the workspace. Thus, the identifiers for a block's neighbors can be obtained by adding or subtracting one distance-unit. Since we are always interested in compact spaces, we can assume that the identifiers range over a finite set of elements. In what follows, we fix an occupancy grid X, and a set of predicates Π_X, which annotate the blocks of the grid.

Robots and System Configurations We assume a system of N mobile robots, where each robot has a unique identifier r from a fixed set R of identifiers. Each robot moves in an occupancy grid in discrete time. That is, we fix a discrete-time unit t and model an individual robot as a dynamical system evolving in discrete steps of t time units. The *state* σ of a robot consists of (1) its position in the space, $\sigma.x$ (which determines a unique block in the occupancy grid) and (2) its velocity configuration, $\sigma.v$, which represents current magnitude and direction of the velocity of the robot. We denote the set of all velocity configurations by V and assume it contains a value 0 denoting that the robot is stationary.[1]

A system with N robots consists of the occupancy grid together with the state of all N robots, such that a consistency condition holds: for each robot state $(\mathbf{x}, \mathbf{v})$, we have that the corresponding block of the occupancy grid is marked occupied, and no two robot states have the same positions (i.e., each block of the occupancy grid is occupied by at most one robot).

[1] In a lower-level dynamical model of a robot, one would need to specify the exact coordinates of the velocity configuration. We will work at the level of more abstract actions and, hence, we do not need to specify the exact representation of the velocity configuration space.

Example 2.1. In the warehouse example, the predicate $occupied(b)$ is true for a block b if the block is currently occupied by an obstacle, the predicate $obj(b)(o)$ is true if object o is currently in block b, the predicate $at(b)(r)$ is true if a robot with identifier r is currently occupying block b. There may be induced constraints on predicates: for example, $at(b)(r) \to \neg occupied(b)$ and $at(b)(r) \to \neg at(b)(r')$ if $r \neq r'$.

Action Primitives Robots traversing a workspace define a dynamic system whose behaviors are represented as a sequence of configurations and transitions from one configuration to the next. Following the AI planning literature [84, 147, 92], we use a set of *action primitives* A_r available to the robot r, which denote simple actions that a robot can perform at any time step. An action primitive is an abstraction of a low-level dynamical controller. For our purposes, the specific details of the control algorithm are not important. In applications where robots' actions are about traversing the environment (such as the warehouse example), we will refer to action primitives as *motion primitives*.

Associated with each action $a \in A_r$ is a *pre-condition* pre_a, which is a formula over the state variables specifying under which conditions the action a can be executed. Executing the action a incurs cost (e.g., energy expenditure), denoted by $cost(a)$. The *effect* of the action, eff_a, is a propositional formula over the state variables describing how they change after applying the action. We write $post_a(s)$ for the state of a robot after the action primitive a is used in the state s.

We use $intermediate_a(s)$ to denote the set of grid blocks through which the robot may traverse when the action $a \in A_r$ is applied at state s, including the beginning and end blocks. A *trajectory* is a sequence of states $s_0 s_1 \ldots$ such that for each $i \geq 0$, there is an action $a \in A_r$ taking the robot from s_i to s_{i+1}.

Example 2.2. Consider a ground robot with five motion primitives: $\{\mathtt{H}, \mathtt{L}, \mathtt{R}, \mathtt{U}, \mathtt{D}\}$, where the primitive H keeps the robot in the same block in the occupancy grid and the primitives L, R, U and D move the robot to the adjacent left, right, upper, and lower blocks, respectively. The availability of a motion primitive may depend on the current state of the robot. For example, if the velocity of the robot in a certain direction is high, the motion primitive to go in the opposite direction may not be available. This is specified by the pre-condition: e.g., $pre_{\mathtt{L}}$ may be $v(r) = 0$, requiring that the robot is at rest. The effect condition, similarly, states the the changes introduced by the action. For example, $eff_{\mathtt{L}}$ specifies the robot is at the block to the left and its velocity is again zero. Assuming L moves exactly one step, we have $intermediate_{\mathtt{L}}(b) = \{b, L(b)\}$, where $L(b)$ is the block immediately to the left of b.

The runtime behavior of the robots in R is described by a discrete-time transition system. A state of the system is a map from each $r \in R$ to a state of r. Let σ_1 and σ_2 be two state vectors, and α_1 be a vector containing as elements the motion primitives applied to the robots in R in state σ_1. Let $a(r) \in A_r$ be the motion primitive applied to robot $r \in R$ in state σ_1. We define a transition $\sigma_1 \xrightarrow{\alpha_1} \sigma_2$ iff

- $\sigma_1(r) \models pre_{a(r)}$ and $\sigma_2(r) = post_{a(r)}(\sigma_1)$ for all $r \in R$.

- $\forall r \in R$, $\mathit{intermediate}_{a(r)}(\sigma_1(r)) \cap \{b \mid \mathit{occupied}(b)\} = \emptyset$, which says that the trajectory of r between the states σ_1 and σ_2 does not pass through an occupied block.

- $\forall r_1, r_2 \in R$, $\mathit{intermediate}_{a(r_1)}(\sigma_1(r_1)) \cap \mathit{intermediate}_{a(r_2)}(\sigma_1(r_2)) = \emptyset$, which captures the property that robots do not collide with each other while doing an (atomic) move from state σ_1 to state σ_2

(Note that the complexity of collision avoidance grows quadratically with the number of robots. We discuss in section 2.2 how this constraint is handled dynamically.) A (multi-robot) trajectory for R is a sequence $\sigma_0\sigma_1\ldots$, where for each $i \geq 0$ and $r \in R$, the projection $\sigma_i(r)\sigma_{i+1}(r)\ldots$ is a trajectory of robot r. That is, a trajectory for the system is the collection of trajectories of all the robots.

We make the simplifying assumption that for all robots in the system, each motion primitive requires the same unit of time for execution. This assumption may not hold for robots with heterogeneous capabilities. Extending our approach to systems where motion primitives take different units of time is possible, at the cost of making the planning algorithms more complex (as was done by Raman et al. [195]).

We continue by introducing propositional logic and linear temporal logic.

Propositional Logic

Let *Var* be a set of propositional variables, which take Boolean values from $\mathbb{B} = \{0, 1\}$, with 0 corresponding to *false* and 1 corresponding to *true*). Formulas in *propositional (Boolean) logic*—which we denote by capital Greek letters—are inductively constructed as follows:

- each $x \in \mathit{Var}$ is a propositional formula; and

- if Ψ and Φ are propositional formulas, so are $\neg\Psi$ and $\Psi \vee \Phi$.

Moreover, we add syntactic sugar and allow the formulas *true*, *false*, $\Psi \wedge \Phi$, $\Psi \rightarrow \Phi$, and $\Psi \leftrightarrow \Phi$, which are defined in the usual way:

$$\begin{aligned} \mathit{true} &:= x \vee \neg x, x \in \mathit{Var} \\ \mathit{false} &:= \neg \mathit{true} \\ \Psi \wedge \Phi &:= \neg(\neg\Psi \vee \neg\Phi) \\ \Psi \rightarrow \Phi &:= \neg\Psi \vee \Phi \\ \Psi \leftrightarrow \Phi &:= (\Psi \rightarrow \Phi) \wedge (\Phi \rightarrow \Psi) \end{aligned}$$

A *propositional valuation* is a mapping $v \colon \mathit{Var} \rightarrow \mathbb{B}$, which maps propositional variables to Boolean values. The semantics of propositional logic is given by a satisfaction relation $\models$ that is inductively defined as follows:

- $v \models x$ if and only if $v(x) = 1$

- $v \models \neg\Psi$ if and only if $v \not\models \Psi$,

- $v \models \Psi \vee \Phi$ if and only if $v \models \Psi$ or $v \models \Phi$

In the case that $v \models \Phi$, we say that v *satisfies* Φ and call it a *model* for Φ. A propositional formula Φ is *satisfiable* if there exists a model v for Φ. The satisfiability problem of propositional logic is deciding whether a given formula is satisfiable. Although this problem is well-known to be NP-complete [33], modern SAT solvers implement optimized heuristic decision procedures that can check satisfiability of formulas with millions of variables [23]. Moreover, SAT solvers also return a model if the input formula is satisfiable.

Specifying Tasks: Linear Temporal Logic

We provide task specifications using *linear temporal logic* (LTL), introduced by Pnueli [189]. Let R be a finite set of indices ranging over identifiers for robots and let Q be a set of *predicates*.

Following the AI planning literature, we consider the robot-specific predicates to describe *fluents* or *action primitives*. Fluents specify predicates about the current state of the robot, e.g., $at(b)(r)$ states that the robot *r* is at the location b. Action primitives specify the capabilities of the robot. For instance, a primitive such as $pick(o)(r)$ specifies an action by which the robot picks up an object *o* in one step.

Syntax LTL is an extension of propositional Boolean logic with modalities that allow expressing temporal[2] properties. Starting with a finite, nonempty set Q of *atomic propositions*, formulas in LTL—which we denote by small Greek letters—are inductively defined as follows:

- each atomic proposition $q \in Q$ is an LTL formula;
- if ψ and φ are LTL formulas, so are $\neg\psi$, $\psi \vee \varphi$, $\mathsf{X}\,\psi$ ("next"), and $\psi\,\mathsf{U}\,\varphi$ ("until").

Again, as in the definition of propositional logic, we add syntactic sugar and allow the formulas $true, false, \psi \wedge \varphi, \psi \rightarrow \varphi, \psi \leftrightarrow \varphi$. Moreover, we define the additional temporal formulas, F ("finally") and G ("globally") by

$$\mathsf{F}\,\psi := true\,\mathsf{U}\,\psi$$
$$\mathsf{G}\,\psi := \neg\,\mathsf{F}\,\neg\psi$$

Semantics LTL formulas are interpreted over infinite words $\tau \in (2^Q)^\omega$, which in our context will be called *trajectories*. For a trajectory τ, and $i \geq 0$, we write $\tau(i)$ for the ith element in the sequence, and $\tau[i,\infty)$ for the infinite suffix of τ starting from the ith element. We define their interpretation using a *valuation function* V. This function maps pairs of LTL formulas and trajectories to Boolean values and is inductively defined as follows:

- $V(p,\tau) = 1$ if and only if $p \in \tau(0)$
- $V(\neg\varphi,\tau) = 1 - V(\varphi,\tau)$

[2] The word *temporal* here refers to order properties of a sequence of events, rather than real-time properties.

- $V(\varphi \vee \psi, \tau) = \max\{V(\varphi, \tau), V(\psi, \tau)\}$
- $V(\mathsf{X}\,\varphi, \tau) = V(\varphi, \tau[1, \infty))$
- $V(\varphi \,\mathsf{U}\, \psi, \tau) = \max_{i \geq 0}\{\min\{V(\psi, \tau[i, \infty)), \min_{0 \leq j < i}\{V(\varphi, \tau[j, \infty))\}\}\}$.

We call $V(\varphi, \tau)$ the *valuation of φ on τ* and say that the *trace τ satisfies φ* if $V(\varphi, \tau) = 1$. The same concept can be expressed by saying that a trace τ is a model for the formula φ, written $\tau \models \varphi$.

Robot Quantifiers We extend the basic logic by allowing outermost quantifiers over the set of identifiers. Formally, formulas of *quantified* LTL are built by existentially or universally quantifying identifier variables in the unary predicates in the formula:

$$\psi \ ::= \ \varphi \mid \exists r.\psi \mid \forall r.\psi$$

Notice that in a model with a fixed set of robots R, the quantifiers correspond to disjunctions or conjunctions over R.

Similarly, in our formulas, we allow syntactic sugar to quantify over blocks in the occupancy grid. For example, we write $\forall b.at(b)(r) \rightarrow \mathit{free}(b)$ to state that the robot r is at a free block.

An LTL formula is *closed* if it does not have any free variables. A *task* is a finite set of closed formulas.

Example 2.3. An example *task* in our setting consists of transporting a set of objects $\{o_i \mid i \in I\}$ to the workstation. We specify the requirement of collecting the object o_i at the location b_i as the temporal logic constraint with nested future operators:

$$\exists r. \left(\begin{array}{c} \mathsf{F}\Big(at(b_i)(r) \wedge pick(o_i)(r) \wedge \\ \mathsf{F}(at(workstation)(r) \wedge drop(o_i)(r))\Big) \end{array} \right) \tag{2.1}$$

for each $i \in I$, where *workstation* is a predicate that describes the location of the workstation. The existential quantification over r specifies that *some* robot has to fulfill the specification (go to a location, pick up an object, and, subsequently, deposit it at the workstation).

System and Environment Assumptions Tasks are always fulfilled under certain assumptions on the system and the environment. *Safety assumptions* specify invariants that must always hold, and are of the form $\mathsf{G}\,\varphi$, where φ is a Boolean combination of predicates. *Liveness assumptions* specify conditions that are satisfied infinitely often, and are of the form $\mathsf{G}\,\mathsf{F}\,\varphi$, where φ is a Boolean combination of predicates.

An example of an environment safety assumption is the presence of a static obstacle:

$$\mathsf{G}(\neg \mathit{free}(b))$$

An example of a system safety assumption is obstacle freedom:

$$\mathsf{G}(at(b)(r) \rightarrow free(b))$$

which states that the robot always avoids the static obstacles given by the predicates $\neg free(b)$. An example of an environment liveness assumption is that a location becomes free infinitely often:

$$\mathsf{G}\,\mathsf{F} free(b)$$

For example, the liveness assumption can encode that a door is infinitely often open, or that a blocked cell is eventually free.

The *planning problem* for a robot asks to compute a trajectory that satisfies the LTL specification

$$\bigwedge_j \mathsf{G}\,\psi_j^s \wedge \bigwedge_j \mathsf{G}\,\mathsf{F}\,\psi_j^l \rightarrow \bigwedge_j \varphi_j^s \wedge \varphi$$

where the antecedent encodes environment safety and liveness assumptions, and the consequent encodes the conjunction of system safety specifications and the task specification.

Dynamic Obstacles A further safety assumption on the system could be collision freedom, that is, no two robots are in the same location at the same time:

$$\forall r.\forall r'.r \neq r' \rightarrow \mathsf{G}(\bigwedge_b at(b)(r) \rightarrow \neg at(b)(r'))$$

or, more generally, freedom from colliding with dynamic obstacles. While in principle all dynamic obstacles can be modeled as part of the workspace and environment assumptions, doing this either makes conservative approximations on the moving obstacles, making the planning problem infeasible for most cases, or makes the planning problem computationally intractable. For example, in the multi-robot scenario, collision avoidance would require global knowledge of the objectives and strategies of all other robots in the workspace.

Instead, `Antlab` makes a design decision: planning is performed under the assumption that there are *no* dynamic obstacles, but local motion planning and collision avoidance protocols are implemented to "patch" the global plan based on the robot's local sensing and communication with nearby robots. Rather than generating *reactive* strategies for the LTL objectives, we plan at two levels. At the high level, we plan under an optimistic assumption (no dynamic obstacles), and generate a sequence of waypoints. At the low level, we implement a local motion planner to find paths between waypoints. The local motion planner avoids obstacles locally and triggers a re-planning step at the high level if the assumptions made at the high level are invalidated and cannot be locally patched (e.g., by a local collision avoidance protocol).

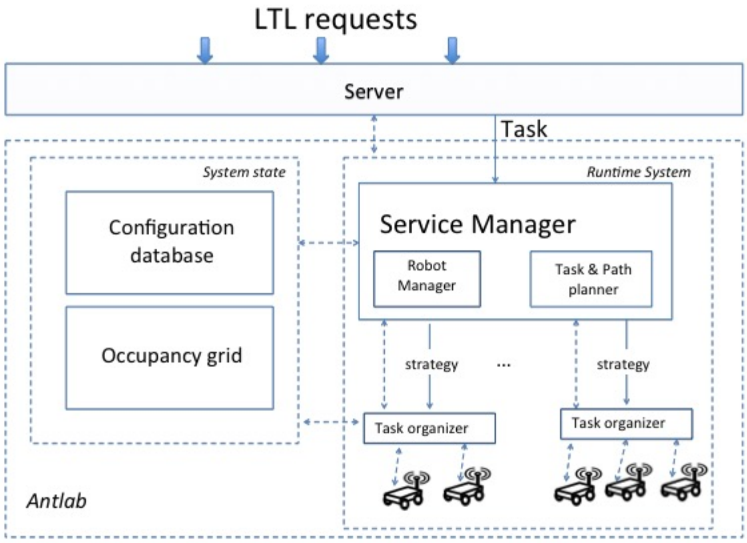


FIGURE 2.1: `Antlab` system architecture

2.2 Antlab Implementation

We now describe the implementation of `Antlab`. The lowest layer of `Antlab`, at the level of robots, uses Robot Operating System [193] (ROS). On top of ROS, we build an actor framework in Python for distributed messaging, based on the XUDD actor framework [227]. The system state is maintained in a database, and the robot manager provides an interface to the database.

Figure 2.1 presents the overall architecture of the system. A front-end application server accepts a stream of LTL tasks, possibly initiated concurrently by different users of the system. The backend consists of two main components: the *System State* and the *Runtime System*. The system state component is a database, which maintains the current state of the world (using an *occupancy grid* data structure to represent a map of the world) as well as the current state of all robots.

The Service manager component constitutes the algorithmic core of the runtime system and has two main components - *Robot Manager* and *Task & Path planner*.

The robot manager keeps track of the current position, availability, and state of all the robots. The runtime system buffers all incoming task requests and periodically invokes the task and path planner. The task and path planner assigns the buffered tasks to some of the available robots (not currently executing any other task) and provides a high-level mission plan (a trajectory for each of the assigned robots such that all tasks are satisfied, if possible). We describe the algorithmic core of the planner in Section 2.3.

Once the planning is performed, the runtime system instantiates a task organizer, which monitors the assigned robots and ensures that the tasks are executed as planned. A robot gets the "ideal" plan from the task organizer as a sequence of waypoints (coordinates to go to). The robots themselves run a ROS navigation stack to implement the low-level sensing, actuation, and motion planning to move from waypoint to waypoint. Using the full navigation stack is necessary because the strategy implemented by the task planner may not take into account dynamic obstacles, which can be realized only through dynamic sensing.

The task organizer also communicates with the robots and tracks that the current plan has not failed. A plan can fail if a robot has deviated from the original ideal plan, for example, due to dynamic obstacles, sensing or actuation imprecision, or violations of the environment assumptions. Finally, a robot itself can fail (e.g., by running out of power); hence the tasks assigned to it fail as well. The task organizer buffers any failed and unfinished tasks for a future assignment by the task and path planner.

Note that the runtime system is highly concurrent: the server, the system state database, the task organizer, and each robot are concurrent components (implemented as actors communicating via message passing). In Figure 2.1, we denote concurrent messages in the system using dotted arrows.

2.3 Task Assignment and Path Planning

Task assignment and execution in `Antlab` happens at two levels. At the static level, the task and path planner solves a *planning problem*. It takes an occupancy grid, a set of robots, and a set of task specifications and generates a trajectory for each robot in the occupancy grid. The combination of all generated trajectories satisfies all the task specifications under the assumption that there are no dynamic obstacles (including those induced by other robots in the system).

To illustrate, for a task specified by (2.1), the output of a successful synthesis problem would consist of plans for robots (one per each object o_i, not necessarily distinct), which together gather all the objects o_i and bring them to the workstation.

The output of the planning problem provides a trajectory per robot as a sequence of action primitives. Note that we assume the world is determined by the occupancy grid and the environment assumptions, and that there are no extraneous disturbances.

At the dynamic level, each robot runs a navigation stack to execute its trajectory stepwise. Each robot computes a local trajectory that executes the steps of the trajectory but takes into account dynamic obstacles.

Problem Definition

We now define the multi-robot task planning problem formally. A *planning problem* instance is given by a five tuple $\mathcal{P} = \langle R, I, A, \Omega, \varphi \rangle$, where

- R is the set of robots,
- $I : R \to X$ maps each robot to a block of the occupancy grid marking its initial state,
- A maps each $r \in R$ to a set of action primitives available to robot r (abusing the notation, we will denote $A(r)$ with A_r)
- Ω is a set of environment assumptions, and

- φ is the LTL specification of the tasks (including all system assumptions).

A multi-robot trajectory is said to be *valid* if it satisfies $\bigwedge \Omega \rightarrow \bigwedge \varphi$. For example, if φ is $\{\exists r. \mathsf{F}\, \phi_1(r), \ldots, \exists r. \mathsf{F}\, \phi_m(r)\}$, a trajectory $\sigma_0 \sigma_1 \ldots \sigma_L$ will be valid if for all $\phi \in \{\phi_1, \ldots, \phi_m\}$ there exists a σ_l, $0 \leq l \leq L$, and a robot $r \in R$, such that $\sigma_l(r)$ satisfies ϕ.

In general, trajectories can be finite or infinite objects. When they are infinite, we shall only consider their finite "lasso-shaped" representations given by a prefix and a loop [30]. Let α be a sequence of motion primitives executed by all the robots in the system. For each robot $r \in R$, for each time instant $t \in \{0, \ldots L\}$, a variable $\alpha(r)(t)$ denotes the motion primitive applied to robot r at step t.

We define the cost of a trajectory σ as

$$cost(\sigma_0 \ldots, \sigma_L) = \sum_{t=1}^{L} \sum_{r \in R} cost(\alpha(r)(t))$$

(it can be viewed as the energy consumption by the robots *in action*). In case the trajectory is infinite, the cost is defined as the weighted sum of the costs of the prefix part and the loop part. A valid trajectory σ is *cost-optimal* if $cost(\sigma) \leq cost(\sigma')$ for every valid trajectory σ'.

Note that the preceding definition does not require the robots to move in sync. Using the "rest" primitive, a robot can wait in its initial state or remain in its final state for an arbitrary amount of time to stretch its length to match with that of the overall system.

We say that an algorithm to solve the planning problem is *sound* if its output, on any problem instance, is a valid trajectory of that instance. We say that the algorithm is *complete* if, for any problem instance, whenever a valid trajectory exists, the algorithm outputs one such valid trajectory.

Constraint-Based Planning

We describe a constraint-based symbolic encoding for cost-optimal trajectories using *bounded synthesis* [206]. As mentioned before, we omit to add constraints for collision avoidance and make the optimistic assumption that trajectories of different robots do not collide. We handle collision avoidance dynamically.

Given $\mathcal{P} = \langle R, I, A, \Omega, \varphi \rangle$ and a fixed length L of the trajectory, we model behaviors of the robots as a boolean combination of linear arithmetic constraints. In the system of constraints, the action primitives of each robot at each state are considered to be the decision variables. For each robot $r \in R$, for each time instant $t \in \{0, \ldots L\}$, we introduce a variable $\sigma_r(t)$ to track the state of the rth robot and, for each time instant $t \in \{0, \ldots L-1\}$, we introduce a variable $\alpha(r)(t)$ to track the action primitive applied to robot r at step t. The objective function is to minimize $cost(\sigma)$ over valid trajectories σ. We add constraints, defined next, to ensure that a valuation to all variables $\sigma_r(t)$ defines a valid trajectory.

Initial State of the Trajectory

At the initial state of the trajectory, the robots will be in their initial locations and stationary.

$$\forall r \in R : \sigma_r(0).\mathbf{x} = I(r) \wedge \sigma_r(0).\mathbf{v} = 0 \tag{2.2}$$

Conformance Between States and Motion Primitives

For each robot $r \in R$, at each time instant t, the state $\sigma_r(t)$ should satisfy the precondition of the motion primitive applied to the robot at time instant t. Moreover, the state $\sigma_r(t)$ should be equal to the postcondition of the motion primitive applied to the robot at time instant $t-1$.

$$\begin{aligned} &\forall r \in R,\ \forall t \in \{0, \dots, L-1\} : \sigma_r(t) \models pre_{\alpha(r)(t+1)} \\ &\forall r \in R,\ \forall t \in \{1, \dots, L\} : \sigma_r(t) = post_{\sigma_r(t-1)}(\alpha(r)(t)) \end{aligned} \tag{2.3}$$

Safety Constraints

The following set of constraints (with $\forall$ again used as a shorthand for conjunctions) ensures that robots maintain safety constraints when they move from one point to another point.

$$\begin{aligned} &\forall r \in R,\ \forall t \in \{0, \dots, L-1\} : \\ &\forall b \in intermediate_{\alpha(r)(t+1)}(\sigma_r(t)) : free(b) \end{aligned} \tag{2.4}$$

We encode environment liveness assumptions $\mathsf{G}\,\mathsf{F} free(b)$ conservatively by specifying the cost of moving to the location b as a cost T incurred by waiting while $\neg free(b)$ holds, before moving from the neighboring location to b when it becomes free. By setting T much higher in comparison with the costs of all action primitives, we ensure that a cost-optimal trajectory uses the assumption only when no other options are available.

Encoding the LTL specification Finally, we provide constraints that ensure the trajectory satisfies the formula. We illustrate our approach on a simple but important special case: reaching a set of goal blocks from a set G. The following constraints ensure that the trajectory satisfies such a specification:

$$\forall g \in G, \exists r \in R, \exists t \in \{0, \dots, L-1\}. \sigma_r(t).\mathbf{x} \in g \wedge \sigma_r(t).\mathbf{v} = 0. \tag{2.5}$$

For general LTL specifications, we first replace the quantifiers with disjunctions or conjunctions to generate a pure LTL formula and then generate the constraints capturing the flattened LTL formula using the *eventuality encoding* of Biere et al. [30]. Though a trace that satisfies an LTL formula is given as an infinite execution path of the system, such a trace can be represented by a finite path in two ways: (i) the finite path is a valid prefix of all its infinite extensions (in case the specification is co-safe), (ii) a portion of the finite path can loop to generate a valid infinite path. [3]

The planning procedure searches over the values $L \in [L_{\text{min}}, L_{\text{max}}]$, which represent the length of a trajectory. For the current choice L, it generates the set of constraints with the

[3] We omit the full description of the encoding here and refer the reader to the original work [30]. Furthermore, in Chapter 3, we discuss the encoding of a reverse problem: given a set of model and non-model traces, infer a formula of minimal size. The two encodings share many conceptual similarities.

following property: if a solution to the set of constraints exists, the trajectories for the robots can be extracted from the solution; otherwise, there is no trajectory of length L satisfying the specification. In the latter case (if no solution exists), we repeat the procedure with a larger value of L.

If all primitives have the same cost, the optimality of the solution is guaranteed by the fact that the solution is found using the minimal number of steps and by elimination of unnecessary moves of robots that do not participate in fulfilling the specification. If, on the other hand, primitives have different cost, an additional optimization process is needed in order to get an optimal solution. The optimization adds additional constraint on final cost to the formula and iterates until it finds the minimal cost for which the formula is still satisfiable.

Note that we do not plan non-colliding paths for the robots; the rationale is that the planning cost is high, but the gains are poor as the uncertainty of the real world often causes imperfect executions of plans anyway. Potential collisions are handled locally. While there is no guarantee that a generated trajectory is collision-free, we do get the following completeness guarantee [202].

Theorem 2.1. Given an input motion planning problem $\mathcal{P}$ with action primitives A and a trajectory length L, if the system of constraints is not satisfiable, there does not exist a valid trajectory of length L that can be synthesized using the primitives in A.

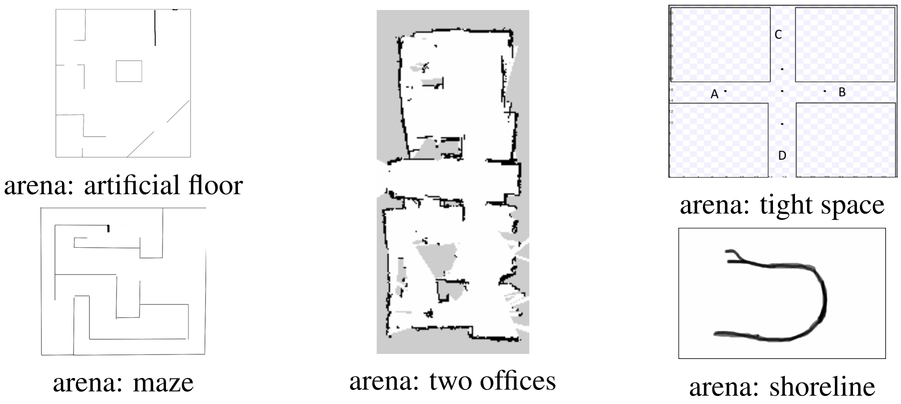
Solving Constraints We have implemented two solvers: a symbolic approach based on the SMT solver Z3 [173] and an enumerative search approach based on domain-independent AI planners (e.g., Metric-FF [111]). For AI planning, we encode the constraints into PDDL, the standard planning language. We compare the two methods in Section 2.4. The choice of the solver is transparent to the rest of the implementation.

We note that the SMT approach supports all LTL specifications "off-the-shelf'. However, while in principle extensions of PDDL support almost all LTL constraints, in our experience, very few planners stably support full LTL. Thus, we pre-process the LTL formulas to provide a sequence of reachability constraints to Metric-FF. (In that way, we are able to support a relevant subset of LTL specifications.)

Dynamic Implementation of Trajectories

The trajectories of the robots found by the planner are implemented on the individual robots using a motion planner of the ROS navigation stack. The motion planner handles dynamic obstacles as well as potential collisions with other robots. Upon sensing an obstacle, the navigation stack adds it to the local cost map and tries to find a local motion plan to avoid the obstacle and continue with the planned trajectory. In cases where local motion planning and obstacle avoidance cannot find a feasible plan, the navigation stack starts recovery behaviors and restarts global planning for the current robot.

We handle potential inter-robot collisions with a local collision avoidance protocol. As noted by Hennes et al. [109], robots are active agents, and treating them as pure dynamic obstacles leads to inefficient trajectories and —in rare cases— collisions. Furthermore, treating



arena: artificial floor

arena: maze

arena: two offices

arena: tight space

arena: shoreline

FIGURE 2.2: Workspaces used in the experiments

robots as passive obstacles often causes robots to approach too close to each other, which leads to long-lasting recoveries of their local planners and oscillations in the trajectories. Thus, in our implementation, we additionally add local communication capability to robots and implement local collision avoidance strategies. By broadcasting its position and velocity to robots in their neighborhood, and by listening to their messages, a robot can determine its local strategy to avoid collisions.

2.4 Evaluation

We evaluate `Antlab`'s behavior on maps of different sizes and shapes and with the different number of robots in the system. We test how well `Antlab` can handle robot (hardware) failure and dynamic collision situations between the agents. Furthermore, we try to identify situations for which multi-robot planning and assignment is meaningful (where the advantage over a simple heuristic is significant). Finally, we examine how the increase in the number of concurrent requests influences the overall performance.

In order to examine different properties, we use different arenas (shown in Figure 2.2). The *arena: two offices* was obtained by mapping two offices in our building with TurtleBots using the ROS SLAM (Simultaneous Localization and Mapping) gmapping package [91]. All the other arenas are artificially created for simulation. We also tested `Antlab` in the *empty arena*: a rectangular space with no obstacles in it (this is justified by a scenario in which robots can go under all the obstacles, as is the case for warehouse pods). The proof of concept is done with 3 TurtleBots in the environment mapped in *arena: two offices*. The testing is done in the Stage simulation environment [4], which enabled us to test on different arenas and vary the number of robots in the arena, hence varying the coverage of the area by robots.

LTL Planning Time and Execution Cost

First, we evaluate `Antlab`'s task assignment and planning on a number of LTL specifications on maps of different sizes and with the different number of robots in the system. We consider the following LTL specifications in our experiments:

TABLE 2.1: Planning and optimization time (in seconds) and trajectory cost for ϕ_2 with increasing number of robots and increasing map size

Small-Size Map				Medium-Size Map				Large-Size Map			
# Robots	average planning time	average optimization time	cost	# Robots	average planning time	average optimization time	cost	# Robots	average planning time	average optimization time	cost
2	4.10	1.92	40.14	2	9.12	171.29	47.72	2	19.89	218.00	85.65
4	0.92	1.75	15.78	4	2.89	164.73	39.16	4	6.90	111.18	44.94
6	0.87	0.87	12.93	6	1.64	6.64	20.72	6	5.57	110.91	42.01
8	0.70	0.86	11.28	8	1.21	1.74	14.55	8	5.56	54.96	34.85

TABLE 2.2: Planning and optimization time (in seconds) and trajectory cost for LTL properties for 8 robots and increasing map size

Property	Small-Size Map			Medium-Size Map			Large-Size Map		
	average planning time	average optimization time	cost	average planning time	average optimization time	cost	average planning time	average optimization time	cost
Repetitive Pick and Drop	0.20	0.50	4.50	1.08	1.92	8.44	2.84	71.47	11.35
Selective Action	0.37	0.62	11.05	1.66	7.57	19.38	6.87	165.92	28.15
Pick and Drop Ordered	0.70	0.86	11.29	1.21	1.74	14.55	5.56	54.96	34.85
Regions Coverage	0.42	0.48	4.44	1.03	1.46	5.44	2.60	0.91	13.75
Sensor measurement	0.58	0.62	13.61	2.40	56.48	22.4	6.97	105.95	35.35

(ϕ_1) *Pick and drop repetitively:* Repeatedly pick an object from the location ℓ and drop it to the location ℓ' (provided that the object would be infinitely often placed at ℓ): $\mathsf{G}\,\mathsf{F}\,put(\ell) \Rightarrow \exists r.\, \mathsf{G}\,\mathsf{F}(pick(\ell)(r) \wedge \mathsf{F}\,drop(\ell')(r))$

(ϕ_2) *Picking and dropping with enforced order:* Pick $p1$ and $p2$, but once picked, drop the object at $d1$ or $d2$ respectively, before picking anything else: $\exists r_1, r_2 \;:\; (\mathsf{F}\,pick(p1)(r_1) \wedge \mathsf{G}(pick(p1)(r_1) \rightarrow ((\neg pick(p2)(r_2) \wedge \neg drop(d2)(r_2))\,\mathsf{U}\,drop(d1)(r_1)))) \wedge (\mathsf{F}\,pick(p2)(r_2) \wedge \mathsf{G}(pick(p2)(r_2) \rightarrow ((\neg pick(p1)(r_1) \wedge \neg drop(d1)(r_1))\,\mathsf{U}\,drop(d2)(r_2))))$

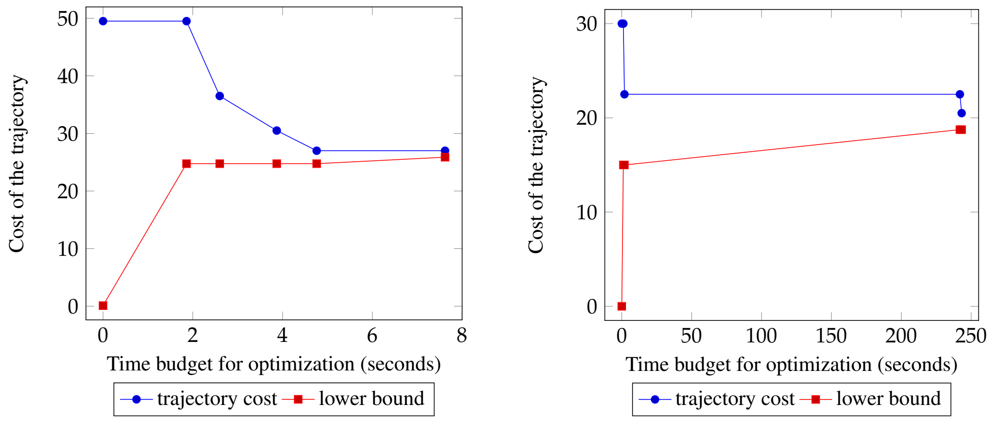
(ϕ_3) *Selective action and measurement with safety restrictions:* The propositions $a1$ and $a2$ denote the operation of acting on the certain place; $m1$, $m2$ denote the subsequent measurement at a different place; $s1$ and $s2$ denote locations that should be occupied at each moment, for safety reasons: $\exists r_1, r_2, r_3, r_4 : \mathsf{F}(((a1(r_1) \wedge \mathsf{F}\,m1(r_1)) \vee (a2(r_2) \wedge \mathsf{F}\,m2(r_2)) \wedge \mathsf{G}(s1(r_3) \wedge s2(r_4)))$

(ϕ_4) *Regions coverage:* From some point onwards, ensure that one of the regions denoted by $s1$, $s2$, $s3$, and $s4$ is always covered by a robot and then eventually point l_1 is visited: $\exists r_1, r_2, r_3, r_4, r_5 : \mathsf{F}(\mathsf{F}\,at(l_1)(r_1) \wedge \mathsf{G}(s1(r_2) \vee s2(r_3) \vee s3(r_4) \vee s4(r_5)))$

(ϕ_5) *Simultaneous sensor measurement:* Measure sensor values at locations $m1$, $m2$, and $m3$ simultaneously, and report the result at one of the report locations $g1, g2, g3$: $\exists r_1, r_2, r_3 : \mathsf{F}((m1(r_1) \wedge m2(r_2) \wedge m3(r_3)) \wedge (\mathsf{F}\,g1(r_1) \vee \mathsf{F}\,g2(r_2) \vee \mathsf{F}\,g3(r_3))$

We created 10 different instances of each of those formulas by picking locations randomly. We used the map *artificial floor* with grid sizes 550 (small, 22 × 25), 864 (medium, 27 × 32), and 2250 (large, 45 × 50). Planning times depend on the map size in grid units. Depending on the motion primitives available, these can be of different size. We used a grid unit of 0.4m (corresponding to the size of the Turtlebot base station).

[IG: make these tables bigger]



(a) For a 3 robots instance

(b) For a 6 robots instance

FIGURE 2.3: Cost improvement over time

TABLE 2.3: Comparing SMT and AI planning for reachability

	Avg Planning Time (sec)			Avg Plan Cost		
Arena name	shoreline	floor	maze	shoreline	floor	maze
SMT-based	11.56	44.63	20	37	31.37	58
Metric-FF	5.44	6.3	15	22	24.25	54

SMT-based Planning Table 2.1 shows times for the SMT-based planner on the different maps as the number of robots increase. Planning (resp., optimization) time is the average time (over ten instances) to get the first plan (resp., optimal plan). Cost is the optimal value to execute the plan. Increasing the coverage of the arena with robots reduces the cost and makes the planning time shorter (first satisfying instance is reached sooner), as each robot's plan is shorter. Table 2.2 shows the same data for ϕ_1-ϕ_5 for 8 robots.

Anytime Optimization The time to find a feasible plan is often significantly shorter than finding the optimal plan for large maps. Once the first satisfying assignment is found, the optimization process iteratively improves the cost: it keeps the number of steps fixed, and invokes the planner with an upper bound on the cost. Figure 2.3 shows how the plans approach the optimal (one with 3 robots, and the other with 6 robots on the map). As expected, the biggest improvements are achieved at the very beginning of the process. Thus, the planning can be run in "anytime" mode and stopped once a cost budget is met or a time budget is exceeded.

AI Planning To compare with an AI planner, we used Temporal Fast Downward (TFD) [80], a planner with temporal logic support. Unfortunately, TFD did not finish on any of these examples. Even on a small empty map (17×19) and a simple reachability objective, TFD was two orders of magnitude slower (103s vs. 0.8s); the time was mostly spent in constructing the state space in memory.

Most classical AI planners optimize for reachability. Thus, we compare the performance of the SMT planner with Metric-FF [111], a state-of-the-art planner, on reachability objectives $\bigwedge_{i=1}^{n} \mathsf{F}\, p_i$, where predicates p_i are locations. Each experiment fixed one of three different arenas

TABLE 2.4: Effect of joint task assignment and planning

	Planning Time (sec)			Plan Cost		
Arena name	shoreline	floor	maze	shoreline	floor	maze
Heuristic assignment	0	0	0	92	25	91
SMT-based assignment and planning	11.56	44.63	20	37	31.37	58

and averaged over 10 different specifications constructed by picking locations randomly. In order to get faster planning for SMT, we relax the optimality requirement and stop the optimization process once it is within 15 step-units (around 6 meters) of the optimal plan and set a timeout of 240s for a request. Metric-FF does not provide any optimality guarantees. Nonetheless, it outperforms SMT-planner both in planning time and in cost (due to suboptimality tolerance), as can be seen in Table 2.3 Both planners time out on two planning tasks.

Our conclusion is that classical AI planners are a better choice for simple reachability goals. However, going beyond reachability objectives requires the SMT planner.

Is Joint Task and Path Planning Necessary? Finally, we evaluate the costs and benefits of using the joint task assignment and planning presented in Section 2.3 in comparison to a naive heuristic that picks the "closest" robot. For general LTL objectives, it is not clear how to define closeness, so we fix reachability objectives and pick the robot closest to the goal in Euclidean metric, ignoring obstacles. The heuristic task assignment is "zero cost", but clearly suboptimal as it does not account for obstacles. The task-to-robot assignment of the algorithms presented in Section 2.3 comes at a cost of the time required for path planning. We use the same setup as when comparing AI planning and SMT-based planning: three arenas, ten reachability specifications.

Table 2.4 shows the summary. For maps for which Euclidean distance is far from the actual traveling distance (such as Shoreline or Maze), joint planning and task assignment can yield significantly better plans. If Euclidean distance is a good approximation (as in the arena *artificial floor*), then the costs are approximately the same (in this experiment, heuristic assignment has a smaller cost due to premature ending of the optimization process in the planner).

Response Time with Concurrency

Next, we measure the effect of concurrent scheduling of tasks –what are the benefits of parallelization and what is the cost of synchronization of robots executing their task at the same time. The experiment is set so that n requests are scheduled concurrently, where each request is a set of tasks, and as soon as a request finishes, the next one is scheduled (thus, there are always n requests executing concurrently). We run 10 requests, each with 3 to 6 tasks, on *Artificial floor* with 8 robots and let n grow up to 5.

As the results (Figure 2.4) suggest, response time improves until we reach 4 simultaneous requests. The delays come from two sources: robot synchronization (which is especially significant if two robots get stuck in a deadlock situation) and sub-optimality due to robots

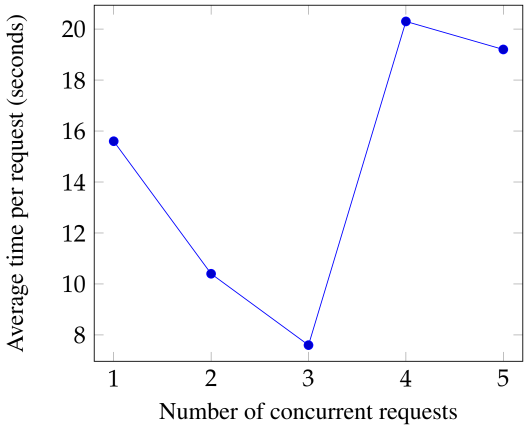


FIGURE 2.4: Execution time vs number of concurrent batches

being busy with a different batch. That is, if the robots in one part of the arena are busy, and a new request to that part arrives, the planner assigns this request to robots that may be far away. A possible solution is to plan based on a partitioning of the arena and only assign robots nearby, even if there is a delay until finishing the current task or to interrupt an active robot to give it a new local task. We address this problem in a more principled way in Chapter 4 by developing a method to plan for already active robots.

Failure Resilience

In this experiment, we measure the effect of `Antlab`'s fault tolerance and task re-assignment on performance. When a robot crashes, it becomes a static obstacle and the system has to reassign its task. Crashed robots can prevent others from finishing the task (e.g., by blocking a path). We explore how often that might happen.

We model crashes probabilistically: a crash can occur in each second with probability p. The probability of a robot not crashing in k seconds is $(1-p)^k$. We fix the number of tasks to 10, and randomly generate the number of locations per task (the locations are generated so that they are, considering static obstacles, achievable). We run experiments for the coverage of 28 square meters (175 square grid units) per robot and vary the probability p. We measure the average time of task completion, as well as the number of locations that `Antlab` did not manage to reach.

We run 60 trials of each experiment. The parameter p is tested for values $\{0, 0.003, 0.005, 0.006, 0.01\}$ which is equivalent to the following probabilities of staying crash-free for a minute: $\{1, 0.83, 0.74, 0.69, 0.55\}$. The initial setting consists of an obstacle-free arena with 9 robots, and robots are given altogether 24 locations to visit in 10 tasks. For higher values of p, we could not obtain meaningful results because often, all the robots would crash before completing the tasks.

Table 2.5 summarizes the results. The baseline is the execution without crashes, which takes about two and half minutes to complete. The execution time increases with the probability of failure. For $p = 0.01$, the average number of locations that the system was unable to visit is tolerable 2.23. However, the execution time almost doubles compared to the baseline.

p	Average execution time (sec)	Average number of locations not visited
0	156.36	0
0.003	220.84	0.42
0.005	241.28	1.13
0.006	244.72	1.12
0.01	290	2.23

TABLE 2.5: Failure resilience test results

Number of patrolling robots	Execution time with collision avoidance turned on (sec)	Execution time with collision avoidance turned off (sec)
0	365	
1	498	563
2	493	551
3	459	667
4	479	596

TABLE 2.6: Navigation in crammed environments

Tight Space Manoeuvring

Finally, we explore the effect of the dynamic collision avoidance mechanism on the overall performance. Due to its assignment mechanisms, Antlab avoids having more robots than necessary in one place (by assigning multiple tasks to a single robot, minimizing the energy spent). Thus, we construct a specific configuration where many robots are located in the same area and therefore collisions are common. We use *arena: tight space* (Figure 2.2). Four robots are set to patrolling from side to side ($A \leftrightarrow B$ and $C \leftrightarrow D$). An additional robot that starts in the middle (*task robot*) gets 10 tasks chosen from A, B, C, and D, in a round-robin fashion. It is also forced to visit the central location between each task, causing congestion in the middle. We run this experiment with 0 to 4 other patrolling robots, and set the priority of the task robot as the highest. We run with both collision avoidance turned on and turned off. Note that even with the collision avoidance turned off, robots will not collide all the time as the lower level navigation stack implements dynamic obstacle avoidance. However, turning collision avoidance on allows robots to communicate their position and goals.

Table 2.6 shows execution times. We note that without the collision avoidance mechanism turned on, robots run into deadlock situations, where none of their recovery behaviors manages to find the way out and the navigation stack gets stuck. Collision avoidance manages to lower the number of deadlocks, though not to eliminate them completely. The results also show that the execution time does not increase with increasing number of patrolling robots. A possible explanation would be that more robots approaching each other might cause earlier stopping and therefore prevent deadlock-like configurations from happening.

2.5 Related work

ROS [193] is a standard software framework for individual robots. Antlab uses ROS at the single robot level. A continuation project, ROS2 [192] was in a nascent phase at the time of developing Antlab. Now it matured and it can handle some of the aspects of Antlab (regarding support for multiple robots), but does not provide synthesis from higher level declarative specifications. The StarL [150] framework unifies programming, specification, and verification of distributed robotic systems. It does not, however, address the specific challenges of a robot service: managing robots, dealing with incoming task requests, dynamic obstacle avoidance, and fault-tolerance.

Planning is a classical problem in AI and robotics [201, 55, 84, 92, 143]. As our work's primary focus is on multiple robots, we discuss that setting in planning. We compare the performance of domain-independent planners to the custom SMT-based planner for LTL tasks. SMT-based synthesis of motion plans was applied to a single robot moving in a workspace containing rectangular obstacles [119] and in synthesizing integrated task and motion plans from partially specified tasks [174, 224]. In multi-robot motion planning, a SMT solver was employed to synthesize a plan for a group of robots from a safe-LTL formula [202] and from a specification that requires a group of robots to reach their preassigned goals while avoiding obstacles and collision with other robots [203]. Unlike these works, Antlab supports the stream of incoming tasks and it does not require that the assignment of robots to tasks is provided to the planning algorithm.

Several prior papers address the problem of generating trajectories for multi-robot systems where the robots are preassigned a set of tasks, whereas Antlab simultaneously assigns robots to tasks and generates trajectories. A subset of these works (e.g. [79, 29, 202, 203]) adopts a centralized approach where a central server is used to synthesize trajectories for a set of robots to reach a set of pre-specified goal locations. The others employ decentralized prioritized planning (e.g. [102, 220, 219]) where, given a fixed set of tasks, the robots in the system coordinate with each other asynchronously to compute the trajectories. Similarly to our work, Turpin et al. [215] assign goals and plan a trajectory simultaneously, but only for simple reachability tasks and requiring that no robot is left idle, rather than optimizing the total cost. Drona [65], a framework for distributed mobile robotics, introduces a multi-robot decentralized motion planner but–unlike Antlab–assumes that robots are preassigned their tasks and that there are no uncertainties in the environment.

In the aforementioned work, as well as in Antlab, the robots are coupled by a no-collision requirement or by some light synchronization requirements. A much tighter coupling, where the actions and states of multiple agents are mutually dependent, is studied in the work of Banusic et al. [24]. The agents there can be seen as components of a larger assembly.

There is an increased interest in using LTL for synthesizing reactive motion plans automatically [136, 229, 64]. However, automated synthesis algorithms scale poorly both with the number of robots and the size of the workspace, and have not proven suitable for multi-robot applications. In order to tackle the scalability issue, several papers synthesize motion plans compositionally (considering conjunctive LTL formulas and synthesizing each conjunct separately) [6, 85]. Even though the compositional approach sometimes outperforms centralized ones, the largest examples so synthesized are still far from real-world test cases. LTLMoP [86] is a modular toolkit that covers different aspects of creating controllers synthesized from LTL or structured English specifications. The main difference from our work is that LTLMoP still resides in a "clean" world assumption, not taking into account robot crashes, imperfect executions of plans, etc.

Our work starts from given motion primitives for the robots of the system. There is orthogonal work that synthesizes motion primitives in a smart way, to achieve just the right level of abstraction coarseness [117, 118], or to be resilient to occasional disturbances [205].

2.6 Conclusion

We have described the design and the implementation of `Antlab`, an infrastructure system for "robots as a service" programming model. We consider `Antlab` to be a successful proof-of-concept for an end-to-end systems for managing and using robot teams. There are, however, many open questions on the way to a production-ready system.

Scalability in `Antlab` can be reached only by performing careful tradeoffs when choosing which assignment or planning algorithm to use and how many robots to have in the workspace: the planning procedure is worst-case exponential in the size of the workspace and the number of robots.

`Antlab` is built on top of existing infrastructure for single robots (ROS). It is, thus, vulnerable to all potential problems at that layer (such as noise and imprecision in sensing and actuation, limited collision-avoidance protocols, etc.). Therefore, we cannot give strong end-to-end real-time guarantees about the execution.

LTL, the task language of `Antlab`, does not support some typical "swarm" actions, such as coordinated work by many robots ("follow the leader"). It would be interesting to extend the declarative task language to tasks of this nature. Further, LTL is still too low-level to describe complex workflows such as industrial processing.

Another pressing problem with LTL as a specification language is that, although succinct, it is a language difficult to master. Aiming to democratize the multi-robot systems development, we ought to help users work with LTL in a more intuitive and reliable manner. We present one way to do so in Chapter 3, based on learning specifications from examples and natural language.

The implementation of `Antlab` features a preliminary version of handling concurrent requests: only the non-busy robots are taken into consideration for executing a newly arrived task. In Chapter 4, we will see how to plan in a principled way also for the robots that are already executing a task.

Chapter 3

Inferring Specifications from Examples

Good specifications are a cornerstone of our efforts to engineer better systems. In Chapter 2, I have proposed Antlab, the system that makes programming of multi-robots applications easier by allowing its users to only provide an LTL specification. And while a specification is less detailed than a corresponding implementation, the problem remains: how to translate the intuition of what the system should do into a formal specification?

Specifications are necessary in many other contexts as well. For instance, a model checking algorithm [21] can prove a system design to be correct. But this proof is only meaningful if the specification correctly captures the requirements of the application. Similarly, while the process of synthesizing reactive systems from their specifications is well researched [206, 196, 158, 169, 96], the success of the synthesized behavior depends on having correct specifications, both of the task and of the environment in which the system operates. The same is true for our efforts to tame learning systems, future or present [125, 48].

Despite the critical role they play, specifications are often taken for granted, assumed to be given by the user. For most users, however, writing correct and complete specifications is hard [114]. Crucial properties of the application are easy to miss, and translating one's intuition into formal representation such as automata or Linear Temporal Logic (LTL) formulas is not straightforward.

A solution to this problem is to allow the users to express their intent in a simpler way and give them automated tools for inferring the full specification. An easy way for users to express their intent is by providing some examples of desired and undesired system's behavior [191]. Other than from users, such examples can also be obtained by examining an existing system [170, 135, 26]. Because a set of examples constitutes only a partial specification, it is often useful to additionally consider any other available modalities that can help resolve ambiguity, such as natural language [197, 145] or interaction with the user [70].

