EFFICIENT TECHNIQUES FOR AUTOMATED PLANNING FOR GOALS IN LINEAR TEMPORAL LOGICS ON FINITE TRACES

FRANCESCO FUGGITTI

A DISSERTATION SUBMITTED TO

THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

YORK UNIVERSITY

TORONTO, ONTARIO

SEPTEMBER 2023

Abstract

One of the greatest challenges of the modern era is to empower AI systems with the ability to deliberate and act autonomously while mitigating the risks that arise from granting such power. To address this challenge, a promising approach is to incorporate behavioral specifications within AI systems using formal languages, especially linear temporal logics. We are interested in efficiently combining temporal logics on finite traces with automated planning, which is an AI model-based approach to producing autonomous behavior and solving the problem of sequential decision-making. Despite the ample literature on the application of linear temporal logics on finite traces, LTL_f and LDL_f, in planning and related fields, limited attention has been given to the study and use of the pure-past linear temporal logics and their potential for specifying temporal goals in planning. Furthermore, the application of temporal logics to other related research areas where planning techniques have been successfully employed, such as business process management and business automation, has been given relatively little focus, and there is a lack of principled research on the topic. In this dissertation, we propose (i) an in-depth study of the pure-past linear temporal logics, (ii) their effective applicability as formal languages to specify temporally extended goals in deterministic and nondeterministic planning, and (iii) the application of planning techniques to solve the declarative trace alignment in business process management while envisioning new methods to solve workflow construction from natural language in business automation. More specifically, we first review the pure-past linear temporal logics, PPLTL and PPLDL, and we show how we can exploit a foundational result on reverse languages to get an exponential improvement over $\text{LTL}_f/\text{LDL}_f$, when computing the corresponding deterministic automata. Given this key result, we introduce an efficient technique to cleverly evaluate the truth of pure-past formulas given the truth value of a small set of subformulas, thus enabling the development of more efficient algorithms. Consequently, in the context of deterministic and nondeterministic planning for pure-past temporally extended goals, we present a novel efficient encoding into standard planning for final-state goals with minimal overhead, and that is at most linear in the size of the goal formula and does not add additional spurious actions. As for declarative trace alignment, we extend process model specifications to full $\text{LTL}_f/\text{LDL}_f$, provide a reduction to cost-optimal planning, and devise new practical encodings. Finally, focusing on the enterprise use of business

ii

automation, we look into the latest techniques in natural language understanding and large language models to translate English instructions to LTL formulas, bridging the gap between the end user and reasoning engines used to construct automatic workflows.

Acknowledgments

We often fall short in expressing our gratitude to others, especially those who profoundly shape our path. In the spirit of sincere acknowledgment, I extend my heartfelt gratitude to the people who have contributed to shaping my personal and professional growth during these years of my Ph.D.

First and foremost, I would like to express my gratitude to my co-supervisors, Giuseppe De Giacomo and Yves Lespérance. I have been incredibly lucky to have had both of you as my supervisors. This dissertation would not have been possible without your expertise, constructive feedback, and support, but, most of all, for your unwavering passion for research that shined through every interaction we had, igniting the same enthusiasm in me.

Next, I would like to thank the other two supervisory committee members, Fabio Patrizi and Franck van Breugel. Your kindness, enlightening conversations, and feedback have been invaluable to me. Your thorough review of early drafts of this dissertation has significantly improved the quality of my work. I would also like to thank the other members of my committee – Blai Bonet, Andrea Marella, and Arik Senderovich – for their feedback and insights that have already sparked ideas and will definitely shape what's next.

Without a doubt, all the projects I worked on during these years benefited from fruitful external collaborations with brilliant people, including Luigi Bonassi, Ramon Fraga Pereira, Alfonso Gerevini, Fabrizio Maggi, Felipe Meneguzzi, Sasha Rubin, and Enrico Scala. Thank you for the countless hours spent discussing ideas and solving problems and for your expertise. These works would not have been as successful without your collaboration.

I have been privileged to work in an amazing lab alongside many talented colleagues and friends who continually motivate and inspire me. I owe a great deal to Marco for our countless conversations that birthed many shared successful projects. The perfect blend of your vast knowledge and humbleness makes you one of the kindest and most talented people I know. A big thank you also goes to Antonio. Without your presence, mentorship, and constructive criticism, the achievements during my Ph.D. would have been likely far less. I would also like to thank Alessandro, Giuseppe, Roberto, and Shufang for their constant willingness to help, technical insights, and shared moments that made the lab an intellectually stimulating environment. I must

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

The ability to make decisions is a critical aspect of human intelligence. Making decisions is vital not only to individuals but also to organizations that rely on them, namely companies, institutions, governments, and users of computer-based systems. Unfortunately, the complexity of decision making can make it challenging to arrive at optimal choices, while poor decisions can have severe consequences. This is why, in the context of computer-based systems, the ultimate goal of the scientific research is to endow such systems with deliberation capabilities. This is among the driving reasons behind the proliferation of Artificial Intelligence (AI) systems. Recent advancements have equipped AI agents with the ability to act autonomously in the surrounding world, that is, without human intervention. At the same time, empowering an AI agent with the ability to self-deliberate its own behavior carries significant risks and so must be balanced with safety measures. Therefore, such an autonomous ability should be guarded by human-guided specifications and oversight, verifiable and comprehensible in human terms and, ultimately, *trustworthy*.

In the last decades, various computational techniques have been proposed to try to address these grand challenges in devising intelligent autonomous systems. To this end, formal languages and logics have stood out as great tools to ensure safety measures of AI agents. In fact, the role of formal languages and logics in Computer Science (CS) and, later, in AI has long been a subject of scientific investigation. The first attempts date back at least to the late '30s when Turing, Church, and Tarski demonstrated the undecidability of first-order logic. However, the real sparkles for the application of logics in CS happened when Church (1957) showed the groundbreaking connection between digital circuits and logic, and when Pnueli (1977) advocated the use of temporal logics for expressing properties of computer programs. These two pioneering works have

given birth to entirely new research fields in CS, such as model checking (Clarke and Emerson, 1981; Quielle and Sifakis, 1981) and reactive system synthesis (Pnueli and Rosner, 1989).

All these applications rely on LTL that is interpreted on infinite sequences of states. However, unlike the original problems in CS, such as the specification of concurrent programs, in AI, tasks are inherently of *finite* nature, meaning that their execution stops after a finite number of steps. Therefore, although AI research started employing LTL, over the years, the AI community has naturally shifted to the use of LTL_f, the *finite* traces variant of LTL. At the beginning, the distinction between interpreting formulas on infinite or finite sequences of states was often blurred but later became more distinctive. Fortunately, focusing on finite-trace variants of LTL also turned out to be advantageous from a practical perspective, given that algorithms for dealing with LTL on infinite traces are computationally challenging (Fogarty et al., 2015), whereas those for finite-trace variants are much better behaved (Tabakov and Vardi, 2005).

Given these remarkable findings, temporal logics on finite traces have been quite favored by the AI and Formal Methods communities. For instance, a non-exhaustive list of works where temporal logics on finite traces have been employed are (De Giacomo and Vardi, 2015, 2016; Zhu et al., 2017; Camacho et al., 2018) for finite temporal synthesis, (Camacho et al., 2017; De Giacomo and Rubin, 2018; Camacho and McIlraith, 2019; Brafman and De Giacomo, 2019a) in the context of fully observable nondeterministic planning, (Lacerda et al., 2015; Brafman et al., 2018; Brafman and De Giacomo, 2019b; De Giacomo et al., 2020b) in the theory of Markov decision processes, (Camacho et al., 2019b; De Giacomo et al., 2019) in the area of reinforcement learning, (De Giacomo et al., 2014, 2016, 2017, 2022) to declaratively specify and monitor business processes, and many others.

Among such applications, a prominent area of AI highly related to the problem of sequential decision-making is that of *Automated Planning* (or simply planning) (Ghallab et al., 2004; Geffner and Bonet, 2013). Planning is the model-based approach to sequential decision-making to generate autonomous behavior. In simpler terms, planning is the research area concerned with the problem for an agent of selecting the actions to do next. Specifically, for a given agent, planning is the problem of finding a sequence of actions mapping a given initial state to a certain goal state, under several configurations of the agent itself and of the environment where the agent acts in. Depending on such configurations, several forms of planning have been identified and studied over the years, ranging from the simplest and basic one (e.g., STRIPS planning (Fikes and Nilsson, 1971)) to the most general one (e.g., decision-theoretic planning (Blythe, 1999) and multi-agent epistemic planning (Bolander and Andersen, 2011)). While basic models make multiple assumptions about the environment (e.g., deterministic, static, fully observable, etc.), some planning problems do not satisfy these strict assumptions. Thus, other variants and extensions of simple forms of planning have been considered. For instance, the model that an agent has of the surrounding environment may be flawed

and imprecise, or there might be unexpected and unpredictable events/changes that need to be taken into account.

An interesting extension of standard planning considers goals that do not refer only to reaching a final goal state but also to intermediate states. Clearly, achieving these kinds of goals is a more general task and faithfully accounts for many real-world scenarios. Indeed, in most cases, realistic goals require properties to hold over time on a sequence of planning states. In the literature, these types of temporal goals are known as *temporally extended goals*, and are often expressed using formal languages that specify properties on finite or infinite sequences of states, including LTL, its variant interpreted on finite sequences LTL$_f$, the extension of LTL$_f$ to regular expressions Linear-time Dynamic Logic on finite sequences (LDL$_f$) (De Giacomo and Vardi, 2013), the Property Specification Language (PSL) (Eisner and Fisman, 2006), and the Computation Tree Logic (CTL) (Clarke and Emerson, 1981), to name a few.

Another reason underpinning the growing popularity of automated planning is represented by its effective use in many applications. Indeed, despite its high computational complexity, classical planning can be efficiently solved by modern heuristic search-based algorithms for most real-world problem instances. For instance, among others, planning has been successfully applied in a number of domains ranging from penetration testing in cybersecurity (Shmaryahu et al., 2018), robotics (Cashmore et al., 2015), web service composition (Chakraborti et al., 2022), building automation solutions for the enterprise (Vukovic et al., 2019), and business process management (De Giacomo et al., 2016, 2017). The latter application is of particular interest in the context of analyzing traces of events produced by a process under execution. In this respect, planning allows scalable solutions to the alignment of traces of events that do not comply with the specified declarative models expressed using finite-trace temporal logics, such as LTL$_f$ (De Giacomo et al., 2017).

In this dissertation, we aim to advance the development of efficient techniques for planning with temporally extended goals expressed in linear temporal logics on finite traces and enable a more effective application of planning within the business process management and business automation areas.

1.2 Motivation

Given the context, the work presented in this dissertation has been motivated by the following gaps identified in the literature. All recent works dealing with linear temporal logics on finite traces have studied and applied formalisms referring only to the present and future. Thus, especially in the AI community, the use of past versions of LTL$_f$ and LDL$_f$, namely Pure-Past Linear Temporal Logic (PPLTL) and Pure-Past Linear Dynamic Logic (PPLDL), has been given only limited attention. However, as pointed out in (Lichtenstein et al., 1985),

in some cases, it is easier and more natural to express properties by referring to the past. For instance, the use of past temporal logics has been advocated in some applications for non-Markovian rewards in Markov decision processes (Bacchus et al., 1996), for non-Markovian models in reasoning about actions (Gabaldon, 2011), for preferred explanations in the context of dynamical diagnosis (Sohrabi et al., 2011), for normative properties in multi-agent systems (Fisher and Wooldridge, 2005; Knobbout et al., 2016; Alechina et al., 2018), and for synthesis (Cimatti et al., 2020). Historically, except for the works just mentioned, PPLTL and PPLDL have been introduced only as a technical means to get results for both LTL and LTL$_f$ (Maler and Pnueli, 1990; Zhu et al., 2019). Moreover, the use of PPLTL and PPLDL could bring great benefits in terms of computational complexity, given their native backward interpretation of traces that simplifies the construction of the corresponding automaton (Chandra et al., 1981).

Looking at temporal logics on finite traces to specify sophisticated goals in planning, different approaches have been developed to solve planning for temporally extended goals in both deterministic and nondeterministic domains. The main idea underlying nearly all such approaches is to compile temporally extended goals into standard "reachability" goals, i.e., into final-state goals.

In deterministic domains, existing encodings of LTL$_f$ into classical planning are either worst-case exponential (Baier and McIlraith, 2006a) or significantly increase the plan length (Torres and Baier, 2015). Additionally, in the nondeterministic setting, state-of-the-art encodings (Camacho et al., 2017; Camacho and McIlraith, 2019) are worst-case exponential for dealing with the stochastic and adversarial nondeterminism, and, for this reason, introduce a lot of bookkeeping machinery that dramatically affects performances. Hence, in both settings, there is still much room for improvement.

Recently, given the good performances shown, planning techniques have been employed in many other closely related fields, including business process management and business automation for the enterprise. Nevertheless, the application of temporal logics to these research areas has attracted comparatively little attention, and there is limited principled work on the topic.

Planning has been particularly fruitful in Business Process Management (BPM), where an interesting problem is the one called trace alignment, also known as conformance checking. The trace alignment problem consists in "aligning" the execution trace of a generic process (e.g., an agent's behavior execution) with minimal-cost modifications in such a way that the resulting trace/execution conforms to a given specification, usually expressed in temporal or dynamic logics on finite traces.

To solve the trace alignment problem, state-of-the-art approaches make use of automated planners. However, existing techniques either focus only on a restricted set of predefined template modeling formulas (i.e. DECLARE (van der Aalst et al., 2009)), which is limited in expressiveness, or do not terminate, as they are based on ad-hoc implementations of the A* algorithm, which do not scale well as the input complexity

increases.

Finally, still in the context of business processes, another research area of prominent application and interest is the development of workflow construction systems for business automation. In general, workflow construction systems are concerned with streamlining routine business processes to improve efficiency. This is often achieved through the automation and integration of various services. However, one of the major obstacles is the high barrier of entry for end users in terms of the expected expertise when writing flow construction specifications. This is among the reasons at the base of the recent use of Natural Language interfaces as means of user input in such applications.

In this setting, there has been limited work on the application of planning for goals expressed using temporal logics as a substrate for workflow specification. Currently, state-of-the-art applications like (Brachman et al., 2022) depend on parser-specific code to look for specific patterns requested by the user. Therefore, significant developer overhead is required to investigate the particular parser used (e.g., parse trees from Abstract Syntax Representations (Astudillo et al., 2020)) to write code for it that is limited in scope (limited by human effort) while eventually producing code is not reusable once the system upgrades to a different parser.

1.3 Dissertation Outline and Contributions

We now present an overview of the contributions of this dissertation divided by chapter. Chapter 2 is a preliminary chapter and provides the necessary background on finite automata theory, linear temporal logics on infinite and finite traces, and automated planning. Chapters 3 to 6 are the core contributions of this dissertation. In these chapters, we focus on pure-past linear temporal logics and their application as formal languages to specify temporally extended goals in deterministic and nondeterministic planning. Then, Chapters 7 and 8 explore the application of automated planning to (i) solve declarative trace alignment in business process management and (ii) help solving workflow construction from natural language instructions. Finally, Chapter 9 includes a summary of the contributions of the dissertation, their impact, and directions for future work.

Let us outline these in more detail.

Chapter 2: Preliminaries

This chapter reviews the relevant background necessary for this dissertation. First, we review finite automata theory, describing the main types of automata on finite words: alternating (AFA), nondeterministic (NFA), and deterministic (DFA). Second, we provide an overview of the linear temporal logics on infinite and finite

traces, namely LTL, LTL_f, and LDL_f. We also provide background on the translation of such temporal logics to automata, which is key to understanding the techniques we later exploit in the dissertation. Finally, we formally define and explain the problem of automated planning in classical settings, nondeterministic settings and what is the role played by temporally extended goals.

Chapter 3: Pure-Past Linear Temporal Logics

In this chapter, we review PPLTL and PPLDL, the pure-past versions of the well-known logics on finite traces LTL_f and LDL_f, respectively. Beyond reviewing the main properties of PPLTL and PPLDL, we show how we can exploit a foundational result on reverse languages from (Chandra et al., 1981) to get an exponential improvement over $\text{LTL}_f/\text{LDL}_f$, when computing the corresponding deterministic automata. To do so, we provide an in-depth theoretical discussion explaining why PPLTL and PPLDL formulas can be translated into their corresponding deterministic automata in worst-case single exponential time and present a concrete algorithm. Moreover, given this key result, we are able to fully formalize the reasoning and expressive power of both PPLTL and PPLDL. In particular, we establish that PPLTL (resp., PPLDL) has the same expressive power as LTL_f (resp., LDL_f), but also observe that transforming a PPLTL (resp., PPLDL) formula into its equivalent LTL_f (resp., LDL_f) is computationally prohibitive. Then, we provide translations in PPLTL of commonly used PDDL3 modal operators and of DECLARE patterns, showing how pure-past formulas can be helpful in expressing properties over time. Finally, we give an overview of the impact that the exponential improvement has of employing pure-past temporal logics on well-known sequential decision-making problems involving temporal specifications, such as planning and decision problems in nondeterministic and non-Markovian domains, compared to LTL_f and LDL_f.

Chapter 4: Handling Pure-Past Linear Temporal Logic Formulas

In this chapter, we formally develop the theoretical foundations to efficiently handle and evaluate PPLTL formulas. Specifically, we exploit the well-known fixpoint characterization of temporal logics formulas on finite traces to characterize when PPLTL formulas become true. From the fixpoint characterization, we observe that (i) given a PPLTL formula, we only need the truth value of its subformulas to evaluate it, (ii) every PPLTL formula can be put in a form where its evaluation depends on the current propositional evaluation and the evaluation of a key set of PPLTL subformulas at the previous instant, and (iii) we can recursively compute and keep the value of such a small set of subformulas as additional propositional variables in the application domain. Therefore, we devise a way to exploit these observations, and formally prove that they suffice to establish the evaluation of a given PPLTL formula. We also provide comprehensive examples showing how the novel evaluation technique is effective and how it relates to automata. Lastly, this novel

evaluation technique enables the development of more efficient algorithms, including the ones about planning for temporally extended goals that we study in Chapters 5 and 6.

Chapter 5: Classical Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic

In this chapter, we explore classical planning for temporally extended goals expressed in PPLTL. We show that although PPLTL is as expressive as LTL$_f$, it is computationally much more well-behaved for planning. Specifically, by exploiting the novel evaluation technique of Chapter 4, we show that planning for PPLTL goals can be encoded into classical planning with minimal overhead, introducing only a number of new fluents that is at most *linear* in the PPLTL goal and no spurious additional actions. Moreover, we formally prove the correctness of the approach. We also implemented an open-source system called Plan4Past, which can be used along with state-of-the-art classical planners to solve the task. An empirical analysis demonstrates the practical effectiveness of Plan4Past, showing that a classical planner generally performs better with our compilation method than with other existing compilation methods for LTL$_f$ goals over the considered benchmarks.

Chapter 6: FOND Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic

In this chapter, we study nondeterministic planning (FOND) for temporally extended goals expressed in PPLTL. Although PPLTL is as expressive as LTL$_f$, FOND planning for PPLTL goals remains EXPTIME-complete, as for standard FOND planning, instead of becoming 2EXPTIME-complete as when LTL$_f$ is used. As for classical planning, we introduce a direct novel technique for FOND planning for PPLTL goals that is optimal with respect to computational complexity and is also effective in practice. In particular, we present a notably simple encoding of FOND planning for PPLTL goals into standard FOND planning for reachability goals, which is based on the observations made in Chapter 4. As was the case for classical planning examined in Chapter 5, the new encoding only introduces few fluents (at most linear in the PPLTL goal) without adding any spurious actions and allows planners to lazily build the relevant part of the deterministic automaton for the goal formula on-the-fly during the planning search. Additionally, we provide an encoding variant that does not introduce derived predicates, which are not well supported by FOND planners. Such an encoding variant is still polynomially related to the size of the PPLTL goal formula. We formally prove the correctness of the approach and implement both encodings in the open-source system Plan4Past, which can be used along with state-of-the-art FOND planners. Finally, we provide experimental evidence of its effectiveness when compared to existing approaches that handle LTL$_f$ temporally extended goals.

7

Chapter 7: Declarative Trace Alignment via Automated Planning

In this chapter, we consider the problem of *declarative* trace alignment, where the process model is a formal specification expressed by LTL_f or LDL_f formulas. After describing the problem of trace alignment in business process management, we provide a formally correct reduction of such a problem to cost-optimal planning that allows us to resort to state-of-the-art automated planners. Then, we devise two different PDDL encodings solving the trace alignment problem along with several optimizations to increase performance and scalability. We implement such encodings in an open-source tool called TraceAligner, which is currently the best-performing tool to align log traces. TraceAligner is released under the MIT license. Finally, we empirically evaluate TraceAligner on a set of extensive benchmarks, showing that our approach significantly outperforms existing ad-hoc solutions.

Chapter 8: Natural Language to Flow Construction via Automated Planning

In this chapter, we describe a practical industrial application combining some of the topics we have previously tackled in the dissertation. In particular, we give an overview of our newly released Python package NL2LTL that leverages the latest techniques in natural language understanding (NLU) and large language models (LLMs) to translate English inputs to LTL formulas. Such an interface allows direct translation to formal languages that a reasoning system can use while, at the same time, allowing the end user to provide inputs in natural language without having to understand the details of the underlying formal language in a system. The package comes with support for a set of default LTL patterns, corresponding to popular DECLARE templates, but is also fully extensible to new domains so adopters of the package can configure it to their needs. The package has been open-sourced and is free to use for the AI community under the MIT license. Finally, we provide a glimpse of the envisioned industrial use of NL2LTL for business automation.

Chapter 9: Conclusions

This chapter concludes the dissertation by providing a summary of the contributions presented. We also provide an overview of the impact that our work has already had on the AI and Formal Methods communities. Finally, we describe the most relevant open research questions that remain to be explored.

Collaborations

All the works presented in this dissertation were developed together with several collaborators. Besides contributing with ideas and discussions, my specific contributions to each work are highlighted in the following paragraphs.

My specific contribution to the joint work in Chapter 3 has concerned the development of the algorithm to transform a PPLTL (resp., PPLDL) formula into a DFA in single exponential time and most of the theoretical results on the relationship between PPLTL/PPLDL and all other related formalisms, including first-order logic, regular expressions, and $\text{LTL}_f/\text{LDL}_f$.

For the works in Chapters 4, 5, and 6, my specific contribution has been both theoretical and practical. On the one hand, I developed the theoretical foundations of the technique, devised the compact encoding, and proved its correctness. On the other hand, I am the main author and contributor of the implementation of the novel encoding in Plan4Past.

In the work in Chapter 7, my contribution has been both theoretical and practical. While I partially contributed to the extension of some theoretical results presented before in (De Giacomo et al., 2017), I devised several new encodings in PDDL independently. From a practical perspective, I am the main author and contributor of the implementation of new encodings in TraceAligner, which I experimentally tested.

Lastly, the work in Chapter 8 was started, and a significant portion of it completed, during an internship in the AI Composition Lab at IBM Research, Cambridge (USA) in Summer 2022, mentored by Tathagata Chakraborti. My specific contribution to this joint work has concerned theoretical and practical aspects, ranging from the problem definition to the implementation.

1.4 Publications

The findings in this dissertation are based on several publications presented, accepted, or to be submitted to premier conferences or journals in Artificial Intelligence. In chronological order:

- Giuseppe De Giacomo, Antonio Di Stasio, Francesco Fuggitti, and Sasha Rubin. *Pure-Past Linear Temporal and Dynamic Logic on Finite Traces*. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence* (IJCAI), pages 4959–4965, Survey Track, 2020.

- Luigi Bonassi, Giuseppe De Giacomo, Marco Favorito, Francesco Fuggitti, Alfonso Emilio Gerevini, and Enrico Scala. *Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic*. In *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling* (ICAPS), pages 61–69, 2023.

  - *This work received the* ***Best Student Paper Award****.*

- Francesco Fuggitti and Tathagata Chakraborti. *NL2LTL – A Python Package for Converting Natural Language (NL) Instructions to Linear Temporal Logic (LTL) Formulas*. In *Proceedings of the Thirty-*

Third International Conference on Automated Planning and Scheduling (ICAPS), System Demonstration Track, 2023, GitHub: `https://github.com/IBM/nl2ltl`.

- − *This work received the* **Best System Demonstration Award Runner-Up***.*

- Luigi Bonassi, Giuseppe De Giacomo, Marco Favorito, Francesco Fuggitti, Alfonso Emilio Gerevini, and Enrico Scala. *FOND Planning for Pure-Past Linear Temporal Logic Goals*. In *Proceedings of the Twenty-Sixth European Conference on Artificial Intelligence* (ECAI), pages 279–286, 2023.

- Giuseppe De Giacomo, Francesco Fuggitti, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. *Declarative Trace Alignment via Automated Planning*. To be submitted to the *Journal of Artificial Intelligence Research*, 2023.

My doctoral work has also partly been devoted to developing and maintaining research tools, which have been published and presented in the system demonstration tracks of premier conferences on Artificial Intelligence or published in software journals. In chronological order:

- Francesco Fuggitti. *LTLf2DFA*. In *Zenodo Repository*, 2019. Doi: *10.5281/zenodo.3888410*, GitHub: `https://github.com/whitemech/LTLf2DFA`, Website: `http://ltlf2dfa.diag.uniroma1.it`.

- Francesco Fuggitti. *FOND4LTLf*. In *Zenodo Repository*, 2021. Doi: *10.5281/zenodo.4876281*, GitHub: `https://github.com/whitemech/FOND4LTLf`.

- Giuseppe De Giacomo and Francesco Fuggitti. *FOND4LTL$_f$: FOND Planning for* LTL$_f$/PPLTL *Goals as a Service*. In *Proceedings of the International Conference on Autonomous Planning and Scheduling* (ICAPS), System Demonstration Track, 2021.

- Marco Favorito and Francesco Fuggitti. *pddl*. In the official *AI-Planning* GitHub organization, 2022. GitHub: `https://github.com/AI-Planning/pddl`.

- Tathagata Chakraborti, Yara Rizk, Vatche Isahagian, Burak Aksar, and Francesco Fuggitti. *From Natural Language to Workflows: Towards Emergent Intelligence in Robotic Process Automation*. In *Proceedings of the Robotic Process Automation Forum at the Business Process Management Conference* (RPA @ BPM), pages 123–137, 2022.

- Francesco Fuggitti and Tathagata Chakraborti. *NL2LTL – A Python Package for Converting Natural Language (NL) Instructions to Linear Temporal Logic (LTL) Formulas*. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence* (AAAI), pages 16428–16430, System Demonstration Track, 2023, GitHub: `https://github.com/IBM/nl2ltl`.