In this chapter, we will consider the problems of synthesizing temporal specifications. We will gradually build towards a multi-modal specification synthesis for a robotic system. As a first step towards it, in Section 3.1, we will consider a problem of inferring a shortest LTL specification that separates two sets of ultimately periodic *traces*, one consisting of positive examples and the other of negative examples.

Unfortunately, both positive *and* negative example traces are not always available in practice. (For instance, when inferring specifications of a system that can only be observed, it is easy to extract the set of positive traces, but very difficult to prove that some examples can never occur.) Absent some domain-specific knowledge on how to derive negative examples, we are left with the problem of learning a specification from positive examples only. This, however, is an ill-defined problem: the shortest specification that accepts the given example traces is *true*, which accepts *all* possible traces, providing us with no useful information. On the other end of the spectrum is a formula that accepts only the given examples and no other trace. Both extremes make little sense.

Thinking metaphorically, the all-encompassing formula *true* is very *loose* ,whereas the formula that accepts only the given example is very *tight* To make this learning problem well-posed, we need to parameterize the problem with a tightness parameter [18, 126]. The intuitive idea of tightness must be concretized so that it enables an efficient learning procedure and that the learned specifications empirically capture relevant properties of the system. We recognize universally very weak automata (UVWs) over infinite words as a suitable formalism. For a UVW, the tightness parameter can be concretized syntactically, making the learning procedure efficient. In Section 3.2, we will see a chain-enumeration algorithm for inferring n-tight UVWs from positive examples only.

Finally, in Section 3.3, I will present LTLTALK, the robotic system that helps its users formalize their intent into a correct LTL specification. LTLTALK synthesizes a set of candidate LTL specifications from a single example trace and a natural language description of the command. In order to choose the correct specification among the set of candidates, LTLTALK interacts with the user by showing different robotic worlds and the robot's actions in them, and asking the user to say whether they match the original intent. Additionally, we use a grammar extension mechanism and a semantic parser to generalize synthesized specifications to parametric task descriptions for subsequent use. Thus, the overall system functions as a way for non-expert users to mold the original formal language (LTL) into a language that comes naturally to them.

3.1 Inferring Specifications from Positive and Negative Examples

In this section, we consider the problem of inferring a formula from both positive and negative examples. This problem arises when formalizing our intuition about the desired behavior or when attempting to make sense of the observed behavior of complex systems (e.g., for debugging, specification mining, or modernization of legacy systems). We focus on learning LTL formulas that distinguish desirable from undesirable executions of a system.

The precise problem we are solving is the following: given a sample $\mathcal{S}$ consisting of two finite sets, one with positive and the other with negative examples, learn an LTL formula φ that is consistent with $\mathcal{S}$ in the sense that all positive examples satisfy φ, whereas all negative examples violate φ.[1] To be as general and as succinct as possible, we here consider examples to be infinite, ultimately periodic words (e.g., traces of a non-terminating system) and assume the standard syntax of LTL. However, our techniques can easily be adapted to the case of finite words and extend smoothly to arbitrary future-time temporal operators, such as “release”, “weak until”, and so on. We fix all necessary definitions and notations in Section 3.1.1.

We will see two novel learning algorithms for LTL formulas from data in this section. The first one is based on SAT solving. The second one leverages learning decision trees as a way to overcome the scalability issues of the first one.

SAT-Based Learning Algorithm

The idea of our first algorithm, presented in Section 3.1.2, is to reduce the problem of learning an LTL formula to a series of satisfiability problems in propositional Boolean logic. With that reduction, we can take advantage of many optimized off-the-shelf SAT solvers to find the solution. Inspired by ideas from bounded model checking [31], our learning algorithm produces a series of propositional formulas $\Phi_n^{\mathcal{S}}$, for increasing values of $n \in \mathbb{N} \setminus \{0\}$ that depend on the sample $\mathcal{S}$. These propositional formulas are so constructed that from a satisfying assignment for them, we can reconstruct an LTL formula consistent with the given sample. By increasing the value of n until $\Phi_n^{\mathcal{S}}$ becomes satisfiable, we obtain an effective algorithm that learns an LTL formula that is guaranteed to classify the examples correctly (given that the sample is non-contradictory).

By design, our SAT-based learning algorithm has three crucial features. First, our algorithm learns LTL formulas of minimal size (i.e., with the minimal number of subformulas). Smaller formulas are generally easier to comprehend by humans than larger ones. Second, once an LTL formula has been learned, our algorithm can be queried for further, distinct formulas consistent with the sample. We believe that this feature is important in practice as it allows generating multiple explanations for the observed data. Third, our algorithm does not rely on an a priori given set of templates. To the best of our knowledge, our SAT-based algorithm was in fact the first learning algorithm not restricted to a fixed class of templates. However, restrictions to the shape of LTL formulas (e.g., the popular GR(1)-fragment of LTL [35]) can easily be encoded if desired.

Learning Algorithm Based on Decision Trees

Our second learning algorithm, which we present in Section 3.1.3, trades in the guarantee of finding minimal solutions for better scalability. The key idea is to perform the learning in two phases. In the first phase, we run the SAT-based learning algorithm described above on various subsets of the examples. This results in a (small) number of LTL formulas, named “LTL primitives”, that classify at least these subsets correctly. In the second phase, we use a standard learning algorithm for decision trees [194] to learn a Boolean combination of these LTL primitives that classifies the whole set of examples correctly, though it might not be

[1] Note that, in contrast to classical computational learning theory [217] and modern statistical machine learning [36, 168], we seek to learn a formula that does not make mistakes on the examples. In other words, it does not assume any noise present in the sample.

minimal. Even though the minimality guarantee is lost, the resulting formulas are still readable for humans (which was the original motivation to look for small formulas): a well-known advantage of decision trees is that they are simple to comprehend due to their rule-based structure.

When learning LTL primitives, we need to carefully choose the subsets of examples so that the learned primitives

1. separate all pairs of positive and negative examples, and
2. are general enough to permit "small" decision trees.

While the first requirement is precise, it allows for a large number of subset selections (some of which may be meaningless, e.g., a full set of provided examples). The second requirement is of a more descriptive nature, and we use it to empirically select the best strategies for choosing subsets.

3.1.1 Preliminaries

Let us begin by setting up basic definitions and notation used throughout.

Finite and Infinite Words

An *alphabet* Σ is a nonempty, finite set. The elements of this set are called *symbols*.

A *finite word* over an alphabet Σ is a sequence $u = a_0 \dots a_n$ of symbols $a_i \in \Sigma$, $i \in \{0, \dots, n\}$. The empty sequence is called *empty word*, and is denoted by ε. The length of a finite word u is denoted by $|u|$, where $|\varepsilon| = 0$. Moreover, Σ^* denotes the set of all finite words over the alphabet Σ, while $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ is the set of all finite, non-empty words over Σ.

An *infinite word* over Σ is an infinite sequence $\tau = a_0 a_1 \dots$ of symbols $a_i \in \Sigma$, $i \in \mathbb{N}$. (We use the symbol τ to stand for *trace*, a common name for infinite words created by a system's execution.) The ith symbol of an infinite word τ is denoted by $\tau(i)$, and the infinite suffix starting at position j by $\tau[j, \infty)$. Given $u \in \Sigma^+$, the infinite word $u^\omega = uu \dots \in \Sigma^\omega$ is the infinite repetition of u. An infinite word τ is called *ultimately periodic* if it is of the form $\tau = uv^\omega$ for a word $u \in \Sigma^*$ and a word $v \in \Sigma^+$. Finally, Σ^ω denotes the set of all infinite words over the alphabet Σ.

Propositional Boolean Logic and Linear Temporal Logic

Here we remind ourselves of the definitions from Section 2.1. For *Var*, a set of propositional variables, we defined formulas in *propositional (Boolean) logic* by:

- each $x \in \mathit{Var}$ is a propositional formula; and
- if Ψ and Φ are propositional formulas, so are $\neg\Psi$ and $\Psi \vee \Phi$.

Moreover, we added syntactic sugar to allow the formulas *true*, *false*, $\Psi \wedge \Phi$, and $\Psi \rightarrow \Phi$.

Linear Temporal Logic (LTL) was defined over a finite, nonempty set Q of *atomic propositions* by

- each atomic proposition $q \in Q$ is an LTL formula;
- if ψ and φ are LTL formulas, so are $\neg\psi$, $\psi \vee \varphi$, $\mathsf{X}\,\psi$ ("next"), and $\psi\,\mathsf{U}\,\varphi$ ("until").

By this inductive definition, every formula consists of its *main operator* and zero, one, or two *child subformulas*. Again, as in the definition of propositional logic, we added syntactic sugar for the formulas $\mathit{true}, \mathit{false}, \psi \wedge \varphi, \psi \rightarrow \varphi$, and for the additional temporal formulas, $\mathsf{F}\,\varphi$ ("finally") and $\mathsf{G}\,\varphi$ ("globally").

While defining F and G is commonly done for reasons of practicality, in the context of our problem it plays a more important role. Since we are interested in finding a formula of the smallest size, and such derived operators reduce the size of the formula, they directly influence what will be considered a correct solution. Such bias is only meaningful insofar as the derived operators are *understandable* for humans.

To define the notion of *size* of an LTL formula φ, we introduce a set of its subformulas $\mathit{subf}(\varphi)$. The set $\mathit{subf}(\varphi)$ is defined inductively over the structure of the formula φ.

- if $\varphi \equiv q \in Q$, then $\mathit{subf}(\varphi) = \{\varphi\}$
- if $\varphi \equiv \psi \oplus \chi$, then $\mathit{subf}(\varphi) = \{\varphi\} \cup \mathit{subf}(\psi) \cup \mathit{subf}(\chi)$, for an operator $\oplus \in \{\wedge, \vee, \rightarrow, \mathsf{U}\}$
- if $\varphi \equiv \oslash\psi$, then $\mathit{subf}(\varphi) = \{\varphi\} \cup \mathit{subf}(\psi)$, for an operator $\oslash \in \{\neg, \mathsf{F}, \mathsf{G}, \mathsf{X}\}$

The *size* of an LTL formula φ, which we denote by $|\varphi|$, is the size of its set of subformulas, $\mathit{subf}(\varphi)$. In other words, a size of an LTL formula is a number of its unique subformulas. One could put forward a different notion of size, one which counts the total number of subformulas. We believe that considering unique subformulas captures in a better way a connection between the size of a formula and how difficult it would be for humans to interpret it.

Finally, let $U = \{\neg, \mathsf{F}, \mathsf{G}, \mathsf{X}\}$ be a set of all unary operators and $B = \{\wedge, \vee, \rightarrow, \mathsf{U}\}$ a set of all binary operators. Considering atomic propositions as nullary operators, we define the set of all operators as $O = Q \cup U \cup B$.

Our SAT-based learning algorithm relies on a canonical syntactic representation of LTL formulas as directed acyclic graphs (we call them *syntax DAGs* of a formula). A syntax DAG is similar to a syntax tree (i.e., the unique tree labeled with atomic propositions as well as Boolean and temporal operators that is derived from the inductive definition of an LTL formula), the only difference being that same subformulas share a node. The number of nodes in a syntax DAG corresponds to the number of unique subformulas of the represented LTL formula, and thus with the size of the formula.

Definition 3.1. A *syntax DAG* of an LTL formula φ is a graph whose nodes are the elements of a set of subformulas $\mathit{subf}(\varphi)$ and there is an edge from every formula to its child subformulas. Each node is labeled by the main operator of the corresponding subformula.

As an example, Figure 3.1b depicts the (unique) syntax DAG of the formula $(p\,\mathsf{U}\,\mathsf{G}\,q) \vee (\mathsf{F}\,\mathsf{G}\,q)$, in which the subformula $\mathsf{G}\,q$ is shared; the corresponding syntax tree is depicted in Figure 3.1a. Note that syntactically distinct formulas have different syntax DAGs.

Samples and Consistency

Throughout this section, we assume that the data we learn from is given as two finite, disjoint sets $\mathcal{P}, \mathcal{N} \subset (2^{Q})^{\omega}$ of ultimately periodic words. The words in $\mathcal{P}$ are interpreted as *positive examples*, while the words in $\mathcal{N}$ are interpreted as *negative examples*. We call the pair $\mathcal{S} = (\mathcal{P}, \mathcal{N})$ a *sample*. Since we want to work with the ultimately periodic words in a sample algorithmically, we assume that they are stored as pairs (u, v) of finite words $u \in (2^{Q})^*$ and $v \in (2^{Q})^+$, which can be accessed individually. To measure the complexity of a sample, we define its *size* to be $|\mathcal{S}| = \sum_{uv^{\omega} \in \mathcal{P} \cup \mathcal{N}} |u| + |v|$.

Given an LTL formula φ and a sample $\mathcal{S} = (\mathcal{P}, \mathcal{N})$, both over a set Q of atomic propositions, we call φ *consistent* with $\mathcal{S}$ if $uv^{\omega} \models \varphi$ for each $uv^{\omega} \in \mathcal{P}$ (i.e., all positive examples satisfy φ) and $uv^{\omega} \not\models \varphi$ for each $uv^{\omega} \in \mathcal{N}$ (i.e., all negative examples do not satisfy φ); in this case, we also say that φ *separates* $\mathcal{P}$ and $\mathcal{N}$. We will say that formula φ is *minimal consistent with the sample* $\mathcal{S}$ if φ is consistent with $\mathcal{S}$ and no consistent LTL formula of smaller size exists.

3.1.2 Inferring a Minimal formula with a SAT-based Learning Algorithm

The fundamental task we solve in this section is:

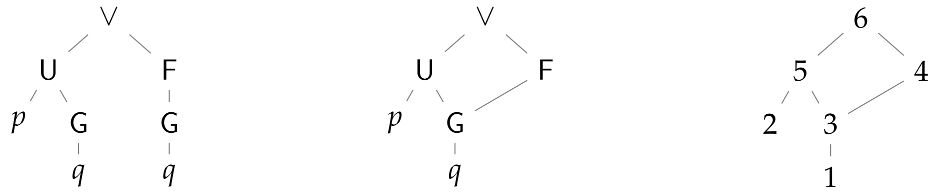
> *"compute a minimal LTL formula consistent with a given sample $\mathcal{S}$".*

We call this task *passive learning of LTL formulas*—as opposed to active learning [10], where the learning algorithm is permitted to actively query for additional data. Note that this problem can have more than one solution as there can be multiple, non-equivalent minimal LTL formulas consistent with a given sample.

Before we explain our learning algorithm in detail, let us comment on the importance of the minimality requirement in the definition above. From one side, we believe that small formulas are easier for humans to comprehend than large ones, which justifies spending effort on learning a smallest formula. However, we do not impose any preference amongst minimal consistent formulas (which is an interesting topic in itself, addressed in Section 3.3). From the other side, we observe that the problem becomes trivial if no restriction on the size is imposed.

Remark 3.1. For $\alpha \in \mathcal{P}$ and $\beta \in \mathcal{N}$, construct a formula $\varphi_{\alpha,\beta}$ with $V(\varphi_{\alpha,\beta}, \alpha) = 1$ and $V(\varphi_{\alpha,\beta}, \beta) = 0$ that describes the first symbol where α and β differ using a sequence of X-operators and an appropriate propositional formula; then, $\bigvee_{\alpha \in \mathcal{P}} \bigwedge_{\beta \in \mathcal{N}} \varphi_{\alpha,\beta}$ is consistent with $\mathcal{S}$ since we assume $\mathcal{P}$ and $\mathcal{N}$ to be disjoint. However, simply characterizing all differences between positive and negative examples is clearly overfitting the sample and, hence, arguably of little help in practice.

Let us now turn to describing our learning algorithm. Its underlying idea is to reduce the construction of a minimal consistent LTL formula to a satisfiability problem in propositional logic and take advantages of existing SAT solvers to find a solution. More precisely, given a sample $\mathcal{S}$ and a natural number $n \in \mathbb{N} \setminus \{0\}$, we construct a propositional formula $\Phi_n^{\mathcal{S}}$ of size polynomial in n and $|\mathcal{S}|$ that has the following two properties:



(a) Syntax tree (b) Syntax DAG (c) Identifiers

FIGURE 3.1: Syntax tree, syntax DAG, and identifiers of the syntax DAG for the LTL formula $(p \,\mathsf{U}\, \mathsf{G}\, q) \vee (\mathsf{F}\, \mathsf{G}\, q)$

1. $\Phi_n^{\mathcal{S}}$ is satisfiable if and only if there exists an LTL formula of size n that is consistent with $\mathcal{S}$; and

2. if v is a model of $\Phi_n^{\mathcal{S}}$, then v contains sufficient information to construct an LTL formula ψ_v of size n that is consistent with $\mathcal{S}$.

By increasing the value of n by one and extracting an LTL formula ψ_v from a model v of $\Phi_n^{\mathcal{S}}$ as soon as it becomes satisfiable (indeed, any model is sufficient), we obtain an effective algorithm that learns an LTL formula of minimal size that is consistent with $\mathcal{S}$. This idea is shown in pseudo code as Algorithm 1. The algorithm will surely terminate because of the existence of a trivial solution from Remark 3.1. The size of this solution provides an upper bound on the value of n.

The key idea of the formula $\Phi_n^{\mathcal{S}}$ is to encode the syntax DAG of an (unknown) LTL formula $\varphi^\star$ with n unique subformulas and then constrain the variables of $\Phi_n^{\mathcal{S}}$ so that $\varphi^\star$ is consistent with the sample $\mathcal{S}$. To simplify our encoding, we assign to each node of this syntax DAG a unique *identifier* $i \in \{1, \ldots, n\}$ so that

a) the identifier of the root is n and

b) if the identifier of an inner node is i, then the identifiers of its children are smaller than i.

Note that such a numbering scheme is not unique for a given syntax DAG, but it entails that the root always has identifier n and the node with identifier 1 is always labeled with an atomic proposition. One assignment of identifiers for the syntax DAG from Figure 3.1b is given in Figure 3.1c.

We encode a syntax DAG using three types of propositional variables:

- $x_{i,\lambda}$ where $i \in \{1, \ldots, n\}$ and $\lambda \in \mathcal{O}$;

Algorithm 1 SAT-based learning algorithm

```
Input: a sample $\mathcal{S}$
$n \leftarrow 0$
repeat
    $n \leftarrow n + 1$
    Construct and solve $\Phi_n^{\mathcal{S}}$
until $\Phi_n^{\mathcal{S}}$ is satisfiable, with a model $v$
Construct and return $\psi_v$
```

TABLE 3.1: Constraints enforcing that the variables $x_{i,\lambda}$ encode a syntax DAG

$$\Big[\bigwedge_{1\leq i\leq n}\bigvee_{\lambda\in O} x_{i,\lambda}\Big]\wedge\Big[\bigwedge_{1\leq i\leq n}\bigwedge_{\lambda\neq\lambda'\in O}\neg x_{i,\lambda}\vee\neg x_{i,\lambda'}\Big] \tag{3.1}$$

$$\Big[\bigwedge_{2\leq i\leq n}\bigvee_{1\leq j<i} l_{i,j}\Big]\wedge\Big[\bigwedge_{2\leq i\leq n}\bigwedge_{1\leq j<j'<i}\neg l_{i,j}\vee\neg l_{i,j'}\Big] \tag{3.2}$$

$$\Big[\bigwedge_{2\leq i\leq n}\bigvee_{1\leq j<i} r_{i,j}\Big]\wedge\Big[\bigwedge_{2\leq i\leq n}\bigwedge_{1\leq j<j'<i}\neg r_{i,j}\vee\neg r_{i,j'}\Big] \tag{3.3}$$

$$\bigvee_{p\in Q} x_{1,p} \tag{3.4}$$

- $l_{i,j}$ where $i \in \{2,\dots,n\}$ and $j \in \{1,\dots,i-1\}$; and
- $r_{i,j}$ where $i \in \{2,\dots,n\}$ and $j \in \{1,\dots,i-1\}$.

Intuitively, the variables $x_{i,\lambda}$ encode labeling of the syntax DAG in the sense that if a variable $x_{i,\lambda}$ is set to *true*, then node i is labeled with λ (recall that each node is labeled with an operator from the set O, which can be either a binary operator, a unary operator or a nullary operator—an atomic proposition). The variables $l_{i,j}$ and $r_{i,j}$, on the other hand, encode the structure of the syntax DAG (i.e., the left and the right child of inner nodes): if variable $l_{i,j}$ ($r_{i,j}$) is set to *true*, then j is the identifier of the left (right) child of node i. By convention, we ignore the variables $r_{i,j}$ if node i of the syntax DAG is labeled with a unary operator; similarly, we ignore both $l_{i,j}$ and $r_{i,j}$ if the node i is labeled with an atomic proposition. Note that in the case of $l_{i,j}$ and $r_{i,j}$, the identifier i ranges from 2 to n because node 1 is always labeled with an atomic proposition and, hence, cannot have children. Moreover, j ranges from 1 to $i-1$ to reflect the fact that identifiers of children have to be smaller than the identifier of the current node.

To enforce that the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$ in fact encode a syntax DAG, we impose the constraints listed in Table 3.1. Formula (3.1) ensures that each node is labeled with exactly one label. Similarly, Formulas (3.2) and (3.3) enforce that each node (except for node 1) has exactly one left and exactly one right child (when deriving the LTL formula from a given model, though, we ignore certain children if the node represents a unary operator or an atomic proposition). Finally, Formula (3.4) makes sure that node 1 is labeled with an atomic proposition.

Let Φ_n^{DAG} now be the conjunction of Formulas (3.1) to (3.4). Then, one can construct a syntax DAG from a model v of Φ_n^{DAG} in a straightforward manner: simply label node i with the unique label λ such that $v(x_{i,\lambda}) = \mathit{true}$, designate node n as the root, and arrange the nodes of the DAG as uniquely described by $v(l_{i,j})$ and $v(r_{i,j})$. Moreover, we can easily derive an LTL formula from this syntax DAG, which we denote by ψ_v. Note, however, that ψ_v describes one LTL formula, but is not yet in any way related to the sample $\mathcal{S}$.

To enforce that ψ_v is indeed consistent with $\mathcal{S}$, we now constrain the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$ further. More precisely, we add for each ultimately periodic word uv^ω in $\mathcal{S}$ a propositional formula $\Phi_n^{u,v}$ that tracks the valuation of the LTL formula encoded by Φ_n^{DAG} (and all its subformulas) on uv^ω. The observation that enables us to do this is the following.

Remark 3.2. Let $uv^\omega \in (2^Q)^\omega$, ψ be an LTL formula over Q, and $k \in \mathbb{N}$. Then, $uv^\omega[|u| + k, \infty) = uv^\omega[|u| + m, \infty)$ with $m \equiv k \mod |v|$. In addition, $V(\varphi, uv^\omega[|u| + k, \infty)) = V(\varphi, uv^\omega[|u| + m, \infty))$ holds for every LTL formula φ.

Intuitively, Remark 3.2 states that the suffixes of a word uv^ω eventually repeat periodically. As a consequence, the valuation of an LTL formula on a word uv^ω can be determined based only on the finite prefix uv (recall that the semantics of temporal operators only depend on the suffixes of a word).

To illustrate this claim, consider the LTL formula $\mathsf{X}\,\varphi$ and assume that we want to determine the valuation $V(\mathsf{X}\,\varphi, uv^\omega[|uv| - 1, \infty))$ (i.e., $\mathsf{X}\,\varphi$ is evaluated at the end of the prefix uv). Then, Remark 3.2 permits us to compute this valuation based on $V(\varphi, uv^\omega[|u|, \infty))$, as opposed to the original semantics of the X-operator, which refers to $V(\varphi, uv^\omega[|uv|, \infty))$ (i.e., the valuation at the next position).

Note that similar ideas can be applied to all other temporal operators as well. Rather than observing only one future timestep, operators U, F and G determine their valuation from the full suffix of a trace. As an example, assume ethat we want to determine the valuation $V(\mathsf{G}\,\varphi, uv^\omega[|u| + k, \infty))$, where $k < |v|$. By Remark 3.2, it is enough to make sure that φ holds for each element of the sequence $uv^\omega[|u| + k, |uv| + k)$.

For $n \in \mathbb{N}$, the formula $\Phi_n^{u,v}$, which requires that our syntax DAG is consistent with the sample, is built over an auxiliary set of propositional variables $y_{i,t}^{u,v}$. Here, $i \in \{1, \ldots, n\}$ is a node in the syntax DAG and $t \in \{0, \ldots, |uv| - 1\}$ is a position in the finite word uv. The meaning of these variables is that the value of $y_{i,t}^{u,v}$ corresponds to the valuation $V(\varphi_i, uv^\omega[t, \infty))$ of the LTL subformula φ_i that is rooted at node i. Note that the set of variables for two distinct words from the sample must be disjoint.

To obtain this desired meaning of the variables $y_{i,t}^{u,v}$, we impose the constraints listed in Table 3.2, which are inspired by bounded model checking [31]. Formula (3.5) implements the LTL semantics of atomic propositions and ensures that if node i is labeled with $q \in Q$, then $y_{i,t}^{u,v}$ is set to *true* if and only if $q \in uv(t)$. Next, Formulas (3.6) and (3.7) implement the semantics of negation and disjunction, respectively: if node i is labeled with $\neg$ and node j is its left child, then $y_{i,t}^{u,v}$ is the negation of $y_{j,t}^{u,v}$; on the other hand, if node i is labeled with $\vee$, node j is its left child, and node j' is its right child, then $y_{i,t}^{u,v}$ is the disjunction of $y_{j,t}^{u,v}$ and $y_{j',t}^{u,v}$. Moreover, Formula (3.8) implements the semantics of the X-operator, following the idea of "returning to the beginning of the periodic part v" as sketched above. Finally, Formula (3.9) implements the semantics of the U-operator. Let us analyze it: the first conjunction in the consequent covers the positions $t \in \{0, \ldots, |u| - 1\}$ in the initial part u, while the second conjunct covers the positions $t \in \{|u|, \ldots, |uv| - 1\}$ in the periodic part v. Thereby, the

TABLE 3.2: Constraints enforcing that the variables $y^{u,v}_{i,t}$ track the valuation of the prospective LTL formula on ultimately periodic words

$$\bigwedge_{1\leq i\leq n}\bigwedge_{q\in Q} x_{i,q} \rightarrow \left[\bigwedge_{0\leq t<|uv|}\begin{cases} y^{u,v}_{i,t} & \text{if } q\in uv(t)\\ \neg y^{u,v}_{i,t} & \text{if } q\notin uv(t)\end{cases}\right] \tag{3.5}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j<i}} (x_{i,\neg}\wedge l_{i,j}) \rightarrow \bigwedge_{0\leq t<|uv|}\left[y^{u,v}_{i,t} \leftrightarrow \neg y^{u,v}_{j,t}\right] \tag{3.6}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,\vee}\wedge l_{i,j}\wedge r_{i,j'}) \rightarrow \bigwedge_{0\leq t<|uv|}\left[y^{u,v}_{i,t} \leftrightarrow (y^{u,v}_{j,t}\vee y^{u,v}_{j',t})\right] \tag{3.7}$$

$$\begin{aligned}\bigwedge_{\substack{1<i\leq n\\ 1\leq j<i}} (x_{i,\mathsf{X}}\wedge l_{i,j}) \rightarrow\\ \left[\left[\bigwedge_{0\leq t<|uv|-1} y^{u,v}_{i,t} \leftrightarrow y^{u,v}_{j,t+1}\right]\wedge\left[y^{u,v}_{i,|uv|-1}\leftrightarrow y^{u,v}_{j,|u|}\right]\right]\end{aligned} \tag{3.8}$$

$$\begin{aligned}\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,\mathsf{U}}\wedge l_{i,j}\wedge r_{i,j'}) \rightarrow\\ \left[\left[\bigwedge_{0\leq t<|u|} y^{u,v}_{i,t} \leftrightarrow \bigvee_{t\leq t'<|uv|}\left[y^{u,v}_{j',t'}\wedge\bigwedge_{t\leq t''<t'} y^{u,v}_{j,t''}\right]\right]\wedge\right.\\ \left.\left[\bigwedge_{|u|\leq t<|uv|} y^{u,v}_{i,t} \leftrightarrow \bigvee_{|u|\leq t'<|uv|}\left[y^{u,v}_{j',t'}\wedge\bigwedge_{t''\in t\looparrowright_{u,v}t'} y^{u,v}_{j,t''}\right]\right]\right]\end{aligned} \tag{3.9}$$

second conjunct relies on an auxiliary set $t \looparrowright_{u,v} t'$ defined by

$$t \looparrowright_{u,v} t' := \begin{cases}\{t,\dots,t'-1\} & \text{if } t<t';\\ \{|u|,\dots,t'-1,t,\dots,|uv|-1\} & \text{if } t\geq t',\end{cases}$$

which contains all positions in v "between t and t'".

Semantics of the derived operators, $\wedge$, $\rightarrow$, F, G can be encoded analogously, as shown in Table 3.3. Moreover, our SAT encoding is extensible, and additional LTL operators such as weak until or weak and strong release can easily be added. To avoid cluttering the presentation, in the rest of the section we will assume only the encoding for the basic operators (Table 3.2).

For each $uv^\omega\in\mathcal{P}\cup\mathcal{N}$, let $\Phi^{u,v}_n$ now be the conjunction of Formulas (3.5) to (3.9). Then, we define

$$\Phi^{\mathcal{S}}_n := \Phi^{DAG}_n \wedge \left[\bigwedge_{uv^\omega\in\mathcal{P}} \Phi^{u,v}_n \wedge y^{u,v}_{n,0}\right]\wedge\left[\bigwedge_{uv^\omega\in\mathcal{N}} \Phi^{u,v}_n \wedge \neg y^{u,v}_{n,0}\right].$$

Note that the subformula $\Phi^{u,v}_n \wedge y^{u,v}_{n,0}$ makes sure that $uv^\omega\in\mathcal{P}$ satisfies the prospective LTL formula (more concretely, uv^ω starting from position 0 satisfies the LTL formula at the root of the syntax DAG), while $\Phi_n^{u,v} \wedge \neg y_{n,0}^{u,v}$ ensures that $uv^\omega \in \mathcal{N}$ does not satisfy it.

To prove the correctness of our learning algorithm, we first establish that the formula $\Phi_n^{\mathcal{S}}$ has in fact the desired properties.

Lemma 3.1. Let $\mathcal{S} = (\mathcal{P}, \mathcal{N})$ be a sample, $n \in \mathbb{N} \setminus \{0\}$, and $\Phi_n^{\mathcal{S}}$ the propositional formula defined above. Then, the following holds:

1. If an LTL formula of size n that is consistent with $\mathcal{S}$ exists, then the propositional formula $\Phi_n^{\mathcal{S}}$ is satisfiable.
2. If $v \models \Phi_n^{\mathcal{S}}$, then ψ_v, the LTL formula derived from the model v, is an LTL formula of size n that is consistent with $\mathcal{S}$.

The proof is a direct consequence of how we defined the propositional formula $\Phi_n^{\mathcal{S}}$. Therefore, it will require a careful application of the definitions from Table 3.2 and Remark 3.2.

Proof. To prove the first statement of Lemma 3.1, assume that φ is an LTL formula of size n that is consistent with $\mathcal{S} = (\mathcal{P}, \mathcal{N})$ and fix identifiers $i \in \{1, \ldots, n\}$ for the nodes of the unique syntax DAG corresponding to φ. Moreover, for each node i, let φ_i be the formula of φ rooted at i.

Based on the LTL formula φ and its subformulas φ_i, we now construct a valuation v for the propositional formula $\Phi_n^{\mathcal{S}}$ as follows:

- we set $v(x_{i,\ell}) = \mathit{true}$ if and only if the node i of the syntax DAG of ψ is labeled with ℓ;
- we set $v(l_{i,j})$ and $v(r_{i,j})$ according to the formula structure and the chosen identifiers of the syntax DAG; and
- for each infinite word $uv^\omega \in \mathcal{P} \cup \mathcal{N}$, for each node i, and each timestep position $t \in \{0, \ldots, |uv| - 1\}$, we set $v(y_{i,t}^{u,v}) = V(\varphi_i, uv^\omega[t, \infty))$.

It is not hard to verify that $v \models \Phi_n^{DAG}$ as the value of the variables $x_{i,\ell}$, $l_{i,j}$, and $r_{i,j}$ have been derived from the syntax DAG of ψ. Moreover, $v \models \Phi_n^{u,v}$ holds for each $uv^\omega \in \mathcal{P} \cup \mathcal{N}$ because the values of the variables $y_{i,t}^{u,v}$ correspond exactly to the valuation of each formula ψ_i on the finite prefix uv. Finally, the fact that φ is consistent with $\mathcal{S}$ implies $v \models y_{n,0}^{u,v}$ for each $uv^\omega \in \mathcal{P}$ (since $V(\varphi, uv^\omega) = 1$ if $uv^\omega \in \mathcal{P}$) and $v \models \neg y_{n,0}^{u,v}$ for each $uv^\omega \in \mathcal{N}$ (since $V(\varphi, uv^\omega) = 0$ if $uv^\omega \in \mathcal{N}$). Thus, $v \models \Phi_n^{\mathcal{S}}$, which proves that $\Phi_n^{\mathcal{S}}$ is satisfiable.

To prove the second statement, assume that $v \models \Phi_n^{\mathcal{S}}$ and let φ_v be the LTL formula obtained from v. In particular, $v \models \Phi_n^{DAG}$, and the variables $x_{i,\ell}$, $l_{i,j}$, and $r_{i,j}$ encode a syntax DAG. For each node i, let now φ_i be the subformula of φ_v rooted at i.

We show by induction over the structure of φ_v that $v(y_{i,t}^{u,v}) = V(\varphi_i, uv^\omega[t, \infty))$ holds for each subformula φ_i of φ_v, each ultimately periodic word $uv^\omega \in \mathcal{P} \cup \mathcal{N}$, and each timestep position $t \in \{0, \ldots, |uv| - 1\}$:

- Let $\varphi_i = q$ for some atomic proposition $q \in Q$. Then, $v(x_{i,q}) = \mathit{true}$, and Formula (3.5) immediately ensures that $v(y_{i,t}^{u,v}) = V(q, uv^\omega[t, \infty))$ holds for each $t \in \{0, \ldots, |uv| - 1\}$.

- Let $\varphi_i = \neg \varphi_j$. Then, $v(x_{i,\neg}) = \mathit{true}$, and applying the induction hypothesis for φ_j yields $v(y^{u,v}_{j,t}) = V(\varphi_j, uv^\omega[t,\infty))$ for each $t \in \{0,\dots,|uv|-1\}$. In this situation, Formula (3.6) enforces that $v(y^{u,v}_{i,t}) = V(\neg\varphi_j, uv^\omega[t,\infty))$ holds for each $t \in \{0,\dots,|uv|-1\}$.

- Let $\varphi_i = \varphi_j \vee \varphi_{j'}$. Then, as in the case of negation, the induction hypotheses together with Formula (3.7) imply that $v(y^{u,v}_{i,t}) = V(\varphi_j \vee \varphi_{j'}, uv^\omega[t,\infty))$ for each $t \in \{0,\dots,|uv|-1\}$.

- Let $\varphi_i = \mathsf{X}\,\varphi_j$. Then, $v(x_{i,\mathsf{X}}) = \mathit{true}$, and applying the induction hypothesis for φ_j yields $v(y^{u,v}_{j,t}) = V(\varphi_j, uv^\omega[t,\infty))$ for each $t \in \{0,\dots,|uv|-1\}$. In this situation, the left conjunct in the consequent of Formula (3.8) enforces that $v(y^{u,v}_{i,t}) = v(y^{u,v}_{j,t+1}) = V(\mathsf{X}\,\varphi_j, uv^\omega[t,\infty))$ holds for each $t \in \{0,\dots,|uv|-2\}$. Moreover, by choosing $k = |v|$ in Remark 3.2, we obtain $V(\varphi_j, uv^\omega[|uv|,\infty)) = V(\varphi_j, uv^\omega[|u|,\infty))$. Thus, the right conjunct in the consequent of Formula (3.8) ensures that $v(y^{u,v}_{i,|uv|-1}) = v(y^{u,v}_{j,|u|}) = V(\mathsf{X}\,\varphi_j, uv^\omega[|uv|-1,\infty))$ holds.

- Let $\varphi_i = \varphi_j \,\mathsf{U}\, \varphi_{j'}$ and, thus, $v(x_{i,\mathsf{U}}) = \mathit{true}$. We split the proof for this operator into four distinct cases.

  1. Let $t \in \{0,\dots,|uv|-1\}$, and assume $V(\varphi_i, uv^\omega[t,\infty)) = 0$, with the reason for evaluating to zero being that $V(\varphi_{j'}, uv^\omega[t',\infty)) = 0$ for all $t' \geq t$ (i.e., $\varphi_{j'}$ never holds after position t). By induction hypothesis, this means that $v(y^{u,v}_{j',t'}) = \mathit{false}$ for $t \leq t' \leq |uv|-1$. Hence, Formula (3.9) enforces $v(y^{u,v}_{i,t}) = 0$.
  2. Let $t \in \{0,\dots,|uv|-1\}$, and assume $V(\varphi_i, uv^\omega[t,\infty)) = 0$ because for all $t' \geq t$ with $V(\varphi_{j'}, uv^\omega[t',\infty)) = 1$ there exists a $t'' \in \{t,\dots,t'-1\}$ such that $V(\varphi_j, uv^\omega[t'',\infty)) = 0$. By induction hypothesis, we obtain that for all $t' \in \{t,\dots,|uv|-1\}$ with $v(y_{j',t'}) = \mathit{true}$ there exists a $t'' \in \{t,\dots,t'-1\}$ with $v(y^{u,v}_{j,t''}) = \mathit{false}$. In this situation, Formula (3.9) enforces $v(y^{u,v}_{i,t}) = \mathit{false}$.
  3. Let $t \in \{0,\dots,|u|-1\}$, and assume $V(\varphi_i, uv^\omega[t,\infty)) = 1$. Then, there exists a $t' \in \{t,\dots,|uv|-1\}$ such that $V(\varphi_{j'}, uv^\omega[t',\infty)) = 1$ and $V(\varphi_j, uv^\omega[t'',\infty)) = 1$ for $t \leq t'' < t'$ (if the smallest position t^* with $V(\varphi_{j'}, uv^\omega[t^*,\infty)) = 1$ is greater than $|uv|-1$, then Remark 3.2 guarantees that there exists an earlier position $t' \in \{t,\dots,|uv|-1\}$ with $V(\varphi_{j'}, uv^\omega[t',\infty)) = 1$ and $V(\varphi_j, uv^\omega[t'',\infty)) = 1$ for $t \leq t'' < t'$). By applying the induction hypothesis, we then obtain $v(y_{j',t'}) = \mathit{true}$ and $v(y_{j,t''}) = \mathit{true}$ for all $t \leq t'' < t'$. In this case, the first conjunct of the consequent of Formula (3.9) enforces $v(y^{u,v}_{i,t}) = 1$.
  4. Let $t \in \{|u|,\dots,|uv|-1\}$, and assume $V(\varphi_i, uv^\omega[t,\infty)) = 1$. Then, there exists a position $t^* \geq t$ such that $V(\varphi_{j'}, uv^\omega[t^*,\infty)) = 1$ and $V(\varphi_j, uv^\omega[t^{**},\infty)) = 1$ for all $t \leq t^{**} < t^*$. By Remark 3.2, this means that there exists a position $t' \in \{|u|,\dots,|uv|-1\}$ such that $V(\varphi_{j'}, uv^\omega[t',\infty)) = 1$ and $V(\varphi_j, uv^\omega[t'',\infty)) = 1$ for all $t'' \in T = \{|u| + m \mid m \equiv (t^{**} \bmod |v|), t^{**} \in \{t,\dots,t^*-1\}\}$. Applying the induction hypothesis now yields

TABLE 3.3: Constraints enforcing that the variables $y^{u,v}_{i,t}$ track the valuation of the prospective LTL formula on ultimately periodic words: derived operators

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,\wedge} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \bigwedge_{0\leq t<|uv|} \Big[y^{u,v}_{i,t} \leftrightarrow (y^{u,v}_{j,t} \wedge y^{u,v}_{j',t}) \Big]$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,\rightarrow} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \bigwedge_{0\leq t<|uv|} \Big[y^{u,v}_{i,t} \leftrightarrow (y^{u,v}_{j,t} \rightarrow y^{u,v}_{j',t}) \Big]$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j<i}} (x_{i,\mathsf{F}} \wedge l_{i,j}) \rightarrow$$
$$\Bigg[\Big[\bigwedge_{0\leq t<|u|} y^{u,v}_{i,t} \leftrightarrow \bigvee_{t\leq t'<|uv|} y^{u,v}_{j,t'} \Big] \wedge \Big[\bigwedge_{|u|\leq t<|uv|} y^{u,v}_{i,t} \leftrightarrow \bigvee_{|u|\leq t'<|uv|} y^{u,v}_{j,t'} \Big] \Bigg]$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j<i}} (x_{i,\mathsf{G}} \wedge l_{i,j}) \rightarrow$$
$$\Bigg[\Big[\bigwedge_{0\leq t<|u|} y^{u,v}_{i,t} \leftrightarrow \bigwedge_{t\leq t'<|uv|} y^{u,v}_{j,t'} \Big] \wedge \Big[\bigwedge_{|u|\leq t<|uv|} y^{u,v}_{i,t} \leftrightarrow \bigwedge_{|u|\leq t'<|uv|} y^{u,v}_{j,t'} \Big] \Bigg]$$

$v(y^{u,v}_{j',t'}) = \mathit{true}$ and $v(y^{u,v}_{j,t''}) = \mathit{true}$ for all $t'' \in T$. Since $t \looparrowright_{u,v} t' \subseteq T$, the second conjunct of the consequent of Formula (3.9) enforces $v(y^{u,v}_{i,t}) = \mathit{true}$.

We have shown that $v(y^{u,v}_{i,t}) = V(\varphi_i, uv^\omega[t,\infty))$ holds for timestep positions $t \in \{0, \ldots, |uv| - 1\}$. But valuations on those timestep positions determine valuations for all $t \in \mathbb{N}$, as discussed in Remark 3.2.

Having established that the variables correctly track the valuation of the formula φ_v on the sample, it is now straightforward to see that φ_v is in fact consistent with $\mathcal{S}$. First, we observe that $\varphi_v = \varphi_n$ because node n is the root of the syntax DAG. Second, since $\Phi^{\mathcal{S}}_n$ enforces $v(y^{u,v}_{n,0}) = \mathit{true}$ for each $uv^\omega \in \mathcal{P}$ and $v(y_{n,0}) = V(\varphi_v)$ holds by the induction above, we obtain $V(\varphi_v, uv^\omega) = 1$. Similarly, we obtain $V(\varphi_v, uv^\omega) = 0$ for each $uv^\omega \in \mathcal{N}$ because $\Phi^{\mathcal{S}}_n$ enforces $v(y^{u,v}_{n,0}) = 0$ if $uv^\omega \in \mathcal{N}$. Thus, φ_v is consistent with $\mathcal{S}$, which proves the second statement. □

A direct consequence of Lemma 3.1 are termination and correctness of Algorithm 1.

Theorem 3.1. Given a sample $\mathcal{S}$, Algorithm 1 terminates eventually and outputs an LTL formula of minimal size that is consistent with $\mathcal{S}$.

Proof. Since there exists a consistent LTL formula for every non-contradictory sample, Part 1 of Lemma 3.1 guarantees that Algorithm 1 terminates. Moreover, Part 2 ensures that the output is indeed an LTL formula that is consistent with $\mathcal{S}$. Since n is increased by one in every iteration of the loop until $\Phi^{\mathcal{S}}_n$ becomes satisfiable, the output of Algorithm 1 is a consistent LTL formula of minimal size. □

Algorithm 2 Learning algorithm based on decision trees

```
Input: a sample $\mathcal{S}$
1: Run Algorithm 1 on small subsets of $\mathcal{P}$ and $\mathcal{N}$ to construct a set $\Pi = \{\varphi_1, \ldots, \varphi_n\}$ of
   LTL formulas such that for each pair $u_1v_1^\omega \in \mathcal{P}$ and $u_2v_2^\omega \in \mathcal{N}$ there exists a $\varphi_i \in \Pi$
   with $V(\varphi_i, u_1v_1^\omega) = 1$ and $V(\varphi_i, u_2v_2^\omega) = 0$
2: Learn a decision tree $T$ with LTL primitives from $\Pi$ as features
3: return $\psi_T$, the resulting Boolean combination of LTL primitives (which is consistent with
   $\mathcal{S}$)
```

It is important to emphasize that the size of $\Phi_n^{\mathcal{S}}$ and, hence, the performance of Algorithm 1 depends on the size of a sample $\mathcal{S} = (\mathcal{P}, \mathcal{N})$, as summarized next.

Remark 3.3. The formula $\Phi_n^{\mathcal{S}}$ ranges over $\mathcal{O}(n^2 + n|\mathcal{S}|)$ variables and is of size $\mathcal{O}(n^2 + n^3 \sum_{uv^\omega \in \mathcal{P} \cup \mathcal{N}} |uv|^3)$.

Remark 3.4. The hardness question for the problem of learning a minimal LTL formula consistent with the sample remains open. For a similar problem, learning a minimal automaton consistent with the sample, Angluin [12] has shown NP-hardness. I believe that learning LTL formulas is NP-hard as well, but was not able to prove so. There are results showing that some fragments of LTL are NP-hard [83], in particular $LTL(\mathsf{X}, \wedge)$, $LTL(\mathsf{F}, \wedge)$, and $LTL(\mathsf{F}, \mathsf{X}, \wedge, \vee)$.

We have described an algorithm that infers a minimal LTL formula from a non-contradicting sample of example traces. It is doing so by iterative SAT solving. The main difference to existing work on the same topic is the fact that our algorithm does not depend on any fixed syntactic templates for the formula. Nonetheless, if the expert knowledge is available, it can be incorporated into our algorithm by constraining the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$.

3.1.3 A Decision Tree Based Learning Algorithm

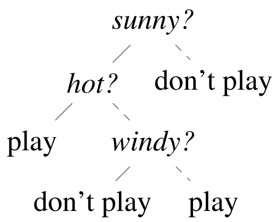
The SAT-based algorithm described in Section 3.1.2 is an elegant, out-of-the-box way to discover minimal LTL formulas describing a sample. Even though it scales well beyond toy examples, its running time seems too prohibitive for real-world examples (as discussed in Section 3.1.4). That is why we now present a second learning algorithm, based on a combination of SAT solving and decision tree learning.

Our second algorithm proceeds in two phases, as outlined in Algorithm 2. In the first phase, we run Algorithm 1 on small subsets of $\mathcal{P}$ and $\mathcal{N}$. This is repeated until we obtain a set Π of LTL formulas (we call them *LTL primitives*) that separate all pairs of words from $\mathcal{P}$ and $\mathcal{N}$. In the second phase, formulas from Π are used as features for a standard decision tree learning algorithm [194]. The resulting decision tree is a Boolean combination of LTL formulas $\varphi_i \in \Pi$ that is consistent with the sample.

Note that this relaxes the problem addressed in Section 3.1.2: we can no longer guarantee to find a formula of minimal size. However, decision trees are among the structures that are the easiest to interpret by end-users. That makes them suitable for our use-case, and the minimality of formulas is replaced by the structural simplicity of decision trees.

Sunny?	Hot?	Windy?	**Play golf or not**
false	true	true	don’t play
true	false	true	don’t play
false	false	false	don’t play
true	false	false	play
true	true	true	play

(a) Training data



(b) learned decision tree

FIGURE 3.2: An example of the features of training data and the learned decision tree

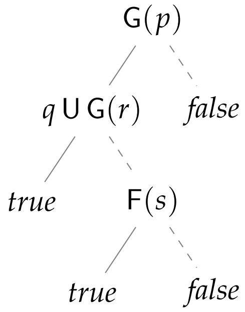


FIGURE 3.3: An example LTL decision tree

Learning Decision Trees

Consider a problem of classifying examples characterized by a number of *features* into a pre-defined set of *classes*. A (classification) decision tree is a tree-shaped structure whose leaf nodes represent different classes, and internal nodes represent tests on the features. It is easy then to classify any given example by following a path dictated by the feature tests at internal nodes and landing in one of the leaf-node classes.

Decision trees can be learned from data. Iterating over internal nodes in a top-down fashion, a learning algorithm picks a decision feature for the node based on a measure of the purity of the split it introduces. (We prefer the feature which separates different classes from the training example, over the one in which classes are kept together.) Different learning algorithms differ by the purity measure they use. The learning process will not attempt to separate the examples from the training set perfectly, as insisting on it might lead to overfitting to the training set.

A simplified, commonly used illustrative example is given in Figure 3.2 . In that example, the data consists of the daily weather information, and the classification is whether people played golf on that day or not (Figure 3.2a). The learned tree (Figure 3.2b) contains feature tests in its internal nodes. Full edges represent that the test is satisfied and dashed edges that it is not. In our example, on a hot and sunny day, the decision tree predicts that people will play golf (left-most branch), whereas on a day that is not sunny, it predicts they won’t play golf, regardless of how windy or hot it is. Further details about decision trees and learning algorithms can be found in standard textbooks, such as [168].

The decision trees we seek to learn have their inner nodes labeled with LTL formulas

Algorithm 3 Obtaining predictor functions under strategy α

```
Input: sample $\mathcal{S} = (\mathcal{P}, \mathcal{N})$, parameters $k$, probDecreaseRate, numRepetitions
$\Pi = \{\}$
while not all pairs separated do
    $\text{prob}_{\mathcal{P}}[t] \leftarrow \frac{1}{|\mathcal{P}|}, \forall t \in \mathcal{P}$; $\text{prob}_{\mathcal{N}}[t] \leftarrow \frac{1}{|\mathcal{N}|}, \forall t \in \mathcal{N}$
    for numRepetitions do
        $P' \leftarrow \text{choose}(\mathcal{P}, \text{prob}_{\mathcal{P}}, k)$
        $N' \leftarrow \text{choose}(\mathcal{N}, \text{prob}_{\mathcal{N}}, k)$
        $\phi = \text{SAT\_separate}(P', N')$
        $\Pi.\text{add}(\phi)$
        $M \leftarrow \{\tau \in \mathcal{P} : \tau \not\models \phi\} \cup \{\tau \in \mathcal{N} : \tau \models \phi\}$
        $\text{prob}_{\mathcal{P}}[\tau] \leftarrow \text{prob}_{\mathcal{P}}[\tau] \cdot \text{probDecreaseRate}, \forall \tau \in \mathcal{P} \cap M$
        $\text{prob}_{\mathcal{N}}[\tau] \leftarrow \text{prob}_{\mathcal{N}}[\tau] \cdot \text{probDecreaseRate}, \forall \tau \in \mathcal{N} \cap M$
        normalize $\text{prob}_{\mathcal{P}}$ and $\text{prob}_{\mathcal{N}}$
        if formulas in $\Pi$ separate all pairs from $\mathcal{P} \times \mathcal{N}$ then
            break
return $\Pi$
```

from Π and their leaves are labeled with either *true* or *false*, as illustrated in Figure 3.3. The LTL formula represented by such a tree T is given by $\psi_T := \bigvee_{\rho \in \mathfrak{P}} \bigwedge_{\varphi \in \rho} \varphi$ where $\mathfrak{P}$ is the set of all paths from the root to a leaf labeled with *true* and $\varphi \in \rho$ denotes that φ occurs on ρ (negated if the path follows a dashed edge). In the example from Figure 3.3, the tree is equivalent to the LTL formula $[\mathsf{G}\, p \wedge (q\, \mathsf{U}\, \mathsf{G}\, r)] \vee [\mathsf{G}\, p \wedge \neg(q\, \mathsf{U}\, \mathsf{G}\, r) \wedge \mathsf{F}\, s]$.

To learn a decision tree over LTL primitives, we perform a preprocessing step and modify the sample as follows. For each word $uv^\omega \in \mathcal{P} \cup \mathcal{N}$, we use the LTL primitives as features and create a Boolean vector of size $|\Pi|$ with the i-th entry set to $V(\varphi_i, uv^\omega)$; this vector is then labeled with *true* if $uv^\omega \in \mathcal{P}$ or with *false* if $uv^\omega \in \mathcal{N}$. In the second step, we apply a standard learning algorithm for decision trees to this modified sample (we used Gini impurity [41] as split heuristic in our experiments). Unlike in the common scenario for using decision tree learning, we are interested in a tree that classifies our sample perfectly. (We let algorithm run until the perfect classification is achieved.)

Obtaining LTL Primitives

Meaningful features are essential for a successful classification using decision trees. In our algorithm, features are generated from the set of LTL primitives Π. We used two different strategies, called Strategy α and Strategy β, for obtaining Π.

Strategy α (Algorithm 3) iteratively chooses subsets $P' \subset \mathcal{P}$ and $N' \subset \mathcal{N}$ of size k according to probability distributions $\text{prob}_{\mathcal{P}}$ and $\text{prob}_{\mathcal{N}}$ on $\mathcal{P}$ and $\mathcal{N}$, respectively. After a formula φ separating P' and N' is found using Algorithm 1 and added to Π, $\text{prob}_{\mathcal{P}}$ and $\text{prob}_{\mathcal{N}}$ are updated to increase the likelihood of any word that is not yet classified correctly by any of the $\varphi \in \Pi$ to be selected. This process is repeated until all pairs of positive and negative examples are separated by some LTL primitive or restarted after a user-given number of iterations (numRepetitions). Although this strategy is, in general, not guaranteed to terminate due to its probabilistic nature, it always did terminate in our experiments.

Algorithm 4 Obtaining predictor functions under strategy β

```
Input: sample $(\mathcal{P}, \mathcal{N})$, parameter $k$
  $\Pi = \{\}$
  $S^* \leftarrow \mathcal{P} \times \mathcal{N}$
  while $S^* \neq \emptyset$ do
      $(P', N') \leftarrow \text{choose}(S^*, \text{uniform}, k)$
      $\phi = \text{SAT\_separate}(P', N')$
      $\Pi.\text{add}(\phi)$
      $S^* \leftarrow S^* \setminus \{(p, n) \in S^* : p \models \phi \wedge n \not\models \phi\}$
  return $\Pi$
```

Strategy β (Algorithm 4) computes LTL primitives in a more aggressive way. Starting with the set $\mathcal{S} = \mathcal{P} \times \mathcal{N}$, it uniformly at random selects k pairs from $\mathcal{S}$ and uses Algorithm 1 to compute an LTL primitive φ that separates those pairs. Then, it removes all pairs separated by φ from $\mathcal{S}$ and repeats the process until $\mathcal{S}$ becomes empty (i.e., all pairs of examples are separated).

3.1.4 Evaluation

In this section, we answer questions that arise naturally: how performant is the SAT learner (Algorithm 1), and what is the performance gain of using decision tree learning (Algorithm 2). Furthermore, what is the complexity of the learned decision trees in terms of the number of decision nodes, and, finally, how do different parameters influence the performance of Algorithm 2. After answering these questions with experiments performed on synthetic data, we will see the usefulness of our algorithms for understanding executions on a use-case of a leader-election algorithm.

We implemented both learning algorithms in a Python tool[2] using Microsoft Z3 [63] as a SAT solver. All experiments were conducted on Debian machines with Intel Xeon E7-8857 CPUs at 3 GHz, using up to 5 GB of RAM.

Performance on Synthetic Data

To simulate real-world use-cases, we generated samples based on common LTL patterns [73], which are shown in Table 3.4. Starting from a pattern formula ψ, we generated sets of random words and separated them into $\mathcal{P}$ and $\mathcal{N}$ depending on whether they are a model of ψ or not.

[2] The source code is publicly available at https://github.com/gergia/samples2LTL and the public interface for the tools is at https://flie.mpi-sws.org/. Motivated by our work, a C++ implementation of the same encoding is available at https://github.com/blickens/flie

TABLE 3.4: Common LTL patterns used in practice [73]

Absence	Existence	Universality
$\mathsf{G}(\neg p_0)$	$\mathsf{F}(p_0)$	$\mathsf{G}(p_0)$
$\mathsf{F}(p_1) \rightarrow (\neg p_0 \,\mathsf{U}\, p_1)$	$\mathsf{G}(\neg p_0 \vee \mathsf{F}(p_0 \wedge \mathsf{F}(p_1)))$	$\mathsf{F}(p_1) \rightarrow (p_0 \,\mathsf{U}\, p_1)$
$\mathsf{G}(p_1 \rightarrow \mathsf{G}(\neg p_0))$	$\mathsf{G}(p_0 \wedge (\neg p_1 \rightarrow (\neg p_1 \,\mathsf{U}(p_2 \wedge \neg p_1))))$	$\mathsf{G}(p_1 \rightarrow \mathsf{G}(p_0))$

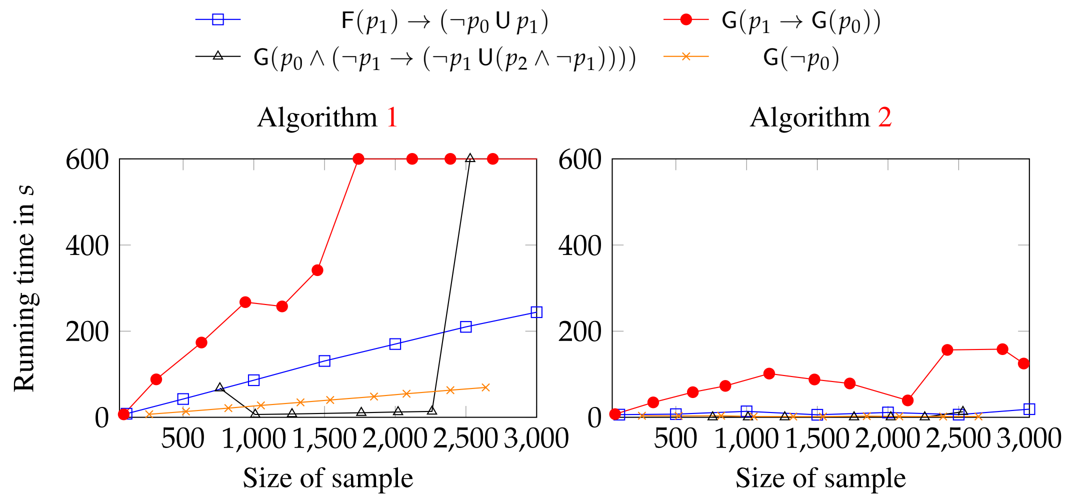


FIGURE 3.4: Comparison of Algorithm 1 and Algorithm 2

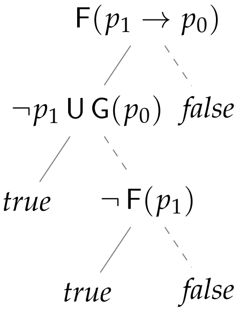

$\mathsf{F}(p_1 \to p_0)$
$\neg p_1 \,\mathsf{U}\, \mathsf{G}(p_0)$
false
true
$\neg\,\mathsf{F}(p_1)$
true
false

FIGURE 3.5: A decision tree obtained from a sample generated from the LTL pattern $\mathsf{G}(p_1 \to \mathsf{G}(p_0))$

Thereby, we fixed $|u| + |v| = 10$ for all words in the sample and added noise in the form of one additional atomic proposition that is not constrained by the pattern formula. The size of the generated samples ranges between 50 and 5000. In total, we generated 192 samples.

Figure 3.4 compares the running times of Algorithm 1 and Algorithm 2 (using Strategy α and $k = 3$) on samples of varying sizes. Overall, Algorithm 1 produces minimal formulas consistent with the sample. It does so even for samples of considerable size, but if the sample size grows beyond 2000 (varies over samples), the SAT-based learner (Algorithm 1) frequently times out (at 600 seconds). When Algorithm 2 (using decision tree learning) is applied to these samples—as shown on the right-hand side of Figure 3.4—none of the computations timed out and the running times significantly improved.

What kind of trees does Algorithm 2 produce? An example output of the algorithm is shown in Figure 3.5. On the figure, if a node evaluates to true, the outgoing full edge should be followed (and the dashed edge otherwise). This example is produced by Algorithm 2 from samples generated by the formula $\mathsf{G}(p_1 \to \mathsf{G}(p_0))$. The obtained tree represents the formula $\psi_t = [\mathsf{G}(p_1 \to p_0) \wedge (\neg p_1 \,\mathsf{U}\, \mathsf{G}(p_0)] \vee [\mathsf{G}(p_1 \to p_0) \wedge \neg(\neg p_1 \,\mathsf{U}\, \mathsf{G}(p_0)) \wedge \neg\,\mathsf{F}(p_1)]$. This is, indeed, one of the largest formulas produced. As Table 3.5 illustrates, Algorithm 2 learns small trees, often with less than five inner nodes. Upon closer inspection, we noticed that it often happens that one of the LTL primitives was the specified formula itself. This suggests

TABLE 3.5: Different parameters used for Algorithm 2

Sampling strategy	Subset size k	Number of timeouts	Avg. running time in s	Avg. number of nodes in a tree
α	3	0 / 192	21.00	3.05
α	6	4 / 192	35.28	1.47
α	10	8 / 192	42.72	1.2
β	3	4 / 192	30.92	1.37
β	6	12 / 192	48.46	1.19
β	10	21 / 192	48.11	1.06

that small subsets already characterize our samples completely.

To be able to compare decision trees to the formulas learned by Algorithm 2, we measure the *size* of a tree T in terms of the size of the formula ψ_T this tree encodes. In our experiments, the formulas learned by Algorithm 2 were on average 1.41 times larger than those learned by Algorithm 1. However, there are outlier trees that are four times bigger than the one learned by Algorithm 1. Nonetheless, about 70 % are of the same size. Even for the outliers, as emphasized previously, readability does not degrade completely because the rule-based structure of decision trees is known to be easily understandable by humans. Note that the runtime and size of decision trees depend on the parameters of Algorithm 2, which we discuss next.

Tuning the Decision Tree-Based Algorithm

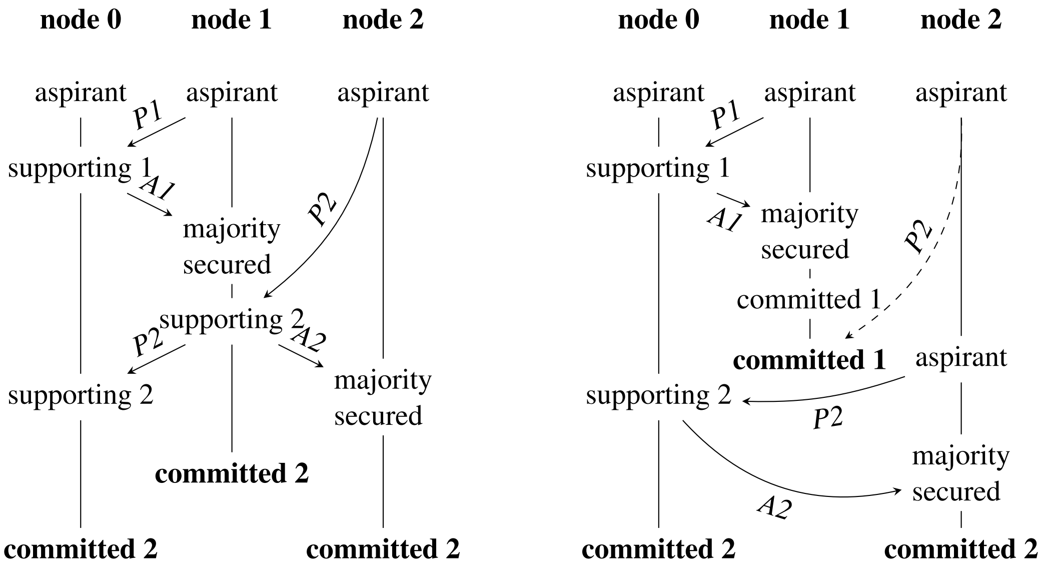
As described in Section 3.1.3, Algorithm 2 can be tuned by various parameters (sampling strategy for obtaining LTL primitives, size of sample subsets, probability increase rate, and the number of repetitions inside a single sampling). In this subsection, we explore how those parameters affect the performance of the algorithm.

Table 3.5 shows the performance of Algorithm 2 for different parameters, averaged over all 192 benchmarks. As the table indicates, the less aggressive method of separating sets, Strategy α, performs better. It seems that if the subset sizes are increased, or Strategy β is used, the sampled subsets already describe the specified formula completely. Finally, we chose Strategy α and $k = 3$ to be our default parameters. Varying the probability decrease rate and the number of repetitions inside a single sampling did not influence the performance much.

Explaining Executions of a Leader Election Protocol

Several methods exist for finding errors or reproducing certain behavior in distributed systems through systematic testing [42, 159]. However, finding an execution and a corresponding schedule is only a first step towards understanding the issue at hand. In the following, we demonstrate how to apply our technique for obtaining a minimal LTL description of a specific inconsistency in a leader election protocol.

The leader election protocol we consider is the *Fast Leader Election* algorithm [128, 165] used by Apache Zookeeper. In this protocol, every node has a unique ID and initially tries to become the leader. To this end, every node sends messages to all other nodes proclaiming its leadership. Upon receiving a message by an aspirant leader with a higher ID, a node gives up its claim and acknowledges its support for the aspirant. If a node learns that an aspirant node has a support of a majority of all nodes, it commits (after waiting for a constant time for new messages) to the aspirant as the leader. Once committed, the node never again changes its decision and informs any other node of its commitment (one example is the message depicted by the dotted arrow in Figure 3.6b). If a node has not committed and learns about another node that has committed, it commits to the same leader.



(a) Consistent schedule for an execution of the leader election protocol

(b) Inconsistent schedule for an execution of the leader election protocol

FIGURE 3.6: Two example executions of the leader election protocol

Figure 3.6a shows an example of a successful leader election with three nodes in an UML-style message sequence chart. The messages exchanged between nodes are proposing the leader i (P_i) and node j acknowledging the claim of a leader (A_j). The arrows indicate exchanged messages and imply precedence of events. Note that not all messages are shown in the figures, but only the ones important for understanding the protocol.

In Figure 3.6a all the nodes have committed to the same leader. On the other hand, Figure 3.6b shows a schedule that ends up in an inconsistent state where nodes are committed to different leaders. This schedule was discovered by the PCTCP algorithm [183], which systematically explores the space of possible executions of distributed algorithms. The situation in Figure 3.6b is caused by asynchronous communication: for performance reasons, nodes commit as quickly as possible and then discard any messages, which otherwise would have changed their commitment (indicated as a dashed line in Figure 3.6b). Note, however, that this is not a bug in Zookeeper's broadcast algorithm, as a leader without a quorum will not be allowed to perform any action in a later phase of the protocol.

To better understand how this inconsistent state arises, our goal is to generate an LTL formula that describes the difference between the schedules in figures 3.6a and 3.6b. To this end, we constructed a sample by generating 20 linearizations of the schedule from Figure 3.6a and 20 linearizations of the schedule from Figure 3.6b. Since we seek an explanation for the inconsistent behavior, the former (with consistent outcomes) correspond to negative examples (set $\mathcal{N}$), and the latter (with inconsistent outcomes) correspond to positive examples (set $\mathcal{P}$). The set of atomic propositions used to construct the examples contains twelve elements: $recv(i, j)$ for $i, j \in \{1,2,3\}$ (meaning that node j received a message from node i) and $comm(i)$ for $i \in \{1,2,3\}$ (meaning that node i committed to a leader).[3]

Finally, we ran Algorithm 1 on this sample. The resulting formula is $\neg recv(2,1) \, \mathsf{U} \, comm(1)$. Intuitively, node 1 did not receive a message from node 2 before it committed to a leader. That is exactly the difference between the schedules in figures 3.6a and 3.6b. Also, it hints at a specific reason for the inconsistency in Figure 3.6b, thus potentially helping the engineers improve the system.

This example illustrates potential uses for the developed learning technique. Note, however, that this experiment still required a significant amount of manual effort, in particular when choosing atomic propositions. In order to apply the technique in practice, more work is needed to automate the process.

3.1.5 Related Work

Learning temporal properties from examples has recently attracted increasing interest, especially in the area of *Signal Temporal Logic (STL)* [160] and *parametric STL* [17]. Examples include the work by Asarin et al. [17], Kong et al. [134, 135], Vaidyanathan et al. [216], and Bartocci, Bortolussi, and Sanguinetti [26]. In contrast to our SAT-based learning algorithm, however, all of these techniques either rely on user-given templates or can only learn formulas from very restricted syntactic fragments. Various techniques for mining LTL specifications [146, 141] and CTL specifications [225] exist, but these also rely on templates or severely restrict the class of formulas. To the best of our knowledge, our SAT-based algorithm was the first that was capable of learning unrestricted LTL formulas without relying on user-given templates. Nonetheless, expert knowledge in the form of constraints on the syntax can easily be encoded if desired.

Camacho et al. [47] use a similar encoding to SAT as the one presented in Section 3.1.2. While they focus on finite traces, the main techniques remain the same. If a perfect separation requirement is relaxed, a Bayesian approach yields significant performance gains [131]. Improving performance is also possible by splitting the learning into two phases: first determining the formula structure and then filling it in to fit the sample [198]. The SYSLITE system [14] improves the scalability of the work presented in this section by encoding the learning of past LTL using bitvectors inside the CVC solver [25].

Our SAT-based learning algorithm is inspired by bounded model checking [31] and the work on learning (minimal) automata over finite words [176, 179]. However, since regular languages are strictly more expressive than LTL (the former being equivalent to monadic second-order logic [45], while the latter being equivalent to first-order logic [129]), automata learning techniques—including active learning algorithms [81, 13] that operate in Angluin's active learning framework [10]—are not immediately applicable.

[3] While we could have included more information into propositions, we had to obscure some in order to avoid solutions which state the obvious, of the form "node 1 committed to node 1 as a leader, while node 2 committed to node 2".

Using decision trees to learn Signal Temporal Logic (STL) formulas has been explored by Bombara et al. [39], whose main contribution is an adaptation of the classical impurity measure to account for STL formulas. However, this work still requires user-defined STL primitives to be provided, which serve as the features for the decision tree learning algorithm. By contrast, our technique uses the SAT-based learning algorithm to infer LTL primitives fully automatically.

Learning of logical formulas has also been studied in the context of *probably approximately correct learning (PAC)* [217]. Grohe et al. [99], for instance, considered learning of first-order definable concepts over structures of small degree. Subsequently, Grohe et al. [98] studied the learning of hypotheses definable using monadic second-order logic on strings. Due to the fundamental differences between PAC learning and the learning model considered here (one being approximate and the other being exact), their techniques cannot easily be applied to our setting.

3.2 Inferring Specifications from Positive Examples Only

In the previous section, we were solving the problem of inferring specifications from a sample consisting of both positive and negative example traces. Unfortunately, both negative *and* positive examples are not always available. For instance, when inferring the specification of a system that is too big to be fully analyzed but whose implementation is given, we can extract traces representing possible executions of the system. Proving that a certain trace is not a possible execution of the system is, however, a model checking problem, which can be infeasible to solve for complex designs. Similarly, we may want to deduce an environment specification from observing the environment, to be used as input to synthesis. For a black-box environment, we can never know that some behavior observation sequence cannot occur. These observations give rise to the question: *Is there some way to learn from positive example traces only?*

As we discussed in the introduction to this chapter, this problem is not well-posed. To define it properly, we need to introduce a measure of how *tight* the desired specification should be. There is a spectrum of possible specification solutions ranging from *true* all the way to the specification that only allows exactly the set of traces in the example set. None of the extremes is satisfactory, so we aim to define a problem parameterized by a *tightness value* n. At the same time, we want to concretize the notion of tightness value in such a way that an efficient learning procedure to learn n-tight specifications can be given. Furthermore, the learned specifications should capture some relevant specification parts of systems (determined empirically) while being easy enough to understand by an engineer.

In this section, we give one such learning procedure. We identified *universal very-weak word automata* (UVWs) over infinite words as an easily readable specification representation with a natural definition of tightness, that lends itself to an efficient learning procedure. This automaton class has been identified as characterizing the class of properties representable both in linear temporal logic (LTL) and in the universal fragment of computation tree logic (ACTL) [157]. While this implies that there are some ω-regular properties that cannot be learned by our framework, the intersection of LTL and ACTL includes the vast majority of specifications found in case studies on specification shapes [3].

By trading away the full ω-regular expressivity, we get multiple advantages that make learning from only positive examples feasible: UVWs can be decomposed into *simple chains* [74], each representing one scenario and how the system satisfying the specification is required to react. Thus, they are easy to examine by a specification engineer. We will demonstrate that the maximum length of such a chain is also a natural notion of the complexity of a specification part, making it a good candidate for the concretization of the desired concept of tightness. Most importantly, simple chains have a natural approximation of language inclusion that enables us to efficiently learn a specification by enumerating all strictest chains that are not in contradiction with any example trace.

As in the previous section, the example traces are given as *ultimately periodic words*, i.e., words of the form uv^ω for some finite words u and v. We remark that there is nothing specific about our example traces being *positive*: the same approached could be used to learn from negative examples only.

We will demonstrate our approach on benchmarks from a case study on the AMBA AHB protocol [94]. Starting from LTL formulas describing the allowed behavior of the AMBA bus clients, we randomly generate sets of positive examples. We run our algorithm on the generated sets of different sizes and note how big the learned UVW is and how long it takes to compute it with our prototype implementation. Our experiments show that if the set of positive examples to learn from is big enough, the algorithm computes a UVW representation of the correct LTL formula. If, on the other hand, only a few positive examples are available, the UVWs grow quite large and are difficult to interpret.

3.2.1 Preliminaries

The basic notions defined in Section 3.1.1, such as *alphabet*, *ultimately periodic infinite words*, and syntax and semantics of LTL play a major role in this section, too. On top of them, we here define additional concepts, necessary for presenting the solution to the problem of learning from the set of positive examples only.

Co-Pareto Front

Let $\mathbb{B} = \{1, 0\}$ denote the set of Boolean values, with 1 representing *true* and 0 representing *false*. Moreover, let $S_1, \ldots, S_m$ be sets and $\sqsubseteq_i$ for $i \in \{1, \ldots, m\}$ be a partial order over the set S_i. Then, we call a function $f \colon S_1 \times \cdots \times S_m \to \mathbb{B}$ monotone if $s_i \sqsubseteq_i s'_i$ for all $i \in \{1, \ldots, m\}$ implies $f(s_1, \ldots, s_m) \leq f(s'_1, \ldots, s'_m)$. Adopting terminology from multicriterial optimization, we say that some tuple $(s_1, \ldots, s_m)$ is a *Pareto optimum* for f if $f(s_1, \ldots, s_m) = 1$ and for no $(s'_1, \ldots, s'_m) \neq (s_1, \ldots, s_m)$ with componentwise inequality $(s'_1, \ldots, s'_m) \sqsubseteq (s_1, \ldots, s_m)$, we have $f(s'_1, \ldots, s'_m) = 1$. The set of Pareto optima is called the *Pareto front*. Likewise, we say that some tuple $(s_1, \ldots, s_m)$ is an element of the *co-Pareto front* if $f(s_1, \ldots, s_m) = 0$ and for all $(s'_1, \ldots, s'_m) \neq (s_1, \ldots, s_m)$ with $(s'_1, \ldots, s'_m) \sqsupseteq (s_1, \ldots, s_m)$, we have $f(s'_1, \ldots, s'_m) = 1$.

Automata over Infinite Words

Given an alphabet Σ, an *automaton* over infinite words is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_I, F)$, where Q is a finite set of *states*, $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*, $Q_I \subseteq Q$ is a set of initial states, and F is a set of *final states*.

Given an infinite word $w = a_0 a_1 \ldots \in \Sigma^\omega$, we say that $\mathcal{A}$ induces a *run* $\pi = \pi_0 \pi_1 \ldots \in Q^\omega$ if $\pi_0 \in Q_I$ and for every $i \in \mathbb{N}$, we have that $(\pi_i, a_i, \pi_{i+1}) \in \delta$. An automaton defines a *language* $\mathcal{L}(\mathcal{A})$, i.e., a subset of Σ^ω that it *accepts*. *Universal Büchi automata* accept all words w for which all (infinite) runs $\pi = \pi_0 \pi_1 \ldots$ induced by the word w visit (some) states from F infinitely often. On the other side, *universal co-Büchi automata* accept all words w for which all (infinite) runs $\pi = \pi_0 \pi_1 \ldots$ induced by the word w visit (all) states from F only finitely often, i.e., there exists an $i \in \mathbb{N}$ such that for every $j \in \mathbb{N}$ with $i \leq j$ we have $\pi_j \notin F$. The final states are also called *rejecting* states in this case.

We say that an automaton is *one-weak* or *very weak* if there exists a ranking function $r \colon Q \to \mathbb{N}$ such that for every $(q, a, q') \in \delta$, we have that either $r(q') < r(q)$ or q and q' are identical. More intuitively, this means that all loops in the automaton are self-loops.

Universal Very Weak Automata

Universal very-weak automata (UVW) are universal co-Büchi automata that are also very-weak. While universal co-Büchi automata are as expressive as Linear Temporal Logic [140], universal very-weak automata are less expressive. They only capture the properties from the intersection of computational tree logic with only universal path quantifiers (ACTL) and linear temporal logic [37, 157].

The language represented by a finite *ω-automaton* (such as a UVW) is uniquely determined by the set of *ultimately periodic words uv^ω* with $u, v \in \Sigma^+$ in the language of the automaton.

An important property of a universal very-weak automaton is that it can be decomposed into *simple chains* [74]. This decomposition enumerates words outside of the language of the automaton in a simple, linear form.

Definition 3.2. A simple chain is a sequence of different states $q_1, \ldots, q_n$ such that for all $i \in \{1, \ldots, n-1\}$, there exists some $a \in \Sigma$ with $(q_i, a, q_{i+1}) \in \delta$. A simple chain is called *longest* (or *maximal*) in an automaton if it cannot be extended by an additional state at the beginning or the end of the sequence without losing the property that it is contained in the automaton. We say that a UVW is in its *decomposed form* if a) there are no transitions between the maximal simple chains of the UVW, and b) for every such simple chain $q_1, \ldots, q_n$, there are no "jumping transitions", i.e., for no $i, j \in \mathbb{N}$ and $a \in \Sigma$, do we have $(q_i, a, q_j) \in \delta$ with $j > i + 1$.

Without loss of generality, we can assume that in a decomposed UVW, every chain has an initial state and the last state is rejecting, as otherwise the whole chain or the last state, respectively, can be removed.

3.2.2 Learning Universal Very-Weak Automata

After introducing the necessary notation, we now describe our approach to learn universal very weak automata (UVW) from positive examples alone. We first define the notion of n-tightness of a UVW, which specifies what languages we want to learn from positive examples alone. We prove that the languages of n-tight automata are unique, which ensures that the learning problem is well-posed.

We then show how the simple chains of n-tight automata can be learned. As per the acceptance condition of a UVW, the chains describe the words to be rejected. Hence, learning n-tight automata amounts to enumerating all simple chains of length up to n that do not reject any of the positive examples. We show that enumerating them all can be posed as the problem of enumerating the *co-Pareto front elements* of a monotone function.

Finally, we show how this insight leads to an efficient learning process: first, the monotone function can be evaluated by solving a relatively simple model checking problem. Second, for enumerating all chains, we can use a Pareto optima enumeration algorithm from existing work, which outputs the co-Pareto front as a byproduct. The last step is then to run the usual simulation-based automaton minimization steps.

Defining Tight Universal Very-Weak Automata

Given a set of *positive* examples $\mathcal{P} \subset \Sigma^\omega$, we want to compute (learn) an automaton $\mathcal{A}$ such that $\mathcal{P} \subseteq \mathcal{L}(\mathcal{A})$. We assume that for each $p \in \mathcal{P}$, we have that $p = u_p(v_p)^\omega$ for some finite words $u_p, v_p \in \Sigma^+$ with $|v_p| \geq 1$.

Since there are infinitely many automata satisfying this condition, we need an optimization criterion for finding the automaton $\mathcal{A}$. Minimizing the number of states of the solution is not a meaningful optimization criterion in this context, as the smallest automaton is always the one with 0 states – such an automaton does not visit final states, and by the acceptance definition of UVW, this means that all words are accepted.

To permit learning from positive examples only, we hence define an alternative learning criterion: we learn the strictest automaton (i.e., the one with the smallest language) that satisfies some syntactic cut-off criterion. For UVWs, there is a natural criterion: the image size of the ranking function, or equivalently, the length of the longest chain in a UVW.

Definition 3.3. Let $\mathcal{P} \subset \Sigma^\omega$ be a set of positive examples and $\mathcal{A}$ be a UVW with $\mathcal{L}(\mathcal{A}) \supseteq \mathcal{P}$. For $n \in \mathbb{N}$, we say that $\mathcal{A}$ is n-tight for $\mathcal{P}$ if the following conditions hold:

1. There does not exist a simple chain of states longer than n in $\mathcal{A}$ (or equivalently, there exists a ranking function proving the very-weakness with co-domain $\{1, \ldots, n\}$),
2. For no other UVW $\mathcal{A}'$ with $\mathcal{P} \subseteq \mathcal{L}(\mathcal{A}') \subset \mathcal{L}(\mathcal{A})$, we have that all simple chains in $\mathcal{A}'$ are of length at most n.

We can show that for a given set of positive examples and a tightness value n, there exists an n-tight UVW. Furthermore, we will see that all n-tight automata have the same language.

Lemma 3.2. Given a set of positive examples $\mathcal{P}$ and some value $n \in \mathbb{N}$, there exists a UVW $\mathcal{A}_{n,\mathcal{P}}$ that is n-tight for $\mathcal{P}$. All other n-tight UVWs have the same language.

Proof. We construct a universal very weak automaton in its decomposed form, i.e., where the UVW consists of a finite set of simple chains without transitions between them. Let C_n be a set of all possible simple chains of length up to n. We ignore the state identities/names, so that a chain of length n is characterized completely by transitions between the states. There are fewer than $2^{|\Sigma|\cdot(2n-1)}$ different transitions (as there can be at most $2^{|\Sigma|\cdot n}$ different self-loops on n states and fewer than $2^{|\Sigma|\cdot(n-1)}$ many transitions between different states). Thus, the set C_n finite. We choose an automaton $\mathcal{A}_{n,\mathcal{P}}$ to consist of the set of all chains $c \in C_n$ such that $\mathcal{P} \subseteq \mathcal{L}(c)$. We will use the symbol $C_{n,\mathcal{P}}$ for that set. The automaton $\mathcal{A}_{n,\mathcal{P}}$, is an n-tight UVW for $\mathcal{P}$. We further claim that its language, $\mathcal{L}(\mathcal{A}_{n,\mathcal{P}}) = \bigcap_{\mathcal{A} \in C_{n,\mathcal{P}}} \mathcal{L}(\mathcal{A})$, is the language of all n-tight automata.

Indeed, for a tighter UVW $\mathcal{A}'$ (i.e., such that $\mathcal{P} \subseteq \mathcal{L}(\mathcal{A}') \subset \mathcal{L}(\mathcal{A}_{n,\mathcal{P}})$) with maximal chain length n, there must exist $\alpha \in \Sigma^\omega$ such that $\alpha \in \mathcal{L}(\mathcal{A}_{n,\mathcal{P}}) \setminus \mathcal{L}(\mathcal{A}')$. The fact that $\alpha \notin \mathcal{L}(\mathcal{A}')$ means that a run of $\mathcal{A}'$ on α will end up in one of its final (rejecting) states going through a chain of up to n states. But by $\mathcal{P} \subseteq \mathcal{L}(\mathcal{A}')$ and by our definition of $\mathcal{A}_{n,\mathcal{P}}$, this chain should be a part of $\mathcal{A}_{n,\mathcal{P}}$. Therefore, $\alpha \notin \mathcal{L}(\mathcal{A}_{n,\mathcal{P}})$, which yields a contradiction.

Let now $\mathcal{A}$ and $\mathcal{A}'$ be two n-tight automata. If they are not equivalent, then there exists a word $\alpha \in \Sigma^\omega \setminus \mathcal{P}$ accepted by one of them but not by the other. Without loss of generality, let $\alpha \notin \mathcal{L}(\mathcal{A})$. Since all chains in $\mathcal{A}$ and $\mathcal{A}'$ are of length at most n, this means that the word is rejected by one of such chains in $\mathcal{A}$. As the chain can be added to $\mathcal{A}'$ without making it reject a word in $\mathcal{P}$, this proves that $\mathcal{A}'$ is not n-tight, yielding a contradiction. Hence, the assumption that the two automata $\mathcal{A}$ and $\mathcal{A}'$ are not equivalent but both n-tight cannot be fulfilled.

□

The lemma shows that for a given set of positive examples $\mathcal{P}$, n-tight automata have a unique language. We will call that language $\mathcal{L}_{n,\mathcal{P}}$. It also shows how such an automaton can be computed: we first enumerate all simple chains of length n that a decomposed automaton accepting all elements from $\mathcal{P}$ could have. Taking these chains together, we obtain an n-tight UVW.

In the next lemma, we want to justify the intuition of the tightness parameter n, that is: the larger n is, the more tightly we can describe the set of examples $\mathcal{P}$.

Lemma 3.3. For $n, n' \in \mathbb{N}$ such that $n \leq n'$, and a given set of examples $\mathcal{P}$, the following inclusion holds: $\mathcal{L}_{n',\mathcal{P}} \subseteq \mathcal{L}_{n,\mathcal{P}}$.

Proof. We will show the claim for $n' = n + 1$ (from which the original claim follows inductively). Using the same notation as in the proof of Lemma 3.2, $C_{n+1,\mathcal{P}}$ is a set of all simple chains of length up to $n+1$ consistent with $\mathcal{P}$. In particular, $C_{n+1,\mathcal{P}}$ contains all chains from $C_{n,\mathcal{P}}$ and also those of length exactly $n+1$. The proof then proceeds as follows: $\mathcal{L}_{n+1,\mathcal{P}} = \bigcap_{\mathcal{A} \in C_{n+1,\mathcal{P}}} \mathcal{L}(\mathcal{A}) = \bigcap_{\mathcal{A} \in (C_{n,\mathcal{P}} \cup (C_{n+1,\mathcal{P}} \setminus C_{n,\mathcal{P}}))} \mathcal{L}(\mathcal{A}) \subseteq \bigcap_{\mathcal{A} \in C_{n,\mathcal{P}}} \mathcal{L}(\mathcal{A}) = \mathcal{L}_{n,\mathcal{P}}$. □

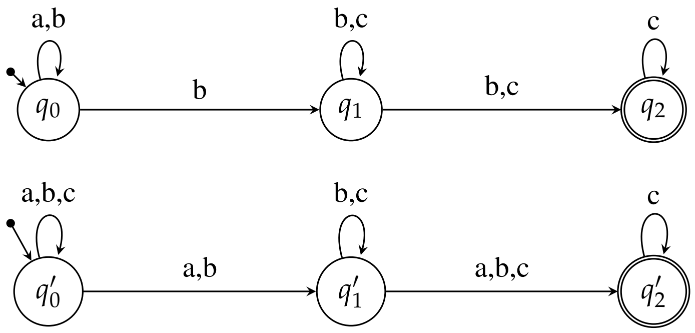


FIGURE 3.7: Two example simple chains, where the lower one is syntactically stronger than the first one.

Enumerating All Simple Chains of a UVW to be Learned

The n-tightness definition of the previous subsection states what language the automaton that we want to learn from a set of positive examples should have. However, enumerating all simple chains that are consistent with the given positive examples is computationally inefficient as their number grows exponentially with n and the size of the alphabet. We show in this subsection how this problem can be mitigated.

To do so, we represent simple chains syntactically by so-called *chain strings*. Then, we define a partial order over these strings that corresponds to language inclusion between automata consisting only of the represented chains. In order to obtain n-tight UVWs, we then only need to enumerate all chain strings that are *strongest* according to this partial order.

We visualize this idea in Figure 3.7 for the case of $n = 3$ and $\Sigma = \{a,b,c\}$. The simple chains given there are represented by the chain strings $(\{a,b\},\{b\},\{b,c\},\{b,c\},\{c\})$ and $(\{a,b,c\},\{a,b\},\{b,c\},\{a,b,c\},\{c\})$, which denote the edge labels along the chain, alternating between self-loops and edges between states. Assuming that both chains are compatible with some set of positive examples over the alphabet $\Sigma = \{a,b,c\}$, the lower one is *stronger* than the upper one in the sense that it rejects strictly more words.

This can be seen from the fact that both chains have the same length, and at each self-loop and each edge between the states, the labels for the lower chain are supersets of the respective labels for the upper automaton. On the chain string level, we can easily see that by looking at every pair of elements in the string and comparing the respective sets for set inclusion. Hence, every rejecting run for the upper chain is a rejecting run for the lower one as well. Chain strings induce a natural order by element-wise inclusion and, as already mentioned, the main idea of our approach is to enumerate only the largest chain strings with respect to the partial order that are consistent with the set of positive examples, which decreases the number of chains to be enumerated.

To simplify the presentation henceforth, the formal definition of a chain string also permits interrupted chains of states, which are not simple chains according to Definition 3.2. Furthermore, we only care about chains in which exactly the first state is initial and exactly the last state is rejecting. Generality is, however, not lost: if a simple chain does not have this form (so it has additional initial or rejecting states), then it contains another shorter simple chain of this form. This shorter simple chain can be extended to a chain of length n by duplicating

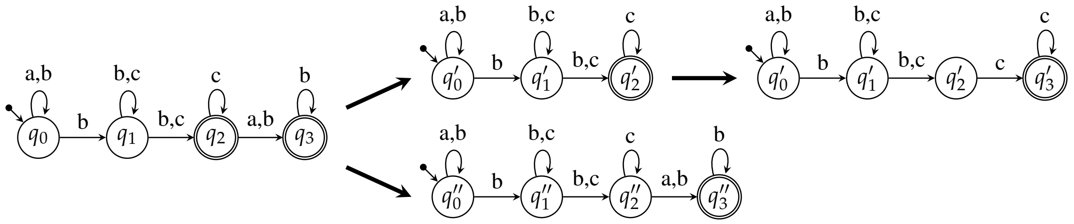


FIGURE 3.8: Splitting a chain with multiple rejecting states

the last (rejecting) state and rerouting the outgoing transitions of the previously last state to the new last state. This yields another chain of length n that is not missed when enumerating *all* maximal (w.r.t. their partial order) chain strings of length n that are compatible with $\mathcal{P}$ according to the definitions to follow. Figure 3.8 depicts this observation. The leftmost chain is split into a chain for the rejecting state q_2 and a chain for the rejecting state q_3. The now shorter chain is post-processed to a longer chain by duplicating the last state.

Definition 3.4. Let Σ and n be given. A *chain string* for Σ and n is of the form $s = (l_1, m_1, l_2, m_2, \dots, l_n) \in (2^\Sigma \times 2^\Sigma)^{n-1} \times 2^\Sigma$. Such a string s induces a chain-like automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_I, F)$ with

- $Q = \{q_1, \dots, q_n\}$;
- $Q_I = \{q_1\}$;
- $\delta = \{(q_i, x, q_i) \mid x \in l_i, i \in \{1, \dots, n\}\} \cup \{(q_i, x, q_{i+1}) \mid x \in m_i, i \in \{1, \dots, n-1\}\}$; and
- $F = \{q_n\}$.

Note that the induced automaton $\mathcal{A}$ consists of at most one single simple chain that is reachable from an initial state.

The main idea of the following enumeration procedure is to cast the problem of finding all strongest simple chains as a problem of finding the *co-Pareto front* of a monotonous function f_n over chain strings. This enables the use of a Pareto front enumeration algorithm [75] for monotone functions to enumerate all simple chains that are consistent with the given positive examples.

Said algorithm finds the Pareto front elements of a rectangular finite subset of $\mathbb{N}^u$ for some $u \in \mathbb{N}$. To make it compatible with the problem of finding simple chains, we have to encode chain strings into $\mathbb{N}^u$. The fact that all chain string elements are powersets enables a relatively simple encoding. We set $u = |\Sigma| \cdot (2n-1)$ and for every chain string $s = (l_1, m_1, l_2, m_2, \dots, l_n) \in (2^\Sigma \times 2^\Sigma)^{n-1} \times 2^\Sigma$, the corresponding encoded string in $\mathbb{N}^u$ is of the form $s' = (l_1^1, \dots, l_1^{|\Sigma|}, m_1^1, \dots, m_1^{|\Sigma|}, l_2^1, \dots, l_2^{|\Sigma|}, \dots, l_n^1, \dots, l_n^{|\Sigma|})$, where every every element l_j^i and m_j^i is either 0 or 1, depending on whether the ith element of Σ is part of the encoded l_j. The order of the elements of Σ used in this encoding is arbitrary but fixed.

A Pareto-front enumeration algorithm necessarily also enumerates the co-Pareto front to be sure it found all Pareto front points [75], which we exploit to find all strongest chain strings, as these form the co-Pareto front. The monotone function itself implements a *model checking* step of all elements in $\mathcal{P}$ against the chain, which is easy to solve due to the lasso-like structure of the examples.

Lemma 3.4. Let $\mathcal{P}$ be a set of positive examples over the alphabet Σ, $n \in \mathbb{N}$, and $f_n \colon (2^\Sigma \times 2^\Sigma)^{n-1} \times 2^\Sigma \to \mathbb{B}$ be a function that maps a chain string over Σ and n to 1 if and only if the automaton induced by the string rejects some element in $\mathcal{P}$. Then, the function f_n is monotone.

Proof. Let $s = (l_1, m_1, l_2, \ldots, m_{n-1}, l_n)$ and $s' = (l'_1, m'_1, l'_2, \ldots, m'_{n-1}, l'_n)$ be two chain strings with $l_i \subseteq l'_i$ for each $i \in \{1, \ldots, n\}$ and $m_i \subseteq m'_i$ for each $i \in \{1, \ldots, n-1\}$. Furthermore, let $\mathcal{A}_s = (Q_s, \Sigma, \delta_s, Q_{I,s}, F_s)$ and $\mathcal{A}_{s'} = (Q_{s'}, \Sigma, \delta_{s'}, Q_{I,s'}, F_{s'})$ be the corresponding UVWs as in Definition 3.4 with $Q_s = Q'_s = \{q_1, \ldots, q_n\}$.

We note that the transition relation in Definition 3.4 is monotone with respect to $l_1 \ldots l_n$ and $m_1 \ldots m_n$. Since we know that $l_i \subseteq l'_i$ for each $i \in \{1, \ldots, n\}$, using this monotonicity, we conclude that $\delta_s \subseteq \delta_{s'}$. Hence, every run π of $\mathcal{A}_s$ for some word $w \in \Sigma^\omega$ is also a run of $\mathcal{A}_{s'}$ for the same word.

As universal automata accept all words that do not induce any rejecting run, this means that all words rejected by $\mathcal{A}_s$ will also be rejected by $\mathcal{A}_{s'}$, and hence, we have $\mathcal{L}(\mathcal{A}_{s'}) \subseteq \mathcal{L}(\mathcal{A}_s)$.

To show that f_n is monotone, recall that the function f_n maps a chain string t to whether the UVW $\mathcal{A}_t$ rejects a word in $\mathcal{P}$, that is, $f_n(t) = 1$ if and only if $\mathcal{P} \not\subseteq \mathcal{L}(\mathcal{A}_t)$. Towards a contradiction, assume that $f_n(\mathcal{A}_s) = 1$ but $f_n(\mathcal{A}_{s'}) = 0$. This means that there exists a word $w \in \mathcal{P}$ such that $w \notin \mathcal{L}(\mathcal{A}_s)$ and $w \in \mathcal{L}(\mathcal{A}_{s'})$. Thus, w witnesses $\mathcal{L}(\mathcal{A}_{s'}) \not\subseteq \mathcal{L}(\mathcal{A}_s)$, which is a contradiction to the previous part of the proof. In conclusion, we obtain that f_n is monotone. □

Because f_n is monotone, we know that the UVW from Lemma 3.2 has the same language as the language defined by the co-Pareto front of f_n. Indeed, the co-Pareto front represents those chain strings that do not reject any word from $\mathcal{P}$, but reject as many other words as possible.

Corollary 3.1. Let $\mathcal{P}$ be a set of positive examples over the alphabet Σ, $n \in \mathbb{N}$, and A be the set of automata induced by the co-Pareto front elements of the function f_n. The language $\bigcap_{\mathcal{A} \in A} \mathcal{L}(\mathcal{A})$ is n-tight for $\mathcal{P}$ and Σ.

Proof. In Lemma 3.2, we have shown that by taking the intersection of the set of all UVWs of length n, consistent with $\mathcal{P}$, we get an n-tight automaton. The mentioned set is denoted by $C_{n,\mathcal{P}}$. Furthermore, we have shown that all n-tight automata have the same, unique n-tight language, which we denote by $\mathcal{L}_{n,\mathcal{P}}$. Let's denote the language $\bigcap_{\mathcal{A} \in A} \mathcal{L}(\mathcal{A})$ by $\mathcal{L}_A$.

It is clear that $\mathcal{L}_{n,\mathcal{P}} \subseteq \mathcal{L}_A$, because $A \subseteq C_{n,\mathcal{P}}$. The other inclusion, $\mathcal{L}_A \subseteq \mathcal{L}_n$, follows from the definition of the set A. Let us assume $\mathcal{L}_A \not\subseteq \mathcal{L}_n$, i.e., that there is a word w such that $w \in \mathcal{L}_A$ and $w \notin \mathcal{L}_n$.

Then, there exists an automaton $\mathcal{D} \in C_{n,\mathcal{P}} \setminus A$ such that $w \notin \mathcal{L}(\mathcal{D})$. The fact that $\mathcal{D}$ is not at the co-Pareto front for function f_n means that there is another automaton on the front, $\mathcal{D}' \in A$, which also rejects w. This is a contradiction with the assumption that $w \in \mathcal{L}_A$.

□

The automaton from this corollary can be built easily, as universal very-weak automata are closed under language intersection by just merging the state sets, transition relations, and initial states [74]. This enables us to simply merge all chains found together into a single UVW. The question, however, is how to do this in an efficient way.

Engineering Considerations of the Learning Algorithm

After the co-Pareto front of strongest chains has been enumerated, the last step in the construction of the UVWs is merging them to a single UVW. We add the chains one by one to a solution UVW. After every such step, we use the automaton minimization techniques described in [3] to reduce the size of the automaton. If the process is stopped prematurely, the result is still useful—a UVW that accepts a superset of the language that the final automaton (given sufficient computation resources) would accept. This property makes it possible to use the algorithm in the *anytime* fashion, stopping it when a given resource budget is exceeded.

It remains to be described how f_n can be computed efficiently. We implemented this process as follows: let $\mathcal{P} = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ and $\mathcal{A} = (Q, \Sigma, \delta, Q_I, F)$ with $Q = \{q_1, \ldots, q_n\}$ be an automaton induced by a chain string to be checked. For every $j \in \{1, \ldots, m\}$, we translate (u_j, v_j) to a deterministic Büchi automaton $\mathcal{A}'$ accepting exactly $u_j(v_j)^\omega$. Such an automaton has $|u_j| + |v_j| + 1$ states. We then check if $\mathcal{A}'$ admits a word rejected by $\mathcal{A}$, i.e., if $\mathcal{L}(\mathcal{A}') \cap \overline{\mathcal{L}(\mathcal{A})} \neq \emptyset$. Since the complement of a universal co-Büchi word automaton can be obtained in the form of a non-deterministic Büchi automaton by just interpreting $\mathcal{A}$ as such, the standard product construction from linear-time model checking can be applied to test if $\mathcal{L}(\mathcal{A}') \cap \overline{\mathcal{L}(\mathcal{A})} \neq \emptyset$. The function f_n can then simply iterate over all examples $j \in \{1, \ldots, m\}$ and test if this is the case for any of them. Whenever it finds that $\mathcal{L}(\mathcal{A})' \cap \overline{\mathcal{L}(\mathcal{A})} \neq \emptyset$ for some automaton $\mathcal{A}'$ built from a positive example, the function f_n returns 1. Otherwise, it returns 0 after iterating through all values for $j \in \{1, \ldots, m\}$.

Note that in an actual implementation of f_n, there is no need to explicitly build $\mathcal{A}'$ or construct the product Büchi automaton. Rather, the implementation can make use of the fact that only the last state of the simple chain is rejecting. So it can compute the states of the product that are reachable and then check if state q_n in the $\mathcal{A}$ component of the product is reachable while at the same time, all characters in v_j are contained in the self-loop label of state q_n. If and only if that is the case, a positive example number j is rejected by $\mathcal{A}$.

3.2.3 Evaluation

We implemented the presented approach in a prototype toolchain, which is available on Github[4]. The enumeration of simple chains is performed by a tool written in `C++`, while the subsequent minimization of the resulting UVWs is implemented in `Python 3`.

In order to assess the performance of our approach on practically relevant properties, we considered the specification of the industrial on-chip bus arbiter of the AMBA AHB bus [15].

[4] https://github.com/TUC-ES/unite

TABLE 3.6: Mean computation times for UVW learning for the ten LTL properties considered in Section 3.3.5

	Property	Time in s	
		chain len. 2	chain len. 3
1)	$\mathsf{G}[a \rightarrow (b \vee c \vee d)]$	0.763	timeout
2)	$\mathsf{G}[a \rightarrow (b \vee c)]$	0.517	1.029
3)	$\mathsf{G}[\mathsf{X}\,\neg a \rightarrow (\neg b \leftrightarrow \mathsf{X}(\neg b))]$	0.493	1.184
4)	$\mathsf{G}[a \rightarrow \neg b]$	0.408	0.713
5)	$\mathsf{G}[a \rightarrow (\neg b \wedge \neg c)]$	0.533	1.059
6)	$\mathsf{G}\,a$	0.526	1.057
7)	$\mathsf{G}[a \rightarrow \mathsf{F}\,b]$	0.442	0.870
8)	$\mathsf{G}[(a \wedge b) \rightarrow \mathsf{X}\,\mathsf{F}(\neg c)]$	0.634	119.123
9)	$\mathsf{G}\,\mathsf{F}\,a$	0.423	0.685
10)	$\mathsf{G}\,\mathsf{F}(\neg a \wedge \neg b)$	0.428	0.702

Specifically, we considered ten assumptions made for the master of the AHB bus, as described in the case study paper on synthesizing AMBA AHB [94]. For simplicity, we abstracted from the concrete variable names and rewrote predicates over categorial values into individual propositions. For instance, the original property A8 [94] referring to a burst sequence of unspecified length (denoted by the value INCR) is $\mathsf{G}[\text{HLOCK} \wedge (\text{HBURST} = \text{INCR}) \rightarrow \mathsf{X}\,\mathsf{F}(\neg\text{REQ\_VLD})]$. It is rewritten into $\mathsf{G}[(a \wedge b) \rightarrow \mathsf{X}\,\mathsf{F}(\neg c)]$. All the resulting formulas are shown on the left-hand-side of Table 3.6.

Except for Property 3, all properties can be represented in UVW form by a single simple chain with two states each. For Property 3, we need two chains of length 3. The properties employing *two to four* atomic propositions have been learned over words with characters that encode this number of propositions. Property 6 has been learned over positive examples in which each character has three proposition values, while for Property 9, we used two propositions. This deviation was necessary to ensure that there are enough distinct positive examples for these properties.

For each property, we computed 50,000 different ultimately periodic words uv^{ω} that satisfy the property, where $|u|$ is of length 0, 1, 2, 3, or 4, while $|v|$ is of length 1, 2, 3, or 4. The characters of the words are the subsets of propositions holding, and all word part lengths are equally likely to be chosen. We also use a uniform probability distribution over the characters when computing the positive examples. Whenever a non-positive example for the property is found during the positive example computation, it is discarded and another example word is computed instead. We ran every experiment on 10 different example sets generated in this way and report the mean values obtained in the following.

The experiments were conducted on a computer with four AMD EPYC 7251 processors running at 2,1 GHz and an x64 version of Linux. The available main memory per run of the learner was restricted to 3 GB. We used a computation time limit of 600 s per learning problem.

Table 3.6 contains the mean computation times for all properties when using all 50,000 positive examples as input in each case.