- Giuseppe De Giacomo, Francesco Fuggitti, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. *A Tool for Declarative Trace Alignment via Automated Planning.* In the *Journal of Software Impacts*, volume 16, May 2023, GitHub: `https://github.com/whitemech/TraceAligner`.

Chapter 2

Preliminaries

In the following sections, we will briefly introduce some of the basic notations and concepts pertaining to finite automata theory, temporal logic formalisms, and automated planning that will be used throughout the rest of the dissertation.

2.1 Finite Automata Theory

Finite-state automata are one of the most fundamental mathematical models of computation in Computer Science. For this reason, finite-state automata are often at the basis of theoretical developments as they represent computational devices, which can be implemented with hardware or software to simulate digital circuits or computer programs (Hopcroft et al., 2001).

2.1.1 Regular Languages

There is a tight connection between automata theory and formal languages. In fact, alphabets, strings, and languages are among the basic concepts revolving around automata theory. Following (Hopcroft et al., 2001), an *alphabet* is a finite, nonempty set of symbols that is commonly denoted as Σ. A *string* (or simply *word*) represents a finite sequence of symbols drawn from an alphabet Σ. The *empty string* is defined as a string without any occurrences of symbols, and it is usually denoted as ϵ. The length of a word w is denoted as $|w|$. The notation Σ^* defines the set of all possible strings, including the empty string, that can be formed using the symbols in an alphabet Σ. Finally, a set of words chosen from some alphabet Σ^* is called a *language*. Given an alphabet Σ and a language $\mathcal{L} \subseteq \Sigma^*$, we say that $\mathcal{L}$ is a language over Σ. Observe that, although languages may have an infinite number of words, the alphabet over which they are defined must be finite.

Regular languages are a specific class of languages that can be expressed with a regular expression (RE)

or a finite automaton. In this way, automata can be seen as devices to tell whether a certain language is *recognized* (i.e., accepted) or not. Also, finite automata are often classified based on the recognized language as, e.g., in the well-known Chomsky hierarchy (Chomsky, 1956).

2.1.2 Deterministic Finite-state Automata

Given a finite nonempty alphabet Σ, a finite word is an element of Σ^*, i.e., a finite sequence $\sigma_0, \ldots, \sigma_n$ of symbols from Σ. Automata on finite words define (finitary) languages, namely, sets of finite words. Following (Vardi, 1996), we recall the definitions of different types of finite-word automata.

A *Deterministic Finite-state Automaton* (DFA) is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where (i) Σ is a finite input *alphabet*; (ii) Q is a finite non-empty set of *states*; (iii) $q_0 \in Q$ is the *initial state*; *(iv)* $\delta : Q \times \Sigma \to Q$ is the *transition function* that prescribes how the automaton moves from a state q to a state q' while reading a symbol σ; *(v)* $F \subseteq Q$ is the set of *accepting states*.

A *run* of an automaton $\mathcal{A}$ on a finite word $w = \sigma_0, \ldots, \sigma_n \in \Sigma^*$ is a finite sequence of states $q_0, \ldots, q_n \in Q$ such that q_0 is the initial state and $q_{i+1} = \delta(q_i, \sigma)$ for $0 \le i < n$. Consequently, deterministic automata have a unique run on a single input. Also, for convenience, when the transition function is deterministic, the transition function can be generalized to a function of the form $\delta^* : Q \times \Sigma^* \to Q$ for sequences of symbols in Σ inductively as $\delta^*(q, \epsilon) = q$ for empty strings and $\delta^*(q, w\sigma) = \delta(\delta^*(q, w), \sigma)$ for any state $q \in Q$, any word $w \in \Sigma^*$ and symbols $\sigma \in \Sigma$. A run on a word w is *accepting* if $\delta^*(q_0, w) \in F$, and the set of words accepted by $\mathcal{A}$, written as $\mathcal{L}(\mathcal{A})$, is called the *language of* $\mathcal{A}$.

Generally, an automaton is represented as an edge-labeled directed graph, where nodes represent automaton states; edges are labeled by symbols in Σ; there is an initial state; and some set of states is considered accepting. A graphical representation of a deterministic automaton is depicted in Figure 2.1. If the transition function is defined for all states in Q and all symbols in Σ (i.e., it is a total function), the DFA is said to be *complete*. An arbitrary DFA can always be completed by introducing a dummy *sink* state and updating the definition of the transition function to the sink state when previously undefined. This procedure is called *completion*.

In general, automata support structural operations. Structural operations change the structure of the automata without modifying the recognized language. Given a DFA $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, there exists a *unique* equivalent *minimal* DFA $\mathcal{A}_m$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_m)$ and $|Q_m| \le |Q|$ and there is no other DFA $\mathcal{A}'$ accepting the same language with fewer states than $|Q_m|$ (modulo state renaming). Moreover, any DFA can be minimized through the well-known *minimization* algorithm. The best complexity known for the minimization algorithm is $\mathcal{O}(n \log n)$, where n is the number of states of the automaton (Hopcroft et al.,
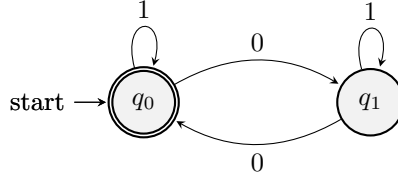
Figure 2.1: A graphical representation of a deterministic finite-state automaton that accepts strings of zeros and ones with an even number of zeros. Circles represent automaton states; double circles represent those automaton states that are accepting. Arrows are labeled with symbols and represent automaton transitions. The initial state is the circle with an incoming edge labeled with "start".

2001).

Another way to simplify an automaton is the *trimming* procedure. Basically, the trimming procedure removes all the unreachable states from the initial state and all the states that do not reach an accepting state, thus leaving only those states that are reachable from the initial state and lead to an accepting state.

Boolean Operations on Deterministic Automata

Deterministic automata are closed under Boolean operations: *complementation*, *intersection*, and *union*. In particular, the complement of a complete DFA $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$ is denoted as $\bar{\mathcal{A}} = \langle \Sigma, Q, q_0, \delta, Q - F \rangle$. Basically, the completion of $\mathcal{A}$ consists of inverting the acceptance condition. Thus, $\bar{\mathcal{A}}$ accepts only the words that are not accepted by $\mathcal{A}$, formally, $\mathcal{L}(\bar{\mathcal{A}}) = \Sigma^* - \mathcal{L}(\mathcal{A})$.

Given two deterministic complete automata $\mathcal{A}_1 = \langle \Sigma, Q_1, q_0^1, \delta_1, F_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, q_0^2, \delta_2, F_2 \rangle$ defined over the same alphabet Σ, their product under one of the Boolean binary operator $\odot \in \{\wedge, \vee\}$ is a DFA $\mathcal{A}_\odot = \langle \Sigma, Q_1 \times Q_2, (q_0^1, q_0^2), \delta, F' \rangle$, where the new transition function δ is defined as $\delta((q_1, q_2), \sigma) = (q_1', q_2')$ if and only if $q_1' = \delta_1(q_1, \sigma)$ and $q_2' = \delta_2(q_2, \sigma)$ and $F' = \{(q_1, q_2) \mid q_1 \in F_1 \odot q_2 \in F_2\}$. Intuitively, the new transition function δ prescribes the simultaneous execution of the two automata. In the case of intersection, the acceptance condition specifies that $\mathcal{A}_\cap$ accepts a word w if both $\mathcal{A}_1$ and $\mathcal{A}_2$ accept w. Thus, the language $\mathcal{L}(\mathcal{A}_\cap)$ recognized by $\mathcal{A}_\cap$ is formally defined as $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. On the contrary, in the case of union, the acceptance condition specifies that $\mathcal{A}_\cup$ accepts a word w if at least one of the two DFAs $\mathcal{A}_1$ or $\mathcal{A}_2$ accepts w. Formally, $\mathcal{L}(\mathcal{A}_\cup) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$.

2.1.3 Nondeterministic Finite-state Automata

A *Nondeterministic Finite-state Automaton* (NFA) is a tuple defined exactly as a DFA except for the transition δ. In the case of NFAs, the transition function δ becomes a transition relation as $\delta : Q \times \Sigma \times Q$. Intuitively, a transition can be seen as a triplet $(q, \sigma, q') \in \delta$. For nondeterministic automata, there can be two triplets $(q, \sigma, q') \in \delta$ and $(q, \sigma, q'') \in \delta$ with $q' \neq q''$. In other words, in nondeterministic automata, for every state

and symbol, there can be multiple possible transitions. Alternatively, a transition function δ returning sets of states can be defined as $\delta(q, \sigma) = \{q' \mid (q, \sigma, q') \in \delta\}$. In this way, deterministic automata can be considered a special case of nondeterministic automata (i.e., when $|\delta(q, \sigma)| = 1$, for all $q \in Q$ and $\sigma \in \Sigma$). As for DFAs, a *run* of an automaton $\mathcal{A}$ on a finite word $w = \sigma_0, \ldots, \sigma_n \in \Sigma^*$ is a finite sequence of states $q_0, \ldots, q_n \in Q$ such that q_0 is the initial state and $(q, \sigma, q_{i+1}) \in \delta$ for $0 \leq i < n$. The run is *accepting* when $q_n \in F$. However, unlike DFAs, an NFA can have multiple runs on a specified input word. Thus, a word w is *accepted* by $\mathcal{A}$ if there *exists* an accepting run of $\mathcal{A}$ on w.

Surprisingly, every language that can be represented by an NFA can also be described by some DFA. In fact, an NFA $\mathcal{A}_n$ can be converted to an equivalent DFA $\mathcal{A}_d$ recognizing the same language (i.e., $\mathcal{L}(\mathcal{A}_n) = \mathcal{L}(\mathcal{A}_d)$) through the so-called *subset construction* (Rabin and Scott, 1959). Intuitively, the subset construction involves building all subsets of the set of states of an NFA. The subset construction is also known as the process of *determinization*. Generally, an NFA characterized by n states (i.e., $|Q| = n$) can be determinized, getting a DFA that is exponentially larger than the NFA, namely with 2^n states. Hence, although both NFAs and DFAs are equally expressive formalisms, NFAs are exponentially more succinct than DFAs. However, empirical evidence has demonstrated that oftentimes, the size of the resulting DFA is *comparable* to or even *smaller* (after minimization) than the original NFA (Tabakov and Vardi, 2005).

Often, in automata theory, a nondeterministic automaton is said to be *nonempty* if $\mathcal{L}(\mathcal{A}) \neq \emptyset$, whereas it is *nonuniversal* when $\mathcal{L}(\mathcal{A}) \neq \Sigma^*$. Checking whether an automaton is nonempty or nonuniversal is one of the most common algorithmic techniques employed in automata theory. For instance, as we will see later, the algorithm for the satisfiability of temporal logic is based on a nonemptiness test.

Boolean Operations on Nondeterministic Automata

As DFAs, nondeterministic automata are closed under intersection and union. Given two nondeterministic automata $\mathcal{A}_1 = \langle \Sigma, Q_1, q_0^1, \delta_1, F_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, q_0^2, \delta_2, F_2 \rangle$ defined over the same alphabet Σ, their intersection is an NFA $\mathcal{A}_\cap$ that simultaneously runs both $\mathcal{A}_1$ and $\mathcal{A}_2$ on the input word. Specifically, the intersection $\mathcal{A}_\cap$ is a tuple $\langle \Sigma, Q_1 \times Q_2, (q_0^1, q_0^2), \delta, F_1 \times F_2 \rangle$, where δ is defined as $((q, t), \sigma, (q', t')) \in \delta$ if and only if $(q, \sigma, q') \in \delta_1$ and $(t, \sigma, t') \in \delta_2$. As in the case of DFAs, the language $\mathcal{L}(\mathcal{A}_\cap)$ recognized by $\mathcal{A}_\cap$ is formally defined as $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

Similarly, given two NFAs as previously defined, the union of these two nondeterministic automata is an NFA $\mathcal{A}_\cup = \langle \Sigma, Q_1 \cup Q_2, q_0, \delta, F_1 \cup F_2 \rangle$, where $\delta = \delta_1 \cup \delta_2$, that is $(q, \sigma, q') \in \delta_1$ if $q \in Q_1$, otherwise $(q, \sigma, q') \in \delta_2$ if $q \in Q_2$, and q_0 is the new starting state. In other words, the union $\mathcal{A}_\cup$ is simply built by introducing a new starting state q_0 with two ϵ-transitions from q_0 to q_0^1 and q_0^2, respectively. In general, ϵ-transitions generalize the definition of NFA, allowing the automaton to have transitions that do not consume any input

symbol. Every NFA with ϵ-transitions can be easily converted to a standard NFA. Moreover, without loss of generality, we assume Q_1 and Q_2 are disjoint. Analogously, the language $\mathcal{L}(\mathcal{A}_\cup)$ recognized by $\mathcal{A}_\cup$ is $\mathcal{L}(\mathcal{A}_\cup) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$.

Regarding complementation, while complementing the acceptance condition suffice for a DFA, since an NFA can have many runs on a given input word, all runs should reject such an input word. Thus, complementing an NFA requires to determinize it first.

2.1.4 Alternating Finite-state Automata

An *Alternating Finite-state Automaton* (AFA) is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where *(i)* Σ is a finite input *alphabet*; *(ii)* Q is a finite non-empty set of *states*; *(iii)* $q_0 \in Q$ is the *initial state*; *(iv)* $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is the *transition function*, where $\mathcal{B}^+(Q)$ is the set of positive Boolean formulas over Q (i.e., built from states of Q using $\wedge$, $\vee$, and the constants *true* and *false*); *(v)* $F \subseteq Q$ is the set of *accepting states*. While nondeterministic automata have the power of existential choice over transitions, alternating automata are computational models with the power of both existential and universal choice over transitions (Brzozowski and Leiss, 1980; Chandra et al., 1981). In fact, the automaton transition function δ can be generally represented with formulas from $\mathcal{B}^+(Q)$, commonly denoted with θ. For instance, the transition $\delta(q, \sigma) = \{q_1, q_2, q_3\}$ of an NFA, can be rewritten as $\delta(q, \sigma) = q_1 \vee q_2 \vee q_3$. While in alternating automata $\delta(q, \sigma)$ can be any arbitrary formula from $\mathcal{B}^+(Q)$, in NFAs no transition uses the $\wedge$ connective, and in DFAs no transition uses the $\vee$ connective. An example of a possible transition of an AFA is $\delta(q_1, \sigma) = q_2 \vee (q_3 \wedge q_4)$, meaning that the automaton accepts the input σw, with σ is a symbol and w is a word, from state q_1 when it accepts the input w from q_2 or from both q_3 and q_4.

The possibility of universal choice in AFA transitions changes the definition of *run* for AFAs. While a run for DFAs and NFAs is defined as a sequence of states, a run for AFAs is a *tree*.

A *tree* T is a directed connected acyclic graph with a (finite or infinite) collection of nodes, which we denote as $\mathsf{nodes}(T)$. In general, in the structure of a tree, one node is called *root*, and it is denoted by ε. Every non-root node has a unique *parent* – we say that s is the *parent* of t, and t is a *child* of s if there is an edge from s to t – and the root ε has no parent. Each node can have an arbitrary number of children, and nodes without any children are called *leaves*. The *level* of a node x (denoted by $\mathsf{level}(x)$) represents the shortest distance or depth of x from the root ε. Obviously, $\mathsf{level}(\varepsilon) = 0$. A *branch* $\beta = x_0, x_1, \ldots$ of a tree is a (finite or infinite) maximal sequence of nodes such that x_0 is the root ε and x_i is the parent of x_{i+1} for all $i \geq 0$. Given a finite alphabet Σ, we define a Σ-*labeled tree* a pair $(T, \mathcal{T})$ of a tree T and a mapping $\mathcal{T}$ from $\mathsf{nodes}(T)$ to Σ assigning a symbol in Σ to every node of T. In this way, $\mathcal{T}$ is often referred to as the

labeled tree. A branch $\beta = x_0, x_1, \ldots$ of $\mathcal{T}$ defines a word $\mathcal{T}(\beta) = \mathcal{T}(x_0), \mathcal{T}(x_1), \ldots$ comprising the sequence of labels assigned along the branch.

Therefore, a run of an AFA $\mathcal{A}$ on a finite word $w = \sigma_0, \ldots, \sigma_n$ can be seen as a finite Q-labeled tree R such that $R(\varepsilon) = q_0$ and given a node x, if $\mathsf{level}(x) = i < n$ with $R(x) = q$ and $\delta(q, \sigma) = \theta$, then the node x has k children $x_1, \ldots, x_k$ for some $k \leq |Q|$, and $\{R(x_1), \ldots, R(x_k)\}$ satisfies θ.

For instance, if $\delta(q_0, \sigma_0) = (q_1 \vee q_2) \wedge (q_3 \vee q_4)$, then the nodes at the first level of the run-tree comprise either q_1 or q_2 and also include q_3 or q_4. The depth of R is at most n, but not all branches need to reach such depth because if $\delta(R(x), \sigma_i) = true$, then the node x does not need to have any children. On the other hand, if $|x| = i < n$ and $R(x) = q$, then we cannot have $\delta(q, \sigma_i) = false$, since false is not satisfiable. A run-tree is *accepting* if all nodes at depth n are labeled by states in F.

Another way to define the acceptance of a word for an AFA is given in terms of a function $Acc : \Sigma^* \to 2^Q$, where $q \in Acc(w)$ is read "input w is accepted from state q". The function Acc is inductively defined as (i) $Acc(\epsilon) = F$, meaning that the empty string ϵ is accepted by the set of final states F; and (ii) $q \in Acc(\sigma w)$ if and only if $V \models \delta(q, \sigma)$ for some $V \subseteq Acc(w)$ with $V \subseteq Q$, meaning that the input σw is accepted from a state q if and only if there exist a set of states V such that $V \subseteq Acc(w)$ that also satisfies the Boolean formula $\delta(q, \sigma)$. Hence, a run of an AFA on a finite word w is accepted if $q_0 \in Acc(w)$. The alternative acceptance condition for an AFA based on the Acc function is equivalent to the one based on the definition of a run-tree. In fact, an easy induction shows that $q \in Acc(w)$ if and only if there is a run-tree on the input w whose root is labeled by q (Vardi, 1996).

Finally, alternating automata have the same expressive power as nondeterministic automata, but they are exponentially more succinct. Indeed, given an AFA $\mathcal{A}_a = \langle \Sigma, Q, q_0, \delta, F \rangle$, we can compute an equivalent NFA $\mathcal{A}_n = \langle \Sigma, 2^Q, \{q_0\}, \delta_n, F_n \rangle$, where $F_n = 2^F$ and $\delta_n(T, a) = \{T' \mid T' \text{ satisfies } \bigwedge_{t \in T} \delta(t, a)\}$ that recognizes the same language, i.e., $\mathcal{L}(\mathcal{A}_a) = \mathcal{L}(\mathcal{A}_n)$ (Fellah et al., 1990). Such a transformation may require an exponentially larger number of states in the resulting NFA $\mathcal{A}_n$, namely $2^{|Q|}$. Additionally, converting an AFA with $|Q| = n$ to an equivalent DFA requires 2^{2^n} states in the worst case (Chandra et al., 1981).

Boolean Operations on Alternating Automata

Contrary to NFAs, AFAs are closed under the three main Boolean operations, namely complementation, intersection, and union. Since intersection and union operations on AFAs are analogous to the ones on NFAs, we only report the complementation.

To begin with, we define the dual operation on formulas in $\mathcal{B}^+(Q)$. In particular, given a formula $\theta \in \mathcal{B}^+(Q)$, the dual of θ, denoted as $\overline{\theta}$, can be obtained applying the following operations: (i) $\overline{q} = q$, (ii) $\overline{true} = false$, (iii) $\overline{false} = true$, (iv) $\overline{\alpha \wedge \beta} = \overline{\alpha} \vee \overline{\beta}$, and (v) $\overline{\alpha \vee \beta} = \overline{\alpha} \wedge \overline{\beta}$. For instance, the dual formula of

$x \vee (y \wedge x)$ is $\overline{x \vee (y \wedge x)} = x \wedge (y \vee x)$.

Therefore, the complement of an AFA $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$ is another AFA denoted as $\bar{\mathcal{A}} = \langle \Sigma, Q, q_0, \bar{\delta}, Q - F \rangle$, where $\bar{\delta}(q, \sigma) = \overline{\delta(q, \sigma)}$ for all $q \in Q$ and $\sigma \in \Sigma$. Basically, we apply the previously defined dual operation to the transition function. Thus, $\bar{\mathcal{A}}$ accepts only the words that are not accepted by $\mathcal{A}$, formally, $\mathcal{L}(\bar{\mathcal{A}}) = \Sigma^* - \mathcal{L}(\mathcal{A})$.

2.2 Linear-time Temporal Logic

Linear-time Temporal Logic (LTL) is a well-known temporal logic with modalities referring to time that was originally introduced in (Pnueli, 1977) as a specification language for the formal verification of computer programs. Consequently, LTL has been widely employed to formally describe complex properties over time in a myriad of applications and contexts, including Formal Methods, Artificial Intelligence, and Business Process Management.

Syntax

An LTL formula φ is defined over a nonempty set of propositional symbols $\mathcal{P}$, it is closed under the Boolean connectives ($\wedge, \vee, \neg$), the unary temporal operator *next-time* (X) and the binary temporal operator *until* (U). Formally, given a set of propositional symbols $\mathcal{P}$, LTL formulas over $\mathcal{P}$ are formally defined by the following grammar:

$$\varphi \quad ::= \quad p \mid \neg \varphi \mid \varphi \wedge \varphi \mid \mathsf{X} \varphi \mid \varphi \, \mathsf{U} \, \varphi$$

where $p \in \mathcal{P}$. There are also other logical and temporal operators that are abbreviations of primitive ones. For common logical abbreviations, we have: (i) $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, (ii) $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$, (iii) $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$, (iv) *true* $\equiv p \vee \neg p$, and (v) *false* $\equiv \neg true$. As for other temporal operators, we have the *eventually* (F) operator $\mathsf{F}\varphi \equiv true \, \mathsf{U} \, \varphi$, the *always* ($\mathsf{G}$) operator $\mathsf{G}\varphi \equiv \neg\mathsf{F}\neg\varphi$; and the *release* ($\mathsf{R}$) operator $\varphi_1 \, \mathsf{R} \, \varphi_2 \equiv \neg(\neg\varphi_1 \, \mathsf{U} \, \neg\varphi_2)$. Two formulas φ and ψ are equivalent $\varphi \equiv \psi$ if and only if they define the same language, i.e., $\mathcal{L}(\varphi) = \mathcal{L}(\psi)$.

Intuitively, the *next-time* operator indicates that a certain proposition must hold in the next time step, whereas the *until* operator indicates that a proposition must hold now and at all points in the future until another proposition becomes true.

Semantics

LTL formulas are interpreted on *infinite traces*, i.e., ω-words over the alphabet $2^{\mathcal{P}}$. A trace $\tau = \tau_0, \tau_1, \dots$ is

an infinite linearly ordered sequence of states where each state τ_i is a set of propositional symbols, namely $\tau_i \in 2^{\mathcal{P}}$, for $i \in \mathbb{N}$. Given an instant i, the state τ_i defines a propositional evaluation. Given an infinite trace τ, an LTL formula φ and a position $i \geq 0$, we inductively define when φ *holds* at position i, written $\tau, i \models \varphi$, as follows:

- $\tau, i \models p$ iff $p \in \tau_i$ (for $p \in \mathcal{P}$);

- $\tau, i \models \neg\varphi$ iff $\tau, i \not\models \varphi$;

- $\tau, i \models \varphi_1 \wedge \varphi_2$ iff $\tau, i \models \varphi_1$ and $\tau, i \models \varphi_2$;

- $\tau, i \models \mathsf{X}\varphi$ iff $\tau, i{+}1 \models \varphi$;

- $\tau, i \models \varphi_1 \, \mathsf{U} \, \varphi_2$ iff there exists $j \geq i$ such that $\tau, j \models \varphi_2$, and $\tau, k \models \varphi_1$ for all k with $i \leq k < j$.

A trace τ *satisfies* φ, written $\tau \models \varphi$, if the trace τ satisfies the formula φ at the first instant, i.e., $\tau, 0 \models \varphi$. Thus, we say that τ is a *model* of φ.

2.3 Linear-time Temporal and Dynamic Logic on Finite Traces

Although LTL has been widely employed as a specification formalism, many interesting problems, especially in Artificial Intelligence, have a finite horizon, assuming that the execution stops after a specific task/goal is achieved. Therefore, many recent works studied and applied versions of LTL evaluated on *finite*, instead of infinite, traces. The difference between infinite and finite length traces might look subtle, but in some cases, the interpretation of a formula on a finite trace completely changes its meaning and its algorithmic usage with respect to the one on infinite traces. While the idea of using finite-trace semantics for LTL appeared in some works in the early 2000s (e.g., in (Bacchus and Kabanza, 1996; Baier and McIlraith, 2006b)), the first formal and thorough study of finite-trace versions of LTL dates back to (De Giacomo and Vardi, 2013).

In this section, we look at Linear-time Temporal Logic interpreted on *finite* traces (LTL$_f$) and its proper extension *Linear-time Dynamic Logic* (LDL$_f$) interpreted on *finite* traces. Analogously to LTL, these logics express temporal properties in a "pure-future" fashion; namely, their temporal operators only refer to the present and to the future.

As previously mentioned, LTL$_f$ and the extension LDL$_f$ have been extensively applied in Artificial Intelligence and Computer Science. For instance, they have been employed as specification formalisms for Planning (Bacchus and Kabanza, 1996; Baier and McIlraith, 2006a; Patrizi et al., 2011; Torres and Baier, 2015; De Giacomo and Rubin, 2018; Camacho and McIlraith, 2019), Synthesis (De Giacomo and Vardi, 2015, 2016; Zhu et al., 2017; Camacho et al., 2018), Reinforcement Learning (Camacho et al., 2019b; De Giacomo

et al., 2019, 2020b), Business Process Management (van der Aalst et al., 2009; Pešić et al., 2010; De Giacomo et al., 2017), and in many other areas.

2.3.1 Linear-time Temporal Logic on Finite Traces

Syntax

The syntax of LTL_f is essentially the same as LTL except for the following additional temporal operators: the *weak-next* (WX) operator defined as $\mathsf{WX}\varphi \equiv \neg\mathsf{X}\neg\varphi$, and the *end* of the trace $\mathsf{final} \equiv \mathsf{WX}\mathit{false}$,

Semantics

As mentioned before, LTL_f formulas are evaluated on finite traces. A finite trace $\tau = \tau_0, \ldots, \tau_n \in (2^{\mathcal{P}})^*$ is a finite sequence of states τ_i, where τ_i at instant i is a propositional interpretation over the alphabet $2^{\mathcal{P}}$. We denote the length $n + 1$ of a trace τ with $\mathsf{length}(\tau)$ and the *last* element of τ by $\mathsf{last}(\tau) = s_n$. Given a finite trace τ, an LTL formula φ and a position $i \geq 0$, we inductively define when φ *holds* at position $i < \mathsf{length}(\tau)$, written $\tau, i \models \varphi$, as follows:

- $\tau, i \models p$ iff $p \in \tau_i$ (for $p \in \mathcal{P}$);

- $\tau, i \models \neg\varphi$ iff $\tau, i \not\models \varphi$;

- $\tau, i \models \varphi_1 \wedge \varphi_2$ iff $\tau, i \models \varphi_1$ and $\tau, i \models \varphi_2$;

- $\tau, i \models \mathsf{X}\varphi$ iff $i < \mathsf{length}(\tau) - 1$ and $\tau, i + 1 \models \varphi$;

- $\tau, i \models \mathsf{WX}\varphi$ iff $i = \mathsf{length}(\tau) - 1$ or $\tau, i + 1 \models \varphi$;

- $\tau, i \models \varphi_1 \mathbin{\mathsf{U}} \varphi_2$ iff there exists $i \leq j < \mathsf{length}(\tau)$ such that $\tau, j \models \varphi_2$, and $\tau, k \models \varphi_1$ for all k with $i \leq k < j$.

An LTL_f formula φ is *true* in τ, formally written as $\tau \models \varphi$, if $\tau, 0 \models \varphi$, namely, it is true in the first instant. Compared to the semantics on infinite traces of LTL, LTL_f semantics is basically the one of LTL but bounded on the indexes for the *next-time* and the *until* operators to characterize the finite nature of the evaluation trace. Interestingly, on finite traces, we have that $\neg\mathsf{X}\varphi \not\equiv \mathsf{X}\neg\varphi$, and $\mathsf{end} \equiv \mathsf{G}\mathit{false}$ characterizes that the trace is ended.

Usually, $\mathsf{sub}(\varphi)$ denotes the set of all *subformulas* of φ obtained from the abstract syntax tree of φ (De Giacomo and Vardi, 2013). For instance, if $\varphi = a \wedge \neg\mathsf{X}(b \vee (c \vee d))$, where a, b, c, d are atomic, then $\mathsf{sub}(\varphi) = \{a, b, c, d, (c \vee d), b \vee (c \vee d), \mathsf{X}(b \vee (c \vee d)), \neg\mathsf{X}(b \vee (c \vee d)), a \wedge \neg\mathsf{X}(b \vee (c \vee d))\}$. Thus, the cardinality of $\mathsf{sub}(\varphi)$, denoted as $|\mathsf{sub}(\varphi)|$, defines the size of an LTL_f formula φ.

In the following, we report some of the most common LTL$_f$ formula example patterns.

Example 2.1. *Remarkable* LTL$_f$ *formula patterns are:*

- Safety*: Gφ, which means "always till the end of the trace φ holds";*

- Liveness*: Fφ, which means "eventually before the end of the trace φ holds";*

- Response*: GFφ, which means "for any point in the trace, there exists a point later in the trace where φ holds";*

- Persistance*: FGφ means that "there is a point in the trace such that from then on until the end of the trace φ holds".*

As shown in Example 2.1, no direct nesting of eventually and always temporal operators is significant in LTL$_f$. However, indirect nesting of eventually and always operators can still produce meaningful and interesting properties. One example could be G($\psi \rightarrow$ Fϕ), which stands for "always, before the end of the trace, if ψ holds, then ϕ will eventually hold".

2.3.2 Linear-time Dynamic Logic on Finite Traces

LDL$_f$ represents an interesting variation of LTL$_f$. In particular, LDL$_f$ can be considered as a merge between LTL$_f$ and Regular Temporal Specifications (RE$_f$), which are basically Regular Expressions (RE) interpreted as a form of temporal specification on finite traces, cf. (De Giacomo and Vardi, 2013).

Although RE$_f$ is strictly more expressive than LTL$_f$, the absence of a direct construct for negation and for conjunction makes RE$_f$ not convenient and suitable to express temporal specifications easily. Furthermore, since negation may give rise to an exponential blow-up of the size of a RE$_f$, operations like intersection and complementation are not straightforward. Therefore, borrowing the syntax from the *Propositional Dynamic Logic* (PDL), a logic for computer programs introduced in (Fischer and Ladner, 1979), De Giacomo and Vardi (2013) introduced LDL$_f$ as a natural extension of LTL$_f$ that also captures RE$_f$ on finite traces.

Syntax

Given a set of propositional symbols $\mathcal{P}$, an LDL$_f$ formula φ is formally defined as follows:

$$
\begin{aligned}
\varphi &\quad ::= \quad tt \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle \varrho \rangle \varphi \\
\varrho &\quad ::= \quad \phi \mid \varphi? \mid \varrho + \varrho \mid \varrho; \varrho \mid \varrho^* \\
\phi &\quad ::= \quad p \mid \neg\phi \mid \phi \wedge \phi
\end{aligned}
$$

where $p \in \mathcal{P}$, tt is a syntactic construct that stands for the true LDL_f formula (not to be confused with the propositional $true \in \mathcal{P}$, which is an abbreviation for $p \vee \neg p$, for some $p \in \mathcal{P}$), and ϱ denotes path expressions, which are RE over propositional formulas ϕ with the addition of the test construct $\varphi?$, typical of PDL. LDL_f shares the common abbreviations for logical operators as in LTL. Other abbreviations are: $[\varrho]\varphi \equiv \neg\langle\varrho\rangle\neg\varphi$ as in PDL, $ff \equiv \neg tt$ for the false LDL_f formula, $\phi \equiv \langle\phi\rangle tt$ to denote the occurrence of the propositional formula ϕ, $\mathsf{end} \equiv [true]ff$ to express that the trace has ended, and $\mathsf{final} \equiv \langle true\rangle\mathsf{end}$ to denote the last element of the trace.

Intuitively, $\langle\varrho\rangle\varphi$ states that, from the current instant in the trace, there exists an execution satisfying the RE ϱ such that its last instant satisfies φ. On the other hand, $[\varrho]\varphi$ states that, from the current instant, all executions satisfying the RE ϱ are such that their last instant satisfies φ. Test constructs put into the execution path a check for the satisfaction of additional LDL_f formulas.

Semantics

Like LTL_f, LDL_f formulas are interpreted on finite traces. Given a finite trace $\tau = \tau_0, \ldots, \tau_n$, we denote by $\tau_{i,j}$ the sub-trace $\tau_i, \ldots, \tau_{j-1}$ if $j \leq \mathsf{length}(\tau)$, or the sub-trace $\tau_i, \ldots, \tau_n$ when $j > \mathsf{length}(\tau)$. Note that when $i \geq \mathsf{length}(\tau)$, $\tau_{i,j}$ denotes the empty trace. Given a finite, possibly empty, trace τ, an LDL_f formula φ, and an instant i, we say that φ *holds* at i, written $\tau, i \models \varphi$, by (mutual) induction, when:

- $\tau, i \models tt$;

- $\tau, i \models \neg\varphi$ iff $\tau, i \not\models \varphi$;

- $\tau, i \models \varphi_1 \wedge \varphi_2$ iff $\tau, i \models \varphi_1$ and $\tau, i \models \varphi_2$;

- $\tau, i \models \langle\varrho\rangle\varphi$ iff there exists a j such that $i \leq j$ and $\tau_{i,j} \in \mathcal{R}(\varrho)$ and $\tau, j \models \varphi$,

where the relation $\tau_{i,j} \in \mathcal{R}(\varrho)$ is inductively defined as:

- $\tau_{i,j} \in \mathcal{R}(\phi)$ if $j = i + 1$, $i < \mathsf{length}(\tau)$, and $\tau_i \models \phi$;

- $\tau_{i,j} \in \mathcal{R}(\varphi?)$ if $j = i$ and $\tau, i \models \varphi$;

- $\tau_{i,j} \in \mathcal{R}(\varrho_1 + \varrho_2)$ if $\tau_{i,j} \in \mathcal{R}(\varrho_1)$ or $\tau_{i,j} \in \mathcal{R}(\varrho_2)$;

- $\tau_{i,j} \in \mathcal{R}(\varrho_1; \varrho_2)$ if there exists $i \leq k \leq j$ such that $\tau_{i,k} \in \mathcal{R}(\varrho_1)$ and $\tau_{k,j} \in \mathcal{R}(\varrho_2)$;

- $\tau_{i,j} \in \mathcal{R}(\varrho^*)$ if $j = i$ or there exists k such that $\tau_{i,k} \in \mathcal{R}(\varrho)$ and $\tau_{k,j} \in \mathcal{R}(\varrho^*)$.

If $i \geq \mathsf{length}(\tau)$, the above definitions still apply. A trace τ *satisfies* an LDL_f formula φ, written $\tau \models \varphi$, if $\tau, 0 \models \varphi$.

In the following, we report some LDL_f formula examples and how such formulas translate to the LTL_f patterns shown in Example 2.1.

Example 2.2. *Remarkable* LDL$_f$ *formula patterns are:*

- Safety*: $[true^*]\varphi$ is equivalent to the* LTL$_f$ *formula* $\mathsf{G}\varphi$;

- Liveness*: $\langle true^*\rangle\varphi$ is equivalent to the* LTL$_f$ *formula* $\mathsf{F}\varphi$;

- Conditional Response*: $[true^*](\varphi_1 \to \langle true^*\rangle\varphi_2)$ is equivalent to the* LTL$_f$ *formula* $\mathsf{G}(\varphi_1 \to \mathsf{F}\varphi_2)$;

- Ordered occurrence*: $\langle true^*;\varphi_1;true^*;\varphi_2;true^*\rangle\mathsf{end}$ is equivalent to the* RE$_f$ $(true^*;\varphi_1;true^*;\varphi_2;true^*)$, *as* LDL$_f$ *captures any* RE$_f$ ϱ *with the formula $\langle\varrho\rangle\mathsf{end}$ (De Giacomo and Vardi, 2013; Brafman et al., 2018);*

- Alternating occurrence*: $\langle(\psi;\varphi)^*\rangle\mathsf{end}$ is equivalent to the* RE$_f$ $(\psi;\varphi)^*$.

The *Alternating occurrence* pattern shown in the previous example is a typical formula that cannot be directly translated to LTL$_f$. In general, LTL$_f$ (and LTL) are not able to capture regular structural properties on a path (Wolper, 1981).

2.3.3 Expressiveness

Generally, two logics are equally expressive when they recognize the same set of languages. In our case, LTL$_f$ is as expressive as First-Order Logic (FOL) on finite linear ordered sequences[1] (Gabbay et al., 1980; De Giacomo and Vardi, 2013) and star-free RE (McNaughton and Papert, 1971). However, RE on finite traces (i.e., RE$_f$) is strictly more expressive than LTL$_f$ (De Giacomo and Vardi, 2013). In particular, RE$_f$ is as expressive as Monadic Second-Order Logic (MSO) on bounded sequences (Khoussainov and Nerode, 2001).

On the other hand, LDL$_f$ has exactly the same expressive power of MSO (De Giacomo and Vardi, 2013), and, therefore, as RE$_f$. As a consequence, from an expressiveness perspective, LDL$_f$ brings advantages over LTL$_f$. In fact, without any additional computational cost, LDL$_f$ is strictly more expressive than LTL$_f$ but keeps almost all the readability and convenience of LTL$_f$.

Finally, according to (De Giacomo and Vardi, 2013), LTL$_f$ and RE$_f$ can be translated in LDL$_f$ in linear time (the inverse does not always hold), whereas the translation of LDL$_f$ in RE$_f$ is, in general, non-elementary. For completeness, we report the following recursive translation function ldlf($\cdot$) from (De Giacomo and Vardi,

---

[1] More precisely, *monadic first-order logic on finite linearly ordered domains*, sometimes also denoted as FO[<].

2013) to translate every LTL$_f$ formula as input into an equivalent LDL$_f$ formula in linear time:

$$
\begin{array}{rcl}
\mathsf{ldlf}(\phi) & = & \langle \phi \rangle tt \\
\mathsf{ldlf}(\neg\varphi) & = & \neg\mathsf{ldlf}(\varphi) \\
\mathsf{ldlf}(\varphi_1 \wedge \varphi_2) & = & \mathsf{ldlf}(\varphi_1) \wedge \mathsf{ldlf}(\varphi_2) \\
\mathsf{ldlf}(\mathsf{X}\varphi) & = & \langle tt \rangle \mathsf{ldlf}(\varphi \wedge \neg\mathsf{end}) \\
\mathsf{ldlf}(\varphi_1 \, \mathsf{U} \, \varphi_2) & = & \langle (\mathsf{ldlf}(\varphi_1)?; tt)^* \rangle \mathsf{ldlf}(\varphi_2 \wedge \neg\mathsf{end})
\end{array}
$$

2.4 From Linear Temporal and Dynamic Logics on Finite Traces to Automata

The translation from LTL$_f$ and LDL$_f$ to automata is one of the main building blocks of several reasoning approaches developed in Computer Science and Artificial Intelligence, such as Reactive Synthesis (De Giacomo and Vardi, 2015; Zhu et al., 2017; Camacho et al., 2018), FOND Planning for temporally extended goals (De Giacomo and Rubin, 2018; Camacho and McIlraith, 2019), and Reinforcement Learning with non-Markovian rewards (Camacho et al., 2019b; De Giacomo et al., 2019, 2020b).

In general, the translation of LTL$_f$ and LDL$_f$ to automata includes several steps. Usually, the first one consists of building an AFA, from which one can derive an NFA, and finally apply determinization techniques to get a DFA. Overall, this approach requires doubly-exponential time (De Giacomo and Vardi, 2013) in the worst case, as the resulting DFA can be of doubly exponential size with respect to the size of the formula.

In the following sections, we only provide the basic algorithm to translate an LTL$_f$ formula to the corresponding finite-state automata (De Giacomo and Vardi, 2013; Brafman et al., 2018), and then we provide an overview of the recent state-of-the-art approaches and tools. For the algorithm's extension to LDL$_f$, we refer the reader to (Brafman et al., 2018).

2.4.1 Translation to Alternating Finite-state Automata

Given an LTL$_f$ formula φ, the set of finite traces τ that satisfy φ defines the language of φ. Formally, $\mathcal{L}(\varphi) = \{\tau \mid \tau \models \varphi\}$. We can build an automaton $\mathcal{A}_\varphi$ that accepts the same set of finite traces that makes φ true, i.e., they define the same language $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A}_\varphi)$. First, we see how one can build an alternating automaton from a given LTL$_f$ formula in linear time (Chandra et al., 1981; Leiss, 1981; Büchi, 1990).

The key idea of the translation algorithm is to use subformulas as states of the automaton and generate suitable transitions mimicking the inductive semantics of temporal formulas (De Giacomo and Vardi, 2013). In particular, the technique uses a generalization of the concept of subformulas derived from the well-known

Fischer-Ladner *closure*, introduced for PDL by Fischer and Ladner (1979), which we denote as $\mathsf{cl}(\varphi)$. We may also refer to the Fischer-Ladner closure simply as the *syntactic closure*. Specifically, the syntactic closure of an LTL$_f$ formula comprises its subformulas, and it is computed by exploiting the well-known *fixpoint characterization* of LTL operators (Gabbay et al., 1980; Manna, 1982; Emerson, 1990) through the following expansion laws:

$$
\begin{aligned}
\mathsf{F}\phi &\equiv \phi \vee \mathsf{X}(\mathsf{F}\phi) \\
\mathsf{G}\phi &\equiv \phi \wedge \mathsf{WX}(\mathsf{G}\phi) \\
\phi_1 \, \mathsf{U} \, \phi_2 &\equiv \phi_2 \vee (\phi_1 \wedge \mathsf{X}(\phi_1 \, \mathsf{U} \, \phi_2)) \\
\phi_1 \, \mathsf{R} \, \phi_2 &\equiv \phi_2 \wedge (\phi_1 \vee \mathsf{WX}(\phi_1 \, \mathsf{R} \, \phi_2)).
\end{aligned}
$$

Here, observe that the laws for the G and R temporal operators are slightly different from the usual LTL laws. The difference lies in the presence of the WX operator instead of the X one. The reason for this change in LTL$_f$ is to account for the finiteness of traces, as the WX operator checks the existence of the next state. When we consider LTL$_f$ formulas, the fixpoint characterization splits the formula into a propositional formula on the current instant and a temporal formula to be checked at the *next* instant. The fixpoint characterization of an LTL$_f$ formula can be computed by recursively applying the following transformation function $\mathsf{xnf}(\cdot)$:

- $\mathsf{xnf}(p) = p$;

- $\mathsf{xnf}(\neg\phi) = \neg\mathsf{xnf}(\phi)$;

- $\mathsf{xnf}(\phi_1 \wedge \phi_2) = \mathsf{xnf}(\phi_1) \wedge \mathsf{xnf}(\phi_2)$;

- $\mathsf{xnf}(\phi_1 \vee \phi_2) = \mathsf{xnf}(\phi_1) \vee \mathsf{xnf}(\phi_2)$;

- $\mathsf{xnf}(\mathsf{X}\phi) = \mathsf{X}\phi$;

- $\mathsf{xnf}(\mathsf{WX}\phi) = \mathsf{WX}\phi$;

- $\mathsf{xnf}(\mathsf{F}\varphi) = \mathsf{xnf}(\varphi) \vee \mathsf{X}(\mathsf{F}\varphi)$;

- $\mathsf{xnf}(\mathsf{G}\varphi) = \mathsf{xnf}(\varphi) \wedge \mathsf{WX}(\mathsf{G}\varphi)$;

- $\mathsf{xnf}(\phi_1 \, \mathsf{U} \, \phi_2) = \mathsf{xnf}(\phi_2) \vee (\mathsf{xnf}(\phi_1) \wedge \mathsf{X}(\phi_1 \, \mathsf{U} \, \phi_2))$;

- $\mathsf{xnf}(\phi_1 \, \mathsf{R} \, \phi_2) = \mathsf{xnf}(\phi_2) \wedge (\mathsf{xnf}(\phi_1) \vee \mathsf{WX}(\phi_1 \, \mathsf{R} \, \phi_2))$.

Proposition 2.3 (Li et al. (2019)). *Every* LTL$_f$ *formula φ can be converted to its* XNF *(neXt Normal Form) in linear-time in the size of the formula. Moreover, $\mathsf{xnf}(\varphi)$ is equivalent to φ.*

Now, the syntactic closure of an LTL$_f$ formula is composed of the resulting subformulas after applying the expansion of the fixpoint characterization. In particular, the syntactic closure $\mathsf{cl}(\varphi)$ of an LTL$_f$ formula φ is the smallest set of LTL$_f$ formulas satisfying the following axioms and rules:

- $\varphi \in \mathsf{cl}(\varphi)$;

- $\neg p \in \mathsf{cl}(\varphi)$ if $p \in \mathsf{cl}(\varphi)$;

- $p \in \mathsf{cl}(\varphi)$ if $\neg p \in \mathsf{cl}(\varphi)$;

- $\varphi_1 \wedge \varphi_2 \in \mathsf{cl}(\varphi)$ implies $\varphi_1, \varphi_2 \in \mathsf{cl}(\varphi)$;

- $\varphi_1 \vee \varphi_2 \in \mathsf{cl}(\varphi)$ implies $\varphi_1, \varphi_2 \in \mathsf{cl}(\varphi)$;

- $\mathsf{X}\varphi \in \mathsf{cl}(\varphi)$ implies $\varphi \in \mathsf{cl}(\varphi)$;

- $\mathsf{WX}\varphi \in \mathsf{cl}(\varphi)$ implies $\varphi \in \mathsf{cl}(\varphi)$;

- $\mathsf{F}\varphi \in \mathsf{cl}(\varphi)$ implies $\varphi, \mathsf{X}(\mathsf{F}\varphi) \in \mathsf{cl}(\varphi)$;

- $\mathsf{G}\varphi \in \mathsf{cl}(\varphi)$ implies $\varphi, \mathsf{WX}(\mathsf{G}\varphi) \in \mathsf{cl}(\varphi)$;

- $\varphi_1 \mathbin{\mathsf{U}} \varphi_2 \in \mathsf{cl}(\varphi)$ implies $\varphi_1, \varphi_2, \mathsf{X}(\varphi_1 \mathbin{\mathsf{U}} \varphi_2) \in \mathsf{cl}(\varphi)$;

- $\varphi_1 \mathbin{\mathsf{R}} \varphi_2 \in \mathsf{cl}(\varphi)$ implies $\varphi_1, \varphi_2, \mathsf{WX}(\varphi_1 \mathbin{\mathsf{R}} \varphi_2) \in \mathsf{cl}(\varphi)$;

- $\mathsf{end} \in \mathsf{cl}(\varphi)$ implies $\mathsf{G}\mathit{false}, \mathit{false} \in \mathsf{cl}(\varphi)$.

Observe that given an LTL$_f$ formula φ, we can compute $\mathsf{cl}(\varphi)$ in linear time. Also, for simplicity, we assume formulas to be in Negation Normal Form (NNF) using the $\mathsf{nnf}(\varphi)$ function. A formula is in NNF when it exploits the common logical abbreviations, and negations are pushed inside the formulas as much as possible, leaving negations only in front of atomic propositions. Putting a formula in NNF can be done in linear time.

The construction of the alternating automaton $\mathcal{A}_\varphi = \langle \Sigma, Q, q_0, \partial, F \rangle$ corresponding to an LTL$_f$ formula φ is straightforward once we characterize the transition function ∂. In particular, the transition function $\partial(\varphi, \Pi)$ takes as input an LTL$_f$ formula φ (implicitly *quoted*, i.e., viewed as an atomic proposition) in NNF and a propositional interpretation Π for $\mathcal{P}$ and returns a positive Boolean formula whose atoms are (implicitly quoted) subformulas of φ.

$$\partial(p, \Pi) = true \text{ if } p \in \Pi$$

$$\partial(p, \Pi) = false \text{ if } p \notin \Pi$$

$$\partial(\neg p, \Pi) = false \text{ if } p \in \Pi$$

$$\partial(\neg p, \Pi) = true \text{ if } p \notin \Pi$$

$$\partial(\varphi_1 \wedge \varphi_2, \Pi) = \partial(\varphi_1, \Pi) \wedge \partial(\varphi_2, \Pi)$$

$$\partial(\varphi_1 \vee \varphi_2, \Pi) = \partial(\varphi_1, \Pi) \vee \partial(\varphi_2, \Pi) \tag{2.1}$$

$$\partial(\mathsf{X}\varphi, \Pi) = \varphi \wedge \neg\mathsf{end} \equiv \varphi \wedge \mathsf{F}\,true$$

$$\partial(\mathsf{WX}\varphi, \Pi) = \varphi \vee \mathsf{end} \equiv \varphi \vee \mathsf{G}\,false$$

$$\partial(\varphi_1 \,\mathsf{U}\, \varphi_2, \Pi) = \partial(\varphi_2, \Pi) \vee (\partial(\varphi_1, \Pi) \wedge \partial(\mathsf{X}(\varphi_1 \,\mathsf{U}\, \varphi_2), \Pi))$$

$$\partial(\mathsf{F}\varphi, \Pi) = \partial(\varphi, \Pi) \vee \partial(\mathsf{X}(\mathsf{F}\varphi), \Pi)$$

$$\partial(\mathsf{G}\varphi, \Pi) = \partial(\varphi, \Pi) \wedge \partial(\mathsf{WX}(\mathsf{G}\varphi), \Pi)$$

Intuitively, for the atomic case, given the subformula p – representing the automaton state – and the symbol Π read from a trace, the function $\partial(p, \Pi)$ evaluates to $true$ if p belongs to Π. The $\partial(p, \Pi)$ function directly follows the semantics of LTL$_f$ formulas. This is clear when comparing it with the expansion laws previously seen.

For empty traces ϵ, we define $\partial(\psi, \epsilon)$ inductively exactly as above except for the following cases:

$$\partial(p, \epsilon) = false$$

$$\partial(\neg p, \epsilon) = false$$

$$\partial(\mathsf{X}\varphi, \epsilon) = false$$

$$\partial(\mathsf{WX}\varphi, \epsilon) = true$$

$$\partial(\varphi_1 \,\mathsf{U}\, \varphi_2, \epsilon) = false \tag{2.2}$$

$$\partial(\mathsf{F}\varphi, \epsilon) = false$$

$$\partial(\mathsf{G}\varphi, \epsilon) = true$$

Observe that $\partial(\varphi, \epsilon)$ is always either $true$ or $false$.

Given an LTL$_f$ formula φ, an equivalent AFA $\mathcal{A}_\varphi$ is a tuple $\mathcal{A}_\varphi = \langle \Sigma, Q, q_0, \partial, F \rangle$, where $\Sigma = 2^{\mathcal{P}}$ is the input alphabet; $Q = \mathsf{cl}(\varphi)$ is the set of states; $q_0 = \varphi$ is the initial state; $\partial(\psi, \Pi)$ is the transition function;

and $F = \{\psi \mid \psi \in Q \text{ and } \partial(\psi, \epsilon)\}$ is the set of accepting states.

2.4.2 Translation to Nondeterministic Finite-state Automata

Interestingly, the $\partial(\psi, \Pi)$ function can be directly employed as an auxiliary function in the algorithm to build the nondeterministic automaton for the LTL$_f$ formula. In fact, Algorithm 2.1 takes as input an LTL$_f$ formula φ and outputs an NFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, Q, q_0, \delta, F \rangle$ that accepts exactly the traces satisfying φ. The algorithm is a variant of the one presented in (De Giacomo and Vardi, 2015). The ∂ function is used in lines 3, 9, and 13. In particular, the one in line 9 is used as a rule to check whether the formula within the condition satisfies the Boolean formula resulting from $\partial(\psi, \Pi)$. This means that in order to get the NFA, we do not need to go through the AFA construction beforehand. In other words, when running the algorithm, we just implement the non-determinization algorithm using recursive calls in line 9 to compute new states of the automaton.