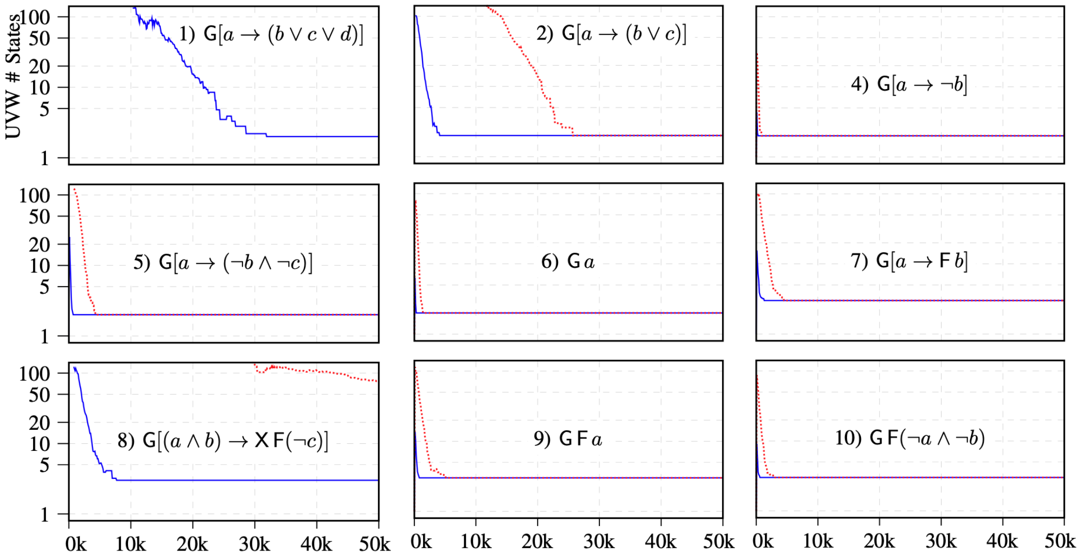


FIGURE 3.9: UVW sizes for nine of the ten examples. The dotted lines are for a UVW chain length of 3, while the solid lines are for a UVW chain length of 2. The number of positive examples given to the learner ranges between 100 and 50,000 and is displayed on the x-axis of each chart. Parts in the charts with absent lines represent timeouts, which were common for low numbers of positive examples.

It can be seen that for most combinations, our approach computes a UVW rather quickly. Only for one property with a higher number of atomic propositions and an unnecessarily long chosen chain length, the toolchain times out.

Figure 3.9 shows for nine of the ten properties how big the computed UVW are, where sizes for both chain lengths of 2 and 3 are reported. Here, we varied the number of positive examples provided to the learner along the X-axis (minimum: 100, in steps of 100). For a very low number of examples, our toolchain often times out. This is caused by the fact that the tightest UVW is often very large when not enough positive examples are available. It can also be observed that for a lower chain length, the computed UVW converges much earlier.

Figure 3.10 depicts the relationship between computation time and the sizes of the computed UVWs in more detail, using Property 3, the one that was left out of Figure 3.9. It can be observed that computation times are very short when enough positive examples are available, and they grow only very mildly with additional positive examples. When, however, not enough examples are available, the approach computes a much larger number of simple chains, which also increases the workload of the UVW minimization heuristic.

Finally, Figure 3.11 depicts the UVWs learned for Property 3, $\mathsf{G}[\mathsf{X}\,\neg a \rightarrow (\neg b \leftrightarrow \mathsf{X}(\neg b))]$. The property can only be learned correctly with a chain length of 3, and the two paths through the UVW from Figure 3.11b show the two ways in which the property can be violated, namely:

(a) after a character with $b = \mathit{true}$ is seen, in the next step both a and b are *false*, and

(a) after a character with $b = \mathit{false}$ is seen, in the next step b is *true* and a is *false*.

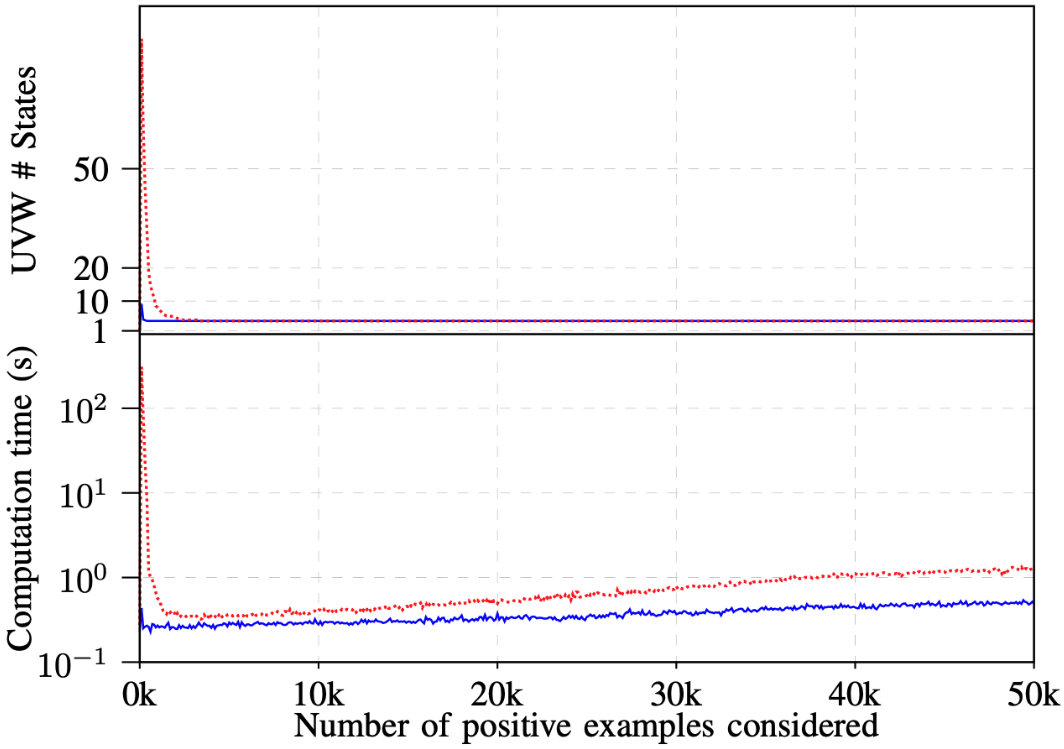
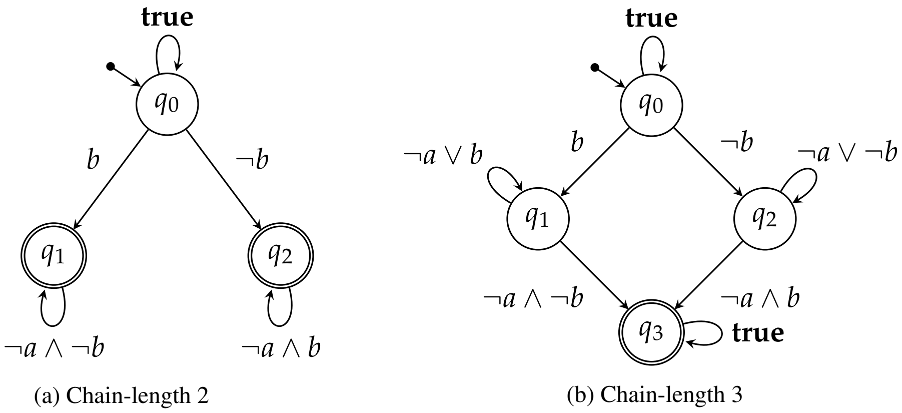


FIGURE 3.10: Joint plot for the computation time and UVW sizes for Property 3.



(a) Chain-length 2

(b) Chain-length 3

FIGURE 3.11: Learned UVW from the positive examples for the LTL property $\mathsf{G}[\mathsf{X}\,\neg a \rightarrow (\neg b \leftrightarrow \mathsf{X}(\neg b))]$.

The automaton has a simple structure and is quite easy to read. The computed UVW for a chain length of 2 (Figure 3.11a) is, as expected, an overapproximation of the language to be learned.

For all ten LTL properties considered in our experiments, the learned UVWs for the correct chain lengths represent the correct languages and have a minimal number of states. Moreover, the resulting UVWs are fairly easy to understand (we refer the reader to the extended version of the experiments, available in the code repository for their depiction), which underpins the use of UVWs as an easy-to-understand specification formalism.

The results of the experiments are telling us that the proposed method works well when many example traces are available, but can not handle when only a few are given. A further challenge is that our method takes the tightness value as its input. It would be valuable if we were able to determine an appropriate tightness value automatically from the sample. This, however, remains an open problem.

3.2.4 Related Work

The problem of automata learning from data traditionally comes in two different settings: *active* [10, 130, 199] and *passive* [182, 110, 179]. In an active setting, the learning algorithm interacts with a *teacher*. The teacher answers two kinds of queries: membership queries (whether a proposed word is in the language of the automaton) and equivalence queries (whether a proposed automaton is the correct one). The learning process stops once the teacher answers an equivalence query positively. Having a teacher that is able to answer equivalence queries is a strong assumption. Our work focuses on the passive setting, where the learning algorithm only has access to data, a set of classified examples.

The standard problem formulation of passive learning is that a sample consists of positive and negative example traces (and this is the setup considered in Section 3.1). For such a setup, several methods have been proposed for learning not only automata [110, 179], but also LTL formulas [178, 47, 131], or STL formulas [39, 170]. None of these methods provides good results when they are presented with only one class of traces—they return a trivial solution, one that accepts (or rejects) all possible traces.

Our problem—learning a specification from system traces— fits into the process mining framework (see Aalst et al. [1] for an overview): given an event log from a process, find a process model that satisfies certain properties. The properties are *fitness* (the model should be consistent with the examples from the log), *precision* (the model should not be overly general, e.g., modeling arbitrary examples), *generalization* (the model should not be overly tight, e.g., consistent only with the examples from the log), and *simplicity* (the model should be simple). Different operationalizations of the four properties give rise to different problem formulations and solutions. By choosing UVWs as our model, we get (structural) simplicity and connect it to the generalization property by the tightness value n, for which we require the tightest possible UVW consistent with the data.

Closely related to our approach is an algorithm by Avellaneda et al. [18] for inferring deterministic automata over finite words (DFAs) from positive finite-word examples alone. Their algorithm searches for an n-state automaton $\mathcal{A}$ that is consistent with the given set of positive examples and for which no other n-state DFA $\mathcal{A}'$ exists such that the language of $\mathcal{A}'$ is a strict subset of the language of $\mathcal{A}$. Both their approach and ours identify the language to be learned in the limit and use a single additional parameter for choosing the complexity of the language to be learned. Unlike in our approach, the resulting language in their algorithm is not unique for a given value of n.

Another direction of previous work is the identification of Live Sequence Charts (LSCs) [152, 151] from system runs. Live Sequence Charts [58] are a specification formalism that is popular for its compliance to the UML standard and the corresponding tools (e.g., IBM RSA). The set of properties representable as Live Sequence Charts, when not using free variables, was shown to be contained in the intersection of LTL and ACTL [138], which is characterized by UVWs (the version with free variables is characterized as a subset of first-order CTL* [59]). The existing work on mining LSCs [152, 151] borrows the concepts of *support* and *consistency* from data mining [103]. With user-defined thresholds for support and consistency, charts are enumerated until one exceeding that threshold is found.

Rather than giving more credibility to patterns occurring most often in the example traces (as is the case when using the notion of support), our method prefers semantically stronger UVWs, controlled by their size. This lets our approach converge to the same property regardless of the distribution of the traces, as long as all traces (in the form of ultimately periodic words) have a non-zero probability of occurring.

A problem related to ours by the fact that the learning happens over (positive) demonstrations only is inverse reinforcement learning [180]. There, however, it is the reward function that is being learned. Obtaining only the reward function does not provide an interpretable task specification. Inspired by inverse reinforcement learning, Vazquez-Chanlatte et al. [218] learn LTL-like temporal specifications from demonstrations. In order to do so, they have to pre-compute the implication lattice between the possible specifications, which limits the applicability of their approach. This is not necessary in our work, as we take advantage of the syntactic approximation of language inclusion between simple chains of UVWs. On the other hand, they successfully handle noise in the sample.

3.3 Interactive Specification Inference for Robotic Systems

In this section, we turn our attention towards interaction between the human and the robot. Our goal is to make it easier for humans to create precise, formal specifications that correspond to their intent. This reminds of a setup in sections 3.1 and 3.2. Indeed, this section will directly build on the SAT-based technique from 3.1. However, fixing the domain to human-robot communication introduces some restrictions. In particular, we cannot hope to obtain more than a handful of examples from the human.

We set our specification language for tasking robots to be the co-safe fragment of linear temporal logic (LTL). LTL comes as a natural choice, given that there are planning and synthesis toolchains from LTL specifications to implementations on robotics platforms, for instance, `Antlab` from Chapter 2 alongside many others [136, 148, 204]. Typical tasks in the robotic domain are co-safe tasks (the ones that can be exemplified by a finite path), and our work focuses on such tasks. Unfortunately, the fact remains that using LTL is not straightforward for untrained users. [115, 72].

Natural language descriptions and programming-by-examples have been considered as alternative, end-user friendly, ways to specify complex tasks. While LTL specifications can be written in structured natural language [137], such systems still require a solid understanding of LTL syntax and semantics. Dialog-based interfaces are usually limited to a fixed set of the most common scenarios [214, 133]; utterances beyond the pre-programmed tasks are rejected.

On the other hand, as we have also seen in sections 3.1 and 3.2, current programming-by-example techniques for temporal specifications (such as automata or LTL) expect many positive and negative examples to be provided [56, 52]. Previous work in programming through examples has shown that it is unreasonable to expect end-users to provide more than a few examples [209]. Moreover, classical results in learning theory [95, 11] show that one cannot identify non-trivial classes of formal languages (which includes LTL)—even in the limit—employing only positive examples. As we discussed in Section 3.2, providing negative examples is challenging; while it is simple to show an execution that accomplishes the task at hand, users are confused when asked to show *how not to* accomplish the task. Thus, in this section, we will identify domain-specific assumptions that will allow us to generate negative examples from the positive ones.

In this section, we apply synthesis from *examples* and *natural language description* to the problem of learning robotic tasks expressed in co-safe LTL formulas. Our approach consists of three components. First, a synthesis procedure that takes a natural language description of a task and an example execution trace from the user and generates a set of candidate LTL specifications. Second, an interactive loop that uses distinguishing examples to identify the correct LTL specification from that set. Third, a generalization step that eventually learns a parameterized LTL specification. The three components ensure the following properties.

- Our approach requires only a single example to be provided by the user, and a few rounds of interaction in which the user judges examples provided by the system.
- We do not require the user to provide negative examples.
- Our technique generalizes from individual examples to learn a parameterized representation and does not require the user to repeat the synthesis for small changes in tasks.

We bias the search in the synthesis procedure based on the natural language description and narrow down the search space based on the following two assumptions about user-provided examples. First, we assume a principle of *no excessive trace*: when a user provides an example trace, we assume that no proper prefix of the trace is sufficient to satisfy the underlying specification. The second principle is similar. When a user provides an example, some of the actions are complex, consisting of a number of *primitive actions*. We assume a principle of *no excessive effort*: every primitive action in the example is necessary to satisfy the specification. Contrapositively, the same example with a strict subset of primitive actions does not satisfy the specification. Thus, we can generate a set of negative examples from a single user-provided example.

We learn a set of candidate LTL specifications that are consistent with the one positive example and the generated negative examples, and are informed by the natural language description. Building upon the work from Section 3.1, we encode the learning problem as a *multi-objective* optimization problem modulo SMT [34]. The optimization objectives approximate the correspondence of the natural language description to the prospective specification.

Given two candidate formulas, we use another instance of multi-objective optimization modulo SMT to generate a world and a robot behavior in it that distinguishes between the two candidates (assuming such a world exists). We interactively show the generated behavior to the user and ask them to judge whether it is consistent with their original command. In this way, we can narrow down candidate solutions to a single one.

While our synthesis procedure learns a specification, one expects that the system maintains a parameterized mapping from natural language descriptions to specifications. For example, it is unreasonable that the user must separately teach the robot to "pick up a red" item and to "pick up a green item". Our system generalizes the learned LTL specification through a form of grammar expansion based on the work in language naturalization [223].

We start with a core DSL to express LTL properties and expand the DSL with new grammar rules whenever the synthesis procedure learns a new mapping. This expansion process, called *naturalization* in the natural language processing literature, allows the system to generalize from previously synthesized specifications and combine them in new ways at a later point. Potential ambiguous parses introduced by the new rules are ranked by a probabilistic model that gets updated through interaction with the user.

We have implemented the synthesis technique in LTLTALK, which implements the synthesis procedure and a planning toolchain on top of a visual interface for a robot navigating a blocks world. (The LTLTALK's codebase [5] and web interface [6] are publicly available.) LTLTALK's interface allows the user to provide natural language descriptions of tasks as well as example executions in the blocks world. An important characteristic of our domain is that we can provide quick visual feedback to the user showing the effect of task execution. Thus, we can use the interface to demonstrate distinguishing worlds in the third phase of synthesis.

We have used LTLTALK to learn LTL specifications for a number of common tasks in a simulated world. We empirically show that most tasks can be learned through the use of just one example, a few (less than 4) interactions, and less than 20 seconds. We furthermore show in a case study how naturalization generalizes beyond the synthesized tasks, effectively increasing the scalability of LTL specification synthesis.

3.3.1 Overview and Motivating Example

Figure 3.12 shows the high-level scheme of LTLTALK. LTLTALK maintains an extensible grammar, mapping natural language utterances into a core language (which corresponds to LTL). When a user issues a (natural language) description of a task, if the description can be parsed into LTLTALK's current core language, a plan is generated and executed. If, on the other hand, LTLTALK cannot parse the command using its current grammar, the system

[5] https://github.com/mpi-sws-rse/ltltalk-interactive-synthesis

[6] https://ltltalk.mpi-sws.org

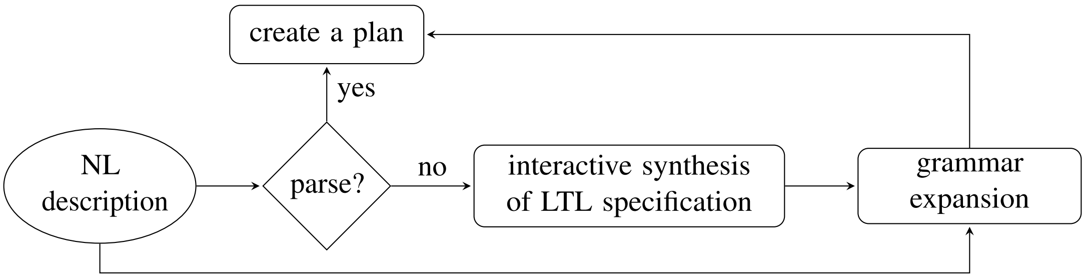


FIGURE 3.12: Schematic view of LTLTALK

synthesizes an LTL specification for the utterance through interaction with the user. Having produced the natural language utterance and its LTL equivalent, LTLTALK expands its formal language by inducing new grammar production rules.

Consider the robot simulation world presented in Figure 3.13. The world includes a robot (the gray cube) that can move about its environment consisting of dry and wet tiles, and pick items of different shapes and colors. LTLTALK internally maintains an extensible grammar, initialized to a version of LTL, that maps known commands to LTL specifications. Consider a situation when a user instructs the robot to *take one red item from (7,4)*, but the robot does not understand the instruction (as this utterance is not yet part of the internal grammar). We show how LTLTALK learns an LTL specification and adds a mapping to its grammar. Initially, LTLTALK asks the user to provide an example for the request by navigating the robot in the simulation world (Figure 3.13a).

Step 1: Generate Candidates

Using the example provided by the user together with the original natural language instruction, LTLTALK creates a number of candidate specifications:

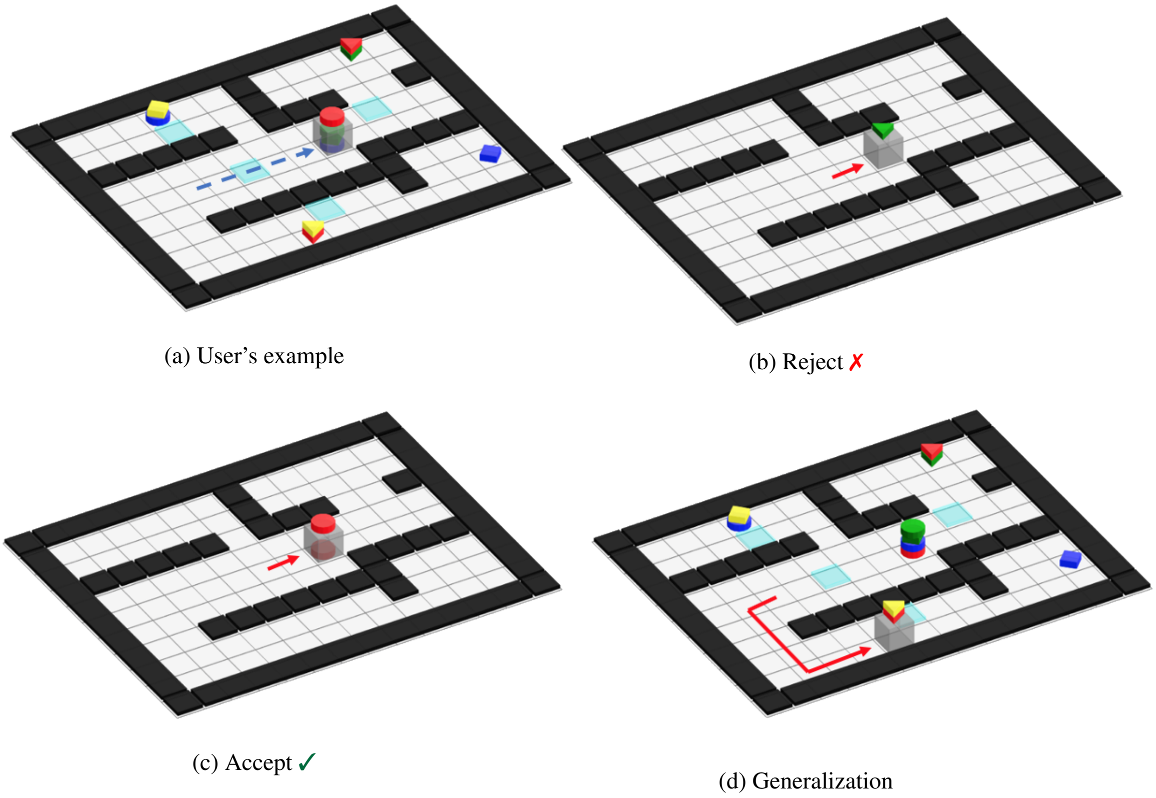
a) **`eventually`** `pick` *`one red`* `item at` $(7,4)$,

b) **`eventually`** `pick` *`one`* `item at` $(7,4)$,

c) **`eventually`** `pick` *`every red`* `item at` $(7,4)$, and

d) **`not`** *`robot`* `at` *`dry`* **`before`** `pick` *`every red`* `item at` $(7,4)$.

The first three instructions correspond to requesting that eventually one red item, or one item of any color, or every red item from the location $(7,4)$ is picked; the fourth one corresponds to asking that eventually a red item from the location $(7,4)$ is picked, but before that the robot must pass over a wet tile. (We will see the full presentation of the core language in Section 3.3.4.) Note that there are still other specifications consistent with the example. However, the number of generated candidates is limited by a hyperparameter of the algorithm and using the natural language instruction makes sure that the found candidates are relevant to the user's intent (full details of the algorithm are presented in Section 3.3.3).



(a) User's example

(b) Reject ✗

(c) Accept ✓

(d) Generalization

FIGURE 3.13: User's interaction with LtlTalk. In 3.13a, the user provides an example for the command *take one red item from 7,4* by moving the robot to $(7,4)$ and picking a red item. Afterwards, based on the example and the (natural language) command, a number of candidate specifications is created. In 3.13b, LtlTalk shows the robot picking one green item, which the user rejects. In 3.13c, LtlTalk shows the robot picking $(7,4)$ without ever going through a wet tile and picking only one red item. The user accepts it, thus narrowing down the set of candidates to a single specification. Finally, 3.13d shows that LtlTalk generalized from the inferred specification and now understands commands such as *take every triangle item from 4,0*.

Step 2: Filter Based on Distinguishing Worlds

In order to determine which one of the candidate specifications is the one that the user had in mind, LtlTalk first shows a world in which there is only one green triangle at the location $(7,4)$ and the robot picking that item (Figure 3.13b). When the user rejects that behavior, LtlTalk can eliminate b) from the list of candidates. When in the next interaction the user accepts the robot reaching the location $(7,4)$ without passing over a water tile and picking only one of the two red items (Figure 3.13c), LtlTalk eliminates specifications c) and d), and concludes that the target specification is **eventually** pick *one red* item at $(7,4)$. The creation of the distinguishing worlds is described in Section 3.3.3.

Step 3: Generalization and Grammar Extension

Having determined the specification corresponding to the natural language description *take one red item from (7,4)*, LtlTalk uses the (*specification*, *NL description*) pair to form a new, generalized rule. It augments the underlying grammar by the new production rule. (This augmentation is described in Section 3.3.4.) With its augmented grammar, LTLTalk is now able to understand expressions such as *take every triangle item from (4,0)*, as illustrated in Figure 3.13d.

3.3.2 Formal Models for Tasks and the World

We first formalize the notion of specification language, world model, and user demonstration.

We use the co-safe fragment of LTL as the specification language. We represent LTL by syntax DAGs. (Both LTL and syntax DAGs are defined in Section 3.1.1.) The semantics of LTL is defined over example traces. The formulas for which there exists a *good prefix* of a trace, i.e., a finite trace whose every extension satisfies the formula, are called *co-safe formulas*.

While LTL formulas are traditionally interpreted over infinite traces, in practical applications (including task-oriented robotic commands), it is often necessary to interpret LTL formulas over finite traces. To facilitate that need, the semantics of LTL can be modified to support evaluation over finite traces, as was done by De Giacomo et al. [61]. We use the finite traces semantics to evaluate candidate formulas on the user-provided finite example traces.

The World Model

The propositional variables in our LTL formulas will come from a *world model* describing the world and the robot's actions in it. We take the planning approach and partition the propositional variables into *fluents* $\mathcal{F}$ and *actions* $\mathcal{A}$ [62]. The set of fluents consists of the propositional or integer variables describing facts about the state of the world; a valuation to the fluents defines a state. An action from $\mathcal{A}$ defines a transition between two states.

A world w is fully described by a tuple $\mathrm{w} = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{A}}, \varphi_{\mathcal{F}}, \varphi_{init})$, where $\mathcal{F}$ is the finite set of *fluents*, $\mathcal{A}$ is the finite set of *actions*, the fluent constraint $\varphi_{\mathcal{F}}$ is an LTL formula on $\mathcal{F}$ describing invariants on the world, the action constraint $\varphi_{\mathcal{A}}$ is an LTL formula describing the effect of an action on the fluents, and φ_{init} is an LTL formula over $\mathcal{F}$ describing the initial state of the world.

For our block world with fixed world width w, world height l, set of colors C, and the set of shapes S, the set of fluents $\mathcal{F}$ contains propositional variables describing the state of the world (locations of obstacles, items, the robot, and what the robot is carrying). For ease of readability, we describe our examples using Prolog-like predicates with relations and (bounded) integer variables. Internally, these predicates are compiled into Boolean propositional variables. We omit a full list of fluents and their propositional encoding but give examples.

We model the blocks world as a set of tiles (i, j), $i \in \{1, \ldots, w\}$, $j \in \{1, \ldots, l\}$. Each tile (i, j) is exactly one of $\mathit{wall}(i, j)$ (obstacle), $\mathit{dry}(i, j)$ (dry tile), or $\mathit{wet}(i, j)$ (wet tile). The predicate $\mathit{at}(x, y)$ denotes the robot is currently at location (x, y), and $\mathit{items}(i, j)(c, s) = k$ indicates there are k items of color c and shape s at location (i, j). The predicate $\mathit{carry}(c, s, k)$ indicates that the robot is carrying k items of color c and shape s.

The fluent constraints specify invariants that must hold on all reachable states. For example, by asserting that the robot can only be in a location within the block world which is not a wall, and that every location is either a wall, a wet tile, or a dry tile:

$$\bigwedge_{i,j} \mathsf{G}\Big(at(i,j) \rightarrow \neg wall(i,j) \wedge (exactlyOne(dry(i,j), wall(i,j), wet(i,j))\Big)$$

(where *exactlyOne* is the propositional formula that ensures exactly one of its inputs is true), or by specifying that the items can be only at locations that are not a wall:

$$\bigwedge_{i,j,c,s,k} \mathsf{G}\Big(items(i,j)(c,s) = k \rightarrow \big(k \geq 0 \wedge (k > 0 \rightarrow \neg wall(i,j))\big)\Big)$$

The initial world φ_{init} defines the locations of wall and wet tiles, the position of the robot, and the location of all the items.

The actions available to the robot are motion in one of the 4 cardinal directions and picking a set of items based on its properties. Moving is expressed by the action $mov(x,y)$, with $x, y \in \{-1, 0, 1\}$ and $|x| + |y| = 1$, and denotes that the robot moved in a relative direction. Picking objects is expressed by the action $pick(c,s) = k$, which indicates that k items of color c and shape s are picked. The action constraints state the effect of an action. Typically, the effect of an action is specified as $\mathsf{G}(\varphi_1 \rightarrow \mathsf{X}(a \rightarrow \varphi_2))$, where $a \in \mathcal{A}$ and φ_1 and φ_2 are propositional formulas on the fluent variables.

For example, the constraint

$$\bigwedge_{i,j} \bigwedge_{(x,y) \in \substack{\{(1,0),(-1,0),\\(0,1),(0,-1)\}}} \mathsf{G}((at(i,j) \wedge \neg wall(i+x, j+y)) \rightarrow \mathsf{X}(mov(x,y) \rightarrow at(i+x, j+y)))$$

states that a move changes the position of the robot to the corresponding cell if the target cell is not a wall. Similarly, we specify that whenever the robot standing on the tile (i,j) picks k items of color c and shape s, these items are removed from the tile.

$$\bigwedge_{i,j} \bigwedge_{\substack{c \in C\\ s \in S}} \bigwedge_{k,l \in \mathbb{N}} \mathsf{G}((items(i,j)(c,s) = l \wedge l \geq k) \rightarrow \mathsf{X}(pick(c,s) = k \rightarrow items(i,j)(c,s) = l - k))$$

Other action constraints specify, for example, that the robot can execute exactly one action in each step:

$$\mathsf{G}\left[\bigvee_{a \in \mathcal{A}} a \wedge \bigwedge_{\substack{a,a' \in \mathcal{A}\\ a \neq a'}} (\neg a \vee \neg a')\right]$$

Given a world $(\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{A}}, \varphi_{\mathcal{F}}, \varphi_{init})$, a finite trace is a *world trace* if it satisfies the fluent constraint, the action constraints, and the initial constraint φ_{init}.

We further assume that the set $\mathcal{A}$ is partially ordered by a relation $\preceq$. This allows us to model related actions. For example, "pick an item" $\prec$ "pick two items".

In this work, we consider task-like specifications: the ones where a concrete change in the environment needs to be accomplished in a one-off manner. We are not interested in infinite executions or specifications that can be satisfied by no action at all. Therefore, we assume that

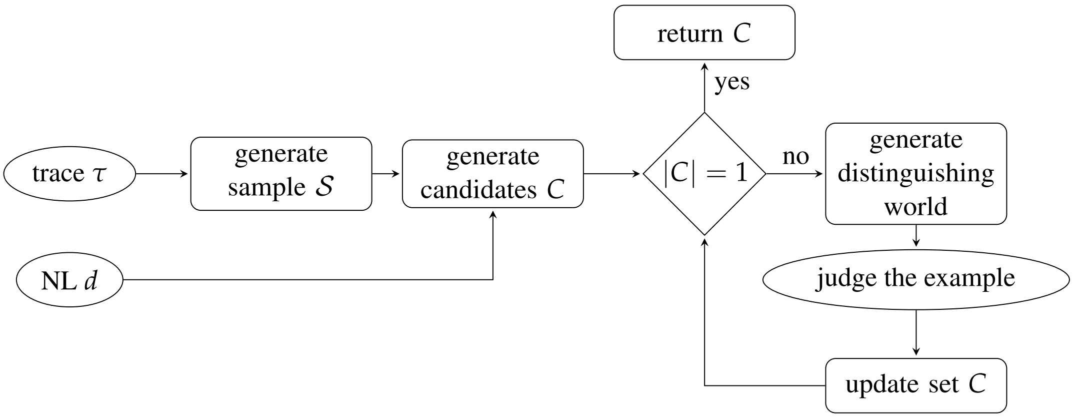


FIGURE 3.14: High-level interactive synthesis algorithm

the user's specifications are co-safe LTL formulas, as they correspond to the described notion of task. Furthermore, we assume that the user is able to demonstrate the intended specification in a given world by a world trace.

3.3.3 Interactive Specification Synthesis

We now describe our algorithm for inferring an LTL specification, starting with a natural language description and an example trace and creating distinguishing traces interactively.

Figure 3.14 shows a high-level view of the interactive synthesis method. The method takes a world trace τ and a natural language description d as an input (provided by a user). First, the trace τ is used to create a sample $\mathcal{S}$ of positive and negative traces. Positive traces satisfy the specification; negative traces do not. From the information present in $\mathcal{S}$ and d, a set of candidate specifications C is generated. In order to determine which one of the candidates is the right one, LTLTALK generates an initial world state and a world trace that are able to distinguish between the two most likely candidates. The user judges the provided example (by answering if the example matches the intent expressed in d). The user's verdict is then used to update the set of candidate specifications until only one candidate remains.

Constructing a Sample from a Single Example

The example provided by the user represents a positive example, a member of $\mathcal{P}$, the one that should be satisfied by the prospective specification. With only this one example, a space of potential specifications is too unconstrained, e.g., *true* would be a legitimate candidate specification, as it is consistent with the example. To shrink that space, we would like to infer a *sample* $\mathcal{S} = (\mathcal{P}, \mathcal{N})$ that also contains negative examples. In order to infer what the set $\mathcal{N}$ might look like, we use two domain-specific heuristic principles: the *principle of no excessive trace* and the *principle of no excessive effort*.

The first principle says that if a prefix of the trace τ would already be a good example for the user's intention, then the user would never bother to give the full trace τ. Therefore, we add all the proper prefixes of the trace τ to $\mathcal{N}$.

The second principle uses the partial order $\prec$ over the set of actions and it says that the user will not demonstrate unnecessarily complex actions. The partial order in our robotic domain is naturally defined for picking actions: each picking action is a complex action consisting of multiple atomic picks of individual items. Thus, the partial order is defined by a subset relation between the picked items. Following the principle, any trace $\tau[a/a']$ that replaces an action a with a strictly lower action a' (i.e., $a' \prec a$) is added to $\mathcal{N}$.

Note that neither of the principles assumes that the user knows how to show a demonstration that is optimal in any sense. Instead, they only assume that the user will not go beyond an already provided demonstration that is consistent with the specification.

For the user's example shown in Figure 3.13a, the robot moving any part of the path from $(2,4)$ to $(7,4)$ constitutes a negative example, according to the principle of no excessive trace. The principle of no excessive action does not apply to this example, since the only picking action in the example is already a primitive action (picking an individual item).

Constructing a Set of Candidate Specifications

Having created the sample $\mathcal{S}$, we are interested in finding a set of candidate specifications that are all consistent with $\mathcal{S}$. (A formula is consistent with the sample, as defined in Section 3.1, if it is satisfied by every trace from $\mathcal{P}$ and by no traces from $\mathcal{N}$.) Moreover, we want to bias the search for candidate specifications using the natural language description d. The problem we solve is the following: given a world w, a sample $\mathcal{S}$, and natural language description d, create a set of candidate specifications C such that

a) all specifications from C are consistent with the sample $\mathcal{S}$

b) $\psi \in$ C maximizes $\lambda(\psi, d)$, an idealized cost function capturing the connection between the specification and the natural language description.

Encoding Samples in SMT

As a first step, for a given sample $\mathcal{S}$ and integer hyperparameters δ and m, we show how finding a set of m formulas of size up to δ that are all consistent with $\mathcal{S}$ can be encoded as an SMT problem.

The encoding builds upon the encoding from Section 3.1. There, we were interested in finding an LTL formula of a fixed size n, consistent with the sample. In this section, we relax the fixed-size requirement, and replace it with maximum size, $\delta \in \mathbb{N}$. (This will allow us to trade-off two desirable properties: the specification simplicity and its similarity to the natural language description.) To this end, we introduce the integer variable $\Delta \in \mathbb{N}$, whose value shall be determined by the solver, with the restriction $\Delta \leq \delta$.

Optimization Modulo Theories For our synthesis procedure, we require the candidate formulas to not only satisfy the constraints imposed by $\Phi_{\delta}^{\mathcal{S}}$, but also to find candidates that *optimize* certain cost functions. For this, we use optimization modulo theories. Recall that an *optimization modulo theories* problem [34] consists of the following components: a set of hard constraints in a background logic (containing Boolean satisfiability but also other theories); a set of soft constraints, each with a cost; and a set of cost functions. In any model, the hard constraints must be met. The soft constraints induce an additional cost function: each constraint may or may not be met, and the cost function adds up the weights of all constraints that are not met. The goal is to find a model that satisfies the hard constraints, and is optimal w.r.t. all the cost functions. Since there are multiple cost functions, an optimal solution is chosen from the Pareto front of solutions. A model is on the Pareto front if there is no model that performs better along all cost functions.

Natural Language Cost Function Our cost function comes from the natural language description. Because the function λ is an idealized function and is not readily available, we have to approximate it. Provided with a rich dataset, one could learn its approximation, as was done by Beltagy et al. [27]. Another option is using natural language processing tools to recover the syntactic structure of the description and connect it to the structure of the formula, as done by Lignos et al. [148].

Rich datasets are not commonly available and mapping linguistic structure to a formula is not helpful if there is only a weak signal present in the natural language description. Therefore, in this work, we approximate $\lambda(\psi, d)$ by

$$L(\psi, d) := \sum_{\psi' \in subf(\psi)} h(lab(\psi'), d)$$

where the label of a formula, $lab(\psi) \in O$, is the top operator of the formula.

The approximation L uses a simple *hints* function h which assigns a score to each label of the subformula based on the natural language description d. Our synthesis method considers function h to be a black box that could be implemented in different ways. In our implementation of the robot simulation world, we define h by $h(o, d) = \frac{overlaps(o,d)}{|o|+|d|}$, which intuitively measures how much overlap there is between the natural language description and the operators and variables of the formula. We are using WordNet [167] lemmas and their synonyms to compute h. Intuitively, for the natural language command $d =$ `take one red item from 7,4` from the motivating example, a propositional variable q referring to red items will have a higher value $h(q, d)$ than the one referring to e.g. blue items.

We add two classes of soft constraints to the hard constraints captured by the formula $\Phi_{\delta}^{\mathcal{S}}$. First, for all $o \in O$, we add a soft constraint $\bigvee_{i \leq \delta} x_{i,o}$ with a weight $h(o, d)$. This constraint suggests that the operator o should occur somewhere in the formula (hence, disjunction) and the strength of the suggestion is $h(o, d)$. Second, in order to avoid maximizing L by adding as many operators as possible, we additionally prefer smaller formulas. We use the variable Δ, which captures the size of the found formula, and add an objective to minimize the value of Δ. The updated formula, combining hard and soft constraints, is named $\Phi_{\delta}^{\mathcal{S},L}$.

Solving the Constraint With this encoding, we query an optimization modulo theories solver to give us m models for $\Phi_{\delta}^{\mathcal{S},L}$, from which we extract the candidate specifications, each of maximum size δ. We iteratively query the solver for a solution, blocking the returned solutions before the next iteration. Note that blocking is syntactic: it is possible that two different, but semantically equivalent formulas could be found (e.g., $\mathsf{F}\, q \,\mathsf{U}\, q$ and $\mathsf{F}\, q$). The candidates are ranked implicitly by the order in which they are found (our optimization constraints prefer candidates that match the natural language description better). However, in order to determine the correct specification from the set of candidates, we have to interact with the user, as described next.

Generating Distinguishing World Traces

In order to narrow down the set of candidates C to a single candidate, LTLTALK iteratively offers (visual) examples consisting of an initial world and a trace in it for the user to judge. The world and the trace are generated in such a way that the user's verdict eliminates some of the candidates from C.

Given the two best ranked candidates, ψ_1 and ψ_2, fluents $\mathcal{F}$, actions $\mathcal{A}$ and the constraints $\varphi_{\mathcal{F}}$ and $\varphi_{\mathcal{A}}$, we aim to find an initial world φ_{init} and a world trace that satisfies one constraint but not the other, i.e., $(\psi_1 \land \neg\psi_2) \lor (\neg\psi_1 \land \psi_2)$. Simply finding a satisfying trace can be done in a standard way, e.g. by a language non-emptiness check for the intersection automaton, or by iterative SAT solving [32]. However, in this case, we need to pay attention to two more conditions:

- the found trace must be a world trace, i.e., it has to satisfy the fluent and action constraints $\varphi_{\mathcal{F}}$ and $\varphi_{\mathcal{A}}$,
- in order to make the user's decision about the shown trace effortless, the created world must be simple and the trace short.

We formalize these requirements as another multi-objective optimization modulo linear arithmetic problem and encode it in an optimizing SMT solver. Assuming the SAT formula that encodes a satisfaction of the difference specification for the trace of length ℓ_τ, we add the following constraints and objectives to it. First, we add the constraints $\varphi_{\mathcal{A}}$ and $\varphi_{\mathcal{F}}$. Then, in order to keep the traces short, we add the objective to minimize ℓ_τ. Finally, we define an integer variable c that captures the complexity of the world by counting the number of fluents that are true in φ_{init} and we require this variable to be minimized. A model for the final formula gives us a world and a trace that will distinguish between the two specification candidates.

3.3.4 Grammar-based Generalization of Learnt Specifications

Natural language commands provided by the user serve two purposes. They are used to prune the candidate specifications in Section 3.3.3, but also to expand the lexicon of the system for later use.

The interactive specification synthesis technique from the previous section produces as output a mapping from a natural language utterance to an LTL specification. However, such a mapping is not robust to parameterization. For example, if a user has run the synthesis procedure to teach the system *pick one red from (7,4)*, it is unreasonable that the system

```
Spec → RobotState | Spec or Spec | Spec and Spec | not Spec |  Spec until
    Spec | eventually Spec | Spec before Spec | Spec; Spec
RobotState → pick QItm at Loc | robot at Loc
Loc   → (Num, Num) | dry | wall | kitchen | bathroom | living room //
    Location attributes

// Item attributes:
QItm   → Quant FItm
Quant  → Univ | Num
Univ   → every | all
FItm   → item | Fltr item
Fltr   → Prop | Fltr and Fltr |
Prop   →  Color |  Shape
Color  → red | blue | green | yellow | ...
Shape  → triangle | square | circle | ...
```

FIGURE 3.15: Core language syntax. Reserved constants and variable names are in italic.

requires them to run the synthesis procedure once again if they wish to *pick one triangle from (6,3)*. In this section, we describe a grammar-based generalization procedure that complements the LTL synthesis procedure from Section 3.3.3 by synthesizing parameterized specifications, thus improving the overall usability of the system.

We first describe our domain-specific core language that maps unambiguously to LTL and which is the starting point of the grammar expansion procedure that we explain next.

The Core Specification Language

The grammar expansion starts with a domain-specific *core* language for specifying tasks in the block world. We call it a core language to emphasize that it will be expanded through the interaction with users. The grammar of the core specification language is shown in Figure 3.15. The basic actions (moving and picking) can be combined using Boolean and temporal connectives.

Note that LTL is contained in the core language. The grammar category `RobotState` corresponds to propositional variables. The keywords **`eventually`**, **`until`**, and **`before`** correspond to the temporal operators, and **`or`** and **`not`** to Boolean operators. We use the semi-colon (;) to denote the chaining of two specifications.

The reason we need a core language is to distinguish the different concepts in the blocks world (positions, items, color, shape, etc.) into separate lexical categories. This enables the grammar expansion procedure to generalize a natural language utterance and induce new production rules. At the level of LTL, the categories are all encoded with propositional variables and syntactic generalization is not possible.

Grammar Expansion

Through interaction with users, the core grammar can be expanded using the *naturalization* process [223]. Naturalization takes a pair of a natural language description and the corresponding formal specification as its input and produces a grammar rule that generalizes from that pair. In LTLTALK, the input for naturalization comes either from the process of interactive synthesis (Section 3.3.3) or by the user giving a different name to an already successfully used specification (or a chain of specifications).

The task of expanding the grammar starts with a natural language description d, the corresponding formal specification s, and the probabilistic parser model p. While s must be fully parsable using the current grammar's production rules, only some parts of d may be parsable. Using this vocabulary, for the example from Section 3.3.1, d = *take one red item from 7,4* and s = **`eventually`** `pick` *`one red`* `item at (7,4)`. In order to induce new rules, the system identifies *matches*—parsable spans appearing in both d and s. In our example, those are *`red`* (category `Prop`), *`one`* (category `Quant`), *`one red`* `item` (category `QItm`), `item` (category `FItm`), and $(7,4)$ (category `Loc`). A set of non-overlapping matches is called a *packing* and is the basis for generating new grammar rules. Examples of packing are {*`red`*, `item` } or {*`one`*}, whereas { *`one red`* `item`, *`one`*} is not a packing because its elements are overlapping.

New grammar rules are introduced through *simple packing*, *best packing*, and *alignment*. (There can be more than one rule introduced per one description-specification pair.) Simple packing considers pre-defined primitive categories for matching (such as colors, shapes, or numbers). The primitive matches in the above example are numbers one, 7, and 4 (category `Num`), and color red (category `Color`). Therefore, a new rule is added to the grammar:

`Spec` → `take Num Color item from (Num, Num)` ≡ **`eventually`** `pick Num Color item from Num Num`

We use the symbol ≡ to connect the right-hand side of the induced rule to the core-language expression with the same semantics. From now on, LTLTALK will understand commands such as *take 2 yellow item from (1,3)*.

Best packing considers maximal packings, i.e., those that would become overlapping by adding any other match, and chooses the packing that scores the best under the model p. The best scoring maximal packing for our example results in the rule

`Stmt` → `take QItm from Loc` ≡ **`eventually`** `pick QItm from Loc.`

Note that this rule is more general than the one generated from the simple packing; with this rule in the grammar, LTLTALK will in the future understand commands such as *take every triangle item from kitchen*. However, it is not the case that best packing is a better method than simple packing: because of its eagerness to generalize as much as possible, it sometimes produces non-desirable new rules.

The third method, alignment, is applicable when the language description d and the specification s are almost identical: the non-identical parts are mapped to each other as synonyms. For more details on grammar learning, see the work of Wang et al. [223].

The added production rules become first-class citizens of the grammar and can be combined with the core grammar rules freely. This enables LTLTALK to learn complex tasks through a combination of grammar-based naturalization and interactive synthesis. For instance, the command *take every triangle item from (4,3) or eventually robot at dry* is parsable immediately after the interactive synthesis of our working example.

LTLTALK can tolerate a number of wrong production rules being introduced in the process of grammar expansion (either by a wrong generalization or by a user's mistake). The model p assigns a score to each derivation. Every time LTLTALK parses a user's command, it offers a ranked list of candidate executions for the user to pick from. The user's choice is then used to update the model p. Thus, an undesirable production rule will be voted down and, in effect, excluded from the grammar.

3.3.5 Evaluation

The LTLTALK implementation consists of three independent modules:

- a front-end module, which shows the simulated world to users and enables them to give examples for their commands (written in JavaScript);
- an interactive synthesis module, which implements the algorithms for synthesizing specifications from a natural language description and a single example, and for generating distinguishing examples, as described in Section 3.3.3 (written in Python). This module uses Z3 [63], an SMT solver that supports optimization modulo theories solving;
- an expanding formal language module, described in Section 3.3.4, implemented on top of the interactive version of the SEMPRE semantic parser toolkit [223] (written in Java). This module additionally uses a thin NLP layer for lemmatization and number normalization [161].

In this section, we first evaluate the interactive synthesis algorithm in terms of performance and its ability to recover the intended specification. Then, we demonstrate how the grammar-based generalization complements interactive synthesis and allows LTLTALK to scale further.

Experimental Setup

We run all the experiments on a machine with four Intel Core i5-4590 CPUs at 3.3 GHz with 15 GB of RAM.

LTL Fragment The exact operators and the set of propositional variables are relevant for the performance of the interactive synthesis algorithm. Ideally, one should use a language that is expressive enough to cover all interesting robot tasks in the blocks world, but is as small as possible. We explicitly use derived LTL operators, as using them makes specifications more compact. Concretely, alongside operators of the propositional logic, we use derived temporal operators $\{\mathsf{F}, \mathsf{U}, \mathsf{B}\}$. Operators F and G are as defined in Section 2.1. The operator B ("before") is defined by $\varphi \,\mathsf{B}\, \psi \equiv \mathsf{F}(\varphi \wedge \mathsf{F}\,\psi)$.

TABLE 3.7: Natural language descriptions and target specifications used for evaluation

task id	natural language description	specification	spec. size
t1	*step into water and then visit (6,4)*	$\neg at(dry)\,\mathsf{B}\,at(6,1)$	4
t2	*reach (4,1), but remain dry in the process*	$at(dry)\,\mathsf{U}\,at(4,1)$	3
t3	*bring one green circle from (7,4) to (3,4)*	$picked(1, green, circle, (7,4))\,\mathsf{B}\,at(3,4)$	3
t4	*take all green from (7,4) to water*	$picked(every, green, *, (7,4))\,\mathsf{B}\,\neg at(dry)$	4
t5	*pick two square items from (4,0)*	$\mathsf{F}(picked(2, *, square, (4,0)))$	2
t6	*get one triangle from (4,0)* *and then one item of any kind from (11,1)*	$picked(1, *, triangle, (4,0)\,\mathsf{B}\,picked(1, *, *, (11,1))$	3
t7	*get one item from (1,2) and one from (3,1)*	$\mathsf{F}(picked(1, *, *, (1,2))) \wedge \mathsf{F}(picked(1, *, *, (3,1))$	5
t8	*get one green and one blue item from (7,4)*	$\mathsf{F}(picked(1, green, *, (7,4))) \wedge \mathsf{F}(picked(1, blue, *, (7,4))$	5
t9	*reach (5,4) by only going through the water*	$\neg at(dry)\,\mathsf{U}\,at(5,4)$	4
t10	*first get one red item from (7,4)* *and afterwards one green item from (10,8)*	$picked(1, red, *, (7,4))\,\mathsf{B}\,picked(1, green, *, (10,8))$	3

For a world of size 14 × 10, propositional variables are defined to determine the robot's location, the quality of the robot's location (whether it is dry or wet), and the quantitative and qualitative properties of the picked items (what combination of color and shape properties, how many items, in numerical and *some* vs. *all* terms). We forbid the nesting of multiple temporal operators beyond the B operator, whose definition involves nested temporal operators.

Benchmarks We created ten tasks, shown in Table 3.7. Each task consists of a natural language task description, a world definition, and one example trace, which are given to LTLTALK, as well as the target specification, which is used to verify whether LTLTALK succeeded in synthesizing it. The specifications differ in their size (the size of their syntax DAG), ranging from two to five. As discussed in Section 3.3.4, these formulas should be viewed as language building blocks that can be combined in more complex formulas using naturalization.

Interactive Synthesis Evaluation

We assess the different parts of the proposed interactive synthesis algorithm in detail. In particular, we are interested in the following research questions:

RQ1 Does the algorithm synthesize specifications with only a few interaction rounds?

RQ2 Do natural language description and user interaction contribute to successful synthesis?

RQ3 How sensitive is the synthesis algorithm to parameter choices?

RQ4 How does our synthesis algorithm compare to enumerative synthesis?

RQ1: Ability to Synthesize Specifications We run our synthesis algorithm on the ten tasks from Table 3.7, setting the maximum depth of the synthesized formula to $\delta = 4$ and the number of initial candidates generated to $m = 5$. Table 3.8 summarizes the results of this experiment. Because different runs of the optimizing solver may produce different models, we execute our algorithm five times per task and report how often out of the five runs the target specification was found, as well as the average number of interaction rounds.