Algorithm 2.1 Algorithm to translate a LTL$_f$ formula into its corresponding NFA

Require: LTL$_f$ formula φ
Ensure: NFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, Q, q_0, \delta, F \rangle$
1: $q_0 \leftarrow \{\varphi\}$
2: $F \leftarrow \{\emptyset\}$
3: **if** $(\partial(\varphi, \epsilon) = true)$ **then**
4:      $F \leftarrow F \cup \{q_0\}$
5: **end if**
6: $Q \leftarrow \{q_0, \emptyset\}$
7: $\delta \leftarrow \emptyset$
8: **while** $(Q \text{ or } \delta \text{ change})$ **do**
9:      **for** $(q \in Q)$ **do**
10:          **if** $(q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi))$ **then**
11:              $Q \leftarrow Q \cup \{q'\}$
12:              $\delta \leftarrow \delta \cup \{(q, \Pi, q')\}$
13:              **if** $(\bigwedge_{(\psi \in q')} \partial(\psi, \epsilon) = true)$ **then**
14:                  $F \leftarrow F \cup \{q'\}$
15:              **end if**
16:          **end if**
17:      **end for**
18: **end while**

The algorithm builds the NFA $\mathcal{A}_\varphi$ for an LTL$_f$ formula φ in a forward fashion. In particular, the algorithm visits every state q seen so far, checks for all the possible transitions from that state, and collects the results, determining the next state q', the new transition (q, Π, q') and if q' is a final state. Intuitively, the auxiliary function ∂ emulates the semantic behavior of every LTL$_f$ subformula after seeing the propositional interpretation Π. Obviously, the algorithm continues until it converges, i.e., when states and transitions do not change.

The states of automaton $\mathcal{A}_\varphi$ are sets of atoms (each atom is a quoted φ subformula) to be interpreted

as conjunctions. The empty conjunction $\emptyset$ stands for *true*. Then, q' is a set of quoted subformulas of φ denoting a minimal interpretation Π such that $q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$. Trivially, we have $(\emptyset, p, \emptyset) \in \delta$ for every $p \in 2^{\mathcal{P}}$. The following result holds:

Theorem 2.4 (De Giacomo and Vardi (2015)). *Algorithm 2.1 is correct, i.e., for every finite trace $\tau : \tau \models \varphi$ if and only if $\tau \in \mathcal{L}(\mathcal{A}_\varphi)$. Moreover, the algorithm terminates in at most an exponential number of steps and generates a set of states Q whose size is at most exponential in the size of the formula φ.*

We further explain how the algorithm works with the following example.

Example 2.5. *In this example, we see the execution of the Algorithm 2.1 given the* LTL$_f$ *formula* $\mathsf{G}(a)$ *with "a" being atomic. At the end of line 7, the components of the* NFA *are the following:* $\mathcal{P} = \{a\}, 2^{\mathcal{P}} = \{\{a\}, \emptyset\}, q_0 = \{\mathsf{G}a\}, Q = \{q_0, \emptyset\}, F = \{q_0, \emptyset\}$, *and* $\delta = \{(\emptyset, \{\}, \emptyset), (\emptyset, \{a\}, \emptyset)\}$. *In the following, we examine each iteration of the algorithm over automaton states:*

1. *Iteration: we pick the first state $q = \{\mathsf{G}a\}$*

   - *with $\Pi = \{a\}$ we have:*

$$
\begin{aligned}
q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\
&\models \partial(\mathsf{G}a, \Pi) \\
&\models \partial(a, \Pi) \wedge \partial(\mathsf{WXG}a, \Pi) \\
&\models true \wedge (\text{``}\mathsf{G}a\text{''} \vee \text{``}\mathsf{G}false\text{''})
\end{aligned}
$$

   The formula $true \wedge (\text{``}\mathsf{G}a\text{''} \vee \text{``}\mathsf{G}false\text{''})$ is a propositional formula with LTL$_f$ *formulas as atoms. As a minimal interpretation we have both $q' = \{\text{``}\mathsf{G}a\text{''}\}$ and $q' = \{\text{``}\mathsf{G}false\text{''}\}$. In both cases, we have that $\partial(\psi, \epsilon) = true$ (where $\psi = \mathsf{G}a$ and $\psi = \mathsf{G}false$), thus our automaton becomes $q_0 = \{\mathsf{G}a\}, Q = \{q_0, \{\mathsf{G}false\}, \emptyset\}, F = \{q_0, \{\mathsf{G}false\}, \emptyset\}$, and $\delta = \{(\emptyset, \{\}, \emptyset), (\emptyset, \{a\}, \emptyset), (q_0, \{a\}, q_0), (q_0, \{a\}, \{\mathsf{G}false\})\}$.*

   - *with $\Pi = \{\}$ we have:*

$$
\begin{aligned}
q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\
&\models \partial(\mathsf{G}a, \Pi) \\
&\models \partial(a, \Pi) \wedge \partial(\mathsf{WXG}a, \Pi) \\
&\models false \wedge (\text{``}\mathsf{G}a\text{''} \vee \text{``}\mathsf{G}false\text{''})
\end{aligned}
$$

29
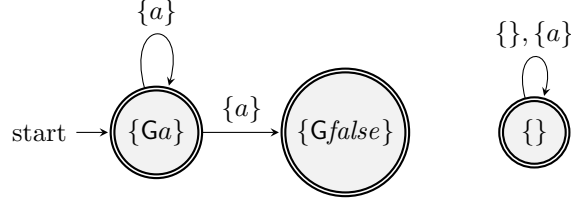
Figure 2.2: The graphical representation of the NFA computed with Algorithm 2.1 corresponding to the LTL$_f$ formula $\varphi = \mathsf{G}(a)$.

> *In this case, the propositional formula $false \wedge (\text{"}\mathsf{G}a\text{"} \vee \text{"}\mathsf{G}false\text{"})$ always evaluates to false. Hence,*
>
> *the automaton does not change.*

2. *Iteration: we pick the new state $q = \{\mathsf{G}false\}$*

- *With both $\Pi = \{\}$ and $\Pi = \{a\}$ we have that:*

$$q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$$

$$\models \partial(\mathsf{G}false, \Pi)$$

$$\models \partial(false, \Pi) \wedge \partial(\mathsf{WXG}false, \Pi)$$

$$\models false \wedge (\text{"}\mathsf{G}false\text{"} \vee \text{"}\mathsf{G}false\text{"})$$

> *As before, the formula is always false. Thus, there are no changes.*

> *The NFA $\mathcal{A}_\varphi = \langle 2^{\{a\}}, Q, q_0, \delta, F \rangle$ is depicted in Figure 2.2.*

The translation algorithm for LDL$_f$ is analogous to the one presented here for LTL$_f$ except for the use of a slightly different definition of the $\partial(\psi, \Pi)$ function, which takes into account the different semantics of LDL$_f$.

Determinization

In order to obtain a DFA, the NFA $\mathcal{A}_\varphi$ can be determinized in exponential time following the procedure in (Rabin and Scott, 1959). Thus, we can transform a LTL$_f$ formula into a DFA of double exponential size. In the following, we report the simple algorithm.

Algorithm 2.2 Algorithm to translate a LTL$_f$ formula into its corresponding DFA

Require: LTL$_f$ formula φ
Ensure: DFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, Q, q_0, \delta, F \rangle$
 1: Compute the NFA with Algorithm 2.1 (*exponential*)
 2: Apply determinization (*exponential*)
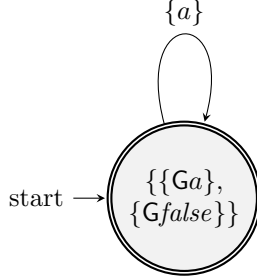
Figure 2.3: The graphical representation of the DFA computed with Algorithm 2.2 corresponding to the LTL$_f$ formula $\varphi = \mathsf{G}(a)$.

Figure 2.3 reports the DFA resulting from the determinization of the NFA in Figure 2.2.

Interestingly, there is also another way to directly evaluate a trace on a DFA that does not require the construction of the automaton. This technique has been introduced in (Brafman et al., 2018) and is called *on-the-fly*. Intuitively, the idea behind the on-the-fly construction is that while reading trace symbols, one can progress all possible states that the NFA can be in and, at the end of the trace, check whether, among those resulting states, there is an accepting state. Formally, let every state be a collection of NFA states as $\mathcal{Q} = \{q_1, \dots, q_n\}$, often called a *macrostate*, and let Π be the next trace symbol. Initially, we have $\mathcal{Q} = \mathcal{Q}_0 = \{q_0\} = \{\{\varphi\}\}$. Then, we can compute the next set of states $\mathcal{Q}'$ as $\mathcal{Q}' = \{q' \mid \exists q \in \mathcal{Q} \text{ such that } q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)\}$. Observe that the condition $q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$ is exactly the same as the one in Algorithm 2.1. By iterating the above procedure, we can check whether a given input trace τ satisfies a formula φ, i.e., $\tau \models \varphi$, starting from the initial macrostate $\mathcal{Q}_0$, if and only if the last state includes $\{true\}$, considering their evaluation in the empty trace $\partial(\psi, \epsilon)$. We refer the reader to (Brafman et al., 2018) for further details.

2.4.3 Other Translations

In the previous sections, we have reviewed the translations to automata that we call *standard* since the construction of the DFA follows the lowest theoretical complexity path. Overall, this approach requires doubly-exponential time (De Giacomo and Vardi, 2013) in the worst case, as the resulting DFA can be of doubly exponential size with respect to the size of the formula. Nonetheless, practical evidence has shown that, when applying determinization, the size of the resulting DFA is typically manageable (Tabakov and Vardi, 2005; Tabajara and Vardi, 2020; Zhu et al., 2021). Clearly, this observation is key when contrasting the recent growing interest in finite trace semantics with older works focusing on infinite trace semantics. In fact, working with the infinite traces semantics is hampered by the well-known determinization intractability of Büchi automata (Fogarty et al., 2015). In this section, we review alternative approaches that exploit other

key observations in translating linear temporal logics on finite traces to automata.

Interestingly, state-of-the-art transformation tools, such as Syft (Zhu et al., 2017), Lisa (Bansal et al., 2020) and Lydia (De Giacomo and Favorito, 2021), use various highly-optimized techniques that are actually nonelementary[2] in the worst case. In particular, they apply aggressive minimization approaches to the (partial) automata built during the construction. Despite such high complexity, minimization turned out to be crucial in order to obtain a scalable approach (Klarlund et al., 2002; Zhu et al., 2021).

The first two alternative techniques to transform an LTL$_f$ formula to a DFA exploits the intermediate translation to FOL. In particular, given that LTL$_f$ has the same expressive power of FOL on finite sequences, Zhu et al. (2017) proposed an encoding of LTL$_f$ formulas into FOL formulas and used the highly optimized Mona tool (Henriksen et al., 1995) to perform the actual translation to DFA. The Mona tool implements an efficient semi-symbolic (i.e., explicit in the states, symbolic in the transitions) representation of the automaton and applies aggressive automata minimization.

Some years later, Bansal et al. (2020) proposed a hybrid approach to the problem of DFA construction. Assuming the starting LTL$_f$ formula φ is of the form $\varphi = \bigwedge_{i=1}^{n} \varphi_i$ where $\varphi_1, \ldots, \varphi_n$ are n subformulas, they decompose the outermost conjunction of φ and transform each subformula φ_i into a DFA $\mathcal{A}_{\varphi_i}$ through Mona. Then, to build the resulting DFA, they compute the product between all automata $\mathcal{A}_{\varphi_i}$, always monitoring the size of such product. In the case where the size of the product automaton exceeds a user-defined threshold, the approach converts the automata into a symbolic representation and continues computing the products, though forgoing minimization. That is why the technique in (Bansal et al., 2020) is referred to as *hybrid*.

Tools associated with techniques in (Zhu et al., 2017; Bansal et al., 2020), Syft and Lisa respectively, have been shown to outperform state-of-the-art tools such as Spot (Duret-Lutz et al., 2016), which implements procedures to translate LTL formulas to automata on infinite words, but can also be used in the case of the finite semantics (i.e., LTL$_f$) by exploiting its encoding into LTL proposed in (De Giacomo and Vardi, 2013). Both Syft and Lisa use Mona as the underlying engine to transform FOL formulas to DFAs. Given the various operations of multiple determinization and projection to handle quantifiers, Mona implements a nonelementary procedure in the worst case. Interestingly, such a high worst-case complexity does not show in practice, and that is the reason why, from a practical perspective, Syft and Lisa perform better than direct techniques previously mentioned, which, instead, are worst-case double exponential.

Lately, starting from the hybrid approach in (Bansal et al., 2020), De Giacomo and Favorito (2021) took a step further and provided a sound and complete technique to directly transform LTL$_f$/LDL$_f$ formulas into a DFA. In particular, De Giacomo and Favorito (2021) introduced a bottom-up, fully *compositional*

---

[2]In computational complexity theory, a nonelementary problem is a problem that does not belong to the elementary class. Therefore, a nonelementary problem has an unbounded number of exponentiations.

approach to automata construction. The approach first computes the automata for the deepest subformulas in the syntactic tree of the formula and then proceeds bottom-up along the tree, incrementally combining the computed automata for the visited subformulas, via automata operations (e.g., union, intersection), based on the encountered operators. De Giacomo and Favorito (2021) implemented their technique in a tool called Lydia, which despite its nonelementary complexity, is now considered the state-of-the-art tool for the translation of temporal logics formulas on finite traces to automata.

2.4.4 Reasoning

As usual in classical logic, we say that an LTL$_f$/LDL$_f$ formula φ is *satisfiable* if it is true in some trace τ and is *valid* if it is true in every trace τ. Moreover, we say that an LTL$_f$/LDL$_f$ formula φ *logically implies* another LTL$_f$/LDL$_f$ formula ψ if and only if ψ is true in all traces where φ is true. The satisfiability and validity of LTL$_f$ and LDL$_f$ formulas are characterized by known complexity results. Specifically, satisfiability, validity, and logical implication for LTL$_f$/LDL$_f$ formulas are PSPACE-complete (De Giacomo and Vardi, 2013). Additionally, since the LTL$_f$/LDL$_f$ are propositionally closed, checking for validity and logical implication can be easily reduced to satisfiability. The classical procedure to check the satisfiability of an LTL$_f$/LDL$_f$ formula is as follows:

Algorithm 2.3 LTL$_f$/LDL$_f$ Satisfiability

Require: Given an LTL$_f$/LDL$_f$ formula φ
 1: Compute the AFA for φ *(linear)*
 2: Compute the corresponding NFA *(exponential)*
 3: Check the NFA for nonemptiness *(nlogspace)*
 4: Return the result of the check

2.5 Automated Planning

One of the central topics in this dissertation is *Automated Planning*, a well-known subfield of research in Artificial Intelligence (AI). Automated Planning, or just planning for short, represents the model-based approach to producing autonomous behavior where the agent behavior is automatically derived from a model of the world state, actions, sensors, and goals. Planning is also connected with the decision making performed by autonomous agents when trying to achieve some goals, which is a fundamental capability that intelligent systems should have.

In the following sections, we will describe in more detail some of the various well-known models to formalize AI planning problems.

2.5.1 Classical Planning

Following Geffner and Bonet (2013), a planning domain model describes the dynamics of an environment and is formally represented by a tuple $\mathcal{M} = \langle 2^{\mathcal{F}}, A, \alpha, tr \rangle$, where $2^{\mathcal{F}}$ is the set of possible states and $\mathcal{F}$ is a set of *fluents*; A is the set of actions; $tr : 2^{\mathcal{F}} \times A \to 2^{\mathcal{F}}$ is the *deterministic* transition function (i.e., $|tr(s, a)| = 1$ for all $s \in 2^{\mathcal{F}}$ and $a \in A$) determining the successor state s' that follows the execution of action a in state s; and $\alpha(s) \subseteq A$ is the set of applicable actions in state s, such that $a \in \alpha(s)$ if and only if $tr(a, s) \neq \emptyset$. In general, a planning state s is a truth assignment for all fluents in $\mathcal{F}$. A classical planning problem is a tuple $\Gamma = \langle \mathcal{M}, s_0, G \rangle$, where $\mathcal{M}$ is a domain model, s_0 is the initial state, and G is the set of goal states.

The mathematical model describing the planning problem Γ is often represented as a *transition system*. Transition systems are commonly depicted as directed edge-labeled graphs with an initial state and a set of goal states. In general, nodes represent domain states, edges are transitions or actions, and a plan is a sequence of actions corresponding to a path from the initial state to one of the goal states. For these reasons, in the specific case of planning, transition systems are also known as *state spaces*. From a common terminology in graph theory, a state s is said to be *reachable* if s is reachable from the initial state s_0.

Clearly, when the problem is large, the explicit enumeration of the state space is not feasible. That is why most often, *compact* or *factored* representations of transition systems have been studied and employed over the years. The key to such compact representations is that states are complete assignments to a set of propositions. In planning, fluents are atomic propositions that are true or false depending on what holds in a given state of the domain. Such a compact representation is known as a STRIPS (Fikes and Nilsson, 1971) representation. However, over the years, other languages have been studied, and the *Planning Domain Definition Language* (PDDL) (McDermott et al., 1998) has become a de-facto standard to represent planning domains. In particular, PDDL can be seen as the extension of STRIPS supporting first-order predicates with variables and constants. Another notable compact representation used in planning, especially in the internals of solvers, is SAS$^+$ (Bäckström and Nebel, 1995).

We can define a compact representation of the planning domain as a tuple $\mathcal{D} = \langle \mathcal{F}, \mathcal{F}_{der}, \mathcal{X}, A, pre, eff \rangle$ where $\mathcal{F}$ is a set of fluents (i.e., a set of positive literals), $\mathcal{F}_{der}$ is a set of derived predicates, $\mathcal{X}$ is a set of axioms, A is a set of action labels, *pre* and *eff* are two functions that define the preconditions and effects of each action $a \in A$.

A planning state s is a subset of $\mathcal{F}$, and a positive literal f holds true in s if $f \in s$; otherwise, f is false in s. Axioms have the form $d \leftarrow \psi$ where $d \in \mathcal{F}_{der}$ and ψ is a propositional formula over $\mathcal{F} \cup \mathcal{F}_{der}$. An axiom $d \leftarrow \psi$ specifies that d is derived to be true from a state s if and only if we can prove that $s \models \psi$, possibly using other axioms from $\mathcal{X}$. We assume that the set of axioms $\mathcal{X}$ is *stratified* (Hoffmann and Edelkamp,

2005) – this guarantees that given a state s and a derived predicate d, it is possible to efficiently and uniquely determine whether d holds true in s. Thus, it is always possible to determine whether a formula ψ over $\mathcal{F} \cup \mathcal{F}_{der}$ is satisfied by a state s. Both functions pre and eff take an action label $a \in A$ as an input and return a propositional formula over $\mathcal{F} \cup \mathcal{F}_{der}$ and an effect eff, respectively. The effect eff is a set of conditional effects each of the form $c \triangleright e$, where c is a propositional formula over $\mathcal{F} \cup \mathcal{F}_{der}$ and $e \subseteq \mathcal{F} \cup \{\neg f \mid f \in \mathcal{F}\}$ is a set of literals from $\mathcal{F}$.

Example 2.6. *In this example, we report a simplified version of the Yale shooting scenario domain. In general, the Yale shooting problem involves shooting at a turkey with the objective of eventually killing it.*

- *Fluents $\mathcal{F} = \{alive, loaded\}$ to represent that the turkey is alive and the gun is loaded in the current situation;*

- *Derived Predicates $\mathcal{F}_{der} = \emptyset$;*

- *Axioms $\mathcal{X} = \emptyset$;*

- *Actions $A = \{load, shoot, wait\}$ to represent the loading of the gun, the shooting, and a no-operation;*

- *Preconditions pre:*

  - *$pre(load) = \neg loaded$;*

  - *$pre(shoot) = true$;*

  - *$pre(wait) = true$;*

- *Effects eff:*

  - *$eff(load) = \{true \triangleright loaded\}$;*

  - *$eff(shoot) = \{loaded \triangleright \neg loaded, loaded \triangleright \neg alive\}$;*

  - *$eff(wait) = \emptyset$.*

An action a can be applied in a state s if $pre(a)$ holds true in s, formally, $s \models pre(a)$. A conditional effect $c \triangleright e$ is triggered in a state s if c is true in s. Applying a in s yields a successor state s' determined by the *deterministic* outcome of $eff(a)$. The new state s' is such that $\forall f \in \mathcal{F}$, f holds true in s' if and only if either (i) f was true in s and no conditional effect $c \triangleright e \in \mathsf{eff}$ triggered in s deletes it ($\neg f \in e$) or (ii) there is a conditional effect $c \triangleright e \in \mathsf{eff}$ triggered in s that adds it ($f \in e$). In case of conflicting effects, similarly to other works (Röger et al., 2014), we assume delete-before-adding semantics.

A classical planning problem combines a domain with an initial state and a goal and consists in looking for a sequence of actions that transforms the initial state into a desired goal state. Formally, a planning problem is a tuple $\Gamma = \langle \mathcal{D}, s_0, G \rangle$, where $\mathcal{D}$ is the domain model, s_0 is the initial state, i.e., an initial assignment to fluents in $\mathcal{F}$, and G is a set of literals over $\mathcal{F}$ called the *reachability* goal, all compactly represented.

A solution to planning problem Γ is a sequence of actions $a \in A$ called *plan* $\pi = a_0, \ldots, a_{n-1}$ such that, when executed, induces a finite *state-trace* $s_0, \ldots, s_n$, where $s_{i+1} = tr(s_i, a_i)$ and $a_i \in \alpha(s_i)$ for $0 \leq i \leq n-1$, and $s_n \models G$. Sometimes, a cost function mapping each action in the model to a non-negative cost determines a preference for plans with lower cost. The cost of a plan is defined as the sum of all action costs: $C_\pi = \sum_{i=0}^{n-1} c(a_i)$. When a plan has minimum cost among the space of all possible plans achieving a certain goal, it is said to be *optimal*. In the case action costs are not defined, costs are assumed to be unitary. Thus, in such a case, the shortest plans are those preferred.

Decision problems of (i) whether a plan exists for a given problem, and (ii) whether a plan of length less than k exists given a positive constant value k, are both PSPACE-complete (Bylander, 1994). In general, these results depend on the fact that plans can become exponentially long. However, if one restricts the plan's length to be polynomial in the size of the planning task, then the problem becomes NP-complete. Although planning problems are theoretically intractable in the worst case, current planning approaches based on heuristic search can typically solve large problem instances fast, regardless of their worst-case guarantees.

2.5.2 Fully Observable Nondeterministic Planning

Although classical planning models a wide variety of sequential decision-making problems, in many real-world environments, the outcome of an action is not always certain and can depend on various factors such as the state of the environment, exogenous and unpredictable events, and even randomness. For these reasons, researchers have studied variations and extensions of classical planning to account for this uncertainty in action effects.

In this dissertation, we focus on problems where the domain is *Fully Observable and NonDeterministic* (FOND). In this way, the planning agent can anticipate and plan for these unpredictable events, allowing it to make more robust and adaptive decisions.

Analogously to the classical planning setting, a FOND domain model is formally described by a tuple $\mathcal{M} = \langle 2^\mathcal{F}, A, \alpha, tr \rangle$. However, unlike classical planning, where actions are deterministic, in FOND planning, some action effects have an uncertain outcome when executing an action. In other words, the transition function $tr : 2^\mathcal{F} \times A \to 2^{2^\mathcal{F}}$ is *nondeterministic* (i.e., $|tr(s, a)| > 1$ for some states s and a). Intuitively, a

nondeterministic domain evolves as follows: from a given state s, the agent chooses a possible action a (i.e., $a \in \alpha(s)$), after which the environment chooses a successor state s' with $s' \in tr(s, a)$. A FOND planning problem is represented as a tuple $\Gamma = \langle \mathcal{M}, s_0, G \rangle$, where $\mathcal{M}$ is a nondeterministic domain model, s_0 is the initial state, and G is the set of goal states.

In general, FOND domain models are represented as transition systems too. However, although counter-intuitive, the automaton built starting from a transition system representing a FOND domain is deterministic. This is because, when progressing, the automaton reads both the action and its effect. Specifically, the nondeterminism in the environment (which is demonic) is not translated into a nondeterminism in the automaton (which is angelic) (De Giacomo and Rubin, 2018).

We compactly represent FOND domains as a tuple $\mathcal{D} = \langle \mathcal{F}, \mathcal{F}_{der}, \mathcal{X}, A, pre, eff \rangle$, where each component is the same as in classical planning, except for the eff function. In particular, while in classical planning the function eff returns only a single effect eff, in FOND planning the function eff returns a set $\{\mathsf{eff}_1, \ldots, \mathsf{eff}_n\}$ of effects, where each effect $\mathsf{eff}_i \in eff(a)$ is a set of conditional effects defined as in classical planning. In FOND, the application of action a in state s yields a successor state s' determined by an outcome *nondeterministically* drawn from $eff(a)$.

Example 2.7. *Following Example 2.6, here we report the nondeterministic variant of the Yale shooting scenario.*

- *Fluents $\mathcal{F} = \{alive, loaded, jammed\}$ to represent that the turkey is alive, the gun is loaded, and the gun is jammed in the current situation;*

- *Derived Predicates $\mathcal{F}_{der} = \emptyset$;*

- *Axioms $\mathcal{X} = \emptyset$;*

- *Actions $A = \{load, shoot, wait\}$ to represent the loading of the gun, the shooting, and a no-operation;*

- *Preconditions pre:*

  - *$pre(load) = \neg loaded$;*

  - *$pre(shoot) = true$;*

  - *$pre(wait) = true$;*

- *Effects eff:*

  - *$eff(load) = \{\{true \triangleright loaded, true \triangleright \neg jammed\}, \{true \triangleright loaded, true \triangleright jammed\}\}$;*

- $eff(shoot) = \{\{loaded \wedge \neg jammed \triangleright \neg loaded, loaded \wedge \neg jammed \triangleright \neg alive, loaded \wedge jammed \triangleright \neg jammed\}\};$

- $eff(wait) = \{\emptyset\}.$

Solutions to FOND Planning

Solutions to FOND planning problems are *policies* that instruct an agent on how to act in an environment with the guarantee of satisfying a given goal specification. A policy is defined as a partial function mapping *non-goal* states into actions. A policy π for a FOND problem Γ, starting from the initial state s_0, induces a set of (possibly infinite) state trajectories (or executions) Λ of the form $\tau = s_0, s_1, \ldots$, where s_0 is the initial state, $s_{i+1} \in tr(s_i, a_i)$, and $a_i = \pi(s_i)$ for $i \geq 0$. If for a certain sequence of states $\tau = s_0, \ldots, s_n$ we have that $\pi(s_n)$ is undefined (no action prescribed), then the generated execution τ is a *finite* trace.

Solution policies guarantee that their executions satisfy the goal. As usual, we consider two kinds of solutions to FOND planning problems: *strong* solutions and *strong-cyclic* solutions (Cimatti et al., 2003). Intuitively, strong policies are solutions whose executions guarantee goal satisfaction, regardless of how the environment reacts to agents' actions. A policy π is a *strong* solution to Γ, if every generated execution is a finite trace τ such that $\mathsf{last}(\tau) \models G$. On the other hand, a policy is a *strong-cyclic* solution to Γ if every generated execution that is a *stochastic-fair* trace is also a finite trace such that $\mathsf{last}(\tau) \models G$. In fact, in the FOND planning literature, the environment is commonly assumed to be *fair*, meaning that the environment manifests the nondeterminism in all of its spectrum. Over the years, a long thread of works studied different notions of fairness, e.g., (Daniele et al., 1999; Pistore et al., 2001; Cimatti et al., 2003; D'Ippolito et al., 2018; Aminof et al., 2020). The two most common notions of fairness are the *stochastic fairness* and *language-theoretic fairness*. The former assumes that some unknown distribution assigns a nonzero probability to each of the alternative effects and that such a distribution is used to select the effect. Formally, given a finite sequence of actions, stochastic fairness prescribes that each finite sequence of possible effects will occur infinitely often (Cimatti et al., 2003; Aminof et al., 2020, 2022). On the contrary, the latter notion can be expressed as a property of traces. In particular, this notion says that if an action a is taken from a state s infinitely often in the trace, and if s' is a possible effect of action a, then s' is the resulting effect of action a from state s infinitely often. Interestingly, when dealing with reachability goals, stochastic and language-theoretic fairness coincide (Aminof et al., 2020).

When a policy π is a solution (either strong or strong-cyclic, depending on the kind of solution we are interested in), we say that π is *winning*. Furthermore, policies can be compactly represented as *finite-state controllers* (FSC) (Geffner and Bonet, 2013). A controller is a finite-state machine that, given the observation

of a state, returns actions. More precisely, an FSC is a tuple $C = \langle Q, T, q_0 \rangle$, where Q is a set of control states, $T : Q \times 2^{\mathcal{F}} \to Q \times A$ is a (partial) fully observable transition function, and $q_0 \in Q$ is the initial controller state. Clearly, transitions do not depend on explicit input sequences but only on the current planning state. Finite-state controllers are essentially Mealy machines (Mealy, 1955) and have a natural graphical representation.

2.5.3 Planning for Temporally Extended Goals

Many real-world scenarios require achieving more general goals than just reachability goals. In most cases, realistic goals require properties to hold over time on a *sequence* of planning states. These types of *temporal goals*, also called *temporally extended goals*, are often expressed using temporal logics on finite or infinite sequences of states, including LTL, LTL_f, LDL_f, PSL, and CTL, just to name a few. Some common formula patterns have also been integrated into the language PDDL3 (Gerevini et al., 2009). Also, recent works witnessed lively interest in the topic. For instance, Bonassi et al. (2022) recently proposed an alternative way of expressing plan constraints and control knowledge through constraints over trajectories of actions rather than states. Temporally extended goals have also been considered over classical goals because they allow us to restrict the manner used by the plan to reach the goals.

Example 2.8. *Consider the well-known FOND domain model called* TRIANGLE-TIREWORLD, *depicted in Figure 2.4. In this domain, locations are connected by roads, and the agent can drive on them. The objective is to drive from one location to another. However, while driving between locations, a tire may go flat, and if there is a spare tire in the location of the car, then the car can use it to fix the flat tire. In Figure 2.4, circles represent locations, roads are represented by arrows, spare tires are depicted as tires, and the agent is depicted as a car. In the* TRIANGLE-TIREWORLD *domain, a classical goal would be reaching a certain*
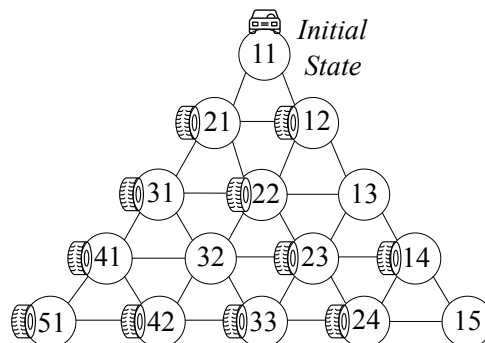
Figure 2.4: TRIANGLE-TIREWORLD FOND domain model.

location (e.g., reaching location 32). On the other hand, a suitable temporal goal would restrict the path

followed by the agent to reach a certain location (e.g., reaching location 32 passing through location 41).

In general, classical goals cannot express any preference or constraint on the plan to reach a chosen location. Instead, temporally extended goals are able to capture a richer class of plans where restrictions on the whole sequence of states must be satisfied as well, also in the presence of nondeterminism.

Planning for temporally extended goals has a long tradition in AI Planning, including pioneering work in the late '90s (Bacchus et al., 1996; Bacchus and Kabanza, 1996; Bacchus et al., 1997; Bacchus and Kabanza, 2000), work on planning via Model Checking (Cimatti et al., 1997; De Giacomo and Vardi, 1999; Giunchiglia and Traverso, 1999), and work on declarative and procedural constraints (Baier and McIlraith, 2006a; Baier et al., 2008). In this dissertation, we focus on temporally extended goals expressed in linear temporal logics on finite traces. Specifically, deterministic planning for LTL_f goals has already been studied and fully characterized in, e.g., (Baier and McIlraith, 2006a; De Giacomo and Vardi, 2013; Torres and Baier, 2015). More recently, nondeterministic planning for LTL_f goals has been fully characterized in, e.g., (De Giacomo and Vardi, 2015; Camacho et al., 2017; De Giacomo and Rubin, 2018; Camacho and McIlraith, 2019). Formally, a planning problem for temporally extended goals is a tuple $\Gamma = \langle \mathcal{D}, s_0, \varphi \rangle$, where $\mathcal{D}$ is a (possibly nondeterministic) domain model, s_0 is the initial state, and φ is a temporal formula over the fluents $\mathcal{F}$.

Plans, Policies, and Strategies

When dealing with temporally extended goals and deterministic domain models, the solution to a planning problem remains a sequence of actions, but in some cases could have additional spurious actions. However, turning to nondeterministic domains, the presence of temporally extended goals changes the definition of policy to a partial function $\pi : (2^{\mathcal{F}})^+ \to A$ mapping traces (i.e., nonempty sequence of states) into applicable actions as temporally extended goals can express non-Markovian properties over traces (Gabaldon, 2011). Therefore, in general, a solution must take into account the histories of states, not just the last one. For this reason and the tight relationship between FOND planning and reactive synthesis (cf. (De Giacomo and Vardi, 2015; De Giacomo and Rubin, 2018; Camacho et al., 2019a)), the updated and more general definition of policy corresponds to the notion of *strategy*. A strategy π for a FOND planning problem Γ, starting from the initial state s_0, induces a set of (possibly infinite) state trajectories (or executions) of the form $\tau = s_0, s_1, \ldots$, where s_0 is the initial state, $s_{i+1} \in tr(s_i, a_i)$, and $a_i = \pi(s_0, \ldots, s_i)$ for $i \geq 0$. If for a certain sequence of states $\tau = s_0, \ldots, s_n$ we have that $\pi(\tau)$ is undefined (no action prescribed), then the generated execution τ is a *finite* trace. Hence, given a temporal goal φ, π is a solution to Γ if every state trajectory induced by π is finite and satisfies φ.

Complexity

The complexity of classical and FOND planning for temporally extended goals is now well-understood. Classical planning for LTL_f goals is PSPACE-complete for deterministic domains, just like for classical reachability goals (Bylander, 1994). In this case, the added expressiveness of LTL_f goals is paid off in terms of algorithmic sophistication but not in worst-case complexity. On the contrary, the complexity of FOND planning for LTL_f goals is EXPTIME-complete in the domain specification as for standard reachability goals (Rintanen, 2004) and 2EXPTIME-complete in the LTL_f goal formula (De Giacomo and Rubin, 2018). In this case, instead, the added expressiveness of LTL_f goals worsens the worst-case goal complexity to 2EXPTIME-complete, compared to the EXPTIME-complete of reachability goals. That is because, in FOND planning, it is required to (implicitly or explicitly) translate LTL_f goal formulas into a DFA, which is double exponential in the worst case.

Encodings to Planning for Reachability Goals

In the literature, two techniques have been mainly exploited to deal with temporally extended goals. The first one uses automata-theoretic approaches, whereas the second one integrates the automaton dynamics directly within the compactly represented domain model. In the following, we are going to describe the main points of the two approaches.

The first approach, formerly developed in a classical setting in (De Giacomo and Vardi, 1999), and later applied to the context of FOND domain models (De Giacomo and Rubin, 2018), is based on automata-theoretic techniques that directly exploit the various properties of automata to compute policies. The latter work is of particular interest because it focuses on FOND planning for LTL_f and LDL_f goals. These automata-theoretic approaches compute the Cartesian product between the automaton resulting from the temporal formula and the automaton representing the domain model. While in classical planning, computing the NFA for the formula suffices (incurring only a worst-case single exponential blow-up), in FOND planning, the automata resulting from the formula must be determinized in some way. Indeed, the automaton determinization step, which involves a worst-case exponential blow-up, is mandatory because otherwise, there is a basic mismatch between the non-determinism of NFAs and the strategic behavior. In particular, while NFAs have perfect foresight (i.e., they see the whole world and guess the best path), strategic behavior means making a decision only based on what has been seen so far. In any case, at the end, a plan/policy can be easily derived from the product automaton.

On the other hand, the second approach has been mainly investigated in (Baier and McIlraith, 2006a; Patrizi et al., 2011; Torres and Baier, 2015; Camacho et al., 2017; Camacho and McIlraith, 2019) and focuses

on integrating the automaton resulting from the goal specification within the domain model, assumed to be compactly represented, e.g., in PDDL. Although there are many variants on how to encode the automaton dynamics, they all share the same underlying idea. Given a planning domain model Γ with an initial state and a goal formula φ, the approach works in the following steps.

Algorithm 2.4 Reduction of a planning for LTL$_f$/LDL$_f$ goals into planning for reachability goals

---
Require: Planning problem Γ, LTL$_f$/LDL$_f$ goal formula φ
 1: Transform the goal formula φ into the corresponding automaton
 2: Build a new planning problem Γ' by augmenting Γ with the states and the dynamics of the automaton
 3: Solve Γ' using any off-the-shelf classical or FOND planner
 4: Extract from Γ' a solution to Γ

With these steps, the problem of planning for a temporal goal is reduced to the problem of planning for simple reachability goals. This is actually very convenient because one can directly use any available classical or FOND off-the-shelf planner to solve the task.

Generally, executions of an augmented planning domain model have two distinct modes. The "world" mode, which follows the domain dynamics, and the "synchronization" mode, in which the automaton is updated. In other words, the "world" mode allows the execution of actions from the original domain model only, whereas, in the "synchronization" mode, only actions that update the state of the automaton can be executed. Intuitively, the new augmented planning problem Γ' has additional parts used to sequentially synchronize the dynamics between the domain and the automaton. Specifically, domain fluents are augmented with automaton states and flags for adequately switching between modes; domain actions preconditions are modified to allow executions only in "world" mode, whereas actions effects are modified to allow executions of synchronization actions; then, the automaton transition function is encoded as a new domain operator with conditional effects; finally, the initial and goal states are modified accordingly. This leads to having plans/executions of the form $\pi = a_1 t_1, \ldots, a_n t_n$, where $a_i \in \mathcal{A}$ is the real domain actions, and $t_1, \ldots, t_n$ are sequences of synchronization actions, which, in the end, are ignored to extract the desired policy. Some encodings, such as (Baier and McIlraith, 2006a), do not require synchronization actions. Thus, the obtained plan for the modified planning problem does not need to be reworked to get the plan for the original planning problem.

As previously mentioned, encodings of temporally extended goals to classical and FOND planning have a long tradition in the planning community (Baier and McIlraith, 2006a; Patrizi et al., 2011; Torres and Baier, 2015; Camacho et al., 2017; Camacho and McIlraith, 2019). Although these works have investigated the problem thoroughly providing several clever optimizations, it is worth mentioning that when the properties of interest correspond to syntactically large formulas whose automata are small, building the full automaton (NFA for classical planning, DFA for FOND planning) *a priori* and encoding the dynamics of the automaton

within a domain specification through additional actions to account for the automaton transition function, as done for example in (Fuggitti, 2019; De Giacomo and Fuggitti, 2021), results in faster encoding times. Obviously, techniques that construct the automaton a priori and blindly encode all automaton transitions are naïve since they do not exploit any insights about the temporally extended goal formulation.

Chapter 3

Pure-Past Linear Temporal Logics

In this chapter, we review PPLTL and PPLDL, the pure-past versions of the well-known logics on finite traces LTL$_f$ and LDL$_f$, respectively. PPLTL and PPLDL are logics about the past, so scan the trace backward from the end towards the beginning. Because of this, we can exploit a foundational result on reverse languages to get an exponential improvement, over LTL$_f$/LDL$_f$, for computing the corresponding DFA. This exponential improvement is reflected in several forms of sequential decision-making problems involving temporal specifications, such as planning and decision problems in nondeterministic and non-Markovian domains. Interestingly, PPLTL (resp., PPLDL) has the same expressive power as LTL$_f$ (resp., LDL$_f$), but transforming a PPLTL (resp., PPLDL) formula into its equivalent LTL$_f$ (resp., LDL$_f$) is quite expensive. Hence, to take advantage of the exponential improvement, properties of interest must be directly expressed in PPLTL/PPLDL.

Part of the content of this chapter has been published in (De Giacomo et al., 2020a) at the International Joint Conference on Artificial Intelligence 2020.

3.1 Motivation

All recent works dealing with linear temporal logics on finite traces have studied and applied formalisms referring only to the present and future. Thus, especially in the AI community, the use of pure-past versions of LTL$_f$ and LDL$_f$, namely Pure-Past Linear Temporal Logic (PPLTL) and Pure-Past Linear Dynamic Logic (PPLDL), have been given only limited attention. However, as pointed out in (Lichtenstein et al., 1985), in some cases, it is easier and more natural to express properties while referring to the past. For instance, using LTL$_f$ or other pure-future formalisms, it is not straightforward to specify that an agent has accomplished a task, and since the agent was decontaminated, it has been in clean areas only.

In fact, over the years, past temporal logics have been taken into account and advocated in some appli-

cations for non-Markovian rewards in MDPs (Bacchus et al., 1996), for non-Markovian models in reasoning about actions (Gabaldon, 2011), for preferred explanations in the context of dynamical diagnosis (Sohrabi et al., 2011), and for normative properties in multi-agent systems (Fisher and Wooldridge, 2005; Knobbout et al., 2016; Alechina et al., 2018). On the other hand, except for the works just mentioned, PPLTL has been introduced only as a technical means to get results for both LTL and LTL$_f$ (Maler and Pnueli, 1990; Zhu et al., 2019). Only recently, Cimatti et al. (2020) proposed a new LTL fragment with both future and past operators. This new fragment, with atoms of arbitrary PPLTL formulas within the scope of future operators, has shown to behave well for the synthesis problem. Nevertheless, they are yet to be better explored.

In this dissertation, instead, we consider PPLTL, and its extension PPLDL, as first-class citizens. In the following, we review the pure-past versions of LTL$_f$ and LDL$_f$, respectively PPLTL and PPLDL, and study their main theoretical properties.

3.2 Pure-Past Linear Temporal Logic

PPLTL is the variant of LTL$_f$ (De Giacomo and Vardi, 2013), reviewed in the previous chapter, that talks about the past instead of the future. Indeed, differently from the pure-future specification languages, we consider *pure-past* formulas to express temporal properties in a "pure-past fashion", i.e., referring only to the present and to the past.

Syntax

A PPLTL formula φ is defined over a set of propositional symbols $\mathcal{P}$, and it is closed under the Boolean connectives ($\wedge, \vee, \neg$), the unary past temporal operator *yesterday* (Y) and the binary past temporal operator *since* (S). Formally, given a set of propositional symbols $\mathcal{P}$, PPLTL formulas over $\mathcal{P}$ are formally defined by the following grammar:

$$\varphi \quad ::= \quad p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{Y}\varphi \mid \varphi \, \mathsf{S} \, \varphi$$

where $p \in \mathcal{P}$. Abbreviations of logical operators are analogous to the ones for pure-future temporal logics, and also for PPLTL, some temporal operators are abbreviations of primitive ones. In particular, we have the *once* (O) operator $\mathsf{O}\varphi \equiv true \, \mathsf{S} \, \varphi$, the *historically* ($\mathsf{H}$) operator $\mathsf{H}\varphi \equiv \neg\mathsf{O}\neg\varphi$; the *weak-yesterday* ($\mathsf{WY}$) operator defined as $\mathsf{WY}\varphi \equiv \neg\mathsf{Y}\neg\varphi$; and the *start* of the trace $\mathsf{start} \equiv \mathsf{WY}\bot$.

Intuitively, the *yesterday* operator indicates that there exists a *previous* state where the property holds, whereas the *since* operator indicates that there was a past state where a proposition held, and a proposition has held *since* then.

Semantics

Interestingly, PPLTL formulas are typically interpreted on *finite traces* because the past has happened and is always bounded by the current instant, namely the last instant, and the starting state of the system – this is important as some other logics have been interpreted on traces that are infinite in both the past and future. As such, every finite state sequence $\tau = \tau_0, \ldots, \tau_n$ is such that $\tau_n \in 2^{\mathcal{P}}$ is the interpretation of the *current* instant, and there always exists a state without its predecessor, namely τ_0.

Given a finite trace τ of length $\mathsf{length}(\tau) = n + 1$, a PPLTL formula φ and a position $i \geq 0$, we inductively define when φ *holds* at position i, written $\tau, i \models \varphi$, as follows:

- $\tau, i \models p$ iff $p \in \tau_i$ (for $p \in \mathcal{P}$);

- $\tau, i \models \neg\varphi$ iff $\tau, i \not\models \varphi$;

- $\tau, i \models \varphi_1 \wedge \varphi_2$ iff $\tau, i \models \varphi_1$ and $\tau, i \models \varphi_2$;

- $\tau, i \models \mathsf{Y}\varphi$ iff $i > 0$ and $\tau, i - 1 \models \varphi$;

- $\tau, i \models \varphi_1 \mathsf{S} \varphi_2$ iff there exists k, with $0 \leq k \leq i$, such that $\tau, k \models \varphi_2$ and $\tau, j \models \varphi_1$ for all j, with $k < j \leq i$.

We say that a PPLTL formula φ is *true* in τ, formally denoted as $\tau \models \varphi$, if $\tau, \mathsf{length}(\tau) - 1 \models \varphi$.

Analogously to the temporal formalisms reviewed in Chapter 2, given a formula φ, we can define its set of subformulas as $\mathsf{sub}(\varphi)$ obtained from the nodes of the syntactic tree of the formula φ. For instance, if $\varphi = a \wedge \neg\mathsf{Y}(b \vee c)$, where a, b, c are atomic, then $\mathsf{sub}(\varphi) = \{a, b, c, (b \vee c), \neg\mathsf{Y}(b \vee c), \mathsf{Y}(b \vee c), a \wedge \neg\mathsf{Y}(b \vee c)\}$. In general, a formula φ has linearly many subformulas. Also, $|\mathsf{sub}(\varphi)|$ defines the size of a PPLTL formula φ.

The following example shows how PPLTL can be easily employed to express temporal properties.

Example 3.1. *The property "we are now at location p_{23} and we have passed through location p_{12}" can be easily expressed in* PPLTL *with* $(p_{23} \wedge \mathsf{O}p_{12})$. *Another interesting property is "every time you took the bus, you bought a new ticket beforehand" that can be expressed in* PPLTL *with* $\mathsf{H}(takeBus \to \mathsf{Y}(\neg takeBus \mathsf{S} buyTicket))$. *This latter* PPLTL *formula recognizes exactly the same set of traces as the corresponding* LTL$_f$ *formula* $(buyTicket \mathsf{R} takeBus) \wedge \mathsf{G}(takeBus \to (buyTicket \vee \mathsf{X}(buyTicket \mathsf{R} \neg takeBus))$ *(Cimatti et al., 2004).*

As shown in Example 3.1, PPLTL formulas have their corresponding translation to LTL$_f$. However, depending on the application, one formalism may be preferred over the other for the readability of the specified property. We will elaborate more on this later in Section 3.5.

3.3 Pure-Past Linear Dynamic Logic

Along the same lines as the development of LDL$_f$ (De Giacomo and Vardi, 2013), PPLDL represents the natural pure-past extension of PPLTL merged with regular expressions RE on finite traces.

Syntax

Given a set of propositional symbols $\mathcal{P}$, a PPLDL formula φ is formally defined as follows:

$$\varphi \quad ::= \quad tt \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\!\langle \varrho \rangle\!\rangle \varphi$$
$$\varrho \quad ::= \quad \phi \mid \varphi? \mid \varrho + \varrho \mid \varrho; \varrho \mid \varrho^*$$

where ϕ denotes propositional formulas over $\mathcal{P}$, tt is the true PPLDL formula, and ϱ denotes path expressions, which are RE over propositional formulas ϕ with the addition of the test construct $\varphi?$, typical of PDL. PPLDL shares the common abbreviations for logical operators as in LDL$_f$. Other abbreviations are the past "box" operator $[\![\varrho]\!]\varphi \equiv \neg\langle\!\langle \varrho \rangle\!\rangle \neg\varphi$, and $\mathsf{start} \equiv [\![true]\!]ff$ to express that the trace has just started.

Intuitively, $\langle\!\langle \varrho \rangle\!\rangle \varphi$ states that there exists a point in the past, reachable (going backward) through the regular expression ϱ from the current instant, where φ holds. On the other hand, $[\![\varrho]\!]\varphi$ states that, from the current instant, all executions satisfying the RE ϱ are such that their initial instant in the past satisfies φ.

Semantics

The semantics of PPLDL formulas is given in terms of finite traces, and it is similar to the one of LDL$_f$. Given a finite trace $\tau = \tau_0, \ldots, \tau_n$, we denote by $\tau_{i,j}$ the sub-trace $\tau_i, \ldots, \tau_j$ if $j < \mathsf{length}(\tau)$, or the sub-trace $\tau_i, \ldots, \tau_n$ when $j \geq \mathsf{length}(\tau)$. Given a finite, possibly empty, trace τ, an LDL$_f$ formula φ, and an instant i, we say that φ *holds* at i, written $\tau, i \models \varphi$, by (mutual) induction, when:

- $\tau, i \models tt$;

- $\tau, i \models \neg\varphi$ iff $\tau, i \not\models \varphi$;

- $\tau, i \models \varphi_1 \wedge \varphi_2$ iff $\tau, i \models \varphi_1$ and $\tau, i \models \varphi_2$;

- $\tau, i \models \langle\!\langle \varrho \rangle\!\rangle \varphi$ iff there exists a j with $0 \leq j \leq i$ such that $\tau_{j,i} \in \mathcal{R}_\mathrm{p}(\varrho)$ and $\tau, j \models \varphi$,

where the relation $\tau_{j,i} \in \mathcal{R}_\mathrm{p}(\varrho)$ is inductively defined as:

- $\tau_{j,i} \in \mathcal{R}_\mathrm{p}(\phi)$ if $j = i - 1$, $i \geq 1$, and $\tau_i \models \phi$;

- $\tau_{j,i} \in \mathcal{R}_\mathrm{p}(\varphi?)$ if $j = i$ and $\tau, i \models \varphi$;

- $\tau_{j,i} \in \mathcal{R}_{\mathrm{p}}(\varrho_1 + \varrho_2)$ if $\tau_{j,i} \in \mathcal{R}_{\mathrm{p}}(\varrho_1)$ or $\tau_{j,i} \in \mathcal{R}_{\mathrm{p}}(\varrho_2)$;

- $\tau_{j,i} \in \mathcal{R}_{\mathrm{p}}(\varrho_1; \varrho_2)$ if there exists $j \leq k \leq i$ such that $\tau_{k,i} \in \mathcal{R}_{\mathrm{p}}(\varrho_1)$ and $\tau_{j,k} \in \mathcal{R}_{\mathrm{p}}(\varrho_2)$;

- $\tau_{j,i} \in \mathcal{R}_{\mathrm{p}}(\varrho^*)$ if $j = i$ or there exists $j \leq k \leq i$ such that $\tau_{k,i} \in \mathcal{R}_{\mathrm{p}}(\varrho)$ and $\tau_{j,k} \in \mathcal{R}_{\mathrm{p}}(\varrho^*)$.

We say that a trace τ *satisfies* a PPLDL formula φ, written $\tau \models \varphi$, if $\tau, \mathsf{length}(\tau) - 1 \models \varphi$.

An example of a PPLDL formula is given in the following.

Example 3.2. *The property "every time, if the cargo-ship departed (cargo), then beforehand there was an alternation of grab and unload of containers" can be expressed in* PPLDL *as*

$$[\![true^*]\!]\big(\langle\!\langle cargo \rangle\!\rangle tt \rightarrow \langle\!\langle (unl; grab)^*; (unl; grab) \rangle\!\rangle \mathsf{start}\big).$$

3.4 From Pure-Past Linear Temporal Logics to Automata

As we have already seen in Chapter 2 (cf. Section 2.4), there is an intimate connection between linear temporal/dynamic logic and automata theory. This is also true for pure-past logics like PPLTL and PPLDL. Actually, as we will see in the following sections, the translation of pure-past temporal and dynamic logics is even more interesting as it directly exploits the "backward" interpretation of pure-past formulas.

3.4.1 Reverse Languages and Alternation

The *reverse* of a trace $\tau = \tau_0, \ldots, \tau_n$ is defined as the trace $\tau^R = \tau_n, \ldots, \tau_0$ that reads the input backwardly, from the last propositional interpretation τ_n till the initial one τ_0. Thus, the reverse of a language $\mathcal{L}$ is denoted as the language $\mathcal{L}^R = \{\tau^R \mid \tau \in \mathcal{L}\}$; namely, it is the set of all reverse traces from the language $\mathcal{L}$.

Interestingly, while the conversion of an AFA with $|Q| = n$ states to an equivalent DFA requires 2^{2^n} states in the worst case (Chandra et al., 1981), the minimal DFA for the *reverse* language is at most single exponentially larger than an equivalent AFA (Chandra et al., 1981). The single exponentially larger AFA can be easily constructed as follows.

Given an AFA $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$ recognizing the language $\mathcal{L}$, we define the DFA $\mathcal{A}^R = \langle \Sigma, S, s_0, \rho, F' \rangle$ that recognizes the reverse language $\mathcal{L}^R$ where:

- $S = 2^Q$;

- $s_0 = F$;

- for $v \in S$ and $\sigma \in \Sigma$, we define $\rho(V, \sigma)$ to be the set of all states q such that $V \models \delta(q, \sigma)$;

- let $V \in F'$ iff $q_0 \in V$.