TABLE 3.8: Performance of interactive synthesis for maximum size $\delta = 4$ and $n = 5$ initial candidates

task id	formula found	overall waiting time [s]	number of interactions
t1	yes (4/5)	4.5	2.8
t2	yes (5/5)	3.6	1.4
t3	yes (5/5)	6.28	2.6
t4	yes (4/5)	12.7	3.4
t5	yes (5/5)	14.4	2.4
t6	yes (5/5)	13.7	2
t7	no (0/5)	6.5	3
t8	no (0/5)	19.1	1.8
t9	yes (5/5)	3.2	2
t10	yes (5/5)	12.0	2.2

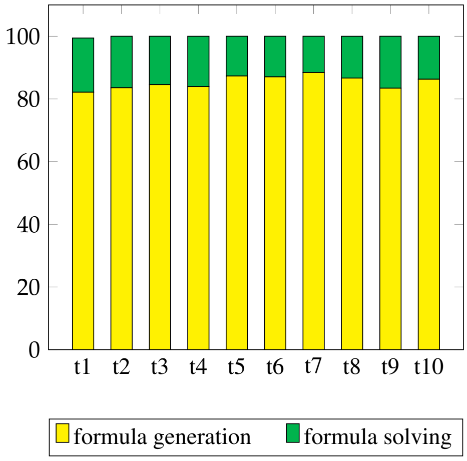
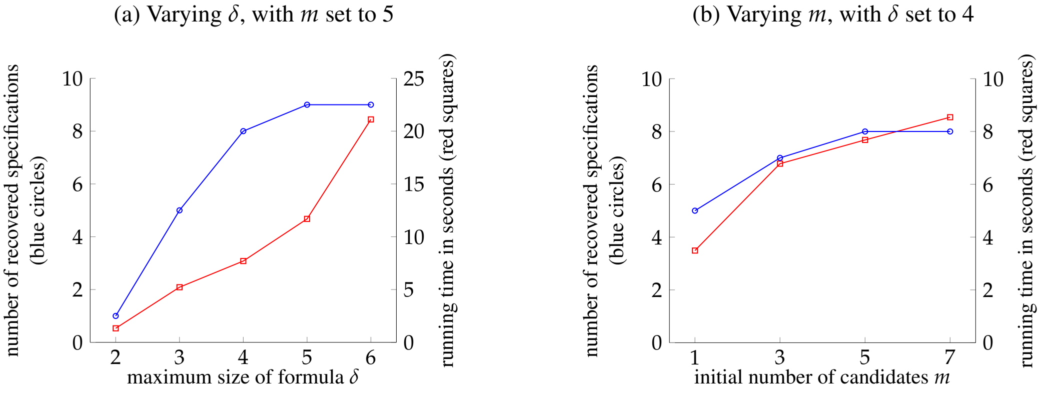


FIGURE 3.16: Relative times needed to generate and to solve formula $\Phi_\delta^{\mathcal{S},L}$

In the majority of cases, LTLTALK found the target formula. It failed to do so for the tasks t7 and t8, for which the target formula has size 5, which is beyond our default limit of $\delta = 4$. The average number of interactions was 2.4, which we consider to be small enough to not overly burden the user.

We measure the user's overall waiting time—the time spent waiting for the system to devise initial candidates as well as to devise distinguishing examples. The average overall waiting time for the user is 9.6 seconds. We note that most of this waiting time is spent on creating the initial candidate set, and in particular, on generating the propositional formula $\Phi_\delta^{\mathcal{S},L}$. Figure 3.16 shows the relative times needed to generate $\Phi_\delta^{\mathcal{S},L}$ compared to the time needed to solve it (i.e., to find the satisfying assignment). We observe that it takes much longer to generate a formula than to solve it. The average times needed for solving formulas from Table 3.8 and for generating distinguishing examples are both below one second. While we do not consider the waiting time to be prohibitively long, we note that a more optimized implementation of the formula generation (e.g. in C++ instead of in Python) would likely reduce the overall running time of our synthesis.


(a) Varying δ, with m set to 5

(b) Varying m, with δ set to 4

FIGURE 3.17: Number of recovered specifications and running time for different choices of hyperparameters m and δ

RQ2: Role of NL and User Interaction We further examine the role that the natural language hints play. We ran the same experiment but supplied no hints information when generating the formula $\Phi_{\delta}^{\mathcal{S},L}$. Without the hints, the intended formula was recovered in only 9 out of 50 cases. Clearly, increasing the size of the initial candidates set m would allow the algorithm to recover the formula without the natural language hints eventually. This would, however, increase the candidate generation time and the number of interactions needed with the user, impacting the user experience.

We next evaluate the role interaction plays in the overall synthesis procedure. In particular, we evaluate whether the initial candidates generation is already enough, i.e., whether it could stand on its own, without additional interaction with the user. Among the 50 runs, the target specification was the top-ranked candidate in 15 cases, it was among the top 3 candidates in 34 cases, and among the top 5 in 38 cases. With non-expert users in mind, we consider only the top-ranked candidate as a successful trial. This means that without interaction, the specification would be successfully recovered in only 15 experiments. We conclude that the interaction is an important part of the synthesis procedure, as it manages to narrow down the set of candidate specifications to a single, correct specification.

RQ3: Varying the Hyperparameters Our synthesis algorithm has two hyperparameters: the maximum size of the specification δ and the number of initial candidates m. The larger these two parameters are, the longer the synthesis runs, but also the more tasks can be successfully solved. We conducted a number of preliminary experiments, varying δ and m. Figure 3.17 shows the number of successfully recovered specifications and the running time for different values of δ (Figure 3.17a) and for different values of m (Figure 3.17b).

For $\delta \geq 5$, there is a possibility to recover all the specifications from benchmarks, since the target specifications do not exceed this depth. However, for $\delta = 5$ or $\delta = 6$, the algorithm successfully recovered only 9 out of 10 specifications. The task t7 (which has size 5) was not recovered, even though the similar task t8 was. A closer inspection reveals that the synthesized specification for t7 was $\mathit{picked}(1,*,*,(1,2))\,\mathsf{B}\,\mathit{picked}(1,*,*,(3,1))$. This shorter specification was consistent with a single example (as were the other candidate specifications), and the natural language description was not able to bias the search towards

TABLE 3.9: Performance of interactive synthesis for maximum size $\delta = 4$ and $n = 5$ initial candidates *without using the no excessive trace principle for generation of negative examples*

task id	formula found	overall waiting time [s]	number of interactions
t1	no (2/5)	3.5	2.8
t2	no (2/5)	2.7	2.6
t3	no (1/5)	2.1	3.4
t4	no (0/5)	4.0	3.0
t5	yes (5/5)	2.3	2.6
t6	no (1/5)	3.9	2.4
t7	no (0/5)	3.5	2.2
t8	no (0/5)	2.5	2
t9	no (0/5)	3.7	3.2
t10	yes (5/5)	4.6	3.2

the correct specification. A different example trace or a better NL similarity function would be necessary to recover the intended specification.

So far, we have insisted on the user providing only a single example. However, depending on the application, it may be the case that asking for few examples would not harm the user experience. To see what the effects of going beyond a single example are, we re-ran the interactive synthesis on the same benchmark set, but with two examples provided for each specification instead of just one. The pre-interaction results improve: the target specification was the top-ranked candidate in 22 cases, it was in the top 3 in 35 cases and in top 5 in 39 cases. Additionally, the specification for the task *t7* is now successfully recovered with $\delta = 5$. Thus, by using more initial examples, one might choose a smaller number of initial candidates, at the expense of higher user effort.

As described in Section 3.3.3, our algorithm relies on two principles for devising a set of negative examples: the principle of no excessive trace and the principle of no excessive effort. We evaluated what the relative contributions of those principles to the performance of the algorithm are by running the same experiment, but giving up on using one of the principles at a time. When negative examples were generated without the no excessive trace principle, the results were significantly worse (as can be seen in Table 3.9). When the principle of no excessive effort was not used, the system was not able to recover *t*8 for the specification depth $\delta = 5$ (which it was able to do using the principle). On the other hand, not using the no excessive effort principle results in reducing overall waiting time for the user to below 10 seconds for each of the tasks.

Finally, we performed an additional experiment, whereby we encoded the generation of distinguishing worlds eagerly, already in the formula $\Phi_{\delta}^{S,L}$. This has potential advantages over the current, lazy generation, approach: no waiting time for the user between interactions, never finding two semantically equivalent specifications (for traces up to a fixed length), and a possibility to take a smarter strategy in choosing which disambiguating example to present the first. An obvious drawback is that the eager generation increases the size of $\Phi_{\delta}^{S,L}$. Unfortunately, we found the eager generation approach not to scale well: the average waiting time for initial candidates rose to 70 seconds, with 10 solver timeouts (for a timeout set to 120 seconds). We thus stick to the lazy generation.

RQ4: Comparison with Enumerative Synthesis A natural baseline for generating initial candidates is enumerative synthesis [7]. We ran two variants of the algorithm: enumerating expressions from the language in the order of size and enumerating expressions from the language biased by their similarity to the natural language description. We enumerated over the same language that was used in the previous experiments, in particular using only those propositional variables that appeared in the example trace. To compare with our approach, we let the enumeration run for 20 seconds (the total time our algorithm at most takes). We then check whether the target specification is contained in the set of specifications consistent with the examples (the user-provided positive and generated negative examples), i.e., whether the system could possibly find it through disambiguating interactions with the user.

The size-based enumeration successfully found four specifications: $t1$, $t2$, $t5$, and $t9$. While $t5$ is the smallest size specification, we observe that none of $t1$, $t2$, and $t9$ contains a picking action. If in an example some items are picked up, there are many more propositional variables to consider, which means that the size of the language to enumerate increases, making the enumeration more difficult. For example, picking a red circle from $(1,2)$ that contains only that item makes the following propositions true: $picked(1, red, *, (1,2))$, $picked(1, *, circle, (1,2))$, $picked(1, red, circle, (1,2))$, $picked(every, red, *, (1,2))$, $picked(every, *, circle, (1,2))$, $picked(every, red, circle, (1,2)))$.

After adding the natural language bias, enumeration is able to successfully find five specifications: $t1$, $t2$, $t3$, $t9$, and $t10$. We conclude that our SMT-based method for deriving initial candidates is superior to enumeration.

Case Study for Grammar-Based Generalization

We illustrate our grammar-based generalization from Section 3.3.4 using case studies. First, we consider the examples from Table 3.7 and show that LTLTALK learns not only the individual demonstrated tasks (t1 - t10), but a whole class of tasks obtained by generalization. Then we show how grammar-based generalization can be used for providing complex but repetitive specifications in an easy way. Finally, we show how the generalization helps with tasks from robotic dialog systems literature.

Generalization of Interactive Synthesis Tasks

Table 3.10 shows induced specification expressions alongside examples of newly parsable commands for a subset of examples from Table 3.7. When a derived production rule does not have `Spec` as its left-hand side, we add to the table a core-language expression with which it shares the semantics, and denote this by the symbol $\equiv$.

TABLE 3.10: Example expressions expanding the core grammar

NL description	generated grammar rule	newly parsable commands
(t1) step into water and then visit $(6,4)$		
	step into water and then visit (Num, Num) step into water and then visit Loc	step into water and then visit $(1,1)$ step into water and then visit kitchen ...
(t2) reach (4,1), but remain dry in the process		
	reach (Num, Num) but remain *dry* in the process reach Loc but remain Loc in the process	reach (10,8) but remain dry in the process ...
(t3) bring one green circle from $(7,4)$ to $(5,4)$		
	bring *one* Color Shape from (Num, Num) to (Num, Num) bring Quant Prop Prop from Loc to Loc	bring every blue square from kitchen to bathroom ...
(t4) take all green from $(7,4)$ to water		
	take *every* color from (Num, Num) to water take Quant Prop from Loc to water take ≡pick	take two squares from (1,3) to water take one item at (2,4) ...
(t10) first get one red item from $(7,4)$ and afterwards one green item from $(10,8)$		
	first get Quant Prop item from Loc and afterwards Quant Prop item from Loc	first get all triangle items from (2,1) and afterwards two red items from kitchen
	from ≡at ...	take one item from kitchen ...

Let us first consider task t4. Its natural language description is *take all green from* $(7,4)$ *to water* and the learnt core language specification is pick *every green* item at (7,4) **before not** *robot* at *dry*. After generalization, the command *take two squares from (1,3) to water* is immediately understood. Namely, using the best packing method, the expression take Quant Prop from Loc to water is added to the language. Using the alignment method, LTLTALK furthermore learnt that *take* can be used interchangeably with pick, and thus an expressions such as *take one item at (2,4)* is now a part of the language.

Note that the system has not generalized over the word *water*. The reason is that *water* is not even partially parsable—it does not appear in the grammar (and the properties of wet tiles can only be expressed by negating the grammatical entity *dry*). Hence, which generalizations are produced also depends on how the grammar is designed.

The generalization from task t3 enables the system to understand expressions such as *bring every blue square from kitchen to bathroom*. However, the same generalization allows for some nonsensical expressions to become parsable: e.g., *bring three yellow red from kitchen to bathroom*. As discussed in Section 3.3.4, this is a property of the best packing technique: it generalizes aggressively, at a price of introducing spurious expressions to the language.

Complex Specifications The naturalization technique proves to be especially useful for scenarios in which the robot does many structurally similar tasks over and over again, e.g., a hospital robot taking different drugs to different patients. In our abstract blocks world, consider a task of distributing items of different colors from a selected location (for instance, (2,3)) to the four corners of the world. Conceptually, the task is very simple: the robot first needs to take all red items to the lower-left corner, then it needs to take all blue items to the upper-left corner, then it needs to take all green items to the lower-right corner, and finally, it needs to take all yellow items to the upper-right corner. Its specification in the core language, shown in Figure 3.18, is complicated and repetitive. Furthermore, the specification is of depth 15. Our interactive synthesis algorithm is not able to uncover such a long specification.

```
pick every red item at (2,3) before robot at (1,1)

and

pick every blue item at (2,3) before robot at (1,10)

and

pick every green item at (2,3) before robot at (14,1)

and

pick every yellow item at (2,3) before robot at (14,10)
```

FIGURE 3.18: Sorting items based on colors using the core language

The solution comes from the modularity of the target specification. Using interactive synthesis, the user can first specify the command *all red from 2,3 to 1,1*. LTLTALK generalizes from the specification and now knows the meaning of any command that corresponds to `Quant Prop from Loc to Loc`. Now the specification from Figure 3.18 can be written by using the conceptual idea of the task, saying *all red from 2,3 to 1,1 and all blue from 2,3 to 1,10 and all green from 2,3 to 15,1 and all yellow from 2,3 to 14,10* (while this specification is also fairly long, it assumes much less knowledge about the core language: only that individual specifications can be connected by the operator `and`).

Finally, this specification can be renamed into *distribute colors from 2,3*. In the future, then, the user could do the same for any other location using a single command.

Dialog Systems Tasks Now we turn our attention to tasks inspired by three studies of robotic dialog systems [133, 214, 188]. The idea of a dialog system is that the robot can ask its user clarifying questions to understand the task. The tasks are of two kinds: *navigation* and *delivery*. Here, we show how LTLTALK is able to do typical navigation and delivery tasks, but also go beyond those.

The tasks of interest are navigation (e.g., *go to the bathroom*) [133, 214], delivery (e.g., *bring a spoon from the living room to the kitchen*) [214], and a complex combination of atomic tasks (e.g., *go to the bathroom and then go to the living room*) [188].

Assume first that a user of LTLTALK tasks the robot to *eventually be at the restroom*. After the user provides the example (and potentially judges the examples provided by LTLTALK), the system learns that *restroom* is the same as *bathroom* (using *alignment*). Similarly, after the user demonstrates the command *go to the kitchen*, LTLTALK knows that the meaning of this phrase corresponds to **`eventually`** `at the kitchen` (using *simple packing*).

It is not only learning of a new phrase that happened, but also generalization. To illustrate, the expression *go to the restroom* is now a part of the language of LTLTALK, as it learnt the meaning of `go to Loc` as well as the synonymity of *bathroom* and *restroom*.

The generalization happens over different grammar categories. For instance, assuming that the user demonstrates the command *go to the kitchen and then go to the living room*, LTLTALK will introduce a new rule to its grammar, encoding that two categories `Spec` can be connected with the connector *and then*, thus forming a new production rule `Spec` → `Spec and then Spec` ≡ `Spec` **`before`** `Spec` (using *best packing*). This allows for generalizing to kinds of actions different than navigation only, e.g., *pick one red item at the kitchen and then go to the living room* is now a part of the LTLTALK's language.

These examples show how LTLTALK's approach can achieve the functionality of the existing dialog systems. These systems ask the user for particular parts of the action, which makes them dependent on knowing the exact structure of possible specifications. LTLTALK, on the other hand, gets the clarification from the user by way of demonstration, therefore providing a more usable solution.

3.3.6 Related work

We presented LTLTALK, a *natural language robotic interface* using interactive specification synthesis *from examples* and *natural language descriptions*. In all three areas (NL communication with robots, synthesis from natural language, and synthesis from examples), recent years have brought a lot of interesting developments. In this section, we relate LTLTALK to the works in these topics.

Natural Language Interfaces for Robotics

In an attempt to provide a more natural specification language for robotics but keep the precision of a formal language, Kress-Gazit et al. [137] propose a controlled, natural looking-language that matches a fragment of LTL. SLURP [148] uses NLP techniques to map the linguistic structure of a command to a fragment of LTL. LTLTALK shares the usage of LTL as its underlying expressive and precise specification language, but adapts the formal language to the users' style through interaction with them.

The grammar expansion technique that we use originates from Voxelurn [223], an NL instruction system for a 3D-blocks building world. There, a user is expected to provide a formal specification for a natural language description that was not understood by the system. LTLTALK removes that burden from its users and enables them to provide an example instead (which is then turned into a formal specification using the interactive synthesis algorithm).

The early work in the robotic dialog systems [166, 133] puts forward the idea of understanding a specification through interactions with the user. They learn the new expressions but do not generalize. Generalization of the learnt expressions is achieved by Thomason et al. [214] using the induction of a CCG grammar from the semantic parsing framework SPF [16]. The commands are limited to delivery and navigation tasks, in contrast to LTLTALK's ability to handle temporal specifications with different propositional variables.

Synthesis from Natural Language

NLyze [101] proposes a natural language interface to spreadsheet programming where a natural language utterance is mapped to a DSL using a semantic parser; the users resolve ambiguity by selecting the correct spreadsheet macro from a ranked list of candidates. SQLizer [232] proposes an NL interface for SQL query programming. (The users are assumed to be unaware of the underlying tables' structure, so providing examples is not an option.) A semantic parser is used as a front-end to generate sketches of SQL queries, and ambiguity is resolved by program repair. Similarly, NaLIR [144] uses an NL interface for SQL programming, and lets users select correct queries. Compared to these systems, LTLTALK *actively* asks for user input by showing new, disambiguating, worlds.

Many successful NL to DSL systems use large datasets of natural language descriptions and corresponding formal language commands to train the synthesizer's model [22, 27, 66, 190]. Instead of requiring a large training set upfront, which is challenging to obtain, LTLTALK learns over time and adapts to idioms and user-specific ways of expressing commands.

Synthesis from Examples

Example-based synthesis techniques have developed several strategies to resolve the inherent ambiguity. Similarly to LTLTALK, Scythe [222, 221], a system for learning SQL queries from input-output examples, uses active querying for disambiguation. Beyond the different application domains, LTLTALK integrates the semantic parser more tightly, as the core language is gradually extended based on user interaction.

Several techniques have been developed to reduce user effort in example-based synthesis. SketchAX [8] leverages properties such as invariance to input perturbations to generate an additional set of examples. Drachsler-Cohen et al. [70] propose a system that interacts with a user by generating abstract examples, which represent a set of concrete examples and thus reduces the rounds of interaction. FlashProg [163] allows users to inspect generated programs in a compact form or asks clarifying questions based on existing test data. Peleg et al. [187] develop a Granular Interaction Model, which on the one hand shows evaluation results at intermediate steps in a program, and on the other hand asks users to inspect generated programs and to provide feedback on parts of it. These techniques do not directly apply to our domain of LTL specification. In particular, unlike the programs in the target domains of these works (integer and bitvector manipulating programs, string processing), LTL specifications are challenging to inspect by users.

Synthesis from Natural Language and Examples

Combining example-based specifications with natural language descriptions has also been explored previously to reduce the number of examples that a user has to provide. Common with LTLTALK, these techniques use the natural language description of a task to bias the search for the correct program towards more likely candidates, but other details differ which make them not immediately applicable to synthesizing LTL specifications.

Manshadi et al. [162] use a dependency parser to bias the search for regular expressions, but the underlying synthesis relies on version space algebra. REGEL [53] uses a semantic parser to obtain the basic scaffolding for the target regular expression from the user's NL description, and then completes the expression using programming-by-example. Nye et al. [181] use a neural network instead of a semantic parser to generate sketches that are filled using enumerative synthesis. MARS [54] encodes synthesis as a MAX-SMT problem, similar to LTLTALK, but trains a neural network to provide the weights. LTLTALK avoids the training phase by adapting over time using an extensible semantic parser. Finally, Raza et al. [197] use a semantic parser to split the natural language description into smaller parts for which the user can separately provide examples. LTLTALK's generalization procedure similarly allows to combine smaller tasks into more complex ones, but the composition happens in the semantic parser itself and does not require re-synthesizing the low-level tasks for new combinations. Neither of the above-mentioned techniques uses active disambiguation through user interaction.

3.4 Conclusion

In this chapter, we described three methods for learning temporal specifications from examples. The first method tackles the problem of learning a minimal LTL formula from a sample consisting of both positive and negative examples. It does so by casting the problem as an instance of propositional satisfiability and using the power of existing SAT solvers.

For many use-cases, however, we typically get only positive examples. Thus, we ventured into specifications from positive examples only. Because the problem of learning from positive examples is ill-defined, we introduced a notion of tightness to define it properly. We recognized UVW as a class of automata suitable for learning under that definition. The results show that this method can successfully work if a large example set is provided to it.

Finally, our third method brings the learning of specifications closer to the real world: we develop LTLTALK, an interface for commanding robots. LTLTALK enables its non-expert users to interact with the robots that expect commands given in LTL.

LTLTALK uses a combination of example-driven synthesis and grammar-based naturalization that allows "one-shot" learning and generalization of LTL specifications from natural language utterances and examples. Because it cannot assume access to a large example set, LTLTALK builds upon the work from the first method (Section 3.1). It resolves the problem of obtaining negative examples (the problem that motivated the second method) by taking advantage of domain-specific knowledge.

Like all techniques based on programming by natural language and examples provided by users, LTLTALK is ultimately incomplete. The candidate generation may fail to find the correct specification. This can, for example, happen if the hyperparameter m (number of initial candidates) is too small. Another example is if no distinguishing worlds are found. That can happen either because the candidate formulas are semantically equivalent or because the bound on lengths of world traces is too small. The result in both of those cases is that the grammar will contain a wrong production rule.

As shown in our experimental evaluation, such incompleteness is rare in practice. Moreover, the advantage of combining with naturalization is that the probabilistic grammar model is robust to a few wrong production rules.

Chapter 4

Planning with Multiple Speculative Initial States

In a multi-robot system that supports asynchronous requests (such as the one described in Chapter 2), a natural question arises: how to plan for the robots already executing some previous task? While in practice the question may get resolved by additional assumptions (e.g., the planning is much faster than the robots' execution), in this chapter, we aim to explore the problem in the most general setup, domain-independent planning.

4.1 Introduction

After plan execution has begun, the need to replan can arise for many reasons. Perhaps the agent's goal has changed [171] or additional goals have become known [28]. Perhaps the environment, or the agent's perception of it, has changed, possibly giving rise to better ways of achieving the agent's goal [51]. Or maybe the original plan was constructed in haste, and now that execution is underway, the agent has the leisure to try to find a better plan to switch to [149]. Regardless of the cause, if the agent is using a planning technique such as forward state-space search, replanning necessitates choosing an initial state for the search.

This small detail, often glossed over in previous work on replanning, raises a vexing problem: with execution underway, this initial state must be one far enough along the current plan that the agent will not encounter it until the replanning process has finished (because otherwise, transitioning to the new plan is not possible). But conservatively choosing an initial state that is too far along the old plan risks inefficiency: the agent's actions will not reflect the new information until the state is reached, possibly causing it to miss opportunities.

The most common solution in current systems appears to be to always choose a transition point that is a fixed time ahead in the future, either as part of system design [97, 164] or through an estimate for the replanning time [149, 200]. But this is at odds with the purpose of automated, domain-independent planning, which is to enable an agent to handle a variety of problems and situations with a single algorithm. In this chapter, we introduce a principled approach to the problem of choosing the initial state for online replanning, which we call the Multiple Initial State Technique (MIST). In MIST, we integrate the choice of the state into the search itself, as only the search has the proper information about the necessary trade-offs.

The structure of MIST resembles that of the classic search algorithm A* [104]. In MIST, however, the open list is initialized with multiple potential initial states rather than just one. These states represent speculations on the state in which the agent might be once the search is done. Open nodes are prioritized by their estimated goal achievement time, taking into account both the makespan of the resulting plan and an estimate of when the planning process itself will finish. In this way, MIST reasons online, during the search, about which initial state is most promising to explore.

In MIST, once the execution goes past a state, that state is no longer viable as an initial state. We also present a variant of MIST that allows the planner to consider initial states that may have already been passed but that are still reachable. For both variants, we prove that the first solution MIST finds is better than any other one it might find by continuing its search. (The claim is conditioned on the suitable properties of heuristic functions, which we will describe later in the chapter.)

To assess the effectiveness of MIST in a concrete yet domain-independent way, we implement it in the Fast Downward planner [108] and extend a set of benchmarks from the International Planning Competition (IPC) to our replanning setting. We find that MIST yields better agent behavior, in the sense of achieving its goals more quickly, than either using a fixed constant or trying to predict replanning time in advance.

4.2 Problem Definition

Although MIST applies to any setting in which planning occurs in the context of ongoing execution, for concreteness, we investigate MIST in classical planning with online goal arrival. The focus is on the moment when a new task arrives while the agent is already executing its *current plan*. This is equivalent to a series of arriving jobs assuming that each task arrives after the previous planning phase is done.

Background

We consider the *finite-domain representation (FDR)* [20, 107] for classical planning tasks:

Definition 4.1. A **planning task** is a tuple (V, A, c, s_0, Γ):

- V is a finite set of *state variables*, each with a finite domain of possible values,
- A is a finite set of *actions*. Each action a is a pair (pre_a, eff_a) of propositional formulas called *preconditions* and *effects*,
- $c\colon A \to \mathbb{R}$ is a function assigning cost to every action,
- s_0 is the *initial state* (complete variable assignment),
- Γ is a propositional formula over the set of states called *goal*.

We denote the set of all complete variable assignments, or *states*, by S. If a state $s \in S$ satisfies a propositional formula Φ, we say that s is *compliant* with Φ and write $s \models \Phi$. A goal Γ is accomplished in a state compliant with it. An action $a \in A$ can only be applied to a state $s \in S$ if $s \models pre_a$. The effect of the action a is described by the propositional formula over state variables, eff_a. The outcome of this application is state denoted by $post_s(a)$.

A solution (*plan*) to a planning task is a sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ with the overall cost $C(\pi) = \sum_{i=1}^{n} c(a_i)$ leading from the initial state s_0 to a state that is compliant with the goal Γ.

In what follows, the action costs c will be interpreted as the time required to execute them.

Continual Online Planning Tasks

We express the notion of continual online planning in the classical planning setting by extending tasks with a second goal, assumed to arrive during the execution of the plan for the original goal.

Definition 4.2. A **continual online planning (COP) task** is a tuple $(V, A, c, \Gamma_{\text{old}}, \Gamma_{\text{new}}, s_0, \pi_{s_0,\Gamma_{\text{old}}})$:

- states V, actions A, and cost function c are as before,
- Γ_{old} is a propositional formula called the *old goal*,
- Γ_{new} is a propositional formula called the *new goal*,
- s_0 is the agent's state at the time when Γ_{new} appeared,
- $\pi_{s_0,\Gamma_{\text{old}}} = \langle a_1 a_2 \ldots a_n \rangle$ is a sequence of actions, taking the agent from the state s_0 to a state compliant with the old goal Γ_{old} (the agent's *current plan*).

Note that, while s_0 is not necessarily the first state of the agent's original plan for Γ_{old}, we disregard all states before the arrival of Γ_{new} and focus on the suffix of the original plan that can still be changed. We assume the actions to be non-interruptible: if Γ_{new} appeared during the execution of an action, s_0 is the state at the end of the action.

We assume for simplicity that Γ_{old} and Γ_{new} are not in direct conflict, i.e., $\Gamma_{\text{old}} \wedge \Gamma_{\text{new}}$ is satisfiable. A solution to a COP task is a plan π consisting of two parts: a prefix of $\pi_{s_0,\Gamma_{\text{old}}}$ and the newly planned extension. There must exist $1 \leq j \leq n$ such that $\pi = \langle a_1 a_2 \ldots a_j b_1 \ldots b_m \rangle$. If the extension $b_1 \ldots b_m$ is not empty, we call the state in which b_1 will be applied the *deviation state*. The state to which the plan π takes the agent must be compliant with $\Gamma_{\text{old}} \wedge \Gamma_{\text{new}}$. A solution is said to be optimal if it minimizes the total planning and execution time, i.e., the time from the arrival of the new job Γ_{new} to the end of the execution of π.

In order to achieve Γ_{new}, it may be useful to deviate from $\pi_{s_0,\Gamma_{\text{old}}}$ early. For example, such a situation occurs if a warehouse robot is moving back to its home base to deliver a package for Γ_{old}, while Γ_{new} requires picking up another package close to where the robot was when Γ_{new} arrived.

Algorithm 5 MIST

```
Input: $s_0$, $\Gamma_{\text{old}}$, $h$, $\Gamma_{\text{new}}$, $\pi_{s_0,\Gamma_{\text{old}}}$, $R$
 1: $\gamma \leftarrow 0$; closed $\leftarrow \emptyset$
 2: open $\leftarrow \{(r,r) \mid r \in R\}$
 3: while open $\neq \emptyset$ do
 4:     $(s, ref_s) \leftarrow \arg\min_{(t,ref_t)\in\text{open}} f(t, ref_t, \gamma)$
 5:     if $(s, ref_s)$ is not consistent with the state of the execution then
 6:         discard $(s, ref_s)$
 7:     if $s \models \Gamma_{\text{old}} \wedge \Gamma_{\text{new}}$ then
 8:         return path to s
 9:     closed $\leftarrow$ closed $\cup \{(s, ref_s)\}$
10:     $\gamma \leftarrow \gamma + 1$
11:     for $t \in$ successors(s) do
12:         $ref_t \leftarrow ref_s$
13:         if $((t, ref_t) \notin (\text{open} \cup \text{closed})$ or $g_{ref_t}(t) < g^{old}_{ref_t}(t)$ then
14:             open $\leftarrow$ open $\cup \{(t, ref_t)\}$
15: return fail
```

Observe that such indirect conflicts can be characterized by the degree to which $\pi_{s_0,\Gamma_{\text{old}}}$ is useful for the combined goal $\Gamma_{\text{old}} \wedge \Gamma_{\text{new}}$. At one extreme end, $\pi_{s_0,\Gamma_{\text{old}}}$ is a prefix of an optimal plan for $\Gamma_{\text{old}} \wedge \Gamma_{\text{new}}$. In this case, it is best to complete the execution of $\pi_{s_0,\Gamma_{\text{old}}}$. At the other extreme end, $\pi_{s_0,\Gamma_{\text{old}}}$ is not useful at all, i.e., no optimal plan for $\Gamma_{\text{old}} \wedge \Gamma_{\text{new}}$ shares a non-empty prefix with $\pi_{s_0,\Gamma_{\text{old}}}$. In that case, it is best to stop $\pi_{s_0,\Gamma_{\text{old}}}$ immediately. However, it is not known upfront where between the two extremes to position the planner. Our aim is to address this trade-off automatically and in full generality.

4.3 The Multiple Initial State Technique (MIST)

Algorithm 5 shows the pseudocode of the MIST algorithm. Its structure closely resembles that of A*, with a few important modifications (marked by the red text color). In contrast to A*, MIST uses a set of potential initial states that we call *reference states*. The reference states are sampled from the plan $\pi_{s_0,\Gamma_{\text{old}}}$ towards the old goal, and are passed to the algorithm as a parameter R. These starting states are different “guesses” on where the agent will be when planning finishes, and they are the potential deviation states for the overall plan.

The open list (*open*) is initialized using the reference states (Line 2). Each element of *open* is a pair of the search node and its corresponding reference state (a candidate for the deviation state). Each newly created search node inherits the reference state of its parent (Line 12).

As in A*, nodes in the open list are expanded in a best-first order according to f, and put into the closed list afterwards. When a node is expanded, its successors are inserted into the open list if they are new, or replaced if they are reached with a lower g-value than before (Line 13). As time passes and execution of the current plan progresses, search nodes whose corresponding plan deviates from the executed actions become invalid and are discarded (Line 6). Line 7 checks the termination condition, reflecting Definition 4.2.

The most important difference to A* is the open list ordering function f. Given a state s, its reference state ref_s, and the number of expansions made by the algorithm so far γ (s_0 is treated as a default parameter), MIST uses $f(s, ref_s, \gamma) = C(\pi_{s_0,ref_s}) + g_{ref_s}(s) + h(s) + os(s, ref_s, \gamma)$. The first part, $C(\pi_{s_0,ref_s})$, represents the time required to move from the initial state s_0 to the reference state that was used to reach s along the execution path. The second part, $g_{ref_s}(s) + h(s)$, is the same as in A*: following our interpretation of action costs as their durations, it combines the time needed to get from the reference state ref_s to the state under consideration s with the estimated time to reach the goal from s. The third part, the overshot function os, models a state becoming irrelevant ($f = \infty$) as the execution passes through it before the estimated end of planning. It is defined as:

$$os(s, ref_s, \gamma) = \begin{cases} 0 & \text{if } (\gamma + \eta(s)) \cdot t_{exp} \leq C(\pi_{s_0,ref_s}) \\ \infty & \text{otherwise} \end{cases}$$

where $\eta(s)$ estimates the remaining number of expansions until a plan is found, t_{exp} is the time per expansion to translate expansions to execution time, and $C(\pi_{s_0,ref_s})$ is the time to reach ref_s along $\pi_{s_0,\Gamma_{\text{old}}}$. We will discuss below how to obtain such estimates by adapting prior work [212, 44].

Having γ as an argument for f has an interesting consequence: it now matters when the function f is evaluated for the relative order of the nodes in *open*. In practice, we do not re-evaluate f on all the nodes in the open list each time the best element is retrieved (Line 4). Instead, we approximate the value of the f-function by keeping the search nodes sorted only by $g + h$, but separately for each reference state. Subsequently, we do the full evaluation only to select the next reference state for which a node should be expanded using the nodes with minimal $g + h$ for each reference state. This approximation is justified by the fact that a changed value of γ affects all the nodes corresponding to the same reference state equally. The loss of precision comes from disregarding differences in η. In preliminary experiments, we also tried a recomputation strategy in exponentially increasing intervals as used in Bugsy [44] and found the difference in solution quality to be negligible compared to this approximation strategy.

Another practical consideration is that we can safely prune a search node (s, ref_s) if s is a reference state itself that is reached by $\pi_{s_0,\Gamma_{\text{old}}}$ after passing through ref_s and $C(\pi_{s_0,ref_s}) + g_{ref_s}(s) \geq C(\pi_{s_0,s})$ holds.

Theoretical Properties

A* is guaranteed to find an optimal solution, provided that the heuristic function is admissible (and nodes can be reopened). A similar guarantee cannot be given for MIST. The essential difference between the two settings (and thus necessarily between the two algorithms) is that in an offline setting, the exploration of the state space during the planning phase comes at no cost. On the other hand, in an online setting, exploring a part of the search space that is not going to be used in the solution can decrease the quality of the final plan, since that time was not used efficiently.

Consider a situation where the only optimal plan deviates at the reference state r, and expanding all the nodes on that plan takes exactly the time that the agent needs to reach r. Expanding any other node will make MIST miss this path. Hence, unless the heuristic functions h and η were perfect, there is no guarantee that MIST will find an optimal solution.

With optimality out of reach, we can prove a simpler property: the stopping criterion of MIST is a correct one. MIST stops the search as soon as the first state compliant with both of its goals is found, which raises the question of whether there is some trade-off between continuing the search and the quality of the solution. We show that continuing the search can not possibly result in a better plan, assuming the heuristic functions h and η are admissible.

We will use $h^*(s)$ to denote the true value of the cost to reach the goal from s, and $\eta^*(s)$ to denote the number of expansions from s to the end of planning. Following the same notation style, $f^*(s, \mathit{refs}_s, \gamma_s)$ and $os^*(s, \mathit{refs}_s, \gamma_s)$ denote the functions f respectively os calculated using $h^*(s)$ and $\eta^*(s)$ instead of the heuristics h and η. We are using the notation $\gamma = \gamma_s$ to indicate that the third argument of the f-function is the value of γ when the node s was explored.

Theorem 4.1. Let h be admissible with respect to planned execution time and η admissible with respect to the number of expansions. Let $\sigma_1 = s_0, s_1, \ldots, s_i, p_1, p_2 \ldots p_m$ be the sequence of states corresponding to the first solution π_1 found by MIST (with the deviation state s_i). Assume the algorithm continued the search and found another solution with the state sequence $\sigma_2 = s_0, s_1, \ldots, s_j, q_1, q_2, \ldots, q_n$ (with the deviation state s_j). Then $f^*(p_m, s_i, \gamma_{p_m}) \leq f^*(q_n, s_j, \gamma_{q_n})$.

Proof.

$$
\begin{aligned}
& f^*(p_m, s_i, \gamma_{p_m}) \\
& = C(\pi_{s_0,s_i}) + g_{s_i}(p_m) + os^*(p_m, s_i, \gamma_{p_m}) \\
& = f(p_m, s_i, \gamma_{p_m}) && (4.1)\\
& \leq f(q_l, s_j, \gamma_{p_m}) && (4.2)\\
& = C(\pi_{s_0,s_j}) + g_{s_j}(q_l) + h(q_l) + os(q_l, s_j, \gamma_{p_m}) \\
& \leq C(\pi_{s_0,s_j}) + g_{s_j}(q_l) + h^*(q_l) + os^*(q_l, s_j, \gamma_{p_m}) && (4.3)\\
& \leq C(\pi_{s_0,s_j}) + g_{s_j}(q_l) + h^*(q_l) + os^*(q_l, s_j, \gamma_{q_l}) && (4.4)\\
& \leq C(\pi_{s_0,s_j}) + g_{s_j}(q_n) + os^*(q_n, s_j, \gamma_{q_n}) && (4.5)\\
& = f^*(q_n, s_j, \gamma_{q_n})
\end{aligned}
$$

The true cost of the solution π_1 is $f^*(p_m, s_i, \gamma_{p_m}) = C(\pi_{s_0,s_i}) + g_{s_i}(p_m) + os^*(p_m, s_i, \gamma_{p_m})$. Following the search structure of MIST, at some point we chose to expand p_m. Since p_m is the last state on the path and our heuristic functions are admissible, the true cost f^* is equal to the cost function f (Equality 4.1). Inequality 4.2 comes from our choice of the state p_m over some state q_l from σ_2.

From the admissibility of h and os (which follows from the admissibility of η) with respect to h^* and os^*, we get Inequality 4.3. The value of os^* for some later point γ_{q_l} when exploring q_l, would be greater or equal to the value for the same state q_l and the reference state s_j at time point γ_{p_m} (Inequality 4.4). This follows from the definition of os^*, which is monotonic when observed as a function of its third argument (time passed so far).

Finally, Inequality 4.5 comes from (a) $\gamma_{q_n} + \eta^*(q_n) = \gamma_{q_l} + \eta^*(q_l)$ and (b) $g_{s_j}(q_l) + h^*(q_l) \leq g_{s_j}(q_n)$. Both sides of equality (a) are equal to the overall planning time. Inequality (b) comes from the fact that $h^*(q_l)$ is the smallest cost from q_l to a goal state. Since that goal state is not necessarily q_n, we get the inequality. □

4.4 MIST for Recoverable Tasks

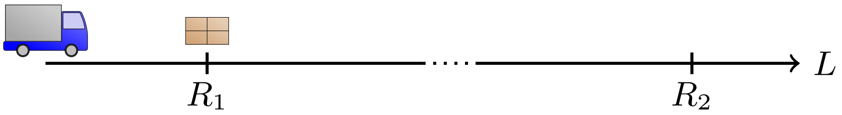


FIGURE 4.1: A truck executing a plan when a new goal appears.

Depending on how the reference states are selected from the initially computed plan, situations may arise in which MIST's replanning strategy incurs a large cost. Consider the example from Figure 4.1. The truck is heading to location L when a new goal, *to pick up and deliver a package to* L appears. It may happen that the planner is not able to find a plan before the truck has already passed the first reference state R_1, which is closest to the package. MIST requires the truck to continue following the current plan until the next reference state R_2 and only then turn back to pick up the package. Depending on the distance between R_1 and R_2, this can cause an arbitrarily large cost penalty compared to turning around at some intermediate location. Even if reference states are frequent, MIST can still incur a large penalty if it repeatedly misses its predicted planning time.

An engineering solution addressing these issues could be to pause the execution for some time if planning is expected to finish soon. Here, we instead suggest a variant of MIST, that we call MIST$_{\text{rec}}$, as an algorithmic solution for a class of COP tasks that satisfy a form of recoverability. This notion of recoverability requires that, for every sequence of actions of the current plan, there is a sequence of actions that reverses the effects of it. In the following definition, we will slightly abuse the previously introduced notation, denoting by $post_s(\alpha)$ a state of the agent after applying *a sequence of actions* α beginning with in state s.

Definition 4.3. For a COP task $(V, A, c, \Gamma_{\text{old}}, \Gamma_{\text{new}}, s_0, \pi_{s_0,\Gamma_{\text{old}}})$, let $s_0, \ldots, s_n$ be the state sequence induced by $\pi_{s_0,\Gamma_{\text{old}}}$. We denote the action subsequence taking the agent from s_i to s_j, with $i < j$, by $\vec{\alpha}_{i,j}$. The task is said to be **recoverable** if, for every such $\vec{\alpha}_{i,j}$, there exists an action sequence $\overleftarrow{\alpha}_{i,j}$ such that $post_{s_j}(\overleftarrow{\alpha}_{i,j})$ is compliant with every formula that appears as a precondition or goal in s_i.

This restriction gives the planner more room for error in the prediction of when re-planning will terminate: when a reference state s is passed, the planner may still finish computing a plan for s, because there is a recovery sequence that brings the agent to a state where that plan is applicable.

Many COP applications are naturally recoverable. Examples include warehouse logistics and various types of manufacturing problems. Furthermore, recoverability relates to known notions of invertibility and undoability, and prior work has established methods to test these properties [112, 60]. Our solution considers the recoverability sequence to be known. In our experiments, we focus on domains where each action has an immediate inverse action of the same cost.

In order to adapt Algorithm 5, two things need to be changed. First, the states of $\pi_{s_0,\Gamma_{\text{old}}}$ that the agent's execution already went through should not be discarded, because the agent can still come back to them. Second, we have to change the definition of the function f to reflect the possibility of returning to a reference state.

Reminding ourselves, f is defined as a sum of three parts: $C(\pi_{s_0,ref_s}) + [g_{ref_s}(s) + h(s)] + os(s, ref_s, \gamma)$, with os being either 0 (if planning is estimated to finish before the execution reaches ref_s), or infinity (otherwise). We adapt os to consider the possibility of using the recovery sequence to move back to ref_s, redefining it as

$$os(s, ref_s, \gamma) = C(\vec{\alpha}) + C(\overleftarrow{\alpha}) + \max((\gamma + \eta(s)) \cdot t_{exp} - C(\pi_{s_0,\Gamma_{\text{old}}}), 0)$$

Like before, os evaluates to 0 if planning is expected to finish in time. Otherwise, it now describes the additional execution time incurred by moving past the reference state ($\vec{\alpha}$) and back ($\overleftarrow{\alpha}$). If planning takes longer than total the execution of $\pi_{s_0,\Gamma_{\text{old}}}$, then the agent will additionally have to wait in s_n, the last state of $\pi_{s_0,\Gamma_{\text{old}}}$ (described by the last term of os).

In Figure 4.2, we can see a visual explanation for the modified overshot function os: The dashed red bar denotes the time needed to execute the current plan leading to s_n, with $s_n \models \Gamma_{\text{old}}$. The green bar labeled by $\gamma \cdot t_{\text{exp}}$ is the time spent planning so far, and the dashed green bar ($\eta(s) \cdot t_{\text{exp}}$) shows the estimation on when the planning will finish. In the illustrated example, the planning time is estimated to exceed the time when the selected reference state ref_s is reached. The os function describes this additional execution time, plus the time it takes to go back to ref_s.

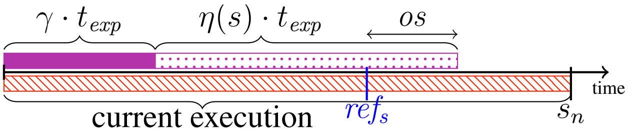


FIGURE 4.2: Overshot.

To prove that MIST$_{\text{rec}}$ retains the same property of the first found solution being the best possible, we need to introduce another mild assumption. Namely, our recovery paths have to be *well-behaved*: applying a sequence of actions and its recovery sequence is of a lower cost than doing the same for any extension of that sequence. With this assumption, we are able to prove hat the overshot function remains admissible.

Lemma 4.1. Assume that η is admissible, and that for a path α that is a prefix of a path α' it holds that $C(\vec{\alpha}) + C(\overleftarrow{\alpha}) \leq C(\vec{\alpha'}) + C(\overleftarrow{\alpha'})$ (well-behaved recovery paths). Then os is admissible, i.e. $os(s, ref_s, \gamma) \leq os^*(s, ref_s, \gamma)$.

Proof. Let $\vec{\alpha}$ be the subsequence of actions on $\pi_{s_0,\Gamma_{\mathrm{old}}}$ taking the agent from the reference state ref_s to the state in which it would be at time $\gamma + \eta(s) \cdot t_{exp}$. Similarly, let $\vec{\alpha^*}$ be the subsequence of actions taking the agent to the state at time $\gamma + \eta^*(s) \cdot t_{exp}$. Since $\eta \leq \eta^*$, $\vec{\alpha}$ must be a subsequence of $\vec{\alpha^*}$. With the assumption of well-behaved recovery paths, we have $\mathsf{C}(\vec{\alpha}) + \mathsf{C}(\overleftarrow{\alpha}) \leq \mathsf{C}(\vec{\alpha^*}) + \mathsf{C}(\overleftarrow{\alpha^*})$, and thus $os \leq os^*$. □

Since os is again admissible, the proof of Theorem 4.1 also applies to MIST$_{\mathrm{rec}}$.

Consider again the logistics example from Figure 4.1. While MIST must continue along the plan until reaching R_2, MIST$_{\mathrm{rec}}$ will be able to finish computing a new plan from the (already passed) reference state R_1, and can turn around without having to go to R_2 first.

4.5 Evaluation

We implemented MIST in Fast Downward [108]. As explained earlier, in our implementation we use a standard A* open list for each reference state, using the MIST extensions to the f-function only to select the open list to be used for the next expansion to avoid having to re-sort the open list. For MIST$_{\mathrm{rec}}$, our implementation assumes that each action has an inverse action with the same cost.

Like Bugsy, we estimate the remaining number of expansions as $\eta = \mathit{delay} * d$ [44, 69], where *delay* is the (moving) average number of expansions between inserting a node into the open list and expanding it, and d is an estimation of the remaining steps to the goal (like h, but ignoring action costs). The expansion delay is important to counteract *search vacillation* [69], i.e., the search fluctuating between different solution paths and, in our case, potentially different reference states. For d, we do not use the distance estimate of the current state, but instead the minimal distance of any evaluated state that corresponds to the considered reference state to make the planning time estimations more stable.

Our key performance metric is the *goal achievement time* (GAT), i.e. overall time for online planning and execution, measured from the moment when the new goals appear. We measure this time as a number of expansions to make the experiments more robust. Action costs are translated into execution time using an instance-specific factor from cost to expansions (we give more details in the next subsection).

In all experiments, the popular FF heuristic [113] is used to guide the search. For the expansion delay, we use a moving average over the last 100 expansions. The experiments were run on a cluster of 2.20 GHz Intel Xeon E5-2660 CPUs. The time and memory limits were set to 30 minutes and 4 GB, respectively.

Benchmarks

We adapted the IPC domains Elevators, Logistics, Rovers, Transport, and VisitAll to our setting, as representatives of applications where (i) goals are of an additive nature and there are no conflicts between them and (ii) all action sequences $\vec{\alpha}$ have a recovery sequence $\overleftarrow{\alpha}$ with the same cost. Criterion (ii) is required for our implementation of MIST$_{\mathrm{rec}}$. We furthermore experiment with Tidybot, which we adapted to satisfy (ii). In Tidybot, there are cases where

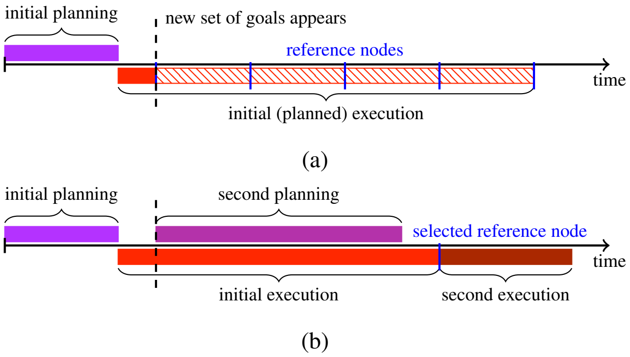


FIGURE 4.3: Generating benchmark instances.

objects are placed behind each other, and the robot cannot reach behind the object in the front. We added an un-finish action to ensure recoverability. However, previously finished objects must be picked up again in these cases, necessitating the planner to falsify and re-achieve previously achieved goals. We assume actions to be non-interruptible.

In some of our benchmark domains, recovery sequences with lower cost can exist. For example, there could be shortcuts to inverting the agent's movements. In the Rovers domain, photos would not need to be "un-taken". In such cases, our implementation of MIST$_{\text{rec}}$ is pessimistic and more practical implementations may achieve lower plan costs.

The instances were adapted by splitting the set of goals in two: the first half is available in the beginning, and the other one becomes available later. The second set of goals is scheduled to appear during the execution of the first computed plan to obtain interesting instances. The process is illustrated in Figure 4.3. The initially computed plan is being executed as a new job arrives (Figure 4.3a). Here, the planner considers 5 reference states as potential initial states for the new plan. The planner has computed an updated plan that starts from the second-to-last reference state (Figure 4.3b). The initial plan is executed until that point before switching to the new plan. The goal achievement time is the time from the start of the second planning phase to the end of the overall execution.

In order to obtain interesting benchmark instances, we tried to ensure that the second planning phase starts and ends during the first planned execution. Thus, we generated the instances such that the second set of goals appears after 10% of the initial plan is executed. Furthermore, we estimated the length of the second planning phase by running the planner offline for both goals from the original start state, and used that to generate different experimental setups where the second planning phase is estimated to end at $E = 0.2, 0.3, \ldots, 0.9$ of the initially planned execution. This is achieved by adjusting the factor for the translation of the action cost to execution time, thereby changing the agent's actions duration.

Results

We compare MIST to the following baselines, which represent different traditional approaches to solving COP tasks:

- *finish*: Finish execution and plan only for the new goals.

- *stop*: Stop execution and re-plan from the current state.
- *approximate*: Approximate the duration of the re-planning phase, and use the state where the agent is expected to be at that time as the deviation state. We use the same estimation for the number of expansions as MIST, i.e. $\eta(m) = \mathit{delay} * d(m)$, using the average expansion delay from the initial planning phase and the estimated distance of the current state.
- *fixed latency*: Stop execution at a fixed point in time (we test values of $10^1, 10^2, \ldots, 10^7$ expansions for this time point). We also consider a theoretical oracle configuration that chooses the best-performing time point to stop the execution (out of the tested values) per instance.

MIST has one important parameter: the selection of the reference states. In our implementation, we set a number of reference states n_R, which are then selected in uniform intervals from the current plan. Figure 4.4 shows the goal achievement time (in number of expansions) for different values of n_R across our full benchmark set. If there are too few reference states, the algorithm does not have the best starting point for the next plan available. On the other hand, the performance also decreases if too many reference states are used, as it becomes more difficult to settle on the most promising one quickly (especially if the planning time estimation is not very accurate). Across the tested numbers of reference states, MIST chooses nodes for expansion corresponding to the reference state which is used for the solution 38% of the time on average, more for fewer reference states (55% for $n_R = 3$), and less the more reference states are used (30% for $n_R = 24$). The overall best results are obtained with $n_R = 8$ for MIST and $n_R = 9$ for MIST$_{\text{rec}}$, and we use these settings for the remaining experiments.

Figure 4.5 shows the relative goal achievement time compared to MIST for the considered algorithms for different expected end points of the second planning phase. If the planning time is very short compared to the execution time (small values of E), stopping the execution as soon as possible works well, but loses out compared to MIST if planning is non-trivial ($E > 0.2$). As expected, finishing the execution becomes better with increasing expected planning times, though MIST always performs better. The fixed latency configurations offer some interpolation between the two extremes of stopping or finishing the execution. Given the diversity of our benchmark set, a fixed latency can not accurately predict the planning time, and these configurations are outperformed by MIST. The approximation baseline also

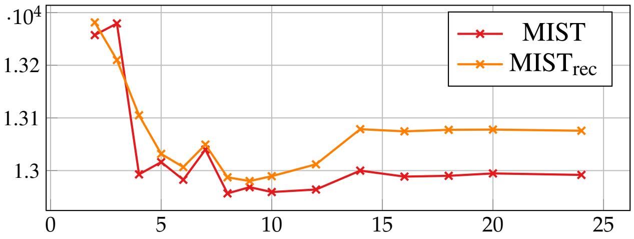


FIGURE 4.4: GAT as geometric mean over all instances (Y-axis) for MIST with different numbers of reference states (X-axis).