The size of $\mathcal{A}^R$ is therefore $2^{\mathcal{O}(|\mathcal{A}|)}$, where $|\mathcal{A}|$ represents the size of $\mathcal{A}$. Moreover, $\mathcal{A}^R$ can be computed in exponential time. In the following, we report the theoretical result from (Chandra et al., 1981) along with a fairly short and completely self-contained proof.

Theorem 3.3 (Chandra et al. (1981)). *Given an* AFA $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$ *recognizing the language* $\mathcal{L}$*, there exists a* DFA $\mathcal{A}^R = \langle \Sigma, S, s_0, \rho, F' \rangle$*, constructed as above, that recognizes the reverse language* $\mathcal{L}^R$*.*

Proof. Before proving the theorem, we introduce a generalization of the Acc function defined in Section 2.1.4 as $q \in Fwd(\tau, X)$, which is intuitively read as "the automaton reads τ forward from state q and results in a state in the set X". Formally, we define the function $Fwd : \Sigma^* \times 2^Q \to 2^Q$ inductively as follows:

$$Fwd(\epsilon, X) = X; \tag{3.1}$$

$$Fwd(\sigma\tau, X) = \{q \mid Fwd(\tau, X) \models \delta(q, \sigma)\}. \tag{3.2}$$

Intuitively, the Fwd function states that either the automaton remains in a state of X if reading the empty string ϵ or the input $\sigma\tau$ is processed forward from all the states q such that the Boolean formula $\delta(q, \sigma)$ is satisfied by $Fwd(\tau, X)$. Observe that the function Acc is definable in terms of the function Fwd, namely, $q \in Acc(\tau)$ if and only if $q \in Fwd(\tau, F)$. In other words, the Fwd function processes the input word τ in the forward direction.

Likewise, to process the input word τ in the backward direction, we define the function $Bck : \Sigma^* \times 2^Q \to 2^Q$ inductively as follows:

$$Bck(\epsilon, X) = X; \tag{3.3}$$

$$Bck(\tau\sigma, X) = Bck(\tau, \{s \mid X \models \delta(s, \sigma)\}). \tag{3.4}$$

By (3.2) and (3.4), it is easy to derive the property $Bck(\tau\sigma, Fwd(\tau', X)) = Bck(\tau, Fwd(\sigma\tau', X))$ replacing X with $Fwd(\tau', X)$. This property corresponds to the shift from $(\tau\sigma)\tau'$ to $\tau(\sigma\tau')$. Then, we show that $Bck(\tau, X) = Fwd(\tau, X)$ for all τ, X. Consider two cases: (i) $\tau = \epsilon$, and (ii) $\tau = \tau'\sigma$ for some τ', σ. For the base case $\tau = \epsilon$, the hypothesis is true from the definitions of Fwd and Bck. For the inductive case, we start from $Bck(\tau, X) = Bck(\tau'\sigma, X)$ and we get $Bck(\tau'\sigma, X) = Bck(\tau'\sigma, Fwd(\epsilon, X))$ by replacing X with $Fwd(\epsilon, X)$. Then, by (3.2) and (3.4), we have that $Bck(\tau'\sigma, Fwd(\epsilon, X)) = Bck(\tau', Fwd(\sigma, X))$. Analogously, by induction on the length of τ', we can show that $Bck(\tau', Fwd(\tau'', X)) = Bck(\epsilon, Fwd(\tau'\tau'', X))$ and therefore that $Bck(\tau', Fwd(\sigma, X)) = Bck(\epsilon, Fwd(\tau'\sigma, X))$. The base case $\tau' = \epsilon$ is immediate. For the

inductive case $\tau' = \tau'''\sigma$, we have that $Bck(\tau', Fwd(\tau'', X)) = Bck(\tau'''\sigma, Fwd(\tau'', X))$, which by (3.2) and (3.4) is equal to $Bck(\tau''', Fwd(\sigma\tau'', X))$. Now, by induction, we have that $Bck(\tau''', Fwd(\sigma\tau'', X)) = Bck(\epsilon, Fwd(\tau'''\sigma\tau'', X))$. Hence, we get $Bck(\epsilon, Fwd(\tau'\tau'', X))$ as required. At this point, by definition of Bck in (3.3), we know that $Bck(\epsilon, Fwd(\tau'\sigma, X)) = Fwd(\tau'\sigma, X)$. Hence, given that $\tau = \tau'\sigma$, we have $Bck(\tau, X) = Fwd(\tau, X)$ as required. Intuitively, $Bck(\tau, X) = Fwd(\tau, X)$ means that the automaton can process the input word τ both forward and backward.

Returning to the proof of Theorem 3.3, we prove, by induction on the input word τ, that $q_0 \in Fwd(\tau, V)$ if and only if the DFA $\mathcal{A}^R$ accepts τ^R from some states in V. For the base case $\tau = \epsilon$, both sides are equivalent to $q_0 \in V$, by (3.1) and (3.3). Now, consider $\tau = \tau'\sigma$. Then, by applying the previous result, we get $Fwd(\tau'\sigma, V) = Bck(\tau'\sigma, V)$. But, $Bck(\tau'\sigma, V) = Bck(\tau', \{s \mid V \models \delta(s, \sigma)\}) = Bck(\tau', \rho(V, \sigma))$ from (3.4) and from the construction of DFA above. Finally, we have $Bck(\tau', \rho(V, \sigma)) = Fwd(\tau', \rho(V, \sigma))$. Thus, $q_0 \in Fwd(\tau'\sigma, V) = Fwd(\tau', \rho(V, \sigma))$ if and only if $\mathcal{A}^R$ accepts $(\tau')^R$ from state $\rho(V, \sigma)$ (by induction), if and only if $\mathcal{A}^R$ accepts $\sigma(\tau')^R = (\tau'\sigma)^R$ from state V, as required. Hence, $\mathcal{A}$ accepts τ if and only if $\mathcal{A}^R$ accepts τ^R. □

3.4.2 Translation to Automata

Given the tight connection between temporal logics on finite traces and finite-state automata, we can take advantage of the single exponential language-theoretic reduction of an AFA to a DFA for the reverse language to get a DFA for PPLTL/PPLDL formulas. Therefore, the DFA resulting from a PPLTL or a PPLDL formula is single exponential in the size of the original formula.

We introduce the syntactic notion of *swap*, denoted as $\cdot^{\text{sw}}$, which, given a PPLTL/PPLDL formula, produces an LTL$_f$/LDL$_f$ formula by *syntactically* replacing each past operator with its corresponding future operator. For PPLTL formulas, the Y operator corresponds to the X operator, and the S operator corresponds to the U operator. On the other hand, for PPLDL formulas, the $\langle\!\langle \varrho \rangle\!\rangle$ operator corresponds to the $\langle \varrho^{\text{sw}} \rangle$, and the $[\![\varrho]\!]$ operator corresponds to $[\varrho^{\text{sw}}]$, where ϱ^{sw} is the regular expression ϱ, with all formulas in test constructs replaced by the corresponding swapped formulas. Formally, we define the *swap* φ^{sw} of a formula φ by induction as follows:

- $p^{\text{sw}} = p$ (for all $p \in \mathcal{P}$) and $tt^{\text{sw}} = tt$;

- $(\neg\varphi)^{\text{sw}} = \neg\varphi^{\text{sw}}$ and $(\varphi_1 \wedge \varphi_2)^{\text{sw}} = \varphi_1^{\text{sw}} \wedge \varphi_2^{\text{sw}}$;

- $(\mathsf{Y}\varphi)^{\text{sw}} = \mathsf{X}\varphi^{\text{sw}}$;

- $(\varphi_1 \mathbin{\mathsf{S}} \varphi_2)^{\text{sw}} = \varphi_1^{\text{sw}} \mathbin{\mathsf{U}} \varphi_2^{\text{sw}}$;

- $(\langle\!\langle\varrho\rangle\!\rangle\varphi)^{\mathsf{sw}} = \langle\varrho^{\mathsf{sw}}\rangle\varphi^{\mathsf{sw}}$ and $([\![\varrho]\!]\varphi)^{\mathsf{sw}} = [\varrho^{\mathsf{sw}}]\varphi^{\mathsf{sw}}$;

- $\phi^{\mathsf{sw}} = \phi$, and $(\varphi?)^{\mathsf{sw}} = (\varphi^{\mathsf{sw}})?$, and $(\rho_1 + \rho_2)^{\mathsf{sw}} = \rho_1^{\mathsf{sw}} + \rho_2^{\mathsf{sw}}$;

- $(\rho_1; \rho_2)^{\mathsf{sw}} = (\rho_1^{\mathsf{sw}}; \rho_2^{\mathsf{sw}})$, and $(\rho^*)^{\mathsf{sw}} = (\rho^{\mathsf{sw}})^*$.

Likewise, we can swap an $\text{LTL}_f/\text{LDL}_f$ formula φ into a PPLTL/PPLDL formula φ^{sw}. Given the set of all traces satisfying a temporal formula ϕ, called the language of ϕ and denoted as $\mathcal{L}(\phi)$, the process of swapping the temporal formula ϕ intuitively consists in finding another temporal formula ψ such that ψ is *true* in exactly those traces that are the reverse of the ones in $\mathcal{L}(\phi)$, namely $\mathcal{L}(\psi) = \mathcal{L}^R(\phi)$. Due to the connection of PPLTL/PPLDL semantics with $\text{LTL}_f/\text{LDL}_f$ semantics, the process of swapping temporal formulas only amounts to a *syntactic* replacement of temporal operators and/or regular expressions. The following lemma summarizes the relation between formulas and their swaps.

Lemma 3.4. *If φ is a PPLTL/PPLDL ($\text{LTL}_f/\text{LDL}_f$, resp.) formula, its swap φ^{sw} is an $\text{LTL}_f/\text{LDL}_f$ (PPLTL/PPLDL, resp.) formula of size $|\varphi|$ such that $\tau \models \varphi$ if and only if $\tau^R \models \varphi^{\mathsf{sw}}$, i.e., $\mathcal{L}^R(\varphi) = \mathcal{L}(\varphi^{\mathsf{sw}})$.*

Proof. The size of the swap is immediate by construction. The reverse language can be shown by structural induction on the formula φ and by applying the semantics of the temporal operators. First, we prove that a PPLTL formula φ is such that $\tau, i \models \varphi$ if and only if $\tau^R, \mathsf{length}(\tau){-}1{-}i \models \varphi^{\mathsf{sw}}$, where φ^{sw} is the corresponding LTL_f swap formula.

- $\varphi = p$. By the PPLTL semantics, we have that $\tau, i \models \varphi$ iff $p \in \tau_i$. By definition of reverse trace, we have that $\tau_i = \tau^R_{\mathsf{length}(\tau)-1-i}$, therefore we have $\tau, i \models \varphi$ iff $p \in \tau^R_{\mathsf{length}(\tau)-1-i}$. Now, $p \in \tau^R_{\mathsf{length}(\tau)-1-i}$ iff $\tau^R, \mathsf{length}(\tau){-}1{-}i \models p = \varphi^{\mathsf{sw}}$. Hence, the thesis holds.

- $\varphi = \mathsf{Y}\varphi'$. By the semantics, we have that $\tau, i \models \mathsf{Y}\varphi'$ iff $i > 0$ and $\tau, i{-}1 \models \varphi'$. Then, by applying the PPLTL semantics and by definition of reverse trace, we have $\mathsf{length}(\tau) - 1 - i > 0$ and $\tau^R, (\mathsf{length}(\tau) - 1 - i) - 1 \models \varphi'$, which are $i < \mathsf{length}(\tau) - 1$ and $\tau^R, \mathsf{length}(\tau){-}2{-}i \models \varphi'$. After that, we have $\tau^R, \mathsf{length}(\tau){-}2{-}i \models \varphi'$ iff $\tau^R, \mathsf{length}(\tau){-}1{-}i \models \mathsf{X}\varphi'^{\mathsf{sw}}$, which is φ^{sw}. Hence, the thesis holds.

- $\phi = \phi_1 \mathsf{S} \phi_2$. By the semantics, we have $\tau, i \models \varphi_1 \mathsf{S} \varphi_2$ iff $\exists k$ with $0 \leq k \leq i$ such that $\tau, k \models \varphi_2$ and $\forall j$ with $k < j \leq i$ such that $\tau, j \models \varphi_1$. By definition of reverse trace, such conditions are iff $\exists k$ with $\mathsf{length}(\tau) - 1 \leq k \leq \mathsf{length}(\tau){-}1{-}i$ such that $\tau^R, k \models \varphi_2$ and $\forall j$ with $k \leq j \leq \mathsf{length}(\tau){-}1{-}i$ such that $\tau, j \models \varphi_1$. Also in this case, by looking at the LTL_f semantics, we obtain that $\tau, i \models \varphi_1 \mathsf{S} \varphi_2$ if and only if $\tau^R, \mathsf{length}(\tau){-}1{-}i \models \varphi_1^{\mathsf{sw}} \mathsf{U} \varphi_2^{\mathsf{sw}}$. Hence, the thesis holds.

- $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \neg\varphi'$. The thesis holds by structural induction.

An analogous line of reasoning can be applied for LTL_f, LDL_f and PPLDL formulas and their swap. $\qquad\square$

The swapping operation must not be confused with the semantic operation of translation from one logic to the other. In the latter case, we want to find the formula of the other logic that is true on *exactly* the same set of traces (keeping the direction of the traces). We will discuss the semantic translation later in Section 3.5.2. To this end, we present two examples that illustrate the syntactic (vs. semantic) relationship between a formula and its swap.

Example 3.5. *Consider the* PPLTL *formula "inRoom $\wedge$ roomDecontaminated $\wedge$ O(getPermit)". The swapped* LTL_f *formula is "inRoom $\wedge$ roomDecontaminated $\wedge$ F(getPermit)". Although the two formulas are syntactically similar, they have different meanings. The former says that the robot is in a decontaminated room, and it acquired the permit to enter the room beforehand. The latter, instead, says that the robot is in a decontaminated room, and later (!), it will get the permit to enter.*

Example 3.6. *Consider the* LTL_f *formula "batteryCharged $\wedge$ F(useNotebook)" and its swapped* PPLTL *formula "batteryCharged $\wedge$ O(useNotebook)". Again, they have two different meanings. While the first says that the battery is now charged and you can eventually use the notebook, the* PPLTL *formula says that you used the notebook in the past, but the battery is charged now.*

We are ready to show that transforming a PPLTL/PPLDL formula into the corresponding DFA can be done in exponential time (vs. double exponential time as for LTL_f/LDL_f formulas):

Theorem 3.7. *For every* PPLTL/PPLDL *formula φ, there exists an equivalent* DFA *$\mathcal{A}_\varphi$ whose size is at most $2^{\mathcal{O}(|\varphi|)}$, and which is computable in at most exponential time.*

Proof. To see the theorem holds, we can swap the given PPLTL/PPLDL formula φ, getting the LTL_f/LDL_f φ^{sw}. Then, from φ^{sw}, we can construct the AFA $A_{\varphi^{\text{sw}}}$, and, finally, build the DFA $\mathcal{A}_\varphi = \mathcal{A}_{\varphi^{\text{sw}}}^R$. By Lemma 3.4 and Theorem 3.3, we get that $\mathcal{A}_\varphi$ has size $2^{\mathcal{O}(|\varphi|)}$ and $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A}_\varphi)$. $\qquad\square$

As a result, as we have seen in Chapter 2 (cf. Section 2.4 – Algorithm 2.2), we can define an analogous algorithm to translate PPLTL/PPLDL formulas into DFAs based on Theorem 3.7.

Algorithm 3.1 Algorithm to translate a PPLTL/PPLDL formula into its corresponding DFA

Require: Given a PPLTL/PPLDL formula φ
 1: Swap φ into the corresponding LTL_f/LDL_f φ^{sw} *(linear)*
 2: Compute AFA for φ^{sw} *(linear)*
 3: Compute DFA for the reverse language from AFA *(exponential)*

Observe that Algorithm 3.1 allows computing the DFA corresponding to a PPLTL/PPLDL formula in worst-case *single* exponential time vs. the worst-case double exponential time of Algorithm 2.2 for the LTL_f/LDL_f

case. This implies that using pure-past temporal logics as a means to express temporal properties reduces the theoretical complexity of several problems, as we will see later in Section 3.7.

Here, it is important to observe that this procedure is the best-known way to transform a PPLTL/PPLDL formula to DFA from a theoretical standpoint. In fact, to transform a PPLTL/PPLDL formula into a DFA, it is also possible to follow similar approaches to the standard one for LTL$_f$/LDL$_f$ formulas, seen in Section 2.4. Nevertheless, given that the PPLTL/PPLDL semantics (backward evaluation) is opposite to how automata read input symbols of traces, the first step of the transformation would need to go through special versions of an AFA (e.g., the two-way AFA (Vardi, 1998; Finkbeiner and Sipma, 2004)), accounting for the backward direction of traces and for which the subsequent transformation to a standard NFA comes at an additional cost (Geffert and Okhotin, 2014). However, in this dissertation, we have not delved further into these options.

3.5 Relationship to Other Formal Languages

We have just shown that when PPLTL and PPLDL are employed as temporal specification formalisms, they give an exponential advantage over LTL$_f$ and LDL$_f$, respectively. This is possible by exploiting the well-known language theoretic property characterizing alternating automata. However, having a computational advantage in transformation is not often a helpful and reliable indicator of differences or similarities in expressive power. Therefore, we examine in detail the general relationship of PPLTL and PPLDL to other formal languages.

3.5.1 Expressive Power

We start by establishing that PPLTL and LTL$_f$ have the same expressive power by using FOL as an intermediate logic. As mentioned in Section 2.3.3, FOL formulas are interpreted on finite traces viewed as labeled linear orders. In such settings, FOL formulas can use: (*i*) variables x that vary over instants and that can be quantified existentially and universally, (*ii*) the binary predicate $<$ denoting the order of instants, (*iii*) equality $=$ between instants, and *(iv)* unary (sometimes called monadic) predicates P for the labels. See (De Giacomo and Vardi, 2013) for formal definitions.

We start by observing that PPLTL and LTL$_f$ can be translated into FOL on finite traces by mimicking the semantics of these logics as FOL formulas, and can be done in linear time:

Theorem 3.8 (De Giacomo and Vardi (2013); Zhu et al. (2019))**.** PPLTL *and* LTL$_f$ *can be translated into* FOL *on finite traces in linear time.*

For the converse, it is known that FOL (on finite traces) can be translated into LTL$_f$ (Gabbay et al., 1980).

Here, we exploit this result to show that FOL can also be translated into PPLTL.

Theorem 3.9 (cf. (Kamp, 1968)). FOL *on finite traces can be translated into both* PPLTL *and* LTL$_f$.

Proof. Given an FOL formula φ replace $x < y$ by $y < x$ to get an FOL φ^{sw} for the reverse language, i.e., $w \models \varphi$ if and only if $w^R \models \varphi^{\text{sw}}$. Then, translate the FOL formula φ^{sw} into an equivalent LTL$_f$ formula ψ (Gabbay et al., 1980). Then, the PPLTL formula ψ^{sw} is equivalent to the original FOL formula φ. □

Putting these results together, we immediately get the following.

Theorem 3.10. PPLTL *and* LTL$_f$ *have the same expressive power.*

Considering the results on LTL$_f$ in (De Giacomo and Vardi, 2013), we can fully characterize the expressive power of PPLTL.

Theorem 3.11. PPLTL *has exactly the same expressive power as* FOL *on finite traces, namely, star-free regular expressions.*

Now, we study the expressive power of PPLDL. Similar to what we did for PPLTL, we exploit the already known expressive power equivalences with intermediate logics to get results for PPLDL. In particular, we follow results from (De Giacomo and Vardi, 2013), which use RE on finite traces to derive results for LDL$_f$.

Theorem 3.12. RE *on finite traces is as least as expressive as* PPLDL.

Proof. We can apply Theorem 3.7 to get a DFA and then use Kleene's Theorem (existence of a finite automaton for any regular expression) to get an equivalent regular expression. □

The reverse direction of the previous theorem holds.

Theorem 3.13. PPLDL *is as least as expressive as* RE *on finite traces.*

Proof. Given a regular expression ϱ, we compute the reverse regular expression and return $\langle\!\langle \varrho \rangle\!\rangle$start. □

At this point, given that RE has the same expressive power as MSO on bounded sequences (De Giacomo and Vardi, 2013), it is easy to derive the expressiveness relationship of PPLDL.

Theorem 3.14. PPLDL *has the same expressive power as* RE, *and as* MSO *on finite traces.*

Putting everything together, we immediately get the following characterization.

Theorem 3.15. PPLDL *has the same expressive power as* LDL$_f$.

3.5.2 Translations to Other Formal Languages

In this section, we are interested in studying ways to translate pure-future temporal and dynamic logics on finite traces (LTL$_f$ and LDL$_f$) into semantically equivalent pure-past temporal and dynamic logics (PPLTL and PPLDL), respectively.

Starting with PPLTL and LTL$_f$, the results we have just seen in the previous section give us a way to translate one into the other and vice versa. In particular, we can first translate LTL$_f$ (PPLTL, resp.) into FOL and then translate FOL into PPLTL (LTL$_f$, resp.). However, we remark that the transformation of an FOL formula into an LTL$_f$ (PPLTL, resp.) formula can be, in general, non-elementary (i.e., not bounded by any finite tower of exponentials) in the size of the FOL formula (Gabbay, 1987). Hence, exploiting an intermediate translation to FOL to get the translation of LTL$_f$ (PPLTL, resp.) formulas into PPLTL (LTL$_f$, resp.) formulas is suboptimal in general.

In fact, we can do better using the following result from the literature:

Theorem 3.16 (Maler and Pnueli (1990)). *A* DFA *accepting star-free regular languages can be translated into a* PPLTL *formula whose size is at most exponentially larger.*

Given this result, we can provide a better translation, which gives us the best-known upper bound for the translation. Whether such a bound is tight is still an open problem.

Theorem 3.17. *For every* PPLTL *(resp.,* LTL$_f$*) formula* φ*, there exists an equivalent* LTL$_f$ *(resp.,* PPLTL*) formula whose size is at most triply exponential in the size of* φ*, and which is computable in at most triply exponential time.*

Proof. Given a PPLTL formula φ, we can build an equivalent DFA $\mathcal{A}_\varphi$ by Theorem 3.7. In general, the DFA may be exponentially larger than φ. Given the DFA, we reverse all of its transitions, thus obtaining an NFA $\mathcal{A}_\varphi^R$ that accepts the reverse of the language of $\mathcal{A}_\varphi$. Then, we can determinize the NFA $\mathcal{A}_\varphi^R$ to get an equivalent DFA $\mathcal{A}_\varphi'^R$. Clearly, the DFA $\mathcal{A}_\varphi'^R$ may be exponentially larger than $\mathcal{A}_\varphi^R$. At this stage, we apply the result in Theorem 3.16 to transform the DFA $\mathcal{A}_\varphi'^R$ into an equivalent PPLTL formula ψ. Finally, by computing the swap $\psi^{\mathtt{sw}}$ of the PPLTL formula ψ, we get a formula that recognizes the reverse language of ψ. In particular, $\psi^{\mathtt{sw}}$ represents the LTL$_f$ formula equivalent to the original PPLTL formula φ. All these transformations may incur in three exponential blowups.

Similarly, we can obtain a PPLTL formula from an LTL$_f$ one. Given an LTL$_f$ formula φ, we can build an equivalent DFA $\mathcal{A}_\varphi$ that may be double-exponentially larger than φ. Then, we apply Theorem 3.16 to get an equivalent PPLTL formula, which may be single-exponentially larger. $\qquad\square$

Next, we bring attention to PPLDL and LDL$_f$. We provide a translation bound (and algorithm) that is
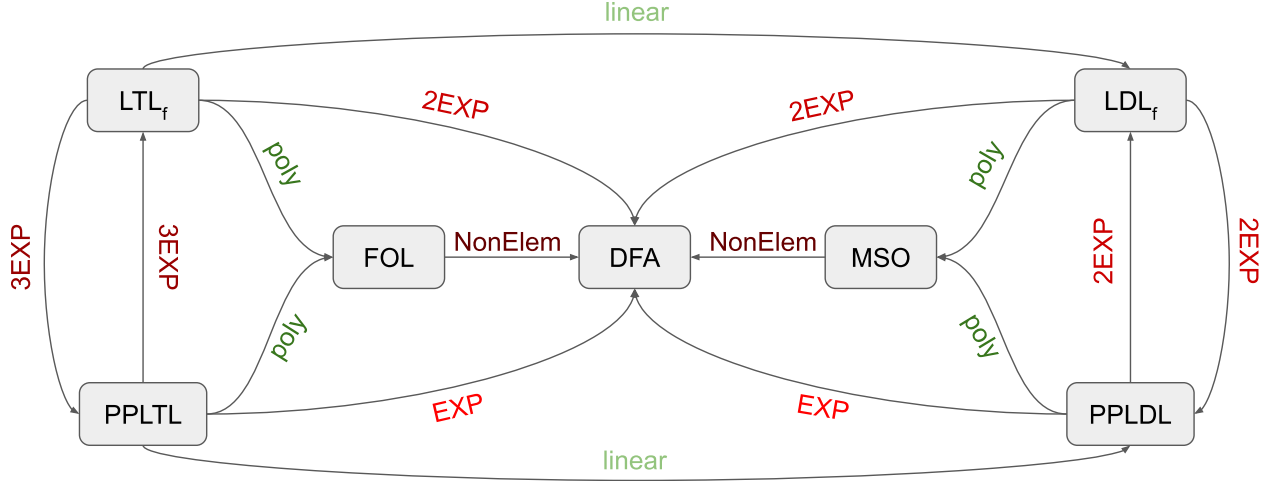
Figure 3.1: Overview of the main transformations among linear temporal and dynamic logics, first-order and second-order logics, and automata.

the best-known upper bound. However, also in this case, whether these bounds can be further improved or are tight remains open.

Theorem 3.18. *For every* PPLDL *(resp.,* LDL$_f$*) formula* φ*, there exists an equivalent* LDL$_f$ *(resp.,* PPLDL*) formula whose size is at most doubly exponential in the size of* φ*, and which is computable in doubly exponential time.*

Proof. From a PPLDL formula φ, we can build an equivalent DFA that may be exponentially larger as per Theorem 3.7. Then, using the Kleene's Theorem, we convert the obtained DFA into a regular expression that may be exponentially larger. Finally, we convert the resulting regular expression into an LDL$_f$ formula with constant blow-up (De Giacomo and Vardi, 2013).

The converse case, i.e., from LDL$_f$ to PPLDL, follows directly by considering the swapped formulas. ☐

Figure 3.1 reports an overview of the main transformations between linear temporal and dynamic logics with other well-known formal languages and automata.