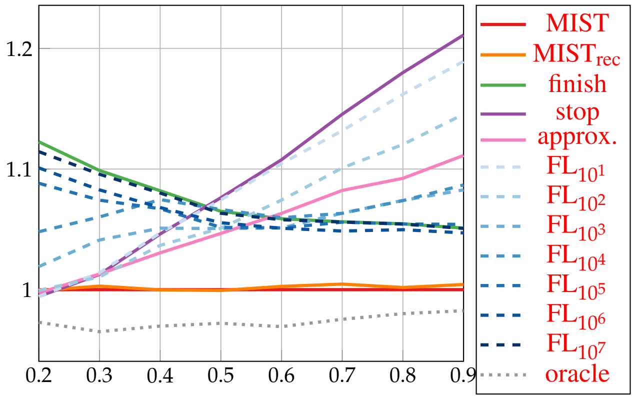


FIGURE 4.5: GAT as geometric mean over all instances relative to MIST (Y-axis) for $E = 0.2, 0.3, \ldots, 0.9$ (X - axis).

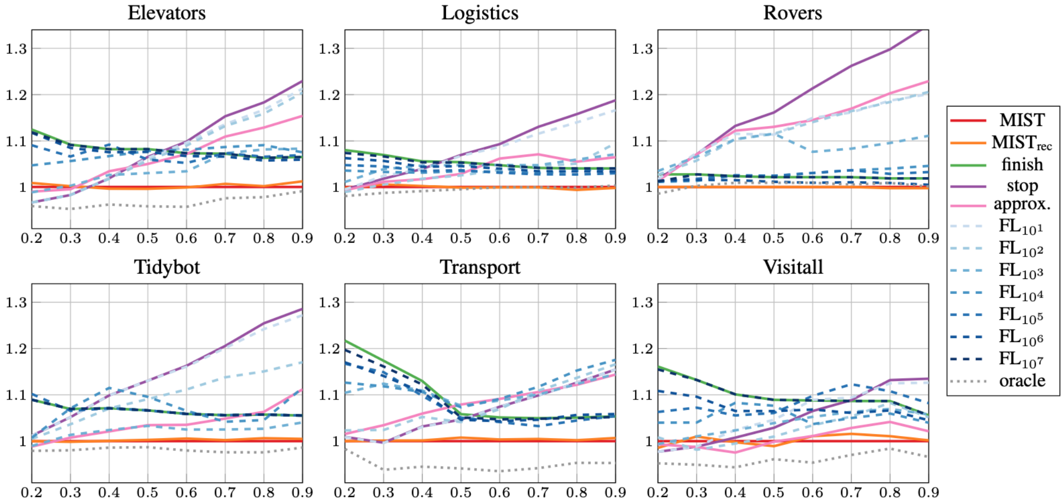


FIGURE 4.6: GAT relative to MIST (Y-axis) for $E = 0.2, 0.3, \ldots, 0.9$ (X-axis).

works well for short planning times, but tends to overestimate the duration of the re-planning phase. On average, MIST reduces the goal achievement time by 8.6% compared to stopping and re-planning immediately, by 6.8% compared to finishing the planned execution, and by 5.1% compared to approximating the re-planning time.

Figure 4.6 gives more insight into the individual domains. The observations from the overall results hold across most domains, with minor exceptions. On VisitAll, the *approximate* baseline comes very close to MIST on average, beating it for some values of E. This can can be attributed to the offline planning time estimation being more accurate. While that also helps MIST to select the correct reference state to expand towards more frequently (46% of the time compared to 35% on other domains), MIST can still suffer from the added overhead. In the Rovers domain, MIST and MIST$_{\text{rec}}$ outperform all competitors for all values of E, and may even beat the oracle (which can be inaccurate if the best deviation state is between two of the considered time points). Both *stop* and *approximate* perform particularly poorly in that domain, with up to 35% respectively 23% worse goal achievement time compared to MIST when considering large expected planning times. Conversely, *finish* comes close to MIST, which indicates that interrupting the execution while re-planning is particularly costly in that domain.

MIST$_{\text{rec}}$ and MIST exhibit similar performance. This suggests two conclusions. First, the way we generate testing instances averages out the edge cases in which MIST$_{\text{rec}}$ significantly outperforms MIST (illustrated in Figure 4.1). Second, even though MIST$_{\text{rec}}$ does not prune reference nodes, it is able to effectively focus its search effort just as well as MIST.

4.6 Related Work

One example of the need for replanning in an industrial context is on-line goal arrival in manufacturing. Ruml et al. [200] address printing systems, where additional pages to print arrive asynchronously. They use a hand-tuned constant to represent the maximum planning time and predict when the new plan can start; if this bound is exceeded, the planning process is interrupted and re-tried later when the printing plant is assumed to be less busy and easier to plan for. Similarly, but in the context of robot navigation, Likhachev et al. [149] use a predefined constant to predict the time when a new plan will be found during an anytime search, allowing them to select the state at which the new plan will take control from the currently executing plan. Systems designed for spacecraft control commonly incorporate replanning, often by fixing a latency within which the planner must react [97, 164], thereby fixing the initial state for the revised plan. In our work, we aim for a more principled and flexible approach that eschews hand-tuned constants.

In some situations, it can be assumed that replanning time will be negligible [132, 156, 155]. This again allows selection of the initial state in advance, in particular stopping the execution as early as possible when a new plan can be assumed to be available. We consider that approach as one of our baselines.

The setting in which replanning is driven by new goals appearing online has been called *continual online planning (COP)* [28, 142, 43]. In that work, COP tasks are defined as Markov Decision Processes where additional goals arrive stochastically at each time step and so world states are extended with the current goal set. In the work presented in this chapter, we leave aside any assumptions about goal arrival distributions and focus on the fundamental question of how the plan search for the combination of an old goal and a new goal can be formulated in the context of a currently executing plan.

In what is perhaps the closest work to ours, Cashmore et al. [49] consider replanning in the context of plan execution. They use a feature of PDDL2.2, called timed initial literals (TILs), to represent non-interruptible commitments of the currently executing plan and an *ad hoc* extension of the domain model, called a *bail-out action generator*, to provide alternatives for interruptible actions. Their approach always discards the remainder of the original plan, thus avoiding the need to choose an initial state but assuming that the system can safely idle if necessary during replanning. Furthermore, their approach is inherently tied to PDDL planning, focusing on the intricacies of interrupting individual actions, rather than a generic solution for state-space search, as described in this chapter.

The concern of time passing during planning has been addressed by Cashmore et al. [50], although without the complexity of a concurrently executing plan. Their main focus lies on finding plans that respect external temporal constraints, such as a deadline for arriving at a bus stop. An associated line of more theoretical work [208, 57] takes as its objective to maximize plan feasibility, leaving aside the notion of overall goal achievement time that we address here. There are also several time-aware heuristic search methods, such as Bugsy [44] and Deadline-Aware Search [69]. However, these methods do not consider concurrent plan execution and merely search from a given static initial state.

4.7 Conclusion

In Chapter 2, we encountered the problem of serving asynchronous task requests in a multi-robot system. In the `Antlab` implementation, we solved it by simply considering only idle robots as candidates for executing new tasks. In this chapter, we examined how the active robots (the ones already executing some previous plans) could be included in the planning process. We did so in the general setting of domain-independent planning.

We proposed a technique called MIST, which addresses the problem of selecting an initial state for planning in the context of an already-executing plan. MIST plans for multiple potential initial states simultaneously. In the process of speculative planning, it reasons about its own planning time and adjusts its actions accordingly. We investigated the performance of MIST on modified planning benchmarks and found that MIST outperformed both the approximate and the fixed latency baseline.

Chapter 5

Reinforcement Learning with Non-Markovian Rewards

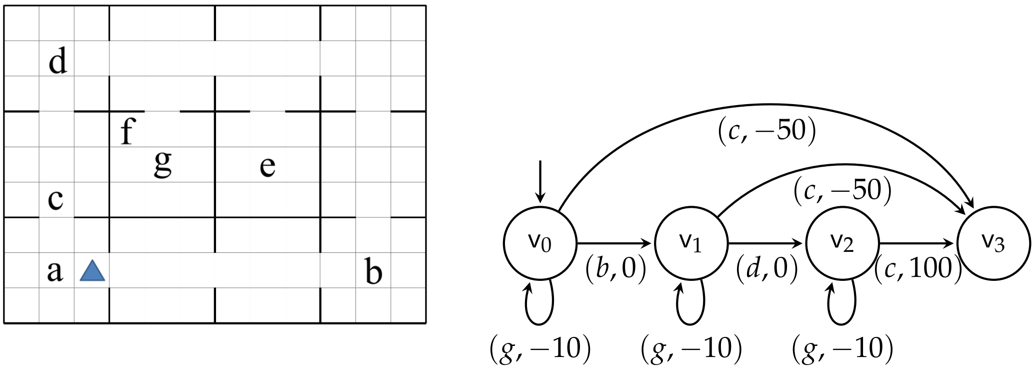
In this chapter, we will not consider specifications given explicitly as temporal logic formulas, as we did in previous chapters. Instead, we consider specifications that are given implicitly, as rewards which the agent receives during its exploration of the environment. We will still keep our focus on the *temporally extended* tasks, the ones that impose strict temporal relations between different actions of the agent.

The temporally extended tasks do not fit naturally to the model of reinforcement learning (RL), which is typically used to learn from rewards. Indeed, the RL problem is modeled so that the agent's rewards disregard the history of the agent's behavior, and they depend only on the last state and action. Prior work assumed that the user explicitly provided the temporal dependence and incorporated it into an RL algorithm. Our approach does not need such an assumption: it uses automata learning to uncover the (temporally extended) reward. We implement the approach in the algorithm called Joint Inference of Reward Machines and Policy (JIRP).

In the second part of the chapter, we explore the middle ground between the two extremes: knowing everything about the reward (as assumed by prior work) and knowing nothing about the reward (and using automata learning to uncover it). The middle ground is allowing the user to provide advice to the learning algorithm. The advice we expect to get from users is in the form of describing the behaviors that seem promising (but without quantifying or guaranteeing that an agent following the advice will indeed obtain any reward).

5.1 Introduction

Learning from rewards is typically done using the model of reinforcement learning (for a broad overview, see the classical textbook by Sutton et al. [210]). Reinforcement learning assumes the environment in which an intelligent agent operates to be modeled by a Markov Decision Process (MDP): the states of the MDP capture the relevant information about the environment, while state-action pairs are equipped with rewards that either reinforce desired or penalize undesired behaviors. In many RL tasks, however, the agent receives its reward sparsely for complex actions over a long period of time.



(a) The office world

(b) Reward machine

FIGURE 5.1: Running example

Learning an optimal policy in such settings can not be achieved using standard RL algorithms: those settings do not map naturally to MDPs as the reward does not depend on the immediate state of the environment and the chosen action but rather on the history of the actions that the agent has performed—in other words, the reward function is *non-Markovian*. A similar problem occurs under different guises: a non-Markovian dynamics of the MDP or partial observability that creates an illusion of a non-Markovian environment. We focus on the non-Markovian reward formulation of the problem and will later draw connections to the related formulations.

Clearly, a reward function is Markovian or non-Markovian only with respect to the underlying MDP, and one can augment the states of any MDP with (relevant parts of) the history to obtain an equivalent problem with a Markovian reward function. However, the exact augmentation is crucial: if done naively, the augmented state space becomes too large to be computationally tractable. To overcome this problem, finite-state machines or temporal logic formulas have been proposed to concisely capture the temporal nature of non-Markovian reward functions and make the RL task feasible [19, 127, 122, 40, 46]. We focus on a specific type of finite-state machines, called *reward machines*, which have, since proposed by Icarte et al. [122], been adopted as a general way to capture non-Markovian reward functions.

5.2 Preliminaries

In this section, we introduce the necessary notation and background on reinforcement learning and reward machines. All the notions will be illustrated using the running example of the office world shown in Figure 5.1a. The agent (its position denoted by a triangle symbol) has to first go to location b, then d, and finally to c, and receives a reward of 100 upon doing so. If it gets to c without respecting the order of going to b and d first, the agent receives a reward of -50. Finally, each time the agent visits g, it gets a reward of -10.

Decision Processes

We begin by defining Markov Decision Processes (MDPs), which model RL problems by incorporating sequential decision making with effects on received rewards and subsequent environment states.

Definition 5.1. A *labeled Markov decision process with non-Markovian rewards (MDP)* is a tuple $\mathcal{M} = (S, s_I, A, p, R, \gamma, P, L)$ consisting of a finite state space S, an agent's initial state $s_I \in S$, a finite set of actions A, and a probabilistic transition function $p\colon S \times A \times S \to [0,1]$. A reward function $R : (S \times A)^+ \times S \to \mathbb{R}$ and a discount factor $\gamma \in [0,1)$ together specify the overall payoff to the agent. Finally, a finite set P of propositional variables, and a labeling function $L : S \times A \times S \to 2^P$ determine the set of relevant high-level events that the agent senses in the environment. We define the size of $\mathcal{M}$, denoted as $|\mathcal{M}|$, to be $|S|$ (i.e., the cardinality of the set S).

Our definition differs from the "usual" modeling used in reinforcement learning (e.g., [210]) in two ways. First, the reward function is defined over the whole history, and such functions are called *non-Markovian*[1]. (A Markovian reward function $R' : S \times A \times S \to \mathbb{R}$ is commonly used in modeling RL problems and is assumed by RL algorithms.) Second, we include a set of propositions: they come from expert knowledge of what is relevant for successfully executing a task and are assumed to be available to the agent.

In our example from Figure 5.1, the state is the agent's position in the grid and the actions available to the agent are movements in the four cardinal directions. The transition function captures the small probability of slipping. The propositions are $P = \{a, b, c, d, e, f, g\}$, and the labeling function L applied to the triple (s, a, s') returns a set of propositions that are true in the state s' (e.g., for the agent that successfully moved from initial position to the left, the labeling function would return $\{a\}$). Note, however, that only labels are not enough: the full state space is still necessary in order to capture the model's dynamics.

A *policy* is a function mapping states in S to a probability distribution over actions in A. At state $s \in S$, an agent using policy π picks an action a with probability $\pi(s, a)$, and the new state s' is chosen with probability $p(s, a, s')$. A policy π and the initial state s_I together determine a stochastic process. A *trajectory* is a realization of this stochastic process: a sequence of states and actions $s_0 a_1 s_1 \ldots a_k s_k$, with $s_0 = s_I$. Its corresponding *label sequence* is $\ell_1 \ell_2 \ldots \ell_k$ where $L(s_i, a_{i+1}, s_{i+1}) = \ell_{i+1}$ for each $0 \leq i < k$. Similarly, the corresponding *reward sequence* is $r_1 r_2 \ldots r_k$, where $r_i = R(s_0 a_1 s_1 \ldots a_i s_i)$, for each $i \leq k$. The overall payoff of the agent is $\sum_i \gamma^i r_i$. We call the pair $(\lambda, \rho) := (\ell_1 \ell_2 \ldots \ell_k, r_1 r_2 \ldots r_k)$ a *trace*.

It is worth noting that not every sequence of labels is a label sequence. Some sequences simply can not be attained by the underlying MDP. We formalize this notion of attainable label sequences in the following definition.

Definition 5.2. Let $\mathcal{M} = (S, s_I, A, p, R, \gamma, P, L)$ be a labeled MDP and $m \in \mathbb{N}$ a natural number. A trajectory $\zeta = s_0 a_1 s_1 \ldots a_k s_k \in (S \times A)^* \times S$ is said to be *m-attainable* if $k \leq m$ and $p(s_{i-1}, a_i, s_i) > 0$ for each $i \in \{1, \ldots, k\}$. Moreover, a trajectory ζ is called *attainable*

[1] The process dynamics remain Markovian. Thus, we keep the name of *Markov* decision processes

if there exists an $m \in \mathbb{N}$ such that ζ is m-attainable. Analogously, we call a label sequence $\lambda = l_1 \dots l_k$ *(m-)attainable* if there exists an (m-)attainable trajectory $s_0 a_1 s_1 \dots s_{k-1} a_k s_k$ such that $l_i = L(s_{i-1}, a_i, s_i)$ for each $i \in \{1, \dots, k\}$

Reward machines

Reward machines [122] are a way to represent a non-Markovian reward function. A reward machine reads a label, responds with a reward, and moves to its next state. Technically, a reward machine is an instance of a Mealy machine [207], with a set of real numbers as its output alphabet and subsets of propositional variables (from the set P defined by the underlying MDP) as its input alphabet.

Definition 5.3. A *reward machine* $\mathsf{A} = (V, \mathsf{v}_I, 2^P, O, \delta, \sigma)$ is defined by a finite, nonempty set V of states, an initial state $\mathsf{v}_I \in V$, an input alphabet 2^P, an output alphabet $O \subseteq \mathbb{R}$, a (deterministic) transition function $\delta \colon V \times 2^P \to V$, and an output function $\sigma \colon V \times 2^P \to O$. We define the size of A, denoted as $|\mathsf{A}|$, to be $|V|$ (i.e., the cardinality of the set V).

A run of a reward machine A on a sequence of labels $\ell_1 \ell_2 \dots \ell_k \in (2^P)^*$ is a sequence $\mathsf{v}_0 (\ell_1, r_1) \mathsf{v}_1 (\ell_2, r_2) \dots \mathsf{v}_{k-1} (\ell_k, r_k) \mathsf{v}_k$ of states and label-reward pairs such that $\mathsf{v}_0 = \mathsf{v}_I$ and for all $i \in \{0, \dots, k-1\}$, we have $\delta(\mathsf{v}_i, \ell_{i+1}) = \mathsf{v}_{i+1}$ and $\sigma(\mathsf{v}_i, \ell_{i+1}) = r_{i+1}$. We write $\mathsf{A}(\ell_1 \ell_2 \dots \ell_k) = r_1 r_2 \dots r_k$ to connect the input label sequence to the sequence of rewards produced by the machine A and say that the machine A is consistent with the trace $(\ell_1 \ell_2 \dots \ell_k, r_1 r_2 \dots r_k)$

We say that a reward machine A *captures* the reward function R of an MDP if for every trajectory $s_0 a_1 s_1 \dots a_k s_k$ and the corresponding label sequence $\ell_1 \ell_2 \dots \ell_k$, the reward sequence that the agent receives equals $\mathsf{A}(\ell_1 \ell_2 \dots \ell_k)$.

A reward machine that captures the reward function of the motivating example is shown in Figure 5.1b. We note here that there can be multiple reward machines that capture the reward function of an MDP—reward machines may differ on a label sequence that does not correspond to any trajectory of an underlying MDP.

Reinforcement Learning With Reward Machines

In reinforcement learning, an agent explores the environment modeled by an MDP, receiving occasional rewards according to the underlying reward function [210]. One way to learn an optimal policy in environments such as our motivating example from Figure 5.1a is tabular Q-learning [226]. There, the value of the function $\mathrm{q}(s, a)$, which represents the expected future reward for the agent taking action a in state s, is iteratively updated. For MDPs with a Markovian reward function, q-learning converges to an optimal policy in the limit, provided that all state-action pairs are seen infinitely often [226].

The QRM algorithm [122] modifies q-learning to learn an optimal policy when the reward function is encoded by a reward machine. Algorithm 6 shows one episode of the QRM algorithm (with episode length *eplength*, learning rate α, and discount factor γ as its hyperparameters and a reward machine as its input). It maintains a set Q of q-functions,

Algorithm 6 QRM_episode

```
Input: a reward machine $(V, \mathsf{v}_I, 2^P, O, \delta, \sigma)$, a set of q-functions $Q = \{\mathrm{q}^{\mathsf{v}} | \mathsf{v} \in V\}$
 1: hyperparameters: episode length eplength, learning rate $\alpha$, discount factor $\gamma$
 2: $s \leftarrow \mathit{InitialState}(); \mathsf{v} \leftarrow \mathsf{v}_I; \lambda \leftarrow []; \rho \leftarrow []$
 3: for $0 \leq t < \mathit{eplength}$ do
 4:     $a \leftarrow \text{GetEpsilonGreedyAction}(\mathrm{q}^{\mathsf{v}}, s)$
 5:     $s' \leftarrow \text{ExecuteAction}(s, a)$
 6:     $\mathsf{v}' \leftarrow \delta(\mathsf{v}, L(s, a, s'))$
 7:     $r \leftarrow \sigma(\mathsf{v}, L(s, a, s'))$ $\boxed{\text{or observe reward in JIRP}}$
 8:     $\mathrm{q}^{\mathsf{v}}(s, a) \leftarrow (1 - \alpha) \cdot \mathrm{q}^{\mathsf{v}}(s, a) + \alpha \cdot (r + \gamma \cdot \max_a \mathrm{q}^{\mathsf{v}'}(s', a))$
 9:     for $\hat{\mathsf{v}} \in V \setminus \{\mathsf{v}\}$ do
10:         $\hat{\mathsf{v}}' \leftarrow \delta(\hat{\mathsf{v}}, L(s, a, s'))$
11:         $\hat{r} \leftarrow \sigma(\hat{\mathsf{v}}, L(s, a, s'))$
12:         $\mathrm{q}^{\hat{\mathsf{v}}}(s, a) \leftarrow (1 - \alpha) \cdot \mathrm{q}^{\hat{\mathsf{v}}}(s, a) + \alpha \cdot (\hat{r} + \gamma \cdot \max_a \mathrm{q}^{\hat{\mathsf{v}}'}(s', a))$
13:     append $L(s, a, s')$ to $\lambda$; append $r$ to $\rho$
14:     $s \leftarrow s'; \mathsf{v} \leftarrow \mathsf{v}'$
15: return $(\lambda, \rho, Q)$
```

denoted as q^{v} for each state v of the reward machine (their initial values can be specified as an input to the algorithm).

The current state v of the reward machine guides the exploration by determining which q-function is used to choose the next action (Line 4). However, in each single exploration step, the q-functions corresponding to all reward machine states are updated (Lines 8 and 12).

The fundamental hypothesis of QRM is that the rewards are known, but the transition probabilities are unknown. When using QRM without rewards explicitly provided (in particular, as a part of JIRP), the rewards must be *observed* (see Line 7). During the execution of the episode, traces (λ, ρ) of the reward machine are collected (Line 13) and returned in the end. While not necessary for q-learning, the traces will be useful in our algorithm to check the consistency of an *inferred* reward machine with rewards received from the environment.

5.3 Joint Inference of Reward Machines and Policies (JIRP)

Given a reward machine, the QRM algorithm learns an optimal policy. In many situations, however, assuming the knowledge of the reward function (and thus the reward machine) is unrealistic. Even if the reward function is known, it can be challenging for users to formalize it in terms of a reward machine. In this section, we describe an RL algorithm that *iteratively* infers (i.e., learns) a correct reward machine together with the optimal policy for a given RL problem and a labeling function, which provides high-level knowledge about relevant events.

Our algorithm combines an automaton learning algorithm to infer hypothesis reward machines and the QRM algorithm for RL on the current candidate. Inconsistencies between the hypothesis machine and the observed traces are used to trigger re-learning of the reward machine. We show that the resulting iterative algorithm converges in the limit almost surely to

Algorithm 7 JIRP

```
1: Initialize the hypothesis reward machine $\mathcal{H}$ with a set of states $V$
2: Initialize a set of q-functions $Q = \{q^v | v \in V\}$
3: Initialize $X = \emptyset$
4: for episode $n = 1, 2, \ldots$ do
5:     $(\lambda, \rho, Q) =$ QRM_episode$(\mathcal{H}, Q)$
6:     if $\mathcal{H}(\lambda) \neq \rho$ then
7:         add $(\lambda, \rho)$ to $X$
8:         infer a new, minimal hypothesis reward machine $\mathcal{H}$ based on the traces in $X$
9:         re-initialize $Q$
```

the reward machine capturing the reward function and to an optimal policy for this reward machine.

JIRP Algorithm

Algorithm 7 describes our JIRP algorithm. It maintains a hypothesis reward machine $\mathcal{H}$ and runs the QRM algorithm to learn an optimal policy (given $\mathcal{H}$). The episodes of QRM are used to collect traces and update q-functions. As long as the traces are consistent with the current hypothesis reward machine $\mathcal{H}$, QRM interacts with the environment using $\mathcal{H}$ to guide the learning process. However, if a trace (λ, ρ) is detected that is inconsistent with the hypothesis reward machine (i.e., $\mathcal{H}(\lambda) \neq \rho$, Line 6), our algorithm records it in a set X (Line 7)—we call the trace (λ, ρ) a *counterexample* and the set X a *sample*. Every time the sample is updated, JIRP infers a *minimal* reward machine (Line 8) that is consistent with the sample (we formalize this shortly).

Note that JIRP infers not an arbitrary consistent reward machine but a *minimal* one (i.e., a consistent reward machine with the smallest number of states among all consistent reward machines). This additional requirement can be seen as Occam's razor strategy [153] and is crucial in that it enables JIRP to converge to an optimal policy in the limit.

Inference of Minimal Reward Machines

Intuitively, a sample $X \subseteq (2^P)^+ \times O^+$ contains a finite number of counterexamples, and we would like to construct a (new) reward machine that is both *minimal* and *consistent with* X.

Task 5.1. Given a finite set $X \subseteq (2^P)^+ \times O^+$, construct a minimal reward machine $\mathcal{H}$ that is consistent with X in that $\mathcal{H}(\lambda) = \rho$ for each $(\lambda, \rho) \in X$.

To learn minimal consistent reward machines, we adopt a popular approach from classical automata learning [110, 179, 175]. Furthermore, the approach is similar to our technique for learning LTL formula from Chapter 3. The idea is to generate a sequence of propositional logic formulas Φ_n^X for increasing values of $n \in \mathbb{N} \setminus \{0\}$ that satisfy two properties:

- Φ_n^X is satisfiable if and only if there exists a reward machine with n states that is consistent with X; and

- a satisfying assignment of the variables in Φ_n^X contains sufficient information to derive such a reward machine.

By starting with $n = 1$ and increasing n by one until Φ_n^X becomes satisfiable, which we check using a SAT solver, we obtain an effective algorithm that learns a minimal reward machine that is consistent with the given sample.

Before we show how to construct the formula Φ_n^X in the remainder, let us briefly introduce the necessary notation. First, we observe that $O_X \subseteq \mathbb{R}$, the set of rewards that appear in the sample X, is finite. Additionally, for a trace $\tau = (\ell_1\ell_2\ldots\ell_k, r_1r_2\ldots r_k) \in (2^P)^* \times O_X^*$, we define the set of *prefixes of* τ by $\mathit{Pref}(\tau) = \{(l_1\ldots l_i, r_1\ldots r_i) \in (2^P)^* \times O_X^* \mid 0 \leq i \leq k\}$. Here, we note that $(\varepsilon, \varepsilon) \in \mathit{Pref}(\tau)$ always holds. We lift this notion to samples $X \subseteq (2^P)^* \times O_X^*$ by $\mathit{Pref}(X) = \bigcup_{\tau \in X} \mathit{Pref}(\tau)$. Finally, we will denote the label sequence obtained by appending to the sequence λ a label l by λl. Similarly, the concatenation of the reward sequence ρ and a reward r will be ρr.

Encoding Reward Machines in Propositional Logic

The encoding of reward machines in propositional logic exploits the observation that once a set V of states and an initial state $\mathsf{v}_I \in V$ is fixed, every reward machine $\mathsf{A} = (V, \mathsf{v}_I, 2^P, O_X, \delta, \sigma)$ is uniquely determined by its transition function δ and its output function σ. Hence, let us fix a set V of states with $|V| = n$ and an initial state $\mathsf{v}_I \in V$.

To encode the transition function and the output function, we introduce two propositional variables: $d_{\mathsf{v},l,\mathsf{w}}$ for $\mathsf{v}, \mathsf{w} \in V$ and $l \in 2^P$; and $o_{\mathsf{v},l,r}$ for $\mathsf{v} \in V$, $l \in 2^P$, and $r \in O_X$. Intuitively, the variable $d_{\mathsf{v},l,\mathsf{w}}$ is set to true if and only if the transition $\delta(\mathsf{v}, l) = \mathsf{w}$ exists in the prospective reward machine, while $o_{\mathsf{v},l,r}$ is set to true if and only if $\sigma(\mathsf{v}, l) = r$.

To ensure that the variables $d_{\mathsf{v},l,\mathsf{w}}$ and $o_{\mathsf{v},l,r}$ indeed encode deterministic functions, we add the following constraints:

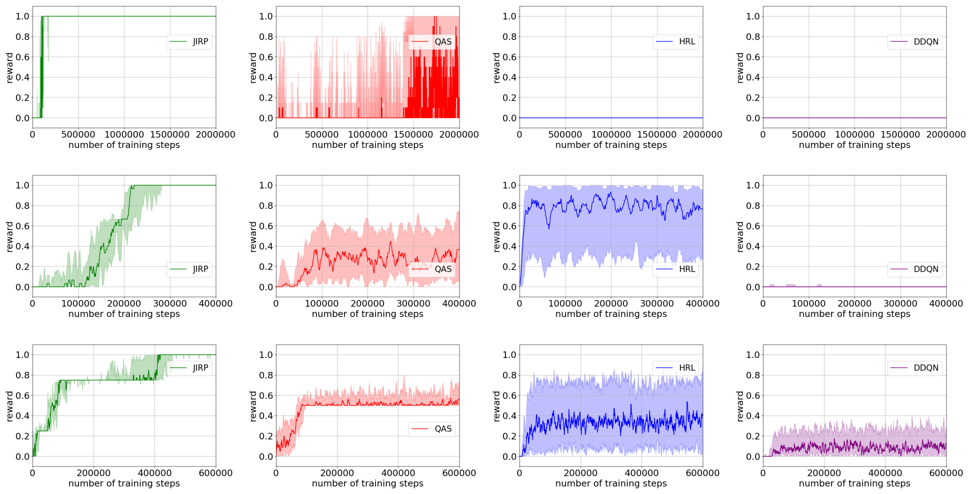
$$\bigwedge_{\mathsf{v} \in V} \bigwedge_{l \in 2^P} \Big[\big[\bigvee_{\mathsf{w} \in V} d_{\mathsf{v},l,\mathsf{w}} \big] \wedge \big[\bigwedge_{\mathsf{w} \neq \mathsf{w}' \in V} \neg (d_{\mathsf{v},l,\mathsf{w}} \wedge d_{\mathsf{v},l,\mathsf{w}'}) \big] \Big] \tag{5.1}$$

$$\bigwedge_{\mathsf{v} \in V} \bigwedge_{l \in 2^P} \Big[\big[\bigvee_{r \in O} o_{\mathsf{v},l,r} \big] \wedge \big[\bigwedge_{r \neq r' \in O} \neg (o_{\mathsf{v},l,r} \wedge o_{\mathsf{v},l,r'}) \big] \Big] \tag{5.2}$$

Note that Formula (5.1) ensures that for every state $\mathsf{v} \in V$ and symbol $l \in 2^P$ the variable $d_{\mathsf{v},l,\mathsf{w}}$ is set to true for exactly one $\mathsf{w} \in V$, whereas Formula (5.2) ensures that for every state $\mathsf{v} \in V$ and symbol $l \in 2^P$ the variable $o_{\mathsf{v},l,r}$ is set to true for exactly one $r \in O$.

Remark 5.1. Given a satisfying assignment $\mathcal{I} \models (5.1) \wedge (5.2)$, we can derive a reward machine $\mathsf{A}_{\mathcal{I}} = (V, \mathsf{v}_I, 2^{\mathcal{P}}, O_X, \delta, \sigma)$ by $\delta(\mathsf{v}, l) = \mathsf{w}$ if and only $\mathcal{I}(d_{\mathsf{v},l,\mathsf{w}}) = 1$, and $\sigma(\mathsf{v}, l) = r$ if and only if $\mathcal{I}(o_{\mathsf{v},l,r}) = 1$.

So far, we have captured the basic structure of a reward machine in our encoding. However, $\mathsf{A}_{\mathcal{I}}$ is not (yet) related to the sample X. We connect it to the sample in the following section.



(a) JIRP (b) QAS (c) HRL (d) DDQN

FIGURE 5.2: Attained rewards of 10 independent simulation runs averaged for every 10 training steps for autonomous vehicle scenario (first row), office world scenario (second row), and Minecraft world scenario (third row).

Consistency with the Sample

To encode consistency with a sample in propositional logic, we introduce new auxiliary variables $x_{\lambda,\mathsf{v}}$ for $(\lambda, \rho) \in \mathit{Pref}(X)$ and $\mathsf{v} \in V$. Intuitively, these variables capture the run of the prospective reward machine on (prefixes of) label sequences in X in the sense that $x_{\lambda,\mathsf{w}}$ is set to true if and only if the prospective reward machine reaches states w after reading λ. To obtain the desired meaning, we add the following constraints:

$$x_{\varepsilon,\mathsf{v}_I} \wedge \bigwedge_{\mathsf{v}\in V\setminus\{\mathsf{v}_I\}} \neg x_{\varepsilon,\mathsf{v}} \tag{5.3}$$

$$\bigwedge_{(\lambda l,\rho r)\in \mathit{Pref}(X)} \bigwedge_{\mathsf{v},\mathsf{w}\in V} (x_{\lambda,p} \wedge d_{\mathsf{v},l,\mathsf{w}}) \rightarrow x_{\lambda l,\mathsf{w}} \tag{5.4}$$

$$\bigwedge_{(\lambda l,\rho r)\in \mathit{Pref}(X)} \bigwedge_{\mathsf{v}\in V} x_{\lambda,\mathsf{v}} \rightarrow o_{\mathsf{v},l,\mathsf{r}} \tag{5.5}$$

Intuitively, Formula (5.3) ensures that the reward machine starts in the initial state q_I, while Formula (5.4) ensures that the variables $x_{\lambda,\mathsf{w}}$ encode valid runs of the prospective reward machine on prefixes of label sequences in X. Formula (5.5) enforces that the prospective reward machine outputs the correct rewards.

The conjunction of Formulas (5.1), (5.2) (ensuring that the reward machine is well defined), (5.3), (5.4), and (5.5) (ensuring that the reward machine is consistent with the sample X) is the desired propositional formula Φ_n^X. The satisfying assignment for this formula yields a minimal reward machine consistent with the sample.

5.4 JIRP Case Studies

In this section, we compare JIRP to other algorithms that might be brought to bear in a non-Markovian environment, for three different scenarios:

(1) an autonomous vehicle scenario, (2) an office world scenario as in the running example, adapted from [122], and (3) a Minecraft world scenario adapted from [9].

We compare the following four different methods:

1) JIRP: We have implemented a prototype of JIRP, which uses the tabular q-learning method [226] and the libalf library [38] to infer minimal reward machines.

2) QAS (q-learning in augmented state space): to incorporate the extra information of the labels (i.e., high-level events in the environment), we perform tabular q-learning [226] in an augmented state space with an extra binary vector representing whether each label has been encountered or not. We choose the learning rate to be $\alpha = 0.8$ and the discount factor to be $\gamma = 0.9$.

3) HRL (hierarchical reinforcement learning): following [139], we define one option for each propositional variable $p \in \mathcal{P}$, and the option terminates whenever p becomes true.

4) DDQN (deep reinforcement learning with double q-learning) [106]: fhe neural network used has 6 fully-connected layers and 64 neurons per layer. The feature inputs of the neural network are the past 200 labels along the trajectory and the current MDP state.

Before reporting on the results of the comparison, we give a short summary of algorithmic optimizations implemented in JIRP.

Optimizations

When implementing JIRP, we introduced two algorithmic optimizations: (1) batching of counterexamples and (2) transfer of q-functions.

Algorithm 7 infers a new hypothesis reward machine whenever a counterexample is encountered. This could potentially incur a high computational cost in frequently inferring the hypothesis reward machines. In order to adjust the frequency of inferring new reward machines, we batch encountered counterexamples. After each period of N episodes (where $N \in \mathbb{Z}_{>0}$ is a user-defined hyperparameter), we infer a new hypothesis reward machine (if there are some counterexamples encountered).

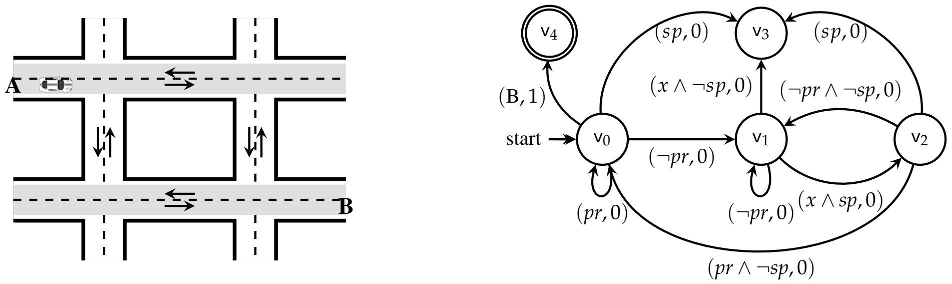
Optimization (2) addresses the fact that in Algorithm 7, after a new hypothesis reward machine is inferred, the q-functions are re-initialized and the experiences from the previous iteration of RL are not utilized. We try to preserve the information from past hypothesis reward machines. Criterion for preserving is based on a notion of *equivalent states*, as defined below.

Definition 5.4. Given a reward machine A and a state $\mathsf{v} \in V$, let $\mathsf{A}[\mathsf{v}]$ be the machine with v as the initial state. For two reward machines A and $\hat{\mathsf{A}}$ over the sets V and $\hat{V}$ of states, respectively, two states $\mathsf{v} \in V$ and $\hat{\mathsf{v}} \in \hat{V}$ are equivalent, denoted by $\mathsf{v} \sim \hat{\mathsf{v}}$, if and only if $\mathsf{A}[\mathsf{v}](\lambda) = \mathsf{A}[\hat{\mathsf{v}}](\lambda)$ for all label sequences λ.

To determine pairwise equivalent states, one can use a simple modification of Hopcroft's partition refinement algorithm [116]. More precisely, if A has m states and Â has n states, one can decide the equivalence of every pair of states in time $\mathcal{O}(2^P \cdot (m+n)^2)$.

Autonomous Vehicle Scenario

We consider the following autonomous vehicle scenario, sketched in Figure 5.3a. As is common in many countries, some of the roads are priority roads. While traveling on a priority road, a car has the right-of-way and does not need to stop at intersections. In the example of Figure 5.3a, all the horizontal roads are priority roads (indicated by gray shading), whereas the vertical roads are ordinary roads.



(a) Map of a residential area.

(b) Reward machine for the autonomous vehicle.

FIGURE 5.3: Autonomous vehicle scenario and a reward machine capturing the reward function

Let us assume that the task of the autonomous vehicle is to drive from position "A" on the map to position "B" while obeying the traffic rules. To simplify matters, we are here only interested in the traffic rules concerning the right-of-way and how the vehicle acts at intersections with respect to the traffic from the intersecting roads. Moreover, we make the following two further simplifications: (1) the vehicle correctly senses whether it is on a priority road and (2) the vehicle always stays in the road and goes straight forward while not at the intersections.

The vehicle is obeying the traffic rules if and only if

- it is traveling on an ordinary road and stops for exactly one time unit at the intersections;
- it is traveling on a priority road and does not stop at the intersections.

We intend to achieve the above task in the setting of episodic RL. Specifically, after each episode of 100 time units, the vehicle receives a reward of 1 if it reached B while obeying the traffic rules; otherwise it receives a reward of 0. It can be seen that the (implicit) reward function is non-Markovian (the reward depends not only on the current state, but also on the history of states from which one can decide if the traffic rules were obeyed). The set of actions is $A = \{\text{straight, left, right, stay}\}$, corresponding to going straight, turning left, turning right and staying in place. For simplicity, we assume that the labeled MDP is deterministic (i.e, the slip rate is zero for each action).

The set of propositional variables is $\{sp, pr, \mathrm{B}, x\}$ and the labeling function L is defined by

$$\begin{aligned} sp \in L(s,a,s') &\Leftrightarrow a = \text{stay}, \\ pr \in L(s,a,s') &\Leftrightarrow s'.priority = true, \\ x \in L(s,a,s') &\Leftrightarrow s \in \mathcal{X}, \\ \mathrm{B} \in L(s,a,s') &\Leftrightarrow s'.x = x_{\mathrm{B}} \wedge s'.y = y_{\mathrm{B}}, \end{aligned}$$

where $s'.priority$ is a Boolean variable that is true if and only if s' is on a priority road, $\mathcal{X}$ represents the set of locations where the vehicle is entering an intersection, $s'.x$ and $s'.y$ are the x and y coordinate values at state s, and x_{B} and y_{B} are x and y coordinate values at B (see Figure 5.3a).

The reward machine in Figure 5.3b captures the reward function for traffic rules. The state v_0 denotes that the vehicle is on a priority road. In this state, the vehicle ends up in the sink state (v_3) if it stops. While on a priority road, the vehicle does not have to pay attention to intersections.

However, if the vehicle transitions to the state v_1 (that is, if it enters a non-priority road), it has to stop for exactly one time unit at the intersection (state v_2). At all the transitions mentioned thus far, the vehicle only gets a reward of 0. Only upon transitioning from v_0 to v_4 does it get a reward 1.

Note that this is not a unique reward machine capturing our reward function. For instance, there could be other reward machines that would describe what happens upon reading B in the state v_1. (However, this will never occur in the given scenario.)

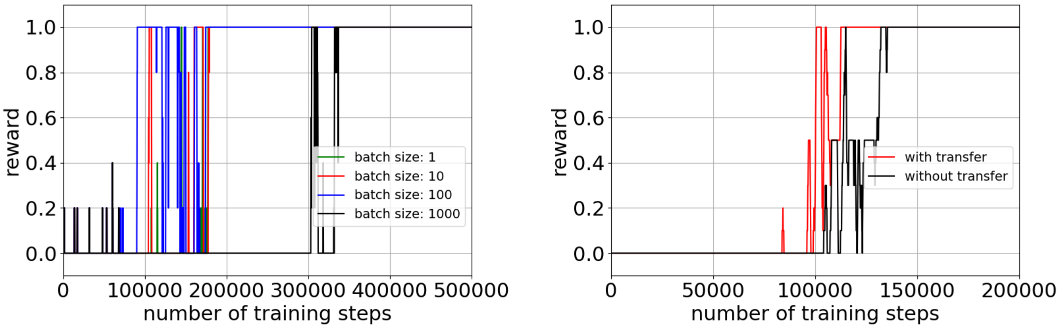
We set $eplength = 100$, $N = 100$ (we discuss the choice of N in the following subsection), and used the transfer of q-functions for the JIRP method. The first row of Figure 5.2 shows the attained rewards with the four different methods in the autonomous vehicle scenario. The y-axis shows the reward obtained after after the number of training steps shown on the x-axis. JIRP converges to optimal policies within 100,000 training steps, while QAS does not converge to optimal policies, HRL and DDQN are stuck with near-zero cumulative reward for up to two million training steps.

Batch Sizes

To test the influence of different batch sizes of counter-examples, we perform JIRP with four different batch sizes of counter-examples: $N = 1$, $N = 10$, $N = 100$ and $N = 1000$. Table 5.1 shows the average computation time for 10 independent runs with the four different batch sizes in autonomous vehicle scenario. Figure 5.4a shows the attained median rewards of 10 independent simulation runs with the four different batch sizes in the autonomous vehicle scenario. We observe that with increased batch size, the computation time decreases as the frequency of inferring new hypothesis reward machines decreases (as shown in Table 5.1). On the other hand, the number of training steps necessary for convergence increases when N

TABLE 5.1: Average computation time (in seconds) for JIRP and four different batch sizes in autonomous vehicle scenario.

	$N = 1$	$N = 10$	$N = 100$	$N = 1000$
time (s)	3995.99	3193.280	2110.23	1311.79



(a) Four different batch sizes.

(b) With and without the transfer of q-functions.

FIGURE 5.4: The effects of algorithmic optimizations: attained median rewards of 10 independent simulation runs in the autonomous vehicle scenario.

becomes larger than 100 (as shown in Figure 5.4a), as slow updating of hypothesis reward machines can cause delay for the optimal convergence.

Transfer of Q-functions

To test the influence of the transfer of q-functions, we perform JIRP in the autonomous vehicle scenario with and without the transfer of q-functions. As can be seen in Figure 5.4b, JIRP with the transfer of q-functions converges to an optimal policy at about 110,000 training steps, while JIRP without the transfer of q-functions converges to an optimal policy at about 140,000 training steps. Therefore, the transfer of q-functions improves sampling efficiency of JIRP.

Office World Scenario

We consider the office world scenario in the 9×12 grid-world [122]. The agent has four possible actions at each time step: move north, move south, move east and move west. After each action, the robot may slip to each of the two adjacent cells with the probability of 0.05, respectively. We use three tasks with different high-level structural relationships among subtasks such as getting coffee, getting mails and going to the office.

We use the same hyperparameters as those in the autonomous vehicle scenario. The second row of Figure 5.2 shows the cumulative rewards with the four different methods in the office world scenario. JIRP converges to an optimal policy within 150,000 training steps, while QAS and HRL reach only 40% and 80% (respectively) of the optimal median cumulative reward within 400,000 training steps, and DDQN is stuck with near-zero attained reward for up to 400,000 training steps.

Minecraft World Scenario

We consider the Minecraft example in a 21×21 gridworld [9]. The four actions and the slip rates are the same as in the office world scenario. We use four tasks including making a plank, making a stick, making a bow and making a bridge.

We use the same hyperparameters as those in the autonomous vehicle scenario. The third row of Figure 5.2 shows the cumulative rewards with the four different methods in the Minecraft world scenario. JIRP converges to an optimal policy within 600,000 training steps, while QAS, HRL and DDQN reach only 50%, 40% and 20% (respectively) of the optimal median cumulative reward within 600,000 training steps.

5.5 RL in non-Markovian Environments with Advice (JIRP^{Adv})

In previous sections, we argued how unrealistic it is to assume that a user can provide a reward machine capturing the non-Markovian reward function. However, it is reasonable to assume that there will be users who can provide some intuition about the reward function. The JIRP method does not support that—it will blindly explore and update the hypothesis reward machine based on the encountered examples.

In this section, we want to use the intuition that users have and let our algorithm accept advice from them. We think of advice as indicating which of the label sequences observed by the agent are *promising* (i.e., for which sequences the agent could get a reward, without determining the numerical value of the reward), and which are not. A label sequence is a word of finite length, and we represent advice as a set of deterministic finite automata (DFAs) that accept promising label sequences. The key argument in favor of choosing DFAs as a formalism for advice is that they are widely known among engineers. Furthermore, many declarative specification languages (such as regular expressions or linear temporal logic) can be translated into DFAs.

Our algorithm is an extension of JIRP, presented in previous sections. Thus, we name it JIRP^{Adv}, to indicate that it takes advice from the user about the reward machine and uses it to learn more efficiently. Advice can express the full information about when the reward machine will give a reward, as well as no information at all. Importantly, the advice is not assumed to be perfect: the algorithm can handle wrong advice.

JIRP^{Adv} takes a set of advice DFAs as its input. Interacting with the environment, the agent iteratively learns a reward machine as described in previous sections. We extend the SAT-based encoding from Section 5.3 to learn a reward machine that is not only *consistent* with the observed rewards, but also *compatible* with the advice. The advice reduces the space of all possible reward machines consistent with the observations, thus speeding-up the convergence. If an an incompatible advice is given by the user, JIRP^{Adv} eventually discards that advice and recovers the learning process.

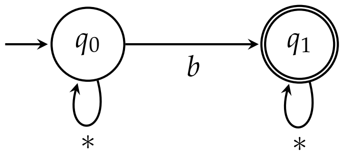


FIGURE 5.5: An advice compatible with the reward machine from Figure 5.1b. The advice says that the label b must be a part of the sequence that gets a reward.

Advice

We use a set of *deterministic finite automata (DFAs)* to model advice from the user. Formally, an advice DFA $\mathcal{D}$ is given by $\mathcal{D} = (Q, q_I, \Sigma, \delta, F)$. It consists of a nonempty, finite set Q of states, an initial state $q_I \in Q$, an input alphabet Σ (here: $\Sigma = 2^P$), a transition function $\delta \colon Q \times \Sigma \to Q$, and set $F \subseteq Q$ of final states.

The core idea of an advice DFA is to guide the inference of reward machines by providing information about which label sequences might result in a reward and which cannot result in a reward. More precisely, the meaning of an advice DFA $\mathcal{D}$ is that all label sequences $\lambda \in \mathcal{L}(\mathcal{D})$ *can* (but do not need to) obtain a *positive reward*, whereas all label sequences $\lambda \notin \mathcal{L}(\mathcal{D})$ *must not* receive a positive reward. Hence, an advice DFA acts as a binary classifier indicating which explorations are promising and which are not.

In our algorithm, advice DFAs are used to restrict the space in which to search for the true reward machine (i.e., the one capturing the reward function R of the MDP). We do so by only constructing candidate reward machines that are *compatible* with the advice DFAs given by the user, as defined below. This makes it possible to learn the correct machine with fewer examples and speeds up the overall convergence of our RL algorithm.

Definition 5.5. We say that an advice DFA $\mathcal{D}$ and a reward machine A are *compatible* if for all nonempty attainable label sequences $\ell_1\ell_2\ldots\ell_k \in (2^P)^+$ of the MDP $\mathcal{M}$ with $\mathsf{A}(\ell_1\ell_2\ldots\ell_k) = r_1r_2\ldots r_k$ it holds that $r_k > 0$ implies $\ell_1\ell_2\ldots\ell_k \in \mathcal{L}(\mathcal{D})$.

We have chosen DFAs as the means of providing advice for two reasons. First, DFAs are a simple formalism, familiar to many engineers and data scientists, that admit effective translations from other common formalisms, such as regular expressions or Linear Temporal Logic. Second, DFAs have a simple, binary semantics, which does not permit to express different reward values for different behaviors. This reflects the observation that it is less demanding for humans to give advice suggesting “what must not be done” than providing the exact, quantitative value.

Figure 5.5 shows an example of an advice DFA compatible with the reward machine from Figure 5.1b. Finally, note that our definitions so far only permit advice for positive rewards. This is not a restriction since we can simply introduce a second “type” of advice DFA for negative rewards. For the sake of a simpler presentation, however, the remainder of this chapter focuses on advice DFAs for positive rewards only.

Similar to the notion of compatibility of an advice and a reward machine, we say that a trace $(\ell_1\ell_2\ldots\ell_k, r_1r_2\ldots r_k)$ is *included* in advice $\mathcal{D}$ if $r_k > 0$ implies $\ell_1\ell_2\ldots\ell_k \in \mathcal{L}(\mathcal{D})$.

Algorithm 8 The JIRP^{Adv} algorithm

```
Input: A set $D = \{\mathcal{D}_1, \ldots, \mathcal{D}_\ell\}$ of advice DFAs
 1: Initialize an empty sample $X \leftarrow \emptyset$
 2: Initialize a reward machine $\mathsf{A}$ with state set $V$ that is compatible with $D$
 3: Initialize a set of $q$-functions $Q = \{q^{\mathsf{v}} \mid \mathsf{v} \in V\}$
 4: for episodes $i = 1, 2, \ldots$ do
 5:     $(\lambda, \rho, Q) \leftarrow \mathrm{QRM}(\mathsf{A}, Q)$
 6:     if $\mathsf{A}(\lambda) \neq \rho$ then
 7:         Add $(\lambda, \rho)$ to $X$
 8:     if $(\lambda, \rho)$ is not included a DFA in $\mathcal{D} \in D$ then
 9:         Remove $\mathcal{D}$ from $D$
10:     if $X$ or $D$ have changed then
11:         Infer a new minimal reward machine $\mathsf{A}$, consistent with $X$ and compatible with $D$
12:         Re-initialize $Q$ (or transfer $q$-functions)
```

Reinforcement Learning with Advice

Our advice-guided RL algorithm, JIRP^{Adv}, is shown as Algorithm 8. It is a simple extension of JIRP that takes into account the user's advice. It maintains a hypothesis reward machine A and runs the QRM algorithm to learn an optimal policy (wrt. A). The episodes of QRM are used to collect traces and update q-functions. As long as the traces are consistent with the current hypothesis reward machine A and included each advice DFAs in D, QRM interacts with the environment using A to guide the learning process.