In light of the previously presented results, while PPLTL and PPLDL allow for exponentially smaller equivalent DFA compared to LTL$_f$ and LDL$_f$, translating LTL$_f$/LDL$_f$ into PPLTL/PPLDL to take advantage of the exponential advantage is not advisable because the translation itself is too expensive. Therefore, the properties of interest should be *naturally* expressible directly in PPLTL/PPLDL to really exploit the exponential improvement when translating them into automata.

Finally, later in the dissertation, especially in the evaluation experiments of Chapters 5 and 6, we will need to manually translate PPLTL formulas to semantically equivalent LTL$_f$ ones (and vice versa) for comparability reasons between our system and existing ones. Given that two formulas are semantically equivalent if they

56

define the same language, we can prove equivalence in terms of languages, exploiting the fact that both PPLTL and LTL$_f$ can be translated into DFAs, for which language equivalence is easy (indeed since every regular language has a unique minimal DFA modulo state renaming, language equivalence reduces to solving graph isomorphism). We employed this technique to formally check the correctness of translations for both tables (PDDL3 and DECLARE) in the next section and all the formulas in our experiments. We computed the minimal DFAs for the formulas using the open-source tool LTLf2DFA[1].

3.6   Examples

The limited attention given in some areas of AI – automated planning included – to the pure-past temporal logics is not a good indicator of their relevance and benefits when employed as specification formalisms. In fact, contrary to expectations, there are several properties that are natural to express using the past. For instance, "serve coffee to X only if it has been requested and not already served in the past" is written in PPLTL as "H(*CoffeeServed* → Y(¬*CoffeeServed* S *CoffeeRequested*))". In general, as mentioned at the beginning of this chapter, PPLTL and PPLDL have already been used in several contexts, e.g.,in (Bacchus et al., 1996; Fisher and Wooldridge, 2005; Gabaldon, 2011; Knobbout et al., 2016; Alechina et al., 2018; Cimatti et al., 2020), and have shown to be helpful for specifying temporally extended properties.

Therefore, we provide some examples of PPLTL formulas – almost all applications have shown a preference for PPLTL over PPLDL– to demonstrate that PPLTL is an appropriate and helpful formalism to specify temporal properties or tasks in AI.

In many cases, we want the agent to achieve a goal g after some condition c has been met. In this setting, we identify the *Immediate-Response* pattern as $g \wedge Y(c)$ and the *Bounded-Response* pattern $g \wedge Y^i(c)$ for $1 \leq i \leq n$, where n is the time-bound within which the agent achieves the goal g. These and other patterns have been employed in the context of MDP rewards in Bacchus et al. (1996).

Then, among common formulas, we also find the *Strict-Sequence* pattern as $O(a \wedge Y(O(b \wedge Y(O(\dots)))))$ forcing the agent to achieve tasks $a, b, \dots$ sequentially, and the *Eventually-All* pattern as $\bigwedge_{i=1}^{n} O(a_i)$ requiring to eventually achieve all tasks a_i.

Additionally, widely used formula patterns can be found in PDDL3 (Table 3.1) that standardized certain modal operators (Gerevini et al., 2009) and in DECLARE (Table 3.2) that is the *de-facto* standard encoding language for Business Processes behaviors (van der Aalst et al., 2009). Table 3.1 and Table 3.2 are both a non-exhaustive list of such common patterns, including their translation to equivalent LTL$_f$ formulas (De Giacomo et al., 2014; Camacho et al., 2019a). A contribution of this dissertation is the translation to their

---

[1] http://ltlf2dfa.diag.uniroma1.it

PDDL3 Operator	Equivalent PPLTL Formula	Equivalent LTL$_f$ Formula
(at-end θ)	θ	$\mathsf{F}(\theta \wedge \mathsf{final})$
(always θ)	$\mathsf{H}(\theta)$	$\mathsf{G}(\theta)$
(sometime θ)	$\mathsf{O}(\theta)$	$\mathsf{F}(\theta)$
(sometime-after $\theta_1\ \theta_2$)	$(\neg\theta_1 \mathbin{\mathsf{S}} \theta_2) \vee \mathsf{H}(\neg\theta_1)$	$\mathsf{G}(\theta_1 \to \mathsf{F}(\theta_2))$
(sometime-before $\theta_1\ \theta_2$)	$\mathsf{H}(\theta_1 \to \mathsf{Y}(\mathsf{O}(\theta_2)))$	$\theta_2 \mathbin{\mathsf{R}} \neg\theta_1$
(at-most-once θ)	$\mathsf{H}(\theta \to (\theta \mathbin{\mathsf{S}} (\mathsf{H}(\neg\theta) \vee \mathsf{start})))$	$\mathsf{G}(\theta \to (\theta \mathbin{\mathsf{U}} (\mathsf{G}(\neg\theta) \vee \mathsf{final})))$
(hold-during $n_1\ n_2\ \theta$)	$\bigvee_{0 \le i \le n_1}(\theta \wedge \mathsf{Y}^i(\mathsf{start}))\vee$ $\bigwedge_{n_1 < i \le n_2} \mathsf{H}(\theta \vee \mathsf{WY}^i(\mathsf{Y}(true)))$	$\bigvee_{0 \le i \le n_1} \mathsf{X}^i(\theta \wedge \mathsf{final})\vee$ $\bigwedge_{n_1 < i \le n_2} \mathsf{WX}^i(\theta)$
$*$ (hold-after $n\ \theta$)	$\bigvee_{0 \le i \le n}(\theta \wedge \mathsf{Y}^i(\mathsf{start}))\vee$ $\mathsf{O}(\theta \wedge \mathsf{Y}^{n+1}(\mathsf{O}(\mathsf{start})))$	$\bigvee_{0 \le i \le n} \mathsf{X}^i(\theta \wedge \mathsf{final})\vee$ $\mathsf{X}^{n+1}(\mathsf{F}(\theta))$

Table 3.1: PDDL3 operators, their equivalent PPLTL and LTL$_f$ formulas. Superscripts abbreviate nested temporal operators. θ is a propositional formula. The $*$ tags the operator with the corrected LTL$_f$ translation.

equivalent PPLTL formulas.

Notably, many, but not all, formulas have a straightforward translation to the corresponding pure-future LTL$_f$ formula. However, even though a systematic translation between LTL$_f$ and PPLTL (and vice versa) does exist (cf. Section 3.5.2), such a translation is impractical (3EXPTIME). Such a high upper bound means that the systematic translation is not simply based on an induction on the structure of the formula. Hence, some inductive formula patterns like $\alpha \mathbin{\mathsf{U}} \beta$ (or, equivalently, $\alpha \mathbin{\mathsf{S}} \beta$) are not easily translatable.

3.7 Impact of Adopting Pure-Past Temporal and Dynamic Logics

Compared to LTL$_f$/LDL$_f$, the exponential gain in transforming PPLTL/PPLDL formulas into DFAs is reflected in an exponential gain in solving a variety of forms of sequential decision-making problems involving temporal specifications. All these sequential decision-making problems involving temporal specifications exploit the intimate connection between linear temporal logics and automata theory. Such a connection relies on the fact that linear temporal and dynamic logics on finite traces can be transformed into finite automata.

In particular, while Fully Observable Nondeterministic planning (FOND) for LTL$_f$/LDL$_f$ goals has been characterized to be EXPTIME-complete in the domain specification and 2EXPTIME-complete in the LTL$_f$/LDL$_f$ goals (De Giacomo and Rubin, 2018), solving FOND planning for PPLTL/PPLDL goals becomes EXPTIME-complete in the domain and EXPTIME-complete in the PPLTL/PPLDL goals.

Clearly, the exponential gain is only achieved when properties can be succinctly expressed using the past. In fact, if we first express the specification in LTL$_f$/LDL$_f$ and then translate it into PPLTL/PPLDL, we lose the advantage given the impractical systematic translation.

The above-mentioned results can be adapted to handle *stochastically* fair domains (De Giacomo and

DECLARE Template	Equivalent PPLTL Formula	Equivalent LTL$_f$ Formula
init(a)	$\mathsf{O}(a \wedge \neg\mathsf{Y}(true))$	a
existence(a)	$\mathsf{O}(a)$	$\mathsf{F}(a)$
absence(a)	$\neg\mathsf{O}(a)$	$\neg\mathsf{F}(a)$
absence2(a)	$\mathsf{H}(a \to \mathsf{WYH}(\neg a))$	$\neg(\mathsf{F}a \wedge \mathsf{XF}(a))$
choice(a,b)	$\mathsf{O}(a) \vee \mathsf{O}(b)$	$\mathsf{F}(a) \vee \mathsf{F}(b)$
exclusive-choice(a,b)	$(\mathsf{O}(a) \vee \mathsf{O}(b)) \wedge \neg(\mathsf{O}(a) \wedge \mathsf{O}(b))$	$(\mathsf{F}(a) \vee \mathsf{F}(b)) \wedge \neg(\mathsf{F}(a) \wedge \mathsf{F}(b))$
co-existence(a,b)	$\mathsf{H}(\neg a) \leftrightarrow \mathsf{H}(\neg b)$	$\mathsf{F}(a) \leftrightarrow \mathsf{F}(b)$
responded-existence(a,b)	$\mathsf{O}(a) \to \mathsf{O}(b)$	$\mathsf{F}(a) \to \mathsf{F}(b)$
response(a,b)	$(\neg a \, \mathsf{S} \, b) \vee \mathsf{H}(\neg a)$	$\mathsf{G}(a \to \mathsf{F}(b))$
precedence(a,b)	$\mathsf{H}(b \to \mathsf{O}(a))$	$(\neg b \, \mathsf{U} \, a) \vee \mathsf{G}(\neg b)$
succession(a,b)	response$(a,b) \wedge$ precedence(a,b)
chain-response(a,b)	$\mathsf{H}(\mathsf{Y}(a) \to b) \wedge \neg a$	$\mathsf{G}(a \to \mathsf{X}(b))$
chain-precedence(a,b)	$\mathsf{H}(b \to \mathsf{Y}(a))$	$\mathsf{G}(\mathsf{X}(b) \to a) \wedge \neg b$
chain-succession(a,b)	$(\mathsf{H}(\mathsf{Y}(a) \to b) \wedge \neg a) \wedge$ $\mathsf{H}(\mathsf{Y}(\neg a) \to \neg b)$	$\mathsf{G}(a \leftrightarrow \mathsf{X}(b))$
not-co-existence(a,b)	$\mathsf{O}(a) \to \neg\mathsf{O}(b)$	$\mathsf{F}(a) \to \neg\mathsf{F}(b)$
not-succession(a,b)	$\mathsf{H}(b \to \neg\mathsf{O}(a))$	$\mathsf{G}(a \to \neg\mathsf{F}(b))$
not-chain-succession(a,b)	$\mathsf{H}(b \to \neg\mathsf{Y}(a))$	$\mathsf{G}(a \to \neg\mathsf{X}(b))$

Table 3.2: DECLARE templates, their equivalent PPLTL and LTL$_f$ formulas. a, b are atomic propositions.

Rubin, 2018; Aminof et al., 2020) with the same exponential advantage. Observe that when the domain is deterministic, the difference in theoretical complexity between LTL$_f$/LDL$_f$ and PPLTL/PPLDL disappears because, for LTL$_f$/LDL$_f$, we can directly work with an NFA, i.e., and check for nonemptiness (cf. (De Giacomo and Rubin, 2018)). In both cases, the complexity is PSPACE in both the domain and the temporal goal.

An analogous line of reasoning can be exploited to show an exponential improvement in several other contexts, as follows.

- Solving Markov Decision Processes (MDP) with non-Markovian rewards (Bacchus et al., 1996; Thiébaux et al., 2006; Brafman et al., 2018) with PPLTL/PPLDL rewards is EXPTIME-complete in the domain and EXPTIME in PPLTL/PPLDL rewards, while the latter is 2EXPTIME-complete for LTL$_f$/LDL$_f$ rewards (Brafman et al., 2018);

- Reinforcement Learning where rewards are based on traces (De Giacomo et al., 2019; Camacho et al., 2019b) with PPLTL/PPLDL rewards also gains the exponential improvement;

- Planning in non-Markovian domains (Brafman and De Giacomo, 2019a), with both the non-Markovian domain and the goal expressed in PPLTL/PPLDL is EXPTIME-complete in the domain and in the goal, vs. 2EXPTIME-complete in the domain and in the goal in the case these are expressed in LTL$_f$/LDL$_f$;

- Solving non-Markovian decision processes (NMDP) (Brafman and De Giacomo, 2019b), with both the

system dynamics and the rewards expressed using PPLTL/PPLDL, is EXPTIME-complete in the domain and in the rewards specification.

3.8   Summary and Discussion

In this chapter, we considered the Pure-Past Linear Temporal Logic (PPLTL) and its extension Pure-Past Linear Dynamic Logic (PPLDL) as first-class citizens of our research and described their main properties and characteristics. By exploiting a well-known foundational result on reverse languages, we provided an algorithm to translate PPLTL and PPLDL formulas into their corresponding DFA that has an exponential improvement if compared to the algorithm to translate LTL$_f$ and LDL$_f$ formulas into DFA. Then, we have reviewed the relationship between PPLTL/PPLDL and other formal languages, including FOL, RE, and MSO. These known results allowed us to expand the research by establishing that PPLTL and PPLDL have the same expressive power as LTL$_f$ and LDL$_f$, respectively, but transforming a PPLTL or PPLDL formula into its equivalent LTL$_f$ or LDL$_f$ formula is computationally prohibitive (i.e., worst-case 3EXPTIME). Moving beyond considering only theoretical results for PPLTL and PPLDL, we have shown their applicability in a range of domains, giving translations in PPLTL of PDDL3 modal operators and of DECLARE patterns. Finally, we have discussed the impact of using pure-past temporal logics on well-known sequential decision-making problems, such as planning and decision problems in nondeterministic and non-Markovian domains, in comparison with LTL$_f$ and LDL$_f$.

Chapter 4

Handling Pure-Past Linear Temporal Logic Formulas

In this chapter, we focus on PPLTL only and study the theoretical foundations and properties to develop an efficient technique for handling and evaluating PPLTL formulas. We exploit the well-known fixpoint characterization of temporal logics formulas on finite traces to determine when PPLTL formulas become true. Then, given some key observations, we devise a novel technique to evaluate the truth of PPLTL formulas using only a small set of subformulas. We formally prove the correctness of this technique and demonstrate its usefulness in developing more efficient algorithms for various application domains. Specifically, in Chapters 5 and 6, we will examine how this technique can be successfully applied to deterministic and nondeterministic planning.

4.1 Fixpoint Characterization

From the literature, it is well-known that LTL and variants have a convenient fixpoint characterization that allows splitting any formula into a propositional formula to be checked at the current instant and a temporal formula to be checked at the next instant (Gabbay et al., 1980; Manna, 1982; Emerson, 1990) (cf. Section 2.4.1). The fixpoint characterization property has already been exploited in AI, e.g., in the MetateM approach (Barringer et al., 1989), and later under the name of "formula progression" in (Bacchus and Kabanza, 1996), which is perhaps the most influential work on planning for temporally extended goals.

Analogously to the fixpoint characterization of LTL and variants, when we consider PPLTL formulas, such a fixpoint characterization splits the formula into a propositional formula on the current instant and a temporal

formula on the past to be checked at the *previous* instant. However, while the future has not happened yet and needs to be guessed, the past has already happened and needs only to be read. This implies that PPLTL formulas can be easily evaluated by recursively applying their fixpoint characterization. A similar line of reasoning was exploited to conveniently handle non-Markovian rewards expressed using PPLTL in (Bacchus et al., 1997), but never fully formalized.

To begin with, we observe that any sequence of actions produces a trace on which PPLTL formulas can be evaluated. Therefore, while the planning process goes on, sequences of actions are produced, traces are generated, and over them, PPLTL goals can be evaluated. The difficulty is that evaluating PPLTL formulas requires a trace, and searching through traces is quite demanding. Instead, our technique does not consider traces at all. In particular, it exploits the following observations:

- to evaluate the PPLTL goal formula, we only need the truth value of its subformulas;

- every PPLTL formula can be put in a form where its evaluation depends only on the current propositional evaluation and the evaluation of a key set of PPLTL subformulas at the previous instant; and

- one can recursively compute and keep the value of such a small set of formulas as additional propositional variables in the state of the planning domain.

In the following, we detail these observations.

Temporal operators in LTL and LTL$_f$ can be decomposed into present and future components, giving a fixpoint characterization of the *until* operator (U) – we remind the reader that other temporal operators as *eventually* (F) and *always* (G) are abbreviations of formulas involving U:

$$\phi_1 \, \mathsf{U} \, \phi_2 \;\equiv\; \phi_2 \vee (\phi_1 \wedge \mathsf{X}(\phi_1 \, \mathsf{U} \, \phi_2))$$

Analogously, PPLTL formulas can be decomposed into present and past components, given the fixpoint characterization of the since operator:

$$\phi_1 \, \mathsf{S} \, \phi_2 \;\equiv\; \phi_2 \vee (\phi_1 \wedge \mathsf{Y}(\phi_1 \, \mathsf{S} \, \phi_2)).$$

Exploiting this equivalence, the formula decomposition can be computed by recursively applying the following transformation function $\mathsf{pnf}(\cdot)$:

- $\mathsf{pnf}(p) = p$;

- $\mathsf{pnf}(\mathsf{Y}\phi) = \mathsf{Y}\phi$;

- $\mathsf{pnf}(\phi_1 \, \mathsf{S} \, \phi_2) = \mathsf{pnf}(\phi_2) \vee (\mathsf{pnf}(\phi_1) \wedge \mathsf{Y}(\phi_1 \, \mathsf{S} \, \phi_2))$;

- $\mathsf{pnf}(\phi_1 \wedge \phi_2) = \mathsf{pnf}(\phi_1) \wedge \mathsf{pnf}(\phi_2)$;

- $\mathsf{pnf}(\neg\phi) = \neg\mathsf{pnf}(\phi)$.

For convenience, we add $\mathsf{pnf}(\mathsf{O}\phi) = \mathsf{pnf}(\phi) \vee \mathsf{Y}(\mathsf{O}\phi)$.

A formula resulting from the application of $\mathsf{pnf}(\cdot)$ is said to be in Previous Normal Form (PNF). Note that formulas in PNF have proper temporal subformulas (i.e., subformulas whose main construct is a temporal operator) appearing only in the scope of the Y operator. Also, observe that the formulas of the form $\mathsf{Y}\phi$ in $\mathsf{pnf}(\varphi)$ are such that $\phi \in \mathsf{sub}(\varphi)$. It is easy to see that the following proposition holds:

Proposition 4.1. *Every* PPLTL *formula* φ *can be converted to its* PNF *form* $\mathsf{pnf}(\varphi)$ *in linear-time in the size of the formula (i.e.,* $|\mathsf{sub}(\varphi)|$*). Moreover,* $\mathsf{pnf}(\varphi)$ *is equivalent to* φ.

Proof. We can prove $\varphi \equiv \mathsf{pnf}(\varphi)$ by structural induction on the formula.

- $\varphi = p$. Immediate by construction.

- $\varphi = \mathsf{Y}(\varphi')$. Immediate by construction.

- $\varphi = \varphi_1 \, \mathsf{S} \, \varphi_2$. Here, we want to prove the claim $\tau, i \models \varphi_1 \, \mathsf{S} \, \varphi_2 \equiv \tau, i \models \varphi_2 \vee (\varphi_1 \wedge \mathsf{Y}(\varphi_1 \, \mathsf{S} \, \varphi_2))$. By the semantics of S, we have that $\tau, i \models \varphi_1 \, \mathsf{S} \, \varphi_2$ if and only if there exists k, with $0 \leq k \leq i$, such that $\tau_0, \ldots, \tau_k \models \varphi_2$ and $\tau_0, \ldots, \tau_j \models \varphi_1$ for all j with $k < j \leq i$. Now, we distinguish between the cases $k = i$ and $0 \leq k < i$.

  - Let $k = i$. If $k = 1$, then $\tau_0, \ldots, \tau_k \models \varphi_2$ by hypothesis, and thus $\tau_0, \ldots, \tau_i \models \varphi_2$. Hence, we will have $\varphi_2 \vee \ldots$.

  - Let $0 \leq k < i$. Then, we have that the claim is equal to $\tau_0, \ldots, \tau_i \models \varphi_1$ and $\tau_0, \ldots, \tau_{i-1} \models \varphi_1 \, \mathsf{S} \, \varphi_2$, meaning that φ_1 must hold now and $\varphi_1 \, \mathsf{S} \, \varphi_2$ must hold at the previous step. Therefore, the second part can be rewritten as $\tau_0, \ldots, \tau_i \models \mathsf{Y}(\varphi_1 \, \mathsf{S} \, \varphi_2)$, by definition of the Y operator with $i > 0$. Hence, we will have $\varphi_1 \wedge \mathsf{Y}(\varphi_1 \, \mathsf{S} \, \varphi_2)$.

  Combining both cases, we get $\tau_0, \ldots, \tau_i \models \varphi_2 \vee (\varphi_1 \wedge \mathsf{Y}(\varphi_1 \, \mathsf{S} \, \varphi_2))$ as required.

- $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \neg\varphi'$. The thesis holds by structural induction.

Then, since no expansion is applied to the Y operator, the conversion cost is at most linear. $\qquad \square$

4.2 Evaluation of Pure-Past Linear Temporal Logic Formulas

Interestingly, the PNF decomposition allows us to characterize the evaluation of a PPLTL formula φ by only keeping track of the truth values of all subformulas that are within the Y-scope. To do so, we introduce Σ_φ as the set of *propositions* of the form "Yϕ" containing:

- "Yϕ" for each subformula of φ of the form Yϕ;

- "Y$(\phi_1 \mathsf{S} \phi_2)$" for each subformula of φ of the form $\phi_1 \mathsf{S} \phi_2$.

We interpret these specific subformulas as *atomic propositions*, denoting them with quotes and collecting them in a set denoted as Σ_φ. To keep track of the truth of each proposition in Σ_φ, we define a specific interpretation σ:

$$\sigma : \Sigma_\varphi \to \{\top, \bot\}$$

Intuitively, given an instant i, σ_i tells us which propositions related to the previous instant (i.e., in Σ_φ) are true at the instant i. Therefore, a PPLTL formula can simply be evaluated by using the propositional interpretation in the current instant i and the truth value assigned by σ_i to propositions related to the previous instant.

Definition 4.2. *Let s_i be a propositional interpretation over $\mathcal{P}$, σ_i a propositional interpretation over Σ_φ, and ϕ a PPLTL subformula in $\mathsf{sub}(\varphi)$, we define the predicate $\mathsf{val}(\phi, \sigma_i, s_i)$, recursively as follows:*

- $\mathsf{val}(p, \sigma_i, s_i)$ *iff* $s_i \models p$;

- $\mathsf{val}(\mathsf{Y}\phi', \sigma_i, s_i)$ *iff* $\sigma_i \models$ "Yϕ'";

- $\mathsf{val}(\phi_1 \mathsf{S} \phi_2, \sigma_i, s_i)$ *iff* $\mathsf{val}(\phi_2, \sigma_i, s_i) \vee (\mathsf{val}(\phi_1, \sigma_i, s_i) \wedge \sigma_i \models$ "Y$(\phi_1 \mathsf{S} \phi_2)$"*)*;

- $\mathsf{val}(\phi_1 \wedge \phi_2, \sigma_i, s_i)$ *iff* $\mathsf{val}(\phi_1, \sigma_i, s_i) \wedge \mathsf{val}(\phi_2, \sigma_i, s_i)$;

- $\mathsf{val}(\neg\phi', \sigma_i, s_i)$ *iff* $\neg\mathsf{val}(\phi', \sigma_i, s_i)$.

Intuitively, the $\mathsf{val}(\phi, \sigma_i, s_i)$ predicate allows us to determine what is the truth value of any PPLTL formula $\phi \in \mathsf{sub}(\varphi)$ by reading a propositional interpretation s_i from trace τ and keeping track of the truth value of propositions in Σ_φ by means of σ_i. Observe that rules in Definition 4.2 basically follow the PNF transformation rules where subformulas within the Y-scope are interpreted as propositions.

Now, given a trace $\tau = s_0, \ldots, s_n$ over $\mathcal{P}$, we compute a corresponding trace $\tau^{[\varphi]} = \sigma_0, \ldots, \sigma_n$ over Σ_φ, where:

- $\sigma_0($"Yϕ"$) \doteq \bot$ for each "Yϕ" $\in \Sigma_\varphi$;

- $\sigma_i(\text{``}\mathsf{Y}\phi\text{''}) \doteq \mathsf{val}(\phi, \sigma_{i-1}, s_{i-1})$, for all i with $0 < i \leq n$.

We can show that for traces of length 1, the following result holds.

Lemma 4.3. *Let φ be* PPLTL *formula over $\mathcal{P}$, $\phi \in \mathsf{sub}(\varphi)$ a subformula of φ, and $\tau = s_0$ a trace over $\mathcal{P}$ of length 1. Then, $s_0 \models \phi$ iff $\mathsf{val}(\phi, \sigma_0, s_0)$.*

Proof. By structural induction on the formula ϕ.

- $\phi = p$. By definition of $\mathsf{val}(\cdot)$, $\mathsf{val}(p, \sigma_0, s_0)$ iff $s_0 \models p$.

- $\phi = \mathsf{Y}\phi'$. By definition of σ_0, $\sigma_0(\text{``}\mathsf{Y}\phi'\text{''}) = \bot$, and by the semantics, $s_0 \not\models \mathsf{Y}\phi'$. Therefore, the thesis holds.

- $\phi = \phi_1 \mathsf{S} \phi_2$. $\mathsf{val}(\phi_1 \mathsf{S} \phi_2, \sigma_i, s_i)$ iff $\mathsf{val}(\phi_2, \sigma_i, s_i) \vee (\mathsf{val}(\phi_1, \sigma_i, s_i) \wedge \sigma_i \models \text{``}\mathsf{Y}(\phi_1 \mathsf{S} \phi_2)\text{''})$. By definition of $\sigma_0$, $\sigma_0(\text{``}\mathsf{Y}(\phi_1 \mathsf{S} \phi_2)\text{''}) = \bot$, hence the formula above simplifies to $\mathsf{val}(\phi_2, \sigma_i, s_i)$. On the other hand, by the semantics, $s_0 \models \phi_1 \mathsf{S} \phi_2$ iff $s_0 \models \phi_2$. Hence, by induction the thesis holds.

- $\phi = \phi_1 \wedge \phi_2$ or $\phi = \neg\phi'$. The thesis holds by structural induction.

$\square$

Next, we extend the previous result to traces of any length.