If a trace (λ, ρ) is encountered that is inconsistent with the hypothesis reward machine (i.e., $\mathsf{A}(\lambda) \neq \rho$), our algorithm records it in a sample set X (Lines 6 and 7). Similarly, if there exists an advice DFA $\mathcal{D} \in D$ that does not include (λ, ρ), we remove $\mathcal{D}$ from D (Lines 8 and 9). This is necessary because the (λ, ρ) is an actual trace experienced by the agent, showing that the advice was incorrect.

Every time the sample is updated or an advice is removed, JIRP^{Adv} infers a new minimal reward machine A' (Line 11) that is

(a) *consistent* with the sample in the sense that $\mathsf{A}'(\lambda) = \rho$ holds for all $(\lambda, \rho) \in X$ and

(b) compatible with each advice DFA in D.

Note that JIRP^{Adv} infers a *minimal* consistent and compatible reward machine (i.e., one with the fewest number of states among all consistent and compatible reward machines). Also note that we re-use all of JIRP's algorithmic optimizations, such as transfer of q-functions from one reward machine to the next and batching of counterexamples, but we omit their descriptions in the pseudocode.

Adding Compatibility with Advice to the Propositional Encoding

The key part of the Algorithm 8 is Line 11, where a new reward machine is inferred. This inference makes sure that the inferred reward machine is consistent with the advice. To achieve that, we extend the encoding into propositional logic presented in Section 5.3.

Let us fix an advice DFA $\mathcal{D} \in D$, say $\mathcal{D} = (Q_\mathcal{D}, q_{I,\mathcal{D}}, 2^P, \delta_\mathcal{D}, F_\mathcal{D})$. We now show how to constrain the variables $d_{\mathsf{v},l,\mathsf{w}}$ and $o_{\mathsf{w},l,r}$ so that the prospective reward machine is compatible with $\mathcal{D}$. We can then successively add similar constraints until the prospective reward machine is compatible with all advice DFAs from D.

The key idea of our encoding is to track the synchronized runs of the prospective reward machine $\mathsf{A}_\mathcal{I}$ and the advice DFA $\mathcal{D}$.

Furthermore, since our definition of compatibility refers to attainable sequences of the underlying MDP, we want to track the run of an automaton which captures possible transitions of the MDP. Formally, an NFA is a tuple $\mathcal{N} = (Q, q_I, \Sigma, \Delta, F)$ where Q, q_I, Σ, F are as in DFAs and $\Delta \subseteq Q \times \Sigma \times Q$ is the transition *relation*. Similar to DFAs, a run of an NFA $\mathfrak{A}$ on a word $u = a_1 \ldots a_k$ is a sequence $q_0, \ldots q_k$ such that $q_0 = q_I$ and $(q_{i-1}, a_i, q_i) \in \Delta$ for each $i \in \{1, \ldots, k\}$. In contrast to DFAs, however, NFAs permit multiple runs on the same input. An NFA accepts a word u if there is a single run on that word that ends in an accepting state.

We now observe that every MDP $\mathcal{M}$ can be "translated" into an NFA $\mathcal{N}_\mathcal{M}$ that is "equivalent" to the reward machine in a sense that it accepts only the label sequences that are attainable in $\mathcal{M}$.

Lemma 5.1. Given a labeled MDP $\mathcal{M}$, one can construct an NFA $\mathcal{N}_\mathcal{M}$ with $2|\mathcal{M}|$ states that accepts exactly the attainable label sequences of $\mathcal{M}$.

Proof. Given a labeled MDP $\mathcal{M} = (S, s_I, A, p, R, \gamma, P, L)$, we construct an NFA $\mathcal{N}_\mathcal{M} = (Q_\mathcal{M}, q_{I,\mathcal{M}}, 2^P, \Delta_\mathcal{M}, F_\mathcal{M})$ by

- $Q_\mathcal{M} = S$;
- $q_{I,\mathcal{M}} = s_I$;
- $(s, \ell, s') \in \Delta_\mathcal{M}$ if and only if there exists an action $a \in A$ with $L(s, a, s') = \ell$ and $p(s, a, s) > 0$; and
- $F_\mathcal{M} = S$.

A straightforward induction shows that $\lambda \in \mathcal{L}(\mathcal{N}_\mathcal{M})$ holds if and only if λ is an attainable label of $\mathcal{M}$. □

Now we come back to adding constraints which will ensure the compatibility with the advice DFA $\mathcal{D}$. The final goal is to derive a machine $\mathsf{A}_\mathcal{I}$ from the model $\mathcal{I}$ of the propositional encoding. We introduce the following new auxiliary variables to the encoding: $z_{\mathsf{v},q_\mathcal{D},q_\mathcal{M}}$ for $\mathsf{v} \in V$, $q_\mathcal{D} \in Q_\mathcal{D}$, and $q_\mathcal{M} \in Q_\mathcal{M}$. Intuitively, $z_{\mathsf{v},q_\mathcal{D},q_\mathcal{M}}$ is set to true if there exists a label sequence λ such that $\mathsf{A}_\mathcal{I}\colon \mathsf{v}_I \xrightarrow{\lambda} \mathsf{v}$ (upon reading λ, the machine $\mathsf{A}_\mathcal{I}$ will end up in state v), $\mathcal{D}\colon q_{I,\mathcal{D}} \xrightarrow{\lambda} q_\mathcal{D}$ (upon reading λ, the advice automaton $\mathcal{D}$ will end up in state $q_\mathcal{D}$), and $\mathcal{N}_\mathcal{M}\colon q_{I,\mathcal{M}} \xrightarrow{\lambda} q_\mathcal{M}$ (upon reading λ, the NFA tracking attainability of a sequence in the MDP $\mathcal{M}$ will end up in state $q_\mathcal{M}$). [2]

[2] To ease the notation, we have used an intuitive symbol $\mathsf{A}\colon \mathsf{v} \xrightarrow{\lambda} \mathsf{w}$ to abbreviate a run of the reward machine A on the input-sequence λ that starts in v and leads to w. By definition, we have $\mathsf{A}\colon \mathsf{v} \xrightarrow{\varepsilon} \mathsf{v}$ for the empty sequence ε and every state v. We did similarly for runs of a DFA and an NFA.

We now add the following constraints:

$$z_{\mathsf{v}_I, q_{I,\mathcal{D}}, q_{I,\mathcal{M}}} \tag{5.6}$$

$$\bigwedge_{\mathsf{v},\mathsf{v}' \in V} \bigwedge_{l \in 2^P} \bigwedge_{\delta_{\mathcal{D}}(q_{\mathcal{D}}, l) = q'_{\mathcal{D}}} \bigwedge_{(q_{\mathcal{M}}, l, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}}} (z_{\mathsf{v}, q_{\mathcal{D}}, q_{\mathcal{M}}} \land d_{\mathsf{v}, l, \mathsf{v}'}) \rightarrow z_{\mathsf{v}', q'_{\mathcal{D}}, q'_{\mathcal{M}}} \tag{5.7}$$

$$\bigwedge_{\mathsf{v} \in V} \bigwedge_{\substack{\delta_{\mathcal{D}}(q_{\mathcal{D}}, l) = q'_{\mathcal{D}} \\ q'_{\mathcal{D}} \notin F_{\mathcal{D}}}} \bigwedge_{\substack{(q_{\mathcal{M}}, l, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}} \\ q'_{\mathcal{M}} \in F_{\mathcal{M}}}} z_{\mathsf{v}, q_{\mathcal{D}}, q_{\mathcal{M}}} \rightarrow \neg \bigvee_{\substack{r \in R_X \\ r > 0}} o_{\mathsf{v}, l, r} \tag{5.8}$$

Note that Formula (5.6) ensures that the synchronized runs of the prospective reward machine $\mathsf{A}_{\mathcal{I}}$, the NFA $\mathcal{N}_{\mathcal{M}}$, and the advice DFA $\mathcal{D}$ start in their initial states. Formula (5.7) enforces that the variables $z_{\mathsf{v}, q_{\mathcal{D}}, q_{\mathcal{M}}}$ correctly track the synchronized runs. Finally, Formula (5.8) ensures that $\mathsf{A}_{\mathcal{I}}$ is compatible with $\mathcal{D}$ by contraposition: if $\mathsf{A}_{\mathcal{I}}$ has moved to state v after reading a label sequence λ and is now processing a new label l but $\lambda l \notin L(\mathcal{D})$ (indicated by $\mathcal{D}$ reaching a non-final state $\mathsf{q}' \notin F_{\mathcal{D}}$), then the output must not be positive. We denote the conjunction of Formulas (5.6), (5.7), and (5.8) by $\Phi_n^{\mathcal{D}}$. For a set D of advice DFAs, we denote the conjunction of corresponding formulas by Φ_n^D, that is $\Phi_n^D := \bigwedge_{\mathcal{D} \in D} \Phi_n^{\mathcal{D}}$.

The final encoding $\Phi_n^{X,D}$ is given by $\Phi_n^{X,D} := \Phi_n^X \land \Phi_n^D$.

5.6 Optimal Convergence

In this section we prove that our encoding has desirable properties, culminating with the proof that the final RL algorithm (JIRP, as well as its generalization JIRP^{Adv}) converges to an optimal policy. We begin by showing the properties of the encoding.

Theorem 5.1. Let $X \subseteq (2^{\mathcal{P}})^+ \times \mathbb{R}^+$ be a sample and D a finite set of advice DFAs that are compatible with X. Moreover, let $\Phi_n^{X,D} = \Phi_n^X \land \Phi_n^D$ be the propositional encoding as defined above. Then, the following holds:

1. If $\mathcal{I} \models \Phi_n^{X,D}$, then the reward machine $\mathsf{A}_{\mathcal{I}}$ is consistent with X and compatible with each $\mathcal{D} \in D$.

2. If there exists a reward machine with n states that is consistent with X and compatible with each $\mathcal{D} \in D$, then $\Phi_n^{X,D}$ is satisfiable.

We will first show two lemmas: the first one claiming that the formula $\Phi_n^{X,D}$ captures the consistency property (wrt. the sample X), and the second one claiming that it also captures the compatibility property (wrt. the advice D).

Lemma 5.2. Let $\mathcal{I} \models \Phi_n^{X,D}$ and $\mathsf{A}_{\mathcal{I}}$ the reward machine as in Remark 5.1. Then, $\mathsf{A}_{\mathcal{I}}$ is consistent with X (i.e., $\mathsf{A}_{\mathcal{I}}(\lambda) = \rho$ for each $(\lambda, \rho) \in X$).

Proof. Let $\mathcal{I} \models \Phi_n^{X,D}$ and $\mathsf{A}_{\mathcal{I}}$ the reward machine constructed from the model $\mathcal{I}$. To prove Lemma 5.2, we show the following, more general statement by induction over the length of prefixes $(\lambda, \rho) \in \mathit{Pref}(X)$: if $\mathsf{A}_{\mathcal{I}} \colon \mathsf{q}_{\mathsf{I}} \xrightarrow{\lambda} \mathsf{v}$, then

1. $\mathcal{I}(x_{\lambda,\mathsf{v}}) = \mathit{true}$; and

2. $\mathsf{A}_{\mathcal{I}}(\lambda) = \rho$.

Lemma 5.2 then follows immediately from Part 2 since $X \subseteq \mathit{Pref}(X)$.

Base case: Let $(\varepsilon, \varepsilon) \in \mathit{Pref}(X)$. By definition of runs, we know that $\mathsf{A}_{\mathcal{I}} \colon \mathsf{v}_I \xrightarrow{\varepsilon} \mathsf{v}_I$ is the only run of A on the empty word. Similarly, Formula (5.3) guarantees that the initial state v_I is the unique state $\mathsf{v} \in V$ for which $\mathcal{I}(x_{\varepsilon,\mathsf{v}}) = \mathit{true}$ holds. Both observations immediately prove Part 1. Moreover, $\mathsf{A}(\varepsilon) = \varepsilon$ holds by definition of the semantics of reward machines, which proves Part 2.

Induction step: Let $(\lambda l, \rho r) \in \mathit{Pref}(X)$. Moreover, let $\mathsf{A}_{\mathcal{I}} \colon \mathsf{v}_I \xrightarrow{\lambda} \mathsf{v} \xrightarrow{l} \mathsf{w}$ be the unique run of A on λ. By applying the induction hypothesis, we then obtain that both $\mathcal{I}(x_{\lambda,\mathsf{v}}) = \mathit{true}$ and $\mathsf{A}(\lambda) = \rho$ hold.

To prove Part 1, note that $\mathsf{A}_{\mathcal{I}}$ contains the transition $\delta(\mathsf{v}, l) = \mathsf{w}$ since this transition was used in the last step of the run on λl. By construction of $\mathsf{A}_{\mathcal{I}}$, this can only be the case if $\mathcal{I}(d_{\mathsf{v},l,\mathsf{w}}) = \mathit{true}$. Then, however, Formula (5.4) implies that $\mathcal{I}(x_{\lambda l,q}) = \mathit{true}$ because $\mathcal{I}(x_{\lambda,p}) = \mathit{true}$ (which holds by induction hypothesis). This proves Part 1.

To prove Part 2, we exploit Formula (5.5). More precisely, Formula (5.5) guarantees that if $\mathcal{I}(x_{\lambda,\mathsf{v}}) = \mathit{true}$ and the next observed label is l with reward r, then $\mathcal{I}(o_{\mathsf{v},l,r}) = \mathit{true}$. By construction of $\mathsf{A}_{\mathcal{I}}$, this means that $\sigma(\mathsf{w}, l) = r$. Hence, $\mathsf{A}_{\mathcal{I}}$ outputs r in the last step of the run on λl. Since $\mathsf{A}_{\mathcal{I}}(\lambda) = \rho$ (which holds by induction hypothesis), we obtain $\mathsf{A}_{\mathcal{I}}(\lambda l) = \rho r$. This proves Part 2.

Thus, $\mathsf{A}_{\mathcal{I}}$ is consistent with X. □

In the following lemma, we prove that the reward machine derived from the model of the propositional formula is also compatible with the advice.

Lemma 5.3. Let $\mathcal{I} \models \Phi_n^{X,D}$ and $\mathsf{A}_{\mathcal{I}}$ the reward machine derived from it. Then, $\mathsf{A}_{\mathcal{I}}$ is compatible with $\mathcal{D} \in D$: i.e., $\mathsf{A}_{\mathcal{I}}(\ell_1 \ell_2 \ldots \ell_k) = r_1 r_2 \ldots r_k$ and $r_k > 0$ implies $\ell_1 \ell_2 \ldots \ell_k \in L(\mathcal{D})$ for every nonempty, attainable label sequence $\ell_1 \ell_2 \ldots \ell_k$.

Proof. Let $\mathcal{I} \models \Phi_n^{X,D}$ and $\mathsf{A}_{\mathcal{I}}$ be the reward machine derived from it. We first show that $\mathsf{A}_{\mathcal{I}} \colon \mathsf{v}_I \xrightarrow{\lambda} \mathsf{v}$, $\mathcal{D} \colon \mathsf{q}_{\mathsf{I},\mathcal{D}} \xrightarrow{\lambda} \mathsf{q}_{\mathcal{D}}$, and $\mathcal{N}_{\mathcal{M}} \colon q_{I,\mathcal{M}} \xrightarrow{\lambda} q_{\mathcal{M}}$ imply $\mathcal{I}(z_{\mathsf{v},\mathsf{q}_{\mathcal{D}},q_{\mathcal{M}}}) = \mathit{true}$ for all label sequences $\lambda \in (2^{\mathcal{P}})^*$. The proof of this claim proceeds by induction of the length of label sequences, similar to Part 2 in the proof of Lemma 5.2.

Base case: Let $\lambda = \varepsilon$. By definition of runs, the only runs on the empty label sequence are $\mathsf{A}_{\mathcal{I}} \colon \mathsf{v}_I \xrightarrow{\varepsilon} \mathsf{v}_I$, $\mathcal{D} \colon \mathsf{q}_{\mathsf{I},\mathcal{D}} \xrightarrow{\varepsilon} \mathsf{q}_{\mathsf{I},\mathcal{D}}$, and $\mathcal{N}_{\mathcal{M}} \colon q_{I,\mathcal{M}} \xrightarrow{\varepsilon} q_{I,\mathcal{M}}$. Moreover, Formula (5.6) ensures that $\mathcal{I}(z_{\mathsf{v}_I,\mathsf{q}_{\mathsf{I},\mathcal{D}},q_{I,\mathcal{M}}}) = \mathit{true}$, which proves the claim.

Induction step: Let $\lambda = \lambda' \ell$. Moreover, let $\mathsf{A}_{\mathcal{I}} \colon \mathsf{q}_{\mathsf{I}} \xrightarrow{\lambda'} \mathsf{v} \xrightarrow{\ell} \mathsf{w}$, $\mathcal{D} \colon \mathsf{q}_{\mathsf{I},\mathcal{D}} \xrightarrow{\lambda'} \mathsf{p}_{\mathcal{D}} \xrightarrow{\ell} \mathsf{q}_{\mathcal{D}}$, and $\mathcal{N}_{\mathcal{M}} \colon q_{I,\mathcal{M}} \xrightarrow{\lambda'} \mathsf{p}_{\mathcal{M}} \xrightarrow{\ell} \mathsf{q}_{\mathcal{M}}$ be the runs of $\mathsf{A}_{\mathcal{I}}$, $\mathcal{D}$ and $\mathcal{N}_{\mathcal{M}}$ on $\lambda = \lambda' \ell$, respectively. By induction hypothesis, we then know that $\mathcal{I}(z_{\mathsf{v},\mathsf{p}_{\mathcal{D}},\mathsf{p}_{\mathcal{M}}}) = \mathit{true}$. Moreover, $\mathsf{A}_{\mathcal{I}}$ contains the transition $\delta(\mathsf{v},\ell) = \mathsf{w}$ because this transition was used in the last step of the run of $\mathsf{A}_{\mathcal{I}}$ on λ. By construction of $\mathsf{A}_{\mathcal{I}}$, this can only be the case if $\mathcal{I}(d_{\mathsf{v},\ell,\mathsf{w}}) = \mathit{true}$. In this situation, Formula 5.7 ensures $\mathcal{I}(z_{\mathsf{w},\mathsf{q}_{\mathcal{D}},\mathsf{q}_{\mathcal{M}}}) = \mathit{true}$ (since also $\delta_{\mathcal{D}}(\mathsf{p}_{\mathcal{D}},\ell) = \mathsf{q}_{\mathcal{D}}$ and $(\mathsf{p}_{\mathcal{M}},\ell,\mathsf{q}_{\mathcal{M}}) \in \Delta_{\mathcal{M}}$), which proves the claim.

Let now $\lambda = \ell_1\ell_2\ldots\ell_k$ be a nonempty, attainable label sequence (i.e., $k \geq 1$). Moreover, let $\mathsf{A}_{\mathcal{I}}\colon \mathsf{q}_{\mathsf{I}} \xrightarrow{\ell_1\ell_2\ldots\ell_{k-1}} \mathsf{v} \xrightarrow{\ell_k} \mathsf{w}$ be the run of $\mathsf{A}_{\mathcal{I}}$ on λ, $\mathcal{D}\colon \mathsf{q}_{\mathsf{I},\mathcal{D}} \xrightarrow{\ell_1\ell_2\ldots\ell_{k-1}} \mathsf{p}_{\mathcal{D}} \xrightarrow{\ell_k} \mathsf{q}_{\mathcal{D}}$ the run of $\mathcal{D}$ on λ, and $\mathcal{N}_{\mathcal{M}}\colon q_{I,\mathcal{M}} \xrightarrow{\ell_1\ell_2\ldots\ell_{k-1}} \mathsf{p}_{\mathcal{M}} \xrightarrow{\ell} \mathsf{q}_{\mathcal{M}}$ the run of an NFA capturing the attainable sequences. Our induction shows that $\mathcal{I}(z_{\mathsf{v},\mathsf{q}_{\mathcal{D}},\mathsf{q}_{\mathcal{M}}}) = \mathit{true}$ holds in this case.

Towards a contradiction with the statement of the Lemma, assume that $\mathsf{A}_{\mathcal{I}}(\ell_1\ell_2\ldots\ell_k) = r_1r_2\ldots r_k$, $r_k > 0$, and $\ell_1\ell_2\ldots\ell_k \notin \mathcal{L}(\mathcal{D})$. In particular, this means $\mathsf{q}_{\mathcal{D}} \notin \mathsf{F}_{\mathcal{D}}$. Since $\delta_{\mathcal{D}}(\mathsf{p}_{\mathcal{D}},\ell_k) = \mathsf{q}_{\mathcal{D}}$ (which was used in the last step in the run of $\mathcal{D}$ on $\ell_1\ell_2\ldots\ell_k$) and $\mathcal{I}(z_{\mathsf{v},\mathsf{q}_{\mathcal{D}},\mathsf{q}_{\mathcal{M}}}) = \mathit{true}$ (due to the induction above), Formula (5.8) ensures that $\mathcal{I}(o_{\mathsf{v},\ell,r}) = 0$ for all $r \in O$ with $r > 0$. However, Formula (5.2) ensures that there is exactly one $r \in O$ with $\mathcal{I}(o_{\mathsf{v},\ell,r}) = 1$. Thus, there has to exist an $r \in O$ such that $r \leq 0$ and $\mathcal{I}(o_{\mathsf{v},\ell,r}) = 1$. By construction of $\mathsf{A}_{\mathcal{I}}$, this means that the last output r_k of $\mathsf{A}_{\mathcal{I}}$ on reading ℓ_k must have been $r_k \leq 0$. However, our assumption was $r_k > 0$, which is a contradiction. Thus, $\mathsf{A}_{\mathcal{I}}$ is compatible with the advice DFA $\mathcal{D}$. □

We are now ready to prove the correctness of our SAT encoding.

Proof of Theorem 5.1. The proof of Part 1 follows immediately from Lemma 5.2 and Lemma 5.3.

To prove Part 2, let $\mathsf{A} = (V,\mathsf{v}_I,2^P,O,\delta,\sigma)$ be a reward machine with n states that is consistent with X and compatible with each $\mathcal{D} \in D$. From this reward machine, we can derive a valuation $\mathcal{I}$ for the variables $d_{\mathsf{v},l,\mathsf{w}}$ and $o_{\mathsf{v},l,r}$ in a straightforward way (e.g., setting $\mathcal{I}(d_{\mathsf{v},l,\mathsf{w}}) = \mathit{true}$ if and only is $\delta(\mathsf{v},l) = \mathsf{w}$). Moreover, we obtain valuation for the variables $x_{\lambda,\mathsf{v}}$ from the runs of (prefixes) of traces in the sample X, and valuations for the variables $z_{\mathsf{v},\mathsf{p}_{\mathcal{D}},\mathsf{p}_{\mathcal{M}}}$ from the synchronized runs of A, $\mathcal{D}$, and $\mathcal{N}_{\mathcal{M}}$, for each $\mathcal{D} \in D$. Then, $\mathcal{I}$ indeed satisfies $\Phi_n^{X,D}$. □

With Theorem 5.1, we have established that our propositional encoding has the desirable properties. Now we continue building up the proof that JIRP^{Adv} converges to an optimal policy. We first show that JIRP^{Adv} almost surely explores every attainable trajectory in the limit (i.e., with probability 1 when the number of episodes goes to infinity).

Lemma 5.4. Given $m \in \mathbb{N}$, JIRP^{Adv} with *eplen* $\geq m$ almost surely explores every m-attainable trajectory in the limit.

Proof. We use induction over the length i of trajectories to prove that JIRP with *eplength* $\geq m$ explores every m-attainable trajectory with a positive (non-zero) probability.

Base case: The only trajectory of length $i = 0$, s_I, is always explored because it is the initial state of every exploration.

Induction step: Let $i = i' + 1$ and $\zeta = s_0 a_0 s_1 \ldots s_{i'} a_{i'} s_i$ be an m-attainable trajectory of length $i \leq m$. Then, the induction hypothesis yields that JIRP^{Adv} explores each m-attainable trajectory $s_0 a_0 s_1 \ldots s_{i'}$ (of length $i' = i - 1$). Moreover JIRP^{Adv} continues its exploration because $eplength \geq m > i'$. At this point, every action $a_{i'}$ will be chosen with probability at least $\epsilon \times \frac{1}{|A_{s_{i'}}|}$, where $A_{s_{i'}} \subseteq A$ denotes the set of available actions in the state $s_{i'}$ (this lower bound is due to the ϵ-greedy strategy used in the exploration). Having chosen action $a_{i'}$, the state s_i is reached with probability $p(s_{i'}, a_{i'}, s_i) > 0$ because ζ is m-attainable. Thus, the trajectory ζ is explored with a positive probability.

Since JIRP^{Adv} with $eplength \geq m$ explores every m-attainable trajectory with a positive probability, the probability of an m-attainable trajectory not being explored becomes 0 in the limit (i.e., when the number of episodes goes to infinity). Thus, JIRP^{Adv} almost surely (i.e., with probability 1) explores every m-attainable trajectory in the limit. □

As an immediate consequence of Lemma 5.4, we obtain that JIRP^{Adv} almost sure explores every (m-)attainable label sequence in the limit as well.

Corollary 5.1. Given $m \in \mathbb{N}$, JIRP^{Adv} with $eplen \geq m$ almost surely explores every m-attainable label sequence in the limit.

In what follows, we will determine an upper bound on the episode length, which will guarantee that we find all the necessary label sequences to 1) eliminate the advice DFAs incompatible with the true reward machine 2) determine the correct reward machine. The optimal convergence then follows directly from having the correct reward machine.

We first observe that every reward machine A can be translated into a DFA $\mathfrak{A}_{\mathsf{A}}$ that is "equivalent" to the reward machine. This DFA operates over the combined alphabet $2^{\mathcal{P}} \times O$ and accepts a sequence $(\ell_1, r_1) \ldots (\ell_k, r_k)$ if and only if A outputs the reward sequence $r_1 r_2 \ldots r_k$ on reading the label sequence $\ell_1 \ell_2 \ldots \ell_k$.

Lemma 5.5. Given a reward machine $\mathsf{A} = (V, \mathsf{v}_I, 2^P, O, \delta, \sigma)$, one can construct a DFA $\mathcal{A}_{\mathsf{A}}$ with $|\mathsf{A}| + 1$ states such that $\mathcal{L}(\mathcal{A}_{\mathsf{A}}) = \{(l_1, r_1) \ldots (l_k, r_k) \in (2^{\mathcal{P}} \times \sigma) \mid \mathsf{A}(l_1 \ldots l_k) = r_1 \ldots r_k\}$.

Proof. For a given reward machine A, we define a DFA $\mathcal{A}_{\mathsf{A}} = (Q, q_I, 2^P \times O_X, \delta, F)$ by

- $Q = V \cup \{\perp\}$, where $\perp \notin V$;
- $q_I = \mathsf{v}_I$;
- $\delta(\mathsf{v}, (\ell, r)) = \begin{cases} \mathsf{w} & \text{if } \delta(\mathsf{v}, \ell) = \mathsf{w} \text{ and } \sigma(\mathsf{v}, \ell) = r; \\ \perp & \text{otherwise} \end{cases}$
- $F = V$.

In this definition, $\perp$ is a new sink state to which $\mathcal{A}_{\mathsf{A}}$ moves if its input does not correspond to a valid input-output pair produced by A. A straightforward induction over the length of inputs to $\mathcal{A}_{\mathsf{A}}$ shows that it indeed accepts the desired language. In total, $\mathcal{A}_{\mathsf{A}}$ has $|\mathsf{A}| + 1$ states. □

The next lemma tells us that there is a short label sequence witnessing that an advice DFA is incompatible with the true reward machine.

Lemma 5.6. Let A be a reward machine and $\mathcal{D}$ an advice DFA. If $\mathcal{D}$ is not compatible with A, then there exists an attainable label sequence $\ell_1\ell_2\ldots\ell_k$ with $k \leq 2|\mathcal{M}| \cdot (|\mathsf{A}|+1) \cdot |\mathcal{D}|$ such that $\mathsf{A}(\ell_1\ell_2\ldots\ell_k) = r_1r_2\ldots r_k$, $r_k > 0$, and $l_1\ldots l_k \notin L(\mathcal{D})$.

Proof. Let A be a reward machine and $O_X \subseteq \mathbb{R}$ the *finite* set of rewards that A can output. Moreover, let $\mathcal{D} = (Q_{\mathcal{D}}, q_{I,\mathcal{D}}, 2^{\mathcal{P}}, \delta_{\mathcal{D}}, F_{\mathcal{D}})$ be an advice DFA. Our proof proceeds by constructing a sequence of auxiliary automata to derive the desired bound.

First, we construct the DFA $\mathcal{A}^*_{\mathsf{A}} = (Q^*_{\mathsf{A}}, q^*_{I,\mathsf{A}}, 2^{\mathcal{P}} \times O_X, \delta^*_{\mathsf{A}}, F^*_{\mathsf{A}})$ according to Lemma 5.5. Recall that $(l_1,r_1)\ldots(l_k,r_k) \in L(\mathcal{A}^*_{\mathsf{A}})$ if and only if $\mathsf{A}(l_1\ldots l_k) = r_1\ldots r_k$. Moreover, $\mathcal{A}^*_{\mathsf{A}}$ has $|\mathsf{A}|+1$ states.

Second, we modify the DFA $\mathcal{A}^*_{\mathsf{A}}$ so that it only accepts sequences $(l_1,r_1)\ldots(l_k,r_k)$ with $r_k > 0$. To this end, we augment the state space with an additional bit $b \in \{0,1\}$, which tracks whether the most recent reward was 0 or greater than 0. Formally, we define a DFA $\mathcal{A}_{\mathsf{A}} = (Q_{\mathsf{A}}, q_{I,\mathsf{A}}, 2^{\mathcal{P}} \times R, \delta_{\mathsf{A}}, F_{\mathsf{A}})$ by

- $Q_{\mathsf{A}} = Q^*_{\mathsf{A}} \times \{0,1\}$;
- $q_{I,\mathsf{A}} = (q^*_{I,\mathsf{A}}, 0)$;
- $\delta_{\mathsf{A}}\big((v,b),(l,r)\big) = \big(\delta^*_{\mathsf{A}}(v,(l,r)), b\big)$ where $b=1$ if and only if $r > 0$; and
- $F_{\mathsf{A}} = F_{\mathsf{A}} \times \{1\}$.

It is not hard to verify that $\mathcal{A}_{\mathsf{A}}$ indeed has the desired property. Moreover, by Lemma 5.5, $\mathcal{A}_{\mathsf{A}}$ has $2(|\mathsf{A}|+1)$ states.

Third, we adapt the advice DFA $\mathcal{D}$, which works over the alphabet $2^{\mathcal{P}}$, to match the input alphabet $2^{\mathcal{P}} \times O_X$ of $\mathcal{A}_{\mathsf{A}}$. We do so by expanding the domain of the transition function $\delta_{\mathcal{D}}$: it now maps the elements of $S_{\mathcal{A}} \times (2^{\mathcal{P}} \times O_X)$ into S, while disregarding the O_X component. This modification does not change the size of the automaton $\mathcal{D}$.

Fourth, we construct the simple product NFA of $\mathcal{A}_{\mathsf{A}}$, $\mathcal{D}$ and $\mathcal{N}_{\mathcal{M}}$ (the last one is as obtained by Lemma 5.1). This NFA is given by $\mathcal{A} = (Q, q_I, 2^{\mathcal{P}} \times O_X, \Delta, F)$ where

- $S = Q_{\mathsf{A}} \times Q_{\mathcal{D}} \times Q_{\mathcal{M}}$;
- $q_I = (q_{I,\mathsf{A}}, q_{I,\mathcal{D}}, q_{I,\mathcal{M}})$;
- $((q_{\mathsf{A}}, q_{\mathcal{D}}, q_{\mathcal{M}}), (q'_{\mathsf{A}}, q'_{\mathcal{D}}, q'_{\mathcal{M}})) \in \Delta$ iff there exists (l,r) such that $\delta_{\mathsf{A}}(q_{\mathsf{A}},(l,r)) = q'_{\mathsf{A}}$, $\delta_{\mathcal{D}}(q_{\mathcal{D}},(l,r)) = q'_{\mathcal{D}'}$, and $(q_{\mathcal{M}}, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}}$.
- $F = F_{\mathsf{A}} \times (Q_{\mathcal{D}} \setminus F_{\mathcal{D}})$.

By construction, $\mathcal{A}$ accepts a sequence $(l_1,r_1)\ldots(l_k,r_k)$ if and only if $\mathsf{A}(l_1\ldots l_k) = r_1\ldots r_k$ with $r_k > 0$ and $l_1\ldots l_k \notin L(\mathcal{D})$ and the sequence is attainable in the MDP $\mathcal{M}$—in other words, $\mathcal{L}(\mathcal{A})$ contains all sequences that witness that $\mathcal{D}$ is not compatible with A. Moreover, $\mathcal{A}$ has $2(|\mathsf{A}|+1) \cdot |\mathcal{D}| \cdot |\mathcal{M}|$ states.

It is left to show that if $\mathcal{D}$ is not compatible with A, then we can find a witness with the desired length. To this end, it is sufficient to show that if $\mathcal{L}(\mathcal{A}) \neq \emptyset$, then there exists a sequence $(l_1, r_1) \ldots (l_k, r_k) \in L(\mathcal{A})$ with $k \leq 2(|\mathsf{A}| + 1) \cdot |\mathcal{D}| \cdot |\mathcal{M}|$. This fact can be established using a simple pumping argument. To this end, assume that $(l_1, r_1) \ldots (l_k, r_k) \in \mathcal{L}(\mathcal{A})$ with $k > 2(|\mathsf{A}| + 1) \cdot |\mathcal{D}| \cdot |\mathcal{M}|$. Then, there exists a state $q \in Q$ such that the unique accepting run of $\mathcal{A}$ on $(l_1, r_1) \ldots (l_k, r_k)$ visits q twice, say at the positions $i, j \in \{0, \ldots k\}$ with $i < j$. In this situation, however, the NFA $\mathcal{A}$ also accepts the sequence $(l_1, r_1) \ldots (l_i, r_i)(l_{j+1}, r_{j+1}) \ldots (l_k, r_k)$, where we have removed the "loop" between the repeating visits of q. Since this new sequence is shorter than the original sequence, we can repeatedly apply this argument until we arrive at a sequence $(l'_1, r'_1) \ldots (l'_\ell, r'_\ell) \in \mathcal{L}(\mathcal{A})$ with $\ell \leq 2(|\mathsf{A}| + 1) \cdot |\mathcal{D}| \cdot |\mathcal{M}|$. By construction of $\mathcal{A}$, this means that $l'_1 \ldots l'_\ell$ is attainable, $\mathsf{A}(l'_1 \ldots l'_\ell) = r'_1 \ldots r'_\ell$, $r_\ell > 0$, and $l'_1 \ldots l'_\ell \notin L(\mathcal{D})$, which proves the claim. □

Next, we show that if two reward machines are semantically different, then we can bound the length of a trace witnessing this fact.

Lemma 5.7. Let A_1 and A_2 be two reward machines. If A_1 and A_2 differ on an attainable label sequence λ, then the length of λ is at most $2|\mathcal{M}| \cdot (|\mathsf{A}_1| + 1) \cdot (|\mathsf{A}_2| + 1)$.

The proof closely follows the structure of the proof of Lemma 5.6.

Proof. Consider the DFAs $\mathcal{A}_{\mathsf{A}_1}$ and $\mathcal{A}_{\mathsf{A}_2}$ with $|\mathsf{A}_1| + 1$ and $|\mathsf{A}_2| + 1$ states, respectively, obtained as in Lemma 5.5. Consider also the NFA $\mathcal{A}_\mathcal{M}$, which accepts only attainable label sequences, obtained as in Lemma 5.1. This NFA is of size $|\mathcal{M}|$. Let $\mathcal{A}$ be synchronized product of $\mathcal{A}_{\mathsf{A}_1}$, $\mathcal{A}_{\mathsf{A}_2}$, and $\mathcal{A}_\mathcal{M}$, similar to the one in the proof of Lemma 5.6. This automaton has $2|\mathcal{M}| \cdot (|\mathsf{A}_1| + 1) \cdot (|\mathsf{A}_2| + 1)$ states. If $\mathsf{A}_1 \neq \mathsf{A}_2$ on an attainable label sequence, then we can find a sequence which leads in this product to a state where $\mathcal{A}_\mathcal{M}$ accepts (the label sequence is attainable), but exactly one of A_1 and A_2 accepts. To show the bound on the length of this sequence, we will use a similar argument as in the previous lemma. Assume that this sequence is longer than the number of states of the automaton $\mathcal{A}$. This means that there is one element of the sequence that is repeated. But the same product state can be reached by the sequence in which the part between the two repetitions is removed. This procedure can be applied until a new sequence is found, of length at most $2|\mathcal{M}| \cdot (|\mathsf{A}_1| + 1) \cdot (|\mathsf{A}_2| + 1)$. □

With Lemmas 5.6 and 5.7 at hand, we are ready to prove that the correct reward machine will eventually be learned.

Lemma 5.8. Let $\mathcal{M}$ be a labeled MDP and A a true reward machine capturing the rewards of $\mathcal{M}$. Moreover, let $D = \{\mathcal{D}_1, \ldots, \mathcal{D}_\ell\}$ be a set of advice DFAs and $n_{\max} = \max\{|\mathcal{D}_1, \ldots, \mathcal{D}_\ell|\}$. Then, JIRP^{Adv} with *eplen* $\geq \max\left\{2|\mathcal{M}| \cdot (|\mathsf{A}| + 1) \cdot n_{\max}, 2|\mathcal{M}| \cdot (|\mathsf{A}| + 1)^2\right\}$ almost surely learns a reward machine in the limit that is equivalent to A.

Proof. Let $(X_0, D_0), (X_1, D_1), \ldots$ be the sequence of samples and sets of advice DFAs that arise in the run of JIRP^{Adv} whenever a new counterexample is added to X (in Lines 6 and 7 of

Algorithm 8) or an advice DFA is removed from the set D (Lines 8 and 9 of Algorithm 8). Moreover, let $\mathsf{A}_0, \mathsf{A}_1, \ldots$ be the corresponding sequence of reward machines that are computed from (X_i, D_i). Note that constructing a new reward machine is always possible because JIRP^{Adv} makes sure that all advice DFAs in the current set D are compatible with the traces in the sample X.

We first observe three properties of these sequences:

1. The true reward machine A (i.e., the one that captures the reward function R) is consistent with every sample X_i that is generated during the run of JIRP^{Adv}. This is due to the fact that the each counterexample is obtained from an actual exploration of the MDP and, hence, corresponds to the "ground truth".

2. The sequence $X_0, X_1, \ldots$ grows monotonically (i.e., $X_0 \subseteq X_1 \subseteq \cdots$) because JIRP^{Adv} always adds counterexamples to X and never removes them (Lines 6 and 7). In fact, whenever a counterexample (λ, ρ) is added to X_i to form X_{i+1}, then $(\lambda, \rho) \notin X_i$ (i.e., $X_i \subsetneq X_{i+1}$). To see why this is the case, remember that JIRP^{Adv} always constructs hypothesis that are consistent with the current sample (and the current set of advice DFAs). Thus, the current reward machine A_i is consistent with X_i, but the counterexample (λ, ρ) was added because $\mathsf{A}_i(\lambda) \neq \rho$. Thus, (λ, ρ) cannot have been an element of X_i.

3. The sequence $D_0, D_1, \ldots$ decreases monotonically (i.e., $D_0 \supseteq D_1 \supseteq \cdots$) because JIRP^{Adv} always removes advice DFAs from D and never adds any (Lines 8 and 9). Thus, there exists a position $i^\star \in \mathbb{N}$ at which this sequence becomes stationary, implying that $D_i = D_{i+1}$ for $i \geq i^\star$.

Similar to Property 1, we now show that each advice DFA in the set D_i, $i \geq i^\star$, is compatible with the true reward machine A. Towards a contradiction, let $\mathcal{D} \in D_i$ be an advice DFA and assume that $\mathcal{D}$ is not compatible with A. Then, Lemma 5.6 guarantees the existence of a label sequence $l_1 \ldots l_k$ with

$$\begin{aligned} k &\leq 2|\mathcal{M}| \cdot (|\mathsf{A}| + 1) \cdot |\mathcal{D}| \\ &\leq 2|\mathcal{M}| \cdot (|\mathsf{A}| + 1) \cdot n_{\max} \end{aligned}$$

such that $\mathsf{A}(l_1, \ldots l_k) = r_1 \ldots r_k, r_k > 0$, and $l_1 \ldots l_k \notin L(\mathcal{D})$. Since we have chosen $eplen \geq 2(|\mathsf{A}| + 1) \cdot n_{\max}$, Corollary 5.1 guarantees that JIRP^{Adv} almost surely explores this label sequence in the limit. Once this happens, JIRP^{Adv} removes $\mathcal{D}$ from the set D (Lines 8 and 9). Hence, we obtain the following:

4. Every advice DFA in D_i, $i \geq i^\star$, is compatible with the true reward machine A.

Next, we establish the three additional properties about the sub-sequence $\mathsf{A}_{i^\star}, \mathsf{A}_{i^\star+1}, \ldots$ of hypotheses starting at position $i^\star$:

5. The size of the true reward machine A is an upper bound for the size of $\mathsf{A}_{i^\star}$ (i.e., $|\mathsf{A}_{i^\star}| \leq |\mathsf{A}|$). This is due to the fact that A is consistent with every sample X_i (Property 1),

every advice DFA in D_i, $i \geq i^\star$, is compatible with A (Property 4), and JIRP^{Adv} always computes minimal consistent reward machines.

6. We have $|\mathsf{A}_i| \leq |\mathsf{A}_{i+1}|$ for all $i \geq i^\star$. Towards a contradiction, assume that $|\mathsf{A}_i| > |\mathsf{A}_{i+1}|$. Since JIRP^{Adv} always computes consistent reward machines and $X_i \subsetneq X_{i+1}$ if $i \geq i^\star$ (see Property 2), we know that A_{i+1} is not only consistent with X_{i+1} but also with X_i (by definition of consistency). Moreover, JIRP^{Adv} computes minimal consistent reward machines. Hence, since A_{i+1} is consistent with X_i and $|\mathsf{A}_{i+1}| < |\mathsf{A}_i|$, the reward machine A_i is not minimal, which is a contradiction.

7. We have $\mathsf{A}_i \neq \mathsf{A}_j$ for $i \geq i^\star$ and $j \in \{i^\star, \ldots, i\}$—in other words, the reward machines generated during the run of JIRP^{Adv} after the $i^\star$-th recomputation are semantically distinct. This is a consequence of the facts that (λ_j, ρ_j) was a counterexample to A_j (i.e., $\mathsf{A}_j(\lambda_j) \neq \rho_j$) and that JIRP^{Adv} always constructs consistent reward machines (which implies $\mathsf{A}_i(\lambda_i) = \rho_i$).

Properties 5 and 6 now provide $|\mathsf{A}|$ as an upper bound on the size of any reward machine that JIRP^{Adv} constructs after the $i^\star$-th recomputation. Since there are only finitely many reward machines of size at most $|\mathsf{A}|$, Property 7 implies that there exists a $j^\star \geq i^\star$ after which no new reward machine is inferred. Hence, it is left to show that $\mathsf{A}_{j^\star} = \mathsf{A}$ (i.e., $\mathsf{A}_{j^\star}(\lambda) = \mathsf{A}(\lambda)$ for all label sequences λ).

Towards a contradiction, let us assume that $\mathsf{A}_{j^\star} \neq \mathsf{A}$. Then, Lemma 5.7 guarantees the existence of a label sequence $\lambda = l_1 \ldots l_k$ with

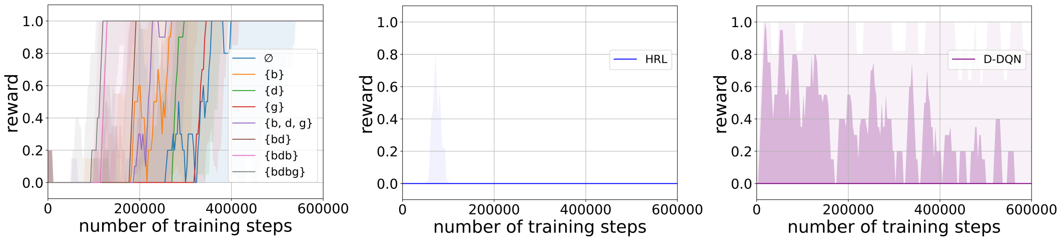
$$\begin{aligned} k &\leq 2|\mathcal{M}| \cdot (|\mathsf{A}_{j^\star}| + 1) \cdot (|\mathsf{A}| + 1) \\ &\leq 2|\mathcal{M}| \cdot (|\mathsf{A}| + 1)^2 \end{aligned}$$

such that $\mathsf{A}_{j^\star}(\lambda) \neq \mathsf{A}(\lambda)$.

Since we have chosen $eplen \geq 2|\mathcal{M}| \cdot (|\mathsf{A}| + 1)^2$, Corollary 5.1 guarantees that JIRP^{Adv} almost surely explores this label sequence in the limit. Thus, the trace (λ, ρ), where $\rho = \mathsf{A}(\lambda)$, is almost surely returned as a new counterexample, resulting in a new sample $X_{j^\star+1}$. This, in turn, causes the construction of a new reward machine, which contradicts the assumption that no further reward machine is generated. Thus, the reward machine $\mathsf{A}_{j^\star}$ is equivalent to the true reward machine A. □

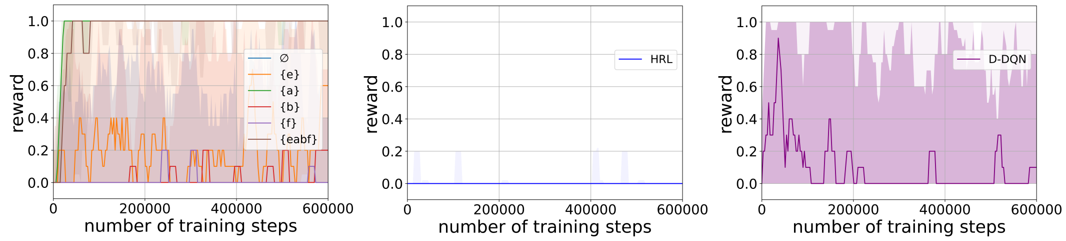
The correctness of JIRP^{Adv} now follows from Lemma 5.8 and the correctness of the QRM algorithm [122]. Lemma 5.8 also provides us with an upper bound on the length of the episodes that JIRP^{Adv} has to explore.

Theorem 5.2. Let $\mathcal{M}$ be a labeled MDP, let A be a true reward machine encoding the rewards of $\mathcal{M}$, and let $D = \{\mathcal{D}_1, \ldots, \mathcal{D}_\ell\}$ be a set of advice DFAs. Moreover, let m be as in Lemma 5.8. Then, JIRP^{Adv} with $eplen \geq m$ almost surely converges to an optimal policy in the limit.



(a) JIRP^{Adv} with advice (b) HRL (c) DDQN

FIGURE 5.6: Attained rewards of 30 independent simulation runs averaged for every 10 training steps each for case study I.



(a) JIRP^{Adv} with advices (b) HRL (c) DDQN

FIGURE 5.7: Attained rewards of 30 independent simulation runs averaged for every 10 training steps each for case study II.

5.7 JIRP^{Adv} Case Studies

In this section, we implement the JIRP^{Adv} approach in two case studies. We compare the three different methods (also used in Section 5.4):[3]

1. JIRP^{Adv}: Our implementation uses the RC2 SAT solver [172] from the PySAT library [124]. A special case of JIRP^{Adv}, when no advice is given, is the JIRP algorithm, from Section 5.3.

2. HRL (hierarchical reinforcement learning): We use a meta-controller for deciding the subtasks (represented by encountering each label) and use the low-level controllers expressed by neural networks [139] for deciding the actions at each state for each subtask.

3. DDQN (deep reinforcement learning with double q-learning): We use the double q-learning method of Hasselt et al. [106]. The DDQN can access the past 200 labels of the trajectory as well as the MDP state.

Figure 5.9 illustrates the advices we use with JIRP^{Adv} in the case studies: DFAs expressing "eventually α_0, then eventually α_1..., then eventually α_{n-1}", noted $\{\alpha_0\alpha_1\cdots\alpha_{n-1}\}$.

Case Study I: Officeworld Domain

This case study relies on the officeworld scenario as introduced in Figure 5.1. We specify a more involved task than that in the motivating example: go to location b, then d, then back to

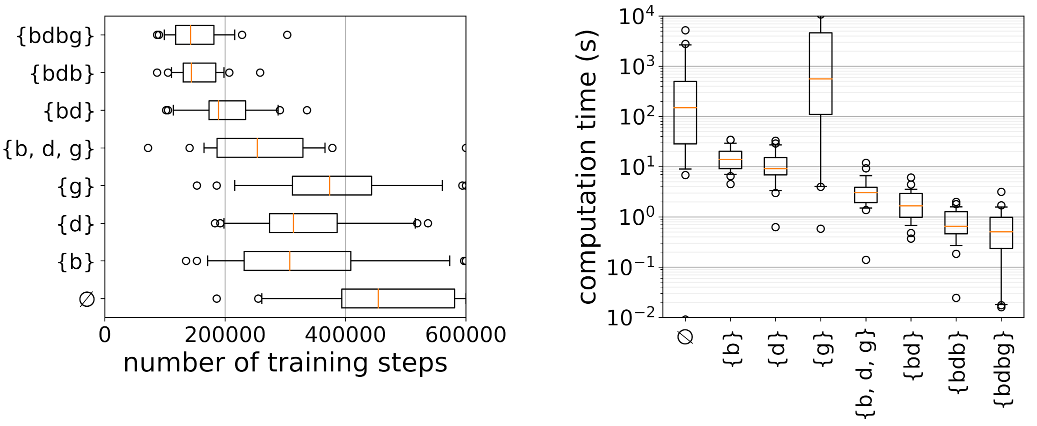
[3] All experiments were conducted on a Vivobook laptop with 1.80-GHz Core i7 CPU and 32-GB RAM

advice	$\emptyset$	$\{b\}$	$\{bd\}$	$\{bdb\}$	$\{bdbg\}$
# steps [$\times 10^3$]	454.5	307.5	189.0	144.0	142.5
inference [s]	149.0	13.9	1.7	0.7	0.5
# inferences	6.1	5.0	3.2	2.3	2.0

TABLE 5.2: Performance of JIRP^{Adv}, for different advice.

b, and finally go to g. We run the JIRP^{Adv} algorithm with different sets of advice DFAs D.

Impact of advice



(a) Training steps for optimal policy convergence.

(b) Cumulative inference time, on a logarithmic scale.

FIGURE 5.8: Distribution of 30 independent simulation runs on each set of advice DFAs with JIRP^{Adv} on case study I.

Figure 5.6a shows the attained reward of JIRP^{Adv} for 8 sets of advice DFAs: $\emptyset$, $\{b\}$, $\{d\}$, $\{g\}$, $\{b,d,g\}$, $\{bd\}$, $\{bdb\}$, $\{bdbg\}$, and Figure 5.8a shows the optimal policy convergence training steps. We observe a clear impact of advices on the performance. It can be seen that the closer the advice is to the ground truth, the faster the maximal reward is reached.

Table 5.2 shows the performance of JIRP^{Adv}, for different advice DFAs. Measured over 30 independent simulation runs, the first row shows the median number of thousands of training steps to convergence; the second row shows the median time needed for reward machines inference; and the third row shows the average number of inferences. With shorter individual inference time and fewer triggers of the reward machine inference, the cumulative inference time drops significantly when using advice, and becomes negligible compared to the RL simulation time with $D \in \{\{bd\}, \{bdb\}, \{bdbg\}\}$.

Figure 5.8b shows the cumulative inference time for the values of D from the set $\{\emptyset, \{b\}, \{d\}, \{g\}, \{b,d,g\}, \{bd\}, \{bdb\}, \{bdbg\}\}$. The advice is clearly helpful, but there is no clear, functional connection between the advice and the algorithm's performance. For instance, the inference time with $D = \{g\}$ increases with respect to the baseline (no advice given).

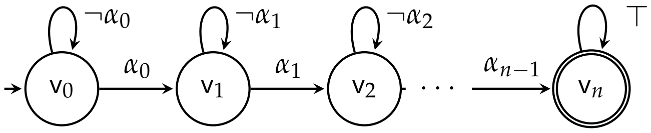


FIGURE 5.9: Advice DFA format used in our case study. The short notation for such DFAs would be $\alpha_0\alpha_1\cdots\alpha_{n-1}$.

Compared to JIRP^{Adv} with $D = \emptyset$ (which corresponds to JIRP), JIRP^{Adv} with $D = \{b\}$, $D = \{bd\}$ and $D = \{bdbg\}$ achieves optimal convergence in 68%, 42% and 31% of the training steps, and with 9.3%, 1.1% and 0.34% cumulative inference time, respectively.

Case Study II: Taxi Domain

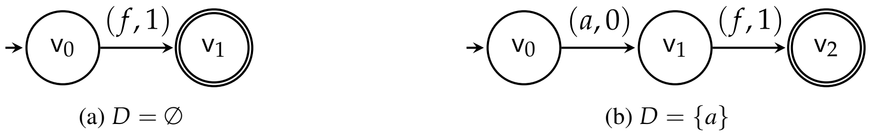
In the second case study, we illustrate a case when the JIRP algorithm is not useful—when the reward function is Markovian. Nonetheless, using advice can be beneficial in such situations.

This experiment is inspired by the OpenAI Gym environment Taxi-v3 (https://gym.openai.com/envs/Taxi-v3/), introduced by Dietterich [67]. The agent, a taxi, navigates on a grid with walls and boarding locations.The agent starts on a random cell. A passenger starts on a random boarding location. The set of actions is $A = \{S, N, E, W, Pickup, Dropoff\}$. The action *Pickup* picks up the passenger present at the agent's current location and has no effect if no passenger is present. The action *Dropoff* drops off the passenger from the taxi to the agent's current boarding location, and has no effect if the taxi is empty or not over any boarding location. The actions S, N, E, W correspond to moving in the four cardinal directions.

We make some assumptions to simplify the problem: the passenger always starts on location A, and the agent starts on a random cell other than a boarding location. We specify the task as carrying the passenger to location B. We define eight labels: a, b, c, d for standing on an empty location (A, B, C, D respectively), and e, f, g, h for standing on a location with the passenger on it (A, B, C, D respectively).

Results

Figure 5.7a presents performances by using 6 different set of advice DFAs: $\emptyset$, $\{e\}$, $\{a\}$, $\{b\}$, $\{f\}$, $\{eabf\}$. Optimal reward is reached only with advices $\{a\}$ and $\{eabf\}$.



(a) $D = \emptyset$

(b) $D = \{a\}$

FIGURE 5.10: The inferred reward machine in all runs of the taxi domain. Omitted transitions are self-looping transitions.

With $D = \emptyset$, only one reward machine is inferred, preventing counterexamples from being registered and new inference to be triggered. This is because the label f (having the passenger on B) is in the sequence if and only if the agent succeeds in the task. Hence the reward machine inferred without advice (Figure 5.10a) is the smallest possible that will correctly classify every attainable label sequence.

The label a, being triggered when the passenger is picked up from A, is helpful to JIRP^{Adv} because it helps the agent track the passenger's location. The advice a lets the reward machine include this label (Figure 5.10b).

This case study shows that advice can be useful even when the reward function is Markovian. Indeed, the advice here was used to infer *how* to reach the state in which a reward is given.

5.8 Related Work

The problem of incorporating high-level temporal knowledge into RL has been studied in hierarchical RL [211, 68, 185]. There, an RL problem is decomposed into a two-level hierarchy: a *meta-controller* decides on a subtask to pursue, and a *controller* decides on actions within the chosen subtask.

The idea of capturing temporal abstraction in a temporal logic or automaton-like form appeared already in the work of Bacchus et al. [19], but an interest in it resurged with the introduction of task monitors [127] and reward machines [122]. They are assumed to be given to the RL agent.

Similar to our work on JIRP, a number of papers followed that suggested learning an automaton-like representation from experience instead of assuming that the user provides them [123, 88, 87, 105]. In JIRP^{Adv} we go a step further and assume that the user is able to provide some knowledge. That knowledge is then incorporated into the learning of reward machines.

The problem statements in the aforementioned papers differ slightly, but in essence, they are solving the same problem: learning a non-Markovian structure of the underlying MDP. This can either be a non-Markovian reward function, a non-Markovian environment dynamics, or partial observability that creates an impression of non-Markovian dynamics. When the MDP is partially observable (POMDP), reward machines need to represent the memory necessary to distinguish between states whose observations coincide. Icarte et al. [120] do this in their algorithm named LRM. The approach taken in JIRP (as well as in JIRP^{Adv}) and LRM have different strengths: JIRP excels when the temporal structure of the rewards contains all the useful information because it learns a smallest and complete representation of the rewards. In this situation, LRM tends to learn larger-than-necessary reward machines, which makes reinforcement learning slower. In cases where the reward function is Markovian, on the other hand, JIRP is not as useful, while LRM can still speed up RL as it can infer useful information about the structure of the POMDP that is not contained in the reward function. The performance of JIRP^{Adv} in such a setup entirely depends on the quality of the received advice.

The notion of advice for RL has been explored in the work by Icarte et al. [121]. There, advice has the form of a temporal logic formula, which can be compiled into a DFA. They assume the availability of *background knowledge functions*, heuristic functions that help the agent follow the advice. No such assumption is necessary for our approach. Focusing more on safety than on helping the learning process, the work on *shields* for RL [5] corrects the agent behavior if it violates the temporal logic specification.

5.9 Conclusion

This chapter focused on specifications that are given implicitly, as rewards that the agent gets. We explored the non-Markovian setup, in which rewards depend on the history of states and actions. Our work is the first to tackle the problem assuming that the (non-Markovian) reward function is not explicitly given to the algorithm. The proposed solution is based on learning automata that capture relevant parts of history, implemented in an algorithm named JIRP. We extended it with the ability to accept advice from users and named that extension JIRP^{Adv}.

One view of our approach is that it helps users with modeling of a problem (in particular, of a state-space). Instead of expecting users to conceptualize states so that the reward function is Markovian, our algorithm infers a suitable state-space through the agent's exploration. Another notable by-product of our approach is the interpretability of the inferred reward function.

A significant drawback of the used constraint-solving methods for automata inference is their scalability. In order to address this challenge, we had to assume that the algorithm receives a set of labels, which define what events bear relevance for getting rewards. While this assumption is not entirely unrealistic, it leaves some burden on the user in an otherwise automated process.

We have shown the optimal convergence property of our algorithms, with an upper bound on the necessary length of the episode. The experiments show that the proposed methods successfully converge to an optimal policy on a number of different small to mid-size benchmarks. Furthermore, they convincingly outperform competing methods.

Chapter 6

Conclusion

In this thesis, we have seen a new model for programming multi-robot systems, one in which robots' work is provided as a service, managed by a centralized backend system. To demonstrate such a programming model, I have implemented a backend system `Antlab`, which accepts declarative specifications (in LTL) and makes sure that the robots of the system execute it.

`Antlab` ensures conformance with the specification by reasoning on two levels. At the first level, it assumes (unrealistically) that the world is ideal (that is, there are no disturbances or unforeseen events, and robots are perfectly synchronized in their actions) and creates a plan for (the subset of) robots with that assumption. At the second level, it locally resolves any conflicts of robots' actions and triggers replanning when necessary. In other words, instead of modeling all things that could possibly go wrong and accounting for them up-front, `Antlab` deals with problems as they arise.

This two-levels approach enables `Antlab` to scale beyond toy examples. However, it also prevents `Antlab` from giving the full end-to-end correctness guarantee for the execution. While realistic scalability remains of utmost importance for `Antlab`, exploring and incorporating different methods (already existing or to appear in the future) for verification of cyber-physical systems would be a way to identify the sweet spot between providing strong guarantees and swift execution.

Choosing LTL as the specification language of `Antlab` enables us to use existing methods for the synthesis and verification of LTL specifications. On the other hand, we have observed that LTL can be difficult for users to master. Furthermore, it does not allow to easily express swarm actions. Thus, it is worth exploring different specification languages in the context of the proposed programming model.

With LTL set as the specification language, I addressed the problem of translating the user's intent into the corresponding LTL specification. To this end, I have presented LTLTALK. It is a tool that uses a single demonstration and a natural language description to devise a set of candidate LTL specifications, which it then narrows down to the correct one by visual interaction with the user. The inferred specification is used to generalize from the provided natural language description, expanding the language of the system for future uses.

We have seen in the thesis that LTLTALK is capable of uncovering and combining common LTL specifications. Interesting future work is examining how well are end-users able to work with LTLTALK, studying it from the perspective of human-machine interaction. In this thesis, I have solved the problem of learning from only positive examples either by identifying a representation form with a suitable notion of tightness (UVW, in Section 3.2), or by utilizing domain knowledge to derive negative examples (Section 3.3). The same problem (learning from positive examples) under different assumptions remains intriguing.

An interesting planning problem arising from the `Antlab`'s programming model is planning for robots that are already executing some previous task. In this thesis, we pose the problem in a general, domain-independent planning form. The proposed solution is a modification of A* search, with speculative initial states. The general problem formulation takes the view of a single agent (which may be a set of robots) and plans for it in a centralized manner. Explicitly considering different robots' plans and planning in a decentralized fashion (perhaps in groups) might bring additional benefits.

In Chapter 5, we have studied the problem of reinforcement learning for temporally extended rewards. In most natural models for such settings, the Markovian property for the reward function does not hold. Therefore, our algorithm interleaves learning of automata capturing the reward function with reinforcement learning. We demonstrate experimentally that this approach outperforms previous approaches and prove that the method converges to an optimal policy.

In order to gain scalability, our method *may* get advice from the user on promising sequences to explore (the advice is formulated as a DFA). It *must* get the set of labels that are relevant for the reward (which are then used as the alphabet of the automaton to be learned). A practically useful extension would be finding a way to automatically infer those labels.

Overall, in this thesis, I argued for a novel multi-robot systems programming model, implemented a proof-of-concept for such a model, and solved specification and planning problems stemming from that model. Beyond technical challenges, there are economic and societal questions about using robots as a shared resource. The answers to those questions are equally important in determining the future of multi-robot programming, as are the technical ones, described in the thesis.

Bibliography

[1] Wil M. P. van der Aalst, Josep Carmona, Thomas Chatain, and Boudewijn F. van Dongen. “A Tour in Process Mining: From Practice to Algorithmic Challenges”. In: *Trans. Petri Nets Other Model. Concurr.* 14 (2019), pp. 1–35.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[3] Keerthi Adabala and Rüdiger Ehlers. “A Fragment of Linear Temporal Logic for Universal Very Weak Automata”. In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. 2018, pp. 335–351. DOI: 10.1007/978-3-030-01090-4_20.

[4] R. Vaughan et al. *Stage simulator*. http://wiki.ros.org/stage. Indigo version.

[5] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. “Safe Reinforcement Learning via Shielding”. In: *AAAI*. AAAI Press, 2018, pp. 2669–2678.

[6] R. Alur, S. Moarref, and U. Topcu. “Compositional Synthesis of Reactive Controllers for Multi-agent Systems”. In: *CAV*. 2016.

[7] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. “Syntax-Guided Synthesis”. In: *Dependable Software Systems Engineering*. Vol. 40. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2015, pp. 1–25.

[8] Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. “Augmented example-based synthesis using relational perturbation properties”. In: *PACMPL* 4.POPL (2020). DOI: 10.1145/3371124.

[9] Jacob Andreas, Dan Klein, and Sergey Levine. "Modular multitask reinforcement learning with policy sketches". In: *ICML'2017*. JMLR. org. 2017, pp. 166–175.

[10] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6. URL: https://doi.org/10.1016/0890-5401(87)90052-6.

[11] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (1987), pp. 87–106.

[12] Dana Angluin. "On the Complexity of Minimum Inference of Regular Sets". In: *Inf. Control.* 39.3 (1978), pp. 337–350.

[13] Dana Angluin and Dana Fisman. "Learning regular omega languages". In: *Theor. Comput. Sci.* 650 (2016), pp. 57–72. DOI: 10.1016/j.tcs.2016.07.031.

[14] M Fareed Arif, Daniel Larraz, Mitziu Echeverria, Andrew Reynolds, Omar Chowdhury, and Cesare Tinelli. "SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces". In: *2020 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2020, pp. 93–103.

[15] *ARM Ltd. Amba specification (rev. 5)*. 2019. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0033/index.html.

[16] Yoav Artzi. *Cornell SPF: Cornell Semantic Parsing Framework*. 2016. eprint: arXiv:1311.3011.

[17] Eugene Asarin, Alexandre Donzé, Oded Maler, and Dejan Nickovic. "Parametric Identification of Temporal Properties". In: *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*. Vol. 7186. Lecture Notes in Computer Science. Springer, 2011, pp. 147–160. URL: https://doi.org/10.1007/978-3-642-29860-8_12.

[18] Florent Avellaneda and Alexandre Petrenko. "Inferring DFA without Negative Examples". In: *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*. 2018, pp. 17–29. URL: http://proceedings.mlr.press/v93/avellaneda19a.html.

[19] Fahiem Bacchus, Craig Boutilier, and Adam J. Grove. "Rewarding Behaviors". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*. 1996, pp. 1160–1167. URL: http://www.aaai.org/Library/AAAI/1996/aaai96-172.php.

[20] Christer Bäckström and Bernhard Nebel. "Complexity Results for SAS$^+$ Planning". In: *Computational Intelligence* 11.4 (1995), pp. 625–655.

[21] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[22] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. “DeepCoder: Learning to Write Programs”. In: *International Conference on Learning Representations (ICLR)*. 2017.

[23] Tomás Balyo, Marijn J. H. Heule, and Matti Järvisalo. “SAT Competition 2016: Recent Developments”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, 2017, pp. 5061–5063.

[24] Gregor B. Banusic, Rupak Majumdar, Marcus Pirron, Anne-Kathrin Schmuck, and Damien Zufferey. “PGCD: robot programming and verification with geometry, concurrency, and dynamics”. In: *ICCPS*. ACM, 2019, pp. 57–66.

[25] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. “CVC4”. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1_14.

[26] Ezio Bartocci, Luca Bortolussi, and Guido Sanguinetti. “Learning Temporal Logical Properties Discriminating ECG models of Cardiac Arrhytmias”. In: *CoRR* abs/1312.7523 (2013). arXiv: 1312.7523. URL: http://arxiv.org/abs/1312.7523.

[27] I. Beltagy and Chris Quirk. “Improved Semantic Parsers For If-Then Statements”. In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. 2016.

[28] J. Benton, Minh B. Do, and Wheeler Ruml. “A Simple Testbed for On-line Planning”. In: *Proceedings of the ICAPS-07 Workshop on Moving Planning and Scheduling Systems into the Real World*. 2007.

[29] J. van den Berg and M. Overmars. “Prioritized motion planning for multiple robots”. In: *IROS*. 2005.

[30] A. Biere, K. Heljanko, T. Junttila, T. latvala, and V. Schuppan. “Linear Encoding of Bounded LTL Model Checking”. In: *LMCS* 2.5:5 (2006), pp. 1–64.

[31] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. “Bounded model checking”. In: *Advances in Computers* 58 (2003), pp. 117–148. DOI: 10.1016/S0065-2458(03)58003-2. URL: https://doi.org/10.1016/S0065-2458(03)58003-2.

[32] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. “Linear Encodings of Bounded LTL Model Checking”. In: *Logical Methods in Computer Science* 2.5 (2006). DOI: 10.2168/LMCS-2(5:5)2006.

[33] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5.

[34] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 194–199.

[35] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of Reactive(1) designs". In: *J. Comput. Syst. Sci.* 78.3 (2012), pp. 911–938. DOI: 10.1016/j.jcss.2011.08.007. URL: https://doi.org/10.1016/j.jcss.2011.08.007.

[36] Avrim Blum, John Hopcroft, and Ravi Kannan. *Foundations of Data Science*. 2018, pp. 1–479. URL: https://www.cs.cornell.edu/jeh/book.pdf.

[37] Mikolaj Bojańczyk. "The Common Fragment of ACTL and LTL". In: *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS*. Ed. by Roberto M. Amadio. Vol. 4962. Lecture Notes in Computer Science. Springer, 2008, pp. 172–185. DOI: 10.1007/978-3-540-78499-9_13.

[38] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. "libalf: The Automata Learning Framework". In: *CAV'2010*. 2010, pp. 360–364. DOI: 10.1007/978-3-642-14295-6_32. URL: https://doi.org/10.1007/978-3-642-14295-6_32.

[39] Giuseppe Bombara, Cristian Ioan Vasile, Francisco Penedo, Hirotoshi Yasuoka, and Calin Belta. "A Decision Tree Approach to Data Classification using Signal Temporal Logic". In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*. ACM, 2016, pp. 1–10.

[40] Ronen I. Brafman, Giuseppe De Giacomo, and Fabio Patrizi. "LTLf/LDLf Non-Markovian Rewards". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. 2018, pp. 1771–1778. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17342.

[41] Leo Breiman, JH Friedman, Richard A Olshen, and Charles J Stone. *Classification and Regression Trees. Wadsworth, 1984*. Routledge, 1993.

[42] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. "A randomized scheduler with probabilistic guarantees of finding bugs". In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*. 2010, pp. 167–178. DOI: 10.1145/1736020.1736040. URL: http://doi.acm.org/10.1145/1736020.1736040.

[43] Ethan Burns, J. Benton, Wheeler Ruml, Sung Wook Yoon, and Minh Binh Do. "Anticipatory On-Line Planning". In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS-12)*. 2012.

[44] Ethan Burns, Wheeler Ruml, and Minh Binh Do. "Heuristic Search When Time Matters". In: *Journal Artificial Intelligence Research* 47 (2013), pp. 697–740.

[45] J. Richard Büchi. "On a decision method in restricted second-order arithmetic". In: *Int. Congr. for Logic, Methodology and Philosophy of Science*. Stanford Univ. Press, 1962, pp. 1–11.

[46] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. "LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning". In: *IJCAI*. ijcai.org, 2019, pp. 6065–6073.

[47] Alberto Camacho and Sheila A. McIlraith. "Learning Interpretable Models Expressed in Linear Temporal Logic". In: *ICAPS*. AAAI Press, 2019, pp. 621–630.

[48] Steven Carr, Nils Jansen, and Ufuk Topcu. "Verifiable RNN-Based Policies for POMDPs Under Temporal Logic Constraints". In: *IJCAI*. ijcai.org, 2020, pp. 4121–4127.

[49] Michael Cashmore, Andrew Coles, Bence Cserna, Erez Karpas, Daniele Magazzeni, and Wheeler Ruml. "Replanning for Situated Robots". In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018*. 2019, pp. 665–673.

[50] Michael Cashmore, Andrew Coles, Bence Cserna, Erez Karpas, Daniele Magazzeni, and Wheeler Ruml. "Temporal Planning while the Clock Ticks". In: *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018*. 2018, pp. 39–46.

[51] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, and Bram Ridder. "Opportunistic Planning for Increased Plan Utility". In: *Proceedings of the ICAPS-16 Workshop on Planning and Robotics (PlanRob 2016)*. 2016.

[52] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. "Active learning for extended finite state machines". In: *Formal Asp. Comput.* 28.2 (2016), pp. 233–263.

[53] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. "Multi-modal Synthesis of Regular Expressions". In: *Programming Language Design and Implementation (PLDI)*. 2020.

[54] Yanju Chen, Ruben Martins, and Yu Feng. "Maximal Multi-layer Specification Synthesis". In: *Foundations of Software Engineering (FSE)*. 2019. DOI: 10.1145/3338906.3338951.

[55] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion*. A Bradford Book, 2005.

[56] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. "Learning Assumptions for Compositional Verification". In: *TACAS*. Vol. 2619. Lecture Notes in Computer Science. Springer, 2003, pp. 331–346.

[57] Andrew Coles, Shahaf Shperberg, Erez Karpas, Solomon Shimony, and Wheeler Ruml. "Beyond Cost-to-go Estimates in Situated Temporal Planning". In: *Proceedings of the ICAPS Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*. 2019.

[58] Werner Damm and David Harel. "LSCs: Breathing Life into Message Sequence Charts". In: *Formal Methods Syst. Des.* 19.1 (2001), pp. 45–80. DOI: 10.1023/A:1011227529550.

[59] Werner Damm, Tobe Toben, and Bernd Westphal. "On the Expressive Power of Live Sequence Charts". In: *Program Analysis and Compilation, Theory and Practice, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*. 2006, pp. 225–246. DOI: 10.1007/978-3-540-71322-7_11.

[60] Jeanette Daum, Álvaro Torralba, Jörg Hoffmann, Patrik Haslum, and Ingo Weber. "Practical Undoability Checking via Contingent Planning". In: *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016*. 2016, pp. 106–114.

[61] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2013.

[62] Giuseppe De Giacomo and Moshe Y. Vardi. "Synthesis for LTL and LDL on Finite Traces". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2015.

[63] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. 2008.

[64] J. A. DeCastro, J. Alonso-Mora, V. Raman, D. Rus, and H. Kress-Gazit. "Collision-Free Reactive Mission and Motion Planning for Multi-Robot Systems". In: *ISRR*. 2015.

[65] A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia. "DRONA: A Framework for Safe Distributed Mobile Robotics". In: *ICCPS*. 2017, pp. 239–248.

[66] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. "Program synthesis using natural language". In: *International Conference on Software Engineering (ICSE)*. 2016. DOI: 10.1145/2884781.2884786.

[67] Thomas G. Dietterich. "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition". In: *CoRR* cs.LG/9905014 (1999). URL: https://arxiv.org/abs/cs/9905014.

[68] Thomas G. Dietterich. "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition". In: *J. Artif. Intell. Res.* 13 (2000), pp. 227–303.

[69] Austin J. Dionne, Jordan Tyler Thayer, and Wheeler Ruml. "Deadline-Aware Search Using On-Line Measures of Behavior". In: *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011*. 2011.

[70] Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. "Synthesis with Abstract Examples". In: *Computer Aided Verification (CAV)*. 2017. DOI: 10.1007/978-3-319-63387-9_13.

[71] Matthew B Dwyer, George S Avrunin, and James C Corbett. "Patterns in property specifications for finite-state verification". In: *Proceedings of the 21st international conference on Software engineering*. 1999, pp. 411–420.

[72] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Patterns in Property Specifications for Finite-State Verification". In: *International Conference on Software Engineering (ICSE)*. 1999. DOI: 10.1145/302405.302672.

[73] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Property Specification Patterns for Finite-state Verification". In: *Proceedings of the Second Workshop on Formal Methods in Software Practice*. FMSP '98. Clearwater Beach, Florida, USA: ACM, 1998, pp. 7–15. ISBN: 0-89791-954-8. DOI: 10.1145/298595.298598. URL: http://doi.acm.org/10.1145/298595.298598.

[74] Rüdiger Ehlers. "ACTL $\cap$ LTL Synthesis". In: *CAV*. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 39–54.

[75] Rüdiger Ehlers. "Computing the Complete Pareto Front". In: *CoRR* abs/1512.05207 (2015). URL: http://arxiv.org/abs/1512.05207.

[76] Rüdiger Ehlers, Ivan Gavran, and Daniel Neider. "Learning Properties in LTL $\cap$ ACTL from Positive Examples Only". In: *FMCAD*. IEEE, 2020, pp. 104–112.

[77] A. Elfes. "Using occupancy grids for mobile robot perception and navigation". In: *IEEE Computer* 22(6) (1989), pp. 46–57.

[78] E. M. Eppstein. *ROS navigation stack*. http://wiki.ros.org/navigation. Indigo version.

[79] M. A. Erdmann and T. Lozano-Pérez. "On multiple moving objects". In: *ICRA*. 1986.

[80] P. Eyerich, R. Mattmüller, and G. Röger. "Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning". In: *Towards Service Robots for Everyday Environments*. Springer, 2012, pp. 49–64. ISBN: 978-3-642-25116-0. DOI: 10.1007/978-3-642-25116-0_6. URL: http://dx.doi.org/10.1007/978-3-642-25116-0_6.

[81] Azadeh Farzan, Yu-Fang Chen, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. "Extending Automated Compositional Verification to the Full Class of Omega-Regular Languages". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 2–17. DOI: 10.1007/978-3-540-78800-3_2.

[82] Maximilian Fickert, Ivan Gavran, Ivan Fedotov, Jörg Hoffmann, Rupak Majumdar, and Wheeler Ruml. "Choosing the Initial State for Online Replanning". In: *AAAI*. AAAI Press, 2021, pp. 12311–12319.

[83] Nathanaël Fijalkow and Guillaume Lagarde. "The Complexity of Learning Linear Temporal Formulas from Examples". In: *CoRR* abs/2102.00876 (2021).

[84] R. Fikes and N. J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving". In: *Artif. Intell.* 2.3/4 (1971), pp. 189–208. DOI: 10.1016/0004-3702(71)90010-5. URL: http://dx.doi.org/10.1016/0004-3702(71)90010-5.

[85] E. Filiot, N. Jin, and J.-F. Raskin. "Antichains and compositional algorithms for LTL synthesis". In: *FMSD* 39.3 (2011), pp. 261–296. DOI: 10.1007/s10703-011-0115-3. URL: http://dx.doi.org/10.1007/s10703-011-0115-3.

[86] C. Finucane, Gangyuan Jing, and H. Kress-Gazit. "LTLMoP: Experimenting with language, Temporal Logic and robot control". In: *IROS*. 2010, pp. 1988–1993.

[87] Daniel Furelos-Blanco, Mark Law, Alessandra Russo, Krysia Broda, and Anders Jonsson. "Induction of Subgoal Automata for Reinforcement Learning". In: *AAAI*. AAAI Press, 2020, pp. 3890–3897.

[88] Maor Gaon and Ronen I. Brafman. "Reinforcement Learning with Non-Markovian Rewards". In: *AAAI*. AAAI Press, 2020, pp. 3980–3987.

[89] Ivan Gavran, Eva Darulova, and Rupak Majumdar. "Interactive synthesis of temporal specifications from examples and natural language". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 201:1–201:26.

[90] Ivan Gavran, Rupak Majumdar, and Indranil Saha. "Antlab: A Multi-Robot Task Server". In: *ACM Trans. Embedded Comput. Syst.* 16.5s (2017), 190:1–190:19.

[91] B. Gerkey and V. Rabaud. *Slam Gmapping package*. https://github.com/ros-perception/slam_gmapping. Indigo version.

[92] M. Ghallab, C. Aeronautiques, C. K. Isi, and D. Wilkins. *PDDL: The Planning Domain Definition Language*. Tech. rep. CVC TR98003/DCS TR1165. Yale Center for Computational Vision and Control, 1998.

[93] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN: 978-1-55860-856-6.

[94] Yashdeep Godhal, Krishnendu Chatterjee, and Thomas A. Henzinger. "Synthesis of AMBA AHB from formal specification: a case study". In: *International Journal on Software Tools for Technology Transfer* 15.5 (2013), pp. 585–601. DOI: 10.1007/s10009-011-0207-9.

[95] E. Mark Gold. "Language Identification in the Limit". In: *Information and Control* 10.5 (1967), pp. 447–474.

[96] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Vol. 2500. Lecture Notes in Computer Science. Springer, 2002.

[97] Nicola Muscettola Gregory, Gregory A. Dorais, Chuck Fry, Richard Levinson, and Christian Plaunt. "IDEA: Planning at the Core of Autonomous Reactive Agents". In: *in Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*. 2002.

[98] Martin Grohe, Christof Löding, and Martin Ritzert. "Learning MSO-definable hypotheses on strings". In: *International Conference on Algorithmic Learning Theory, ALT 2017, 15-17 October 2017, Kyoto University, Kyoto, Japan*. Vol. 76. Proceedings of Machine Learning Research. PMLR, 2017, pp. 434–451. URL: http://proceedings.mlr.press/v76/grohe17a.html.

[99] Martin Grohe and Martin Ritzert. "Learning first-order definable concepts over structures of small degree". In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–12. URL: https://doi.org/10.1109/LICS.2017.8005080.

[100] E. Guizzo. "Three Engineers, Hundreds of Robots, One Warehouse". In: *IEEE Spectrum* 45.7 (2008), pp. 26–34. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2008.4547508.

[101] Sumit Gulwani and Mark Marron. "NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 2014. DOI: 10.1145/2588555.2612177.

[102] Y. Guo and L.E. Parker. "A distributed and optimal motion planning approach for multiple mobile robots". In: *ICRA*. 2002.

[103] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann, 2011. ISBN: 978-0123814791. URL: http://hanj.cs.illinois.edu/bk3/.

[104] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Trans. Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[105] Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, and Daniel Kroening. "DeepSynth: Program Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning". In: *CoRR* abs/1911.10244 (2019).

[106] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proc. AAAI'16*. Phoenix, Arizona: AAAI Press, 2016, pp. 2094–2100. URL: http://dl.acm.org/citation.cfm?id=3016100.3016191.

[107] Malte Helmert. "Concise Finite-Domain Representations for PDDL Planning Tasks". In: *Artificial Intelligence* 173 (2009), pp. 503–535.

[108] Malte Helmert. "The Fast Downward Planning System". In: *J. Artif. Intell. Res.* 26 (2006), pp. 191–246. DOI: 10.1613/jair.1705. URL: https://doi.org/10.1613/jair.1705.

[109] D. Hennes, D. Claes, W. Meeussen, and K. Tuyls. "Multi-robot Collision Avoidance with Localization Uncertainty". In: *AAMAS*. 2012, pp. 147–154. ISBN: 0-9817381-1-7, 978-0-9817381-1-6.

[110] Marijn Heule and Sicco Verwer. "Exact DFA Identification Using SAT Solvers". In: *ICGI'2010'*. Vol. 6339. Lecture Notes in Computer Science. Springer, 2010, pp. 66–79. DOI: 10.1007/978-3-642-15488-1_7.

[111] Jörg Hoffmann. "The metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables". In: *J. Artif. Int. Res.* 20.1 (Dec. 2003), pp. 291–341. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=1622452.1622463.

[112] Jörg Hoffmann. "Where 'Ignoring Delete Lists' Works: Local Search Topology in Planning Benchmarks". In: *J. Artif. Intell. Res.* 24 (2005), pp. 685–758. DOI: 10.1613/jair.1747. URL: https://doi.org/10.1613/jair.1747.

[113] Jörg Hoffmann and Bernhard Nebel. "The FF Planning System: Fast Plan Generation Through Heuristic Search". In: *Journal of Artificial Intelligence Research* 14 (2001), pp. 253–302.

[114] Gerard J. Holzmann. "The Logic of Bugs". In: *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '02/FSE-10. Charleston, South Carolina, USA: Association for Computing Machinery, 2002, 81–87. ISBN: 1581135149. DOI: 10.1145/587051.587064. URL: https://doi-org.ezp-prod1.hul.harvard.edu/10.1145/587051.587064.

[115] Gerard J. Holzmann. "The logic of bugs". In: *Symposium on Foundations of Software Engineering (FSE)*. 2002. DOI: 10.1145/587051.587064.

[116] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. ISBN: 0-201-02988-X.

[117] Kyle Hsu, Rupak Majumdar, Kaushik Mallik, and Anne-Kathrin Schmuck. "Lazy Abstraction-Based Controller Synthesis". In: *ATVA*. Vol. 11781. Lecture Notes in Computer Science. Springer, 2019, pp. 23–47.

[118] Kyle Hsu, Rupak Majumdar, Kaushik Mallik, and Anne-Kathrin Schmuck. "Multi-Layered Abstraction-Based Controller Synthesis for Continuous-Time Systems". In: *HSCC*. ACM, 2018, pp. 120–129.

[119] W. N. N. Hung, X. Song, J. Tan, X. Li, J. Zhang, R. Wang, and P. Gao. "Motion Planning with Satisfiability Modulo Theroes". In: *ICRA*. 2014, pp. 113–118.

[120] Rodrigo A Toro Icarte, Ethan Waldie, Toryn Klassen, Richard Valenzano, Margarita P. Castro, and Sheila A. McIlraith. "Learning Reward Machines for Partially Observable Reinforcement Learning". In: *NeurIPS*. 2019.

[121] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. "Advice-Based Exploration in Model-Based Reinforcement Learning". In: *Canadian Conference on AI*. Vol. 10832. Lecture Notes in Computer Science. Springer, 2018, pp. 72–83.

[122] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. "Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning". In: *ICML'2018*. 2018, pp. 2112–2121. URL: http://proceedings.mlr.press/v80/icarte18a.html.

[123] Rodrigo Toro Icarte, Ethan Waldie, Toryn Q. Klassen, Richard Anthony Valenzano, Margarita P. Castro, and Sheila A. McIlraith. "Learning Reward Machines for Partially Observable Reinforcement Learning". In: *NeurIPS*. 2019, pp. 15497–15508.

[124] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. "PySAT: A Python Toolkit for Prototyping with SAT Oracles". In: *SAT*. 2018, pp. 428–437. DOI: 10.1007/978-3-319-94144-8_26. URL: https://doi.org/10.1007/978-3-319-94144-8_26.

[125] Nils Jansen, Bettina Könighofer, Sebastian Junges, Alex Serban, and Roderick Bloem. "Safe Reinforcement Learning Using Probabilistic Shields (Invited Paper)". In: *CONCUR*. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 3:1–3:16.

[126] Natasha Yogananda Jeppu, Thomas F. Melham, Daniel Kroening, and John O'Leary. "Learning Concise Models from Long Execution Traces". In: *DAC*. IEEE, 2020, pp. 1–6.

[127] Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. "A Composable Specification Language for Reinforcement Learning Tasks". In: *NeurIPS*. 2019, pp. 13021–13030.

[128] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*. 2011, pp. 245–256. DOI: 10.1109/DSN.2011.5958223. URL: https://doi.org/10.1109/DSN.2011.5958223.

[129] Anthony Willem Kamp. "Tense Logic and the Theory of Linear Order". PhD thesis. University of California, Los Angeles, 1968.

[130] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[131] Joseph Kim, Christian Muise, Ankit Shah, Shubham Agarwal, and Julie Shah. “Bayesian Inference of Linear Temporal Logic Specifications for Contrastive Explanations”. In: *IJCAI*. ijcai.org, 2019, pp. 5591–5598.

[132] Russell Knight, Gregg Rabideau, Steve A. Chien, Barbara Engelhardt, and Rob Sherwood. “Casper: Space Exploration through Continuous Planning”. In: *IEEE Intell. Syst.* 16.5 (2001), pp. 70–75.

[133] Thomas Kollar, Vittorio Perera, Daniele Nardi, and Manuela M. Veloso. “Learning environmental knowledge from task-based human-robot dialog”. In: *ICRA*. IEEE, 2013, pp. 4304–4309.

[134] Zhaodan Kong, Austin Jones, Ana Medina Ayala, Ebru Aydin Gol, and Calin Belta. “Temporal logic inference for classification and prediction from data”. In: *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC’14, Berlin, Germany, April 15-17, 2014*. ACM, 2014, pp. 273–282. URL: http://doi.acm.org/10.1145/2562059.2562146.

[135] Zhaodan Kong, Austin Jones, and Calin Belta. “Temporal Logics for Learning and Detection of Anomalous Behavior”. In: *IEEE Trans. Automat. Contr.* 62.3 (2017), pp. 1210–1222. DOI: 10.1109/TAC.2016.2585083. URL: https://doi.org/10.1109/TAC.2016.2585083.

[136] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. “Temporal-Logic-Based Reactive Mission and Motion Planning”. In: *IEEE Transactions on Robotics* (2009).

[137] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. “Translating Structured English to Robot Controllers”. In: *Advanced Robotics* 22.12 (2008). DOI: 10.1163/156855308X344864.

[138] Hillel Kugler, David Harel, Amir Pnueli, Yuan Lu, and Yves Bontemps. “Temporal Logic for Scenario-Based Specifications”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 445–460. ISBN: 3-540-25333-5. DOI: 10.1007/978-3-540-31980-1_29. URL: https://doi.org/10.1007/b107194.

[139] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. “Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation”. In: *NeurIPS’2016*. 2016, pp. 3675–3683.

[140] Orna Kupferman and Moshe Y. Vardi. “Safraless Decision Procedures”. In: *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*. IEEE Computer Society, 2005, pp. 531–542. DOI: 10.1109/SFCS.2005.66.

[141] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. “General LTL Specification Mining (T)”. In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015, pp. 81–92. URL: https://doi.org/10.1109/ASE.2015.71.

[142] Seth Lemons, J. Benton, Wheeler Ruml, Minh Binh Do, and Sung Wook Yoon. “Continual On-line Planning as Decision-Theoretic Incremental Heuristic Search”. In: *Embedded Reasoning, Papers from the 2010 AAAI Spring Symposium, Technical Report SS-10-04*. 2010.

[143] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R.B. Scherl. “GOLOG: A Logic Programming Language for Dynamic Domains”. In: *J. Log. Program.* 31.1-3 (1997), pp. 59–83. DOI: 10.1016/S0743-1066(96)00121-5. URL: http://dx.doi.org/10.1016/S0743-1066(96)00121-5.

[144] Fei Li and H. V. Jagadish. “Constructing an Interactive Natural Language Interface for Relational Databases”. In: *PVLDB* 8.1 (2014). DOI: 10.14778/2735461.2735468.

[145] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. “SUGILITE: Creating Multimodal Smartphone Automation by Demonstration”. In: *CHI*. ACM, 2017, pp. 6038–6049.

[146] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. “Mining assumptions for synthesis”. In: *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. IEEE, 2011, pp. 43–50. URL: https://doi.org/10.1109/MEMCOD.2011.5970509.

[147] V. Lifschitz and W. Ren. “A Modular Action Description Language”. In: *AAAI*. AAAI Press, 2006, pp. 853–859.

[148] Constantine Lignos, Vasumathi Raman, Cameron Finucane, Mitchell P. Marcus, and Hadas Kress-Gazit. “Provably Correct Reactive Control from Natural Language”. In: *Auton. Robots* 38.1 (2015). DOI: 10.1007/s10514-014-9418-8.

[149] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. “ARA*: Anytime A* with Provable Bounds on Sub-Optimality”. In: *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003*. Ed. by Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf. MIT Press, 2003, pp. 767–774.

[150] Y. Lin and S. Mitra. “StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems”. In: *LCTES*. 2015.

[151] David Lo and Shahar Maoz. “Mining Scenario-Based Triggers and Effects”. In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L’Aquila, Italy*. 2008, pp. 109–118. DOI: 10.1109/ASE.2008.21.

[152] David Lo and Shahar Maoz. “Specification mining of symbolic scenario-based models”. In: *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE’08, Atlanta, Georgia, November 9-10, 2008*. 2008, pp. 29–35. DOI: 10.1145/1512475.1512482.

[153] Christof Löding, P. Madhusudan, and Daniel Neider. “Abstract Learning Frameworks for Synthesis”. In: *TACAS*. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 167–185.

[154] Lingzhi Luo, Nilanjan Chakraborty, and Katia P. Sycara. “Multi-robot assignment algorithm for tasks with set precedence constraints”. In: *ICRA*. IEEE, 2011, pp. 2526–2533.

[155] Hang Ma, Wolfgang Hönig, TK Satish Kumar, Nora Ayanian, and Sven Koenig. “Lifelong path planning with kinematic constraints for multi-agent pickup and delivery”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 7651–7658.

[156] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. “Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks”. In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017*. 2017, pp. 837–845.

[157] Monika Maidl. “The Common Fragment of CTL and LTL”. In: *FOCS 2000, Proceedings*. 2000, pp. 643–652. DOI: 10.1109/SFCS.2000.892332. URL: https://doi.org/10.1109/SFCS.2000.892332.

[158] Rupak Majumdar, Kaushik Mallik, Anne-Kathrin Schmuck, and Damien Zufferey. “Assume-Guarantee Distributed Synthesis”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.11 (2020), pp. 3215–3226.

[159] Rupak Majumdar and Filip Niksic. “Why is random testing effective for partition tolerance bugs?” In: *PACMPL* 2.POPL (2018), 46:1–46:24. DOI: 10.1145/3158134. URL: http://doi.acm.org/10.1145/3158134.

[160] Oded Maler and Dejan Nickovic. “Monitoring Temporal Properties of Continuous Signals”. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*. Vol. 3253. Lecture Notes in Computer Science. Springer, 2004, pp. 152–166. URL: https://doi.org/10.1007/978-3-540-30206-3_12.

[161] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. “The Stanford CoreNLP Natural Language Processing Toolkit”. In: *Association for Computational Linguistics (ACL) System Demonstrations*. 2014.

[162] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. “Integrating Programming by Example and Natural Language Programming”. In: *Conference on Artificial Intelligence (AAAI)*. 2013.

[163] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. "User Interaction Models for Disambiguation in Programming by Example". In: *User Interface Software & Technology (UIST)*. 2015. DOI: 10.1145/2807442.2807459.

[164] Conor McGann, Frederic Py, K Rajan, H Thomas, R Henthorn, and R McEwen. "T-rex: A model-based architecture for auv control". In: *3rd Workshop on Planning and Plan Execution for Real-World Systems*. Vol. 2007. 2007.

[165] Andre Medeiros. *ZooKeeper's atomic broadcast protocol: Theory and practice*. 2012.

[166] Çetin Meriçli, Steven D. Klee, Jack Paparian, and Manuela M. Veloso. "An interactive approach for situated task specification through verbal instructions". In: *Autonomous Agents and Multi-Agent Systems (AAMAS)*. 2014.

[167] George A. Miller. "WordNet: A Lexical Database for English". In: *Commun. ACM* 38.11 (1995). ISSN: 0001-0782. DOI: 10.1145/219717.219748.

[168] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN: 978-0-07-042807-2. URL: http://www.worldcat.org/oclc/61321007.

[169] Salar Moarref and Hadas Kress-Gazit. "Automated synthesis of decentralized controllers for robot swarms from high-level temporal logic specifications". In: *Auton. Robots* 44.3-4 (2020), pp. 585–600.

[170] Sara Mohammadinejad, Jyotirmoy V. Deshmukh, Aniruddh G. Puranic, Marcell Vazquez-Chanlatte, and Alexandre Donzé. "Interpretable classification of time-series data using efficient enumerative techniques". In: *HSCC*. ACM, 2020, 9:1–9:10.

[171] Matthew Molineaux, Matthew Klenk, and David Aha. "Goal-driven autonomy in a Navy strategy simulation". In: *Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010.

[172] António Morgado, Carmine Dodaro, and João Marques-Silva. "Core-Guided MaxSAT with Soft Cardinality Constraints". In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Ed. by Barry O'Sullivan. Vol. 8656. Lecture Notes in Computer Science. Springer, 2014, pp. 564–573. DOI: 10.1007/978-3-319-10428-7_41. URL: https://doi.org/10.1007/978-3-319-10428-7_41.

[173] L. De Moura and N. Bjørner. "Z3: an efficient SMT solver". In: *TACAS*. 2008, pp. 337–340.

[174] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki. "SMT-Based Synthesis of Integrated Task and Motion Plans from Plan Outlines". In: *ICRA*. 2014.

[175] Daniel Neider. "Applications of automata learning in verification and synthesis". PhD thesis. RWTH Aachen University, 2014. URL: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2014/5169.

[176] Daniel Neider. “Computing Minimal Separating DFAs and Regular Invariants Using SAT and SMT Solvers”. In: *ATVA*. Vol. 7561. Lecture Notes in Computer Science. Springer, 2012, pp. 354–369.

[177] Daniel Neider, Jean-Raphaël Gaglione, Ivan Gavran, Ufuk Topcu, Bo Wu, and Zhe Xu. “Advice-Guided Reinforcement Learning in a non-Markovian Environment”. In: *AAAI*. AAAI Press, 2021, pp. 9073–9080.

[178] Daniel Neider and Ivan Gavran. “Learning Linear Temporal Properties”. In: *FMCAD*. IEEE, 2018, pp. 1–10.

[179] Daniel Neider and Nils Jansen. “Regular Model Checking Using Solver Technologies and Automata Learning”. In: *NASA Formal Methods*. Vol. 7871. Lecture Notes in Computer Science. Springer, 2013, pp. 16–31.

[180] Andrew Y. Ng and Stuart J. Russell. “Algorithms for Inverse Reinforcement Learning”. In: *ICML*. Morgan Kaufmann, 2000, pp. 663–670.

[181] Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. “Learning to Infer Program Sketches”. In: *International Conference on Machine Learning (ICML)*. 2019.

[182] José Oncina and Pedro Garcia. “Inferring regular languages in polynomial updated time”. In: *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*. World Scientific. 1992, pp. 49–61.

[183] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Berfrouei, and Georg Weissenbacher. “Randomized Testing of Distributed Systems with Probabilistic Guarantees”. In: *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. to appear. 2018.

[184] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[185] Ronald Parr and Stuart J Russell. “Reinforcement learning with hierarchies of machines”. In: *Advances in neural information processing systems*. 1998, pp. 1043–1049.

[186] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[187] Hila Peleg, Sharon Shoham, and Eran Yahav. "Programming Not Only by Example". In: *International Conference on Software Engineering (ICSE)*. 2018. DOI: 10.1145/3180155.3180189.

[188] Vittorio Perera and Manuela M. Veloso. "Handling Complex Commands as Service Robot Task Requests". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2015.

[189] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[190] Illia Polosukhin and Alexander Skidanov. "Neural Program Search: Solving Programming Tasks from Description and Examples". In: *International Conference on Learning Representations (ICLR)*. 2018.

[191] Oleksandr Polozov and Sumit Gulwani. "FlashMeta: a framework for inductive program synthesis". In: *OOPSLA*. ACM, 2015, pp. 107–126.

[192] community project. *ROS2*. https://index.ros.org/doc/ros2/. Accessed: November 2016.

[193] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009.

[194] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN: 1-55860-238-0.

[195] V. Raman, N. Piterman, and H. Kress-Gazit. "Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations". In: *ICRA*. 2013, pp. 4075–4081.

[196] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M Murray, and Sanjit A Seshia. "Reactive synthesis from signal temporal logic specifications". In: *Proceedings of the 18th international conference on hybrid systems: Computation and control*. 2015, pp. 239–248.

[197] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. "Compositional Program Synthesis from Natural Language and Examples". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2015.

[198] Heinz Riener. "Exact synthesis of LTL properties from traces". In: *2019 Forum for Specification and Design Languages (FDL)*. IEEE. 2019, pp. 1–6.

[199] Ronald L. Rivest and Robert E. Schapire. "Inference of Finite Automata Using Homing Sequences". In: *Inf. Comput.* 103.2 (1993), pp. 299–347.

[200] Wheeler Ruml, Minh Binh Do, Rong Zhou, and Markus P. J. Fromherz. "On-line Planning and Scheduling: An Application to Controlling Modular Printers". In: *Journal Artificial Intelligence Research* 40 (2011), pp. 415–468.

[201] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2009.

[202] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia. “Automated composition of motion primitives for multi-robot systems from safe LTL specifications”. In: *IROS*. IEEE. 2014, pp. 1525–1532.

[203] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia. “Implan: Scalable Incremental Motion Planning for Multi-Robot Systems”. In: *ICCPS*. 2016.

[204] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J. Pappas, and Sanjit A. Seshia. “Implan: Scalable Incremental Motion Planning for Multi-Robot Systems”. In: *ICCPS*. IEEE Computer Society, 2016, 43:1–43:10.

[205] Stanly Samuel, Kaushik Mallik, Anne-Kathrin Schmuck, and Daniel Neider. “Resilient Abstraction-Based Controller Design”. In: *CoRR* abs/2008.06315 (2020).

[206] Sven Schewe and Bernd Finkbeiner. “Bounded Synthesis”. In: *ATVA*. Vol. 4762. Lecture Notes in Computer Science. Springer, 2007, pp. 474–488.

[207] Jeffrey O. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008. ISBN: 978-0-521-86572-2. URL: http://www.cambridge.org/gb/knowledge/isbn/item1173872/?site_locale=en_GB.

[208] Shahaf S. Shperberg, Andrew Coles, Bence Cserna, Erez Karpas, Wheeler Ruml, and Solomon Eyal Shimony. “Allocating Planning Effort When Actions Expire”. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*. 2019, pp. 2371–2378.

[209] Rishabh Singh and Sumit Gulwani. “Predicting a Correct Program in Programming by Example”. In: *CAV (1)*. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 398–414.

[210] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[211] Richard S Sutton, Doina Precup, and Satinder Singh. “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning”. In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[212] Jordan T. Thayer and Wheeler Ruml. “Using Distance Estimates in Heuristic Search”. In: *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*. 2009.

[213] Theano Development Team. “Theano: A Python framework for fast computation of mathematical expressions”. In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[214] Jesse Thomason, Shiqi Zhang, Raymond J. Mooney, and Peter Stone. “Learning to Interpret Natural Language Commands through Human-Robot Dialog”. In: *IJCAI*. AAAI Press, 2015, pp. 1923–1929.

[215] M. Turpin, K. Mohta, N. Michael, and V. Kumar. “Goal Assignment and Trajectory Planning for Large Teams of Aerial Robots”. In: *RSS*. 2013.

[216] Prashant Vaidyanathan, Rachael Ivison, Giuseppe Bombara, Nicholas A. DeLateur, Ron Weiss, Douglas Densmore, and Calin Belta. "Grid-based temporal logic inference". In: *56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, December 12-15, 2017*. 2017, pp. 5354–5359. URL: https://doi.org/10.1109/CDC.2017.8264452.

[217] Leslie G. Valiant. "A Theory of the Learnable". In: *Commun. ACM* 27.11 (1984), pp. 1134–1142. DOI: 10.1145/1968.1972. URL: http://doi.acm.org/10.1145/1968.1972.

[218] Marcell Vazquez-Chanlatte, Susmit Jha, Ashish Tiwari, Mark K. Ho, and Sanjit A. Seshia. "Learning Task Specifications from Demonstrations". In: *Neural Information Processing Systems (NeurIPS)*. 2018.

[219] M. Čáp, P. Novák, M. Selecký, J. Faigl, and J. Vokřínek. "Asynchronous decentralized prioritized planning for coordination in multi-robot system". In: *IROS*. 2013.

[220] P. Velagapudi, K. Sycara, and P. Scerri. "Decentralized prioritized planning in large multirobot teams". In: *IROS*. 2010.

[221] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. "Interactive Query Synthesis from Input-Output Examples". In: *SIGMOD Conference*. 2017.

[222] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. "Synthesizing highly expressive SQL queries from input-output examples". In: *Programming Language Design and Implementation (PLDI)*. 2017.

[223] Sida I. Wang, Samuel Ginn, Percy Liang, and Christopher D. Manning. "Naturalizing a Programming Language via Interactive Learning". In: *ACL (1)*. Association for Computational Linguistics, 2017, pp. 929–938.

[224] Y. Wang, N. T. Dantam, S. Chaudhuri, and L. E. Kavraki. "Task and Motion Policy Synthesis as Liveness Games". In: *ICAPS*. 2016, p. 536.

[225] Andrzej Wasylkowski and Andreas Zeller. "Mining temporal specifications from object usage". In: *Autom. Softw. Eng.* 18.3-4 (2011), pp. 263–292. URL: https://doi.org/10.1007/s10515-011-0084-1.

[226] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

[227] C.A. Webber. *XUDD: A Python Actor Model System*. https://github.com/xudd/xudd. Accessed: March 2016, Branch: master.

[228] K. W. Wong, C. Finucane, and H. Kress-Gazit. "Provably-correct robot control with LTLMoP, OMPL and ROS". In: *IROS*. 2013, pp. 2073–2073. DOI: 10.1109/IROS.2013.6696636.

[229] T. Wongpiromsarn, U. Topcu, and R. M. Murray. "Receding Horizon Temporal Logic Planning". In: *IEEE Trans. Automat. Contr.* (2012).

[230] T. Wongpiromsarn, U. Topcu, and R.M. Murray. “Receding Horizon Temporal Logic Planning”. In: *Trans. on Automatic Control* 57.11 (2012), pp. 2817–2830. ISSN: 0018-9286. DOI: 10.1109/TAC.2012.2195811.

[231] Zhe Xu, Ivan Gavran, Yousef Ahmad, Rupak Majumdar, Daniel Neider, Ufuk Topcu, and Bo Wu. “Joint Inference of Reward Machines and Policies for Reinforcement Learning”. In: *ICAPS*. AAAI Press, 2020, pp. 590–598.

[232] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. “SQLizer: query synthesis from natural language”. In: *PACMPL* 1.OOPSLA (2017). DOI: 10.1145/3133887.

Curriculum Vitae

Research Interests

Program Synthesis, Interpretable Learning, Human-Robot Interaction, Software Verification

Education and employment

2016 – 2021 Doctoral student, Max Planck Institute for Software Systems, Kaiserslautern, Germany.

2013 – 2016 Research Engineer, Bellabeat Inc, Zagreb, Croatia.

2011 – 2013 M.Sc., Mathematics and Computer Science, University of Zagreb, Zagreb, Croatia.

2008 – 2011 B.Sc., Mathematics, University of Zagreb, Zagreb, Croatia.