Theorem 4.4. *Let φ be a* PPLTL *formula over $\mathcal{P}$, $\phi \in \mathsf{sub}(\varphi)$ a subformula of φ, τ a trace over $\mathcal{P}$, and $\tau^{[\varphi]}$ the corresponding trace over Σ_φ. Then,*

$$\tau \models \phi \ \ \textit{iff} \ \ \mathsf{val}(\phi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau)).$$

Proof. We prove the thesis by double induction on the length of the trace τ and on the structure of the formula ϕ.

- Base case: $\tau = s_0$. By Lemma 4.3, the thesis holds.

- Inductive step: Let $\tau = \tau_{n-1} \cdot s_n$. By inductive hypothesis, the thesis holds for the trace τ_{n-1} of length $n-1$:

$$\tau_{n-1} \models \phi \ \ \text{iff} \ \ \mathsf{val}(\phi, \mathsf{last}(\tau_{n-1}^{[\varphi]}), \mathsf{last}(\tau_{n-1}))$$

Now, we prove that the thesis also holds for $\tau_{n-1} \cdot s_n$:

$$\tau_{n-1} \cdot s_n \models \phi \ \ \text{iff} \ \ \mathsf{val}(\phi, \mathsf{last}((\tau_{n-1} \cdot s_n)^{[\varphi]}), \mathsf{last}(\tau_{n-1} \cdot s_n))$$

To prove the claim, we proceed by structural induction on the formula, knowing that $\mathsf{last}((\tau_{n-1}\cdot s_n)^{[\varphi]}) = \sigma_n$ and $\mathsf{last}(\tau_{n-1}\cdot s_n) = s_n$:

- $\phi = p$. We have that $\tau_{n-1}\cdot s_n \models p$ iff $s_n \models p$. For the $\mathsf{val}(\cdot)$ predicate we have that $s_n \models p$ iff $\mathsf{val}(p, \sigma_n, s_n)$. Therefore, the thesis holds.

- $\phi = \mathsf{Y}\phi'$. We have that $\tau_{n-1}\cdot s_n \models \mathsf{Y}\phi'$ iff $\tau_{n-1} \models \phi'$. By inductive hypothesis, $\tau_{n-1} \models \phi'$ iff $\mathsf{val}(\phi', \mathsf{last}(\tau_{n-1}^{[\varphi]}), \mathsf{last}(\tau_{n-1}))$. For the $\mathsf{val}(\cdot)$ predicate $\mathsf{val}(\mathsf{Y}\phi', \sigma_n, s_n)$ iff $\sigma_n \models$ "$\mathsf{Y}\phi'$", which in turn is defined as $\mathsf{val}(\phi', \mathsf{last}(\tau_{n-1}^{[\varphi]}), \mathsf{last}(\tau_{n-1}))$. Hence the thesis holds.

- $\phi = \phi_1 \mathsf{S}\phi_2$. In this case it suffices to remember that $\tau_{n-1}\cdot s_n \models \phi_1 \mathsf{S}\phi_2$ iff $\tau_{n-1}\cdot s_n \models \phi_2 \vee (\phi_1 \wedge \mathsf{Y}(\phi_1 \mathsf{S}\phi_2))$. On the other hand, $\mathsf{val}(\phi_1 \mathsf{S}\phi_2, \sigma_n, s_n)$ iff $\mathsf{val}(\phi_2, \sigma_n, s_n) \vee (\mathsf{val}(\phi_1, \sigma_n, s_n) \wedge \sigma_n \models$ "$\mathsf{Y}(\phi_1 \mathsf{S}\phi_2)$"). By structural induction we have that $\tau_{n-1}\cdot s_n \models \phi_1$ iff $\mathsf{val}(\phi_1, \sigma_n, s_n)$, and $\tau_{n-1}\cdot s_n \models \phi_2$ iff $\mathsf{val}(\phi_2, \sigma_n, s_n)$. Moreover, $\tau_{n-1}\cdot s_n \models \mathsf{Y}(\phi_1 \mathsf{S}\phi_2)$ iff $\tau_{n-1} \models \phi_1 \mathsf{S}\phi_2$, and $\sigma_n \models$ "$\mathsf{Y}(\phi_1 \mathsf{S}\phi_2)$" iff $\mathsf{val}(\phi_1 \mathsf{S}\phi_2, \mathsf{last}(\tau_{n-1}^{[\varphi]}), \mathsf{last}(\tau_{n-1}))$. Finally, we have that $\tau_{n-1} \models \phi_1 \mathsf{S}\phi_2$ iff $\mathsf{val}(\phi_1 \mathsf{S}\phi_2, \mathsf{last}(\tau_{n-1}^{[\varphi]}), \mathsf{last}(\tau_{n-1}))$ holds by induction on the length of the trace.

- $\phi = \phi_1 \wedge \phi_2$ or $\phi = \neg\phi'$. The thesis holds by structural induction.

$\square$

Theorem 4.4 gives us the bases of our technique. Specifically, Theorem 4.4 guarantees that by keeping a suitably updated trace σ, we can evaluate our PPLTL goal only using the propositional interpretation in the current instant and the truth value of the "$\mathsf{Y}\phi$" formulas in σ, without considering the entire trace. Moreover, Theorem 4.4 can be seen as another way to explore runs of the DFA corresponding to the PPLTL formula φ. In fact, by computing which subformulas of φ are true at every instant while scanning a given trace τ, one is implicitly building the states of the DFA corresponding to φ on the fly.

4.3   Examples

In this section, we report some examples to clarify how the mathematical construction of the previous section works.

Example 4.5. *Let $\varphi = \mathsf{Y}(a)$ be the PPLTL formula under examination. The set of subformulas of φ are simply $\mathsf{sub}(\varphi) = \{a, \mathsf{Y}(a)\}$, whereas the $\mathsf{pnf}(\varphi) = \mathsf{Y}(a)$. Given the PNF of φ, we can characterize the evaluation of φ in terms of a subset of its subformulas. In this case where $\varphi = \mathsf{Y}(a)$, from its PNF, we get that $\Sigma_\varphi = \{$"$\mathsf{Y}(a)$"$\}$. This means that to evaluate the truth value of φ, we only need to keep track of the truth value of the proposition "$\mathsf{Y}(a)$".*

Now, we consider the trace $\tau = \{a\}, \emptyset$. *At the beginning, we know that* $\sigma_0 = \sigma_0("Y(a)") = \bot$ *by definition. For each subsequent instant of time* i, *following the theory, we can compute the truth value of the proposition* "$Y(a)$" *applying* $\sigma_i("Y(a)") = \mathsf{val}(a, \sigma_{i-1}, s_{i-1})$. *Hence, by applying the rules in Definition 4.2 and consuming the two symbols we have in the trace* τ, *we get that:*

- $\sigma_0("Y(a)") = \bot$ *by definition;*

- $\sigma_1("Y(a)") = \mathsf{val}(a, \sigma_0, s_0) = \mathsf{val}(a, \sigma_0, \{a\}) = \top$ *as* $\{a\} \models a$.

From Theorem 4.4, we have that $\tau \models \varphi$ *iff* $\mathsf{val}(\varphi, \sigma_1, s_1)$ *as* σ_1 *and* s_1 *represent the last instants of the traces* $\tau^{[\varphi]}$ *and* τ, *respectively. Substituting the values, we get* $\{a\}, \emptyset \models Y(a)$ *iff* $\mathsf{val}(Y(a), \{"Y(a)"\}, \{\})$. *Now,* $\mathsf{val}(Y(a), \{"Y(a)"\}, \{\})$ *is true if and only if, by the rules in Definition 4.2,* $\{"Y(a)"\} \models "Y(a)"$, *which is trivially true. Thus, the trace* $\tau = \{a\}, \emptyset$ *makes the* PPLTL *formula* $Y(a)$ *true.*

We can represent the evaluation of propositions in Σ_φ *and the truth value of* φ *graphically as follows.*

Propositions in Σ_φ	**Trace $\tau^{[\varphi]}$**
"$Y(a)$"	$\longrightarrow\!\bigcirc\!\longrightarrow\!\bullet$
Formula	**Trace τ**
$Y(a)$	$\longrightarrow\!\bigcirc\!\longrightarrow\!\bullet$

In particular, we can read the table as the proposition "$Y(a)$" *is true (filled circle) in the last instant of* $\tau^{[\varphi]}$, *and therefore the formula* $Y(a)$ *is true on the trace* τ. *Given that the formula is a* PPLTL *formula, as described in Section 3.2, the formula is true in* τ *if it is satisfied in the last instant of the trace.*

Now, we can consider a slightly more complex example.

Example 4.6. *Let* $\varphi = Y(a) \wedge (\neg b \, S \, c)$ *be the* PPLTL *formula under examination. The set of subformulas of* φ *are* $\mathsf{sub}(\varphi) = \{a, b, c, \neg b, Y(a), (\neg b \, S \, c), Y(a) \wedge (\neg b \, S \, c)\}$, *whereas the* $\mathsf{pnf}(\varphi) = Y(a) \wedge (\neg b \vee (c \wedge Y(\neg b \, S \, c)))$. *As before, we characterize the evaluation of* φ *in terms of a subset of its subformulas, given the* PNF *of* φ. *In this case, we get that* $\Sigma_\varphi = \{"Y(a)", "Y(\neg b \, S \, c)"\}$. *This means that to evaluate the truth value of* φ, *we only need to keep track of the truth value of the propositions* "$Y(a)$" *and* "$Y(\neg b \, S \, c)$".

This time, we consider the trace $\tau = \{a\}, \{a, c\}, \{a, b\}, \{c\}$. *For each subsequent instant of time* i, *following the theory, we compute the truth value of the propositions* "$Y(a)$" *and* "$Y(\neg b \, S \, c)$" *by applying the predicates* $\mathsf{val}(a, \sigma_{i-1}, s_{i-1})$ *and* $\mathsf{val}(\neg b \, S \, c, \sigma_{i-1}, s_{i-1})$. *Hence, for* σ_0, *we have that:*

- $\sigma_0("Y(a)") = \sigma_0("Y(\neg b \, S \, c)") = \bot$ *by definition.*

Analyzing our interpretation σ_1, *we have:*

- $\sigma_1(\text{``Y}(a)\text{''}) = \mathsf{val}(a, \sigma_0, s_0) = \mathsf{val}(a, \sigma_0, \{a\}) = \top \ \textit{as } \{a\} \models a; \textit{ and}$

- $\sigma_1(\text{``Y}(\neg b \,\mathsf{S}\, c)\text{''}) = \mathsf{val}(\neg b \,\mathsf{S}\, c, \sigma_0, s_0)$

$$= \mathsf{val}(c, \sigma_0, s_0) \vee (\mathsf{val}(\neg b, \sigma_0, s_0) \wedge \sigma_0 \models \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''})$$

$$= \mathsf{val}(c, \sigma_0, \{a\}) \vee (\neg\mathsf{val}(b, \sigma_0, \{a\}) \wedge \{\} \models \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''})$$

$$= \bot \vee (\neg\bot \wedge \bot) \ \textit{as } \{a\} \not\models c \textit{ and } \{a\} \not\models b$$

$$= \bot.$$

Therefore, so far, we have $\sigma_0 = \{\}$ and $\sigma_1 = \{\text{``Y}(a)\text{''}\}$. Continuing with σ_2, we get the following:

- $\sigma_2(\text{``Y}(a)\text{''}) = \mathsf{val}(a, \sigma_1, s_1) = \mathsf{val}(a, \{\text{``Y}(a)\text{''}\}, \{a, c\}) = \top \ \textit{as } \{a, c\} \models a; \textit{ and}$

- $\sigma_2(\text{``Y}(\neg b \,\mathsf{S}\, c)\text{''}) = \mathsf{val}(\neg b \,\mathsf{S}\, c, \sigma_1, s_1)$

$$= \mathsf{val}(c, \sigma_1, s_1) \vee (\mathsf{val}(\neg b, \sigma_1, s_1) \wedge \sigma_1 \models \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''})$$

$$= \mathsf{val}(c, \sigma_1, \{a, c\}) \vee (\neg\mathsf{val}(b, \sigma_1, \{a, c\}) \wedge \{\text{``Y}(a)\text{''}\} \models \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''})$$

$$= \top \vee (\neg\bot \wedge \bot) \ \textit{as } \{a, c\} \models c \textit{ and } \{a, c\} \not\models b$$

$$= \top.$$

Thus, $\sigma_2 = \{\text{``Y}(a)\text{''}, \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''}\}$. We go on and compute the last interpretation σ_3 as follows:

- $\sigma_3(\text{``Y}(a)\text{''}) = \mathsf{val}(a, \sigma_2, s_2) = \mathsf{val}(a, \{\text{``Y}(a)\text{''}, \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''}\}, \{a, b\}) = \top \ \textit{as } \{a, b\} \models a; \textit{ and}$

- $\sigma_3(\text{``Y}(\neg b \,\mathsf{S}\, c)\text{''}) = \mathsf{val}(\neg b \,\mathsf{S}\, c, \sigma_2, s_2)$

$$= \mathsf{val}(c, \sigma_2, s_2) \vee (\mathsf{val}(\neg b, \sigma_2, s_2) \wedge \sigma_2 \models \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''})$$

$$= \mathsf{val}(c, \sigma_2, \{a, b\}) \vee (\neg\mathsf{val}(b, \sigma_2, \{a, b\}) \wedge \{\text{``Y}(a)\text{''}, \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''}\} \models \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''})$$

$$= \bot \vee (\neg\top \wedge \top) \ \textit{as } \{a, b\} \not\models c \textit{ and } \{a, b\} \models b$$

$$= \bot.$$

Hence, we have $\sigma_3 = \{\text{``Y}(a)\text{''}\}$.

Now, from Theorem 4.4, we have that $\tau \models \varphi$ iff $\mathsf{val}(\varphi, \sigma_3, s_3)$ as σ_3 and s_3 represent the last instants of the traces $\tau^{[\varphi]}$ and τ, respectively. Substituting the values, we get

$$\{a\}, \{a, c\}, \{a, b\}, \{c\} \models \mathsf{Y}(a) \wedge (\neg b \,\mathsf{S}\, c) \ \textit{ iff } \ \mathsf{val}(\mathsf{Y}(a) \wedge (\neg b \,\mathsf{S}\, c), \{\text{``Y}(a)\text{''}\}, \{c\}).$$

Now, by the rules in Definition 4.2, we have:

$$\mathsf{val}(\mathsf{Y}(a) \wedge (\neg b \,\mathsf{S}\, c), \{\text{``Y}(a)\text{''}\}, \{c\}) \ \textit{ iff } \ \mathsf{val}(\mathsf{Y}(a), \{\text{``Y}(a)\text{''}\}, \{c\}) \wedge \mathsf{val}(\neg b \,\mathsf{S}\, c, \{\text{``Y}(a)\text{''}\}, \{c\}).$$

Analyzing the truth value of both parts, we get:

- $\mathsf{val}(\mathsf{Y}(a), \{\text{``}\mathsf{Y}(a)\text{''}\}, \{c\}) = \top \quad as \quad \{\text{``}\mathsf{Y}(a)\text{''}\} \models \text{``}\mathsf{Y}(a)\text{''}; and$

- $\mathsf{val}(\neg b\,\mathsf{S}\,c, \{\text{``}\mathsf{Y}(a)\text{''}\}, \{c\}) = \mathsf{val}(c, \sigma_3, s_3) \vee (\mathsf{val}(\neg b, \sigma_3, s_3) \wedge \sigma_3 \models \text{``}\mathsf{Y}(\neg b\,\mathsf{S}\,c)\text{''})$

$$= \mathsf{val}(c, \sigma_3, \{c\}) \vee (\neg\mathsf{val}(b, \sigma_3, \{c\}) \wedge \{\text{``}\mathsf{Y}(a)\text{''}\} \models \text{``}\mathsf{Y}(\neg b\,\mathsf{S}\,c)\text{''})$$

$$= \top \vee (\neg\bot \wedge \bot) \; as \; \{c\} \models c \; and \; \{c\} \not\models b$$

$$= \top.$$

Given that both parts evaluate to true, then we can conclude that the trace τ makes the PPLTL formula $\mathsf{Y}(a) \wedge (\neg b\,\mathsf{S}\,c)$ true. Similarly to what we have shown in the previous example, we represent the evaluation of propositions in Σ_φ and the truth value of φ graphically as follows.

Propositions in Σ_φ	**Trace $\tau^{[\varphi]}$**
"$\mathsf{Y}(a)$"	→○→●→●→●
"$\mathsf{Y}(\neg b\,\mathsf{S}\,c)$"	→○→○→●→○
Formula	**Trace τ**
$\mathsf{Y}(a) \wedge (\neg b\,\mathsf{S}\,c)$	→○→●→○→●

In particular, the proposition "$\mathsf{Y}(a)$" is true from the second instant onward, whereas the proposition "$\mathsf{Y}(\neg b\,\mathsf{S}\,c)$" is true only on the last but one instant of the trace $\tau^{[\varphi]}$. Therefore, the formula $\mathsf{Y}(a) \wedge (\neg b\,\mathsf{S}\,c)$ is true on the second and the last instants of the trace τ. Given that the formula is a PPLTL formula and that is satisfied on the last instant of the trace, we conclude that $\tau \models \varphi$.

4.4 Relationship with Automata

As previously mentioned, given that reasoning on linear temporal logics is usually done by relying on automata theory, the clever evaluation of PPLTL formulas that we have shown in this chapter can also be seen as another way to explore the DFA associated with the PPLTL formula. Specifically, the computation of which key subformulas of a certain PPLTL formula are true at every instant of time along the trace essentially results in exploring runs of the DFA of the PPLTL formula. Clearly, the run depends on the symbols present in the trace.

In this section, we go through Examples 4.5 and 4.6 of the previous section and describe how the computation of truth values of formula propositions belonging to Σ_φ relates to the runs of an automaton.

Example 4.7. *In Example 4.5, the PPLTL formula is $\varphi = \mathsf{Y}(a)$ and the trace is $\tau = \{a\}, \emptyset$. By combining the input trace $\tau = \{a\}, \emptyset$ and the trace associated with the PPLTL formula $\tau^{[\varphi]} = \sigma_0, \sigma_1$ (whose components*

σ_0 and σ_1 are computed as shown in Example 4.5), we can construct the following augmented trace $(\tau, \tau^{[\varphi]})$ that simultaneously reads both τ and $\tau^{[\varphi]}$:

$$s_0 = \{a\} \qquad\qquad s_1 = \{\}$$

$$\sigma_0 = \{\} \qquad\qquad \sigma_1 = \{\text{``Y}(a)\text{''}\}$$
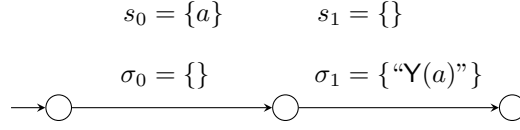
Now, the minimal DFA corresponding to the PPLTL formula $\mathsf{Y}(a)$ is as follows. Here, observe that since transitions in an automaton may be numerous, they are typically compactly represented as Boolean formulas that represent models of the propositional interpretations for each transition. In this example, a represents $\{a\}$, whereas $\neg a$ represents $\emptyset$.
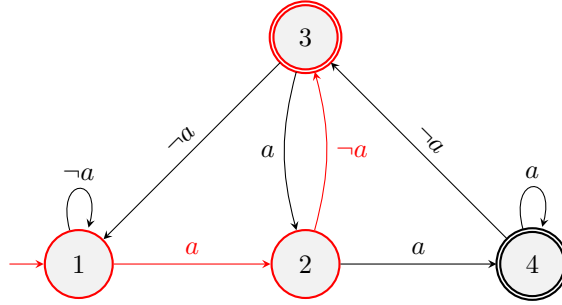
At this point, it is clear that the red colored part of the DFA actually corresponds to the augmented trace when projected on τ. In this respect, we can say that depending on the trace τ, the augmented trace $(\tau, \tau^{[\varphi]})$ computed through the technique previously described corresponds to exploring runs of the DFA of the formula.

The same line of reasoning can be applied to show the relationship between the construction in the Example 4.6 and the runs of the DFA associated with the PPLTL formula.

Example 4.8. *Given the formula* $\varphi = \mathsf{Y}(a) \wedge (\neg b \,\mathsf{S}\, c)$ *and the trace* $\tau = \{a\}, \{a, c\}, \{a, b\}, \{c\}$, *the augmented trace* $(\tau, \tau^{[\varphi]})$ *is constructed as follows.*

$$s_2 = \{a, b\}$$

$$s_0 = \{a\} \qquad s_1 = \{a, c\} \qquad \sigma_2 = \{\text{``Y}(a)\text{''}, \qquad s_3 = \{c\}$$

$$\sigma_0 = \{\} \qquad \sigma_1 = \{\text{``Y}(a)\text{''}\} \qquad \text{``Y}(\neg b \,\mathsf{S}\, c)\text{''}\} \qquad \sigma_3 = \{\text{``Y}(a)\text{''}\}$$
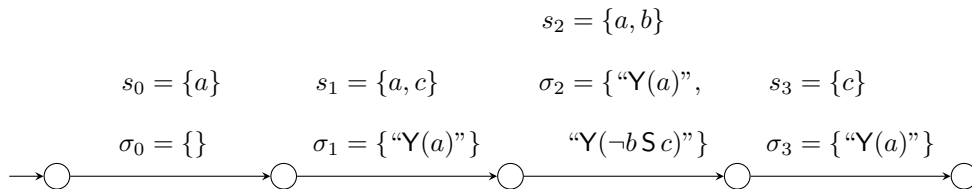
The minimal DFA associated with the PPLTL formula $\mathsf{Y}(a) \wedge (\neg b \,\mathsf{S}\, c)$ is as follows. Again, as before, since transitions in an automaton may be numerous (as in this particular case), they are compactly represented as Boolean formulas that represent models of the propositional interpretations for each transition.
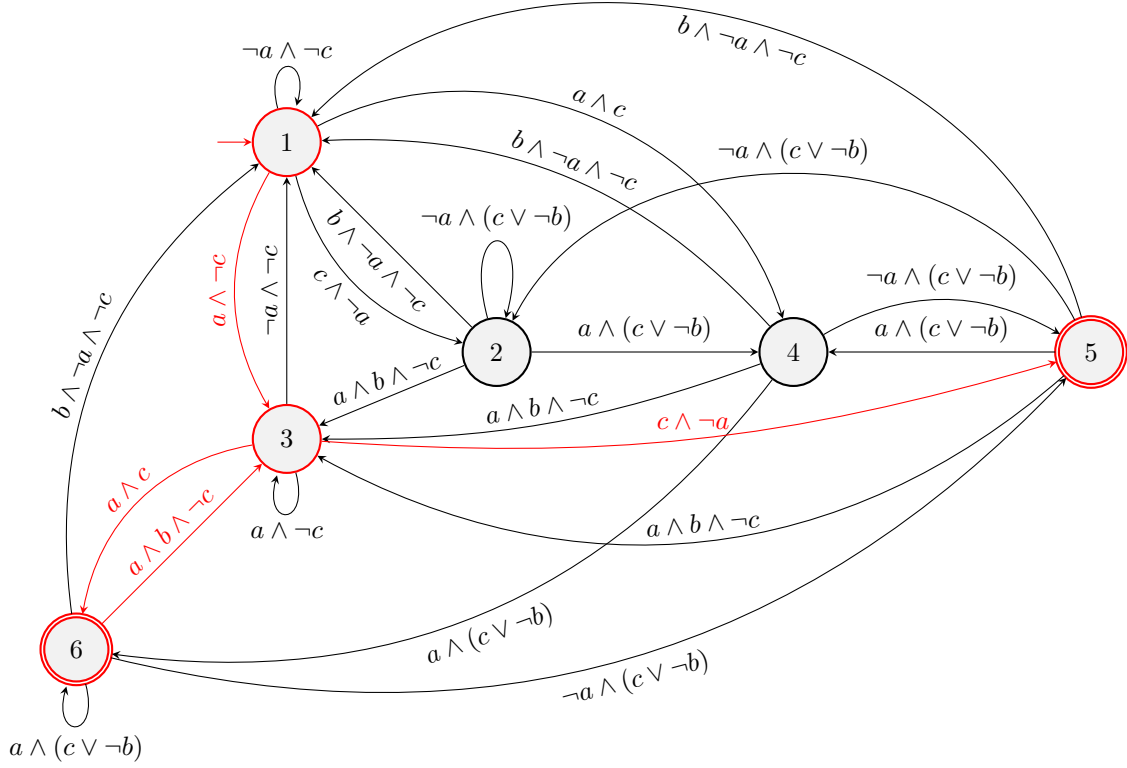
Even a small formula like the one under examination may have several transitions. Here, observe that before compactification, there are as many transitions as models of Boolean formulas that label transitions. As in the previous case, also here, the red colored part of the DFA corresponds to the augmented trace when projected on τ. To evaluate PPLTL formulas, we do not need to build the whole automaton a priori, using our technique suffices.

Interestingly, we can do more than just relate our evaluation technique with runs on DFAs. Although the key set of subformulas in Σ_φ we keep track of does not suffice to fully determine the automaton state, for small automata we can also characterize which subformulas are true in the automaton state. In particular, in cases like the one of Example 4.5, we can show how our technique can be used to build the DFA if one considers every possible run or, equivalently, every possible propositional interpretation from every state of the automaton.

Example 4.9. *Given the PPLTL formula $\mathsf{Y}(a)$, its subformulas are $\mathsf{sub}(\varphi) = \{a, \mathsf{Y}(a)\}$, and $\Sigma_\varphi = \{\text{“}\mathsf{Y}(a)\text{”}\}$. The set of all possible propositional interpretations to be considered are $\{\}$ and $\{a\}$. We can build the DFA for $\mathsf{Y}(a)$ by evaluating from every state and for each propositional interpretation read which subformulas are true in the successor state. The resulting DFA is as follows.*
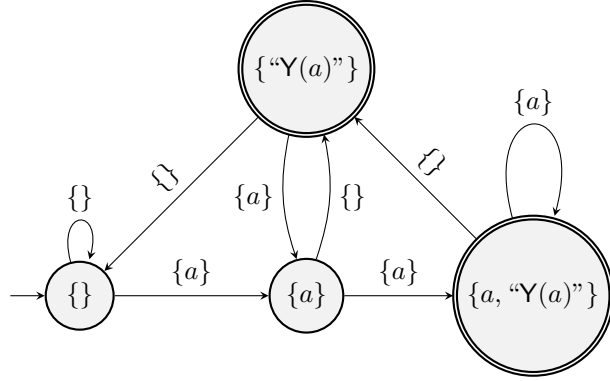
Finally, observe that while for this simple case, the resulting DFA *corresponds to the minimal* DFA *of* $\mathsf{Y}(a)$, *this is not always the case for any formula.*

4.5   Summary and Discussion

In this chapter, we have reviewed the fixpoint characterization of PPLTL formulas and shown how to exploit it to derive a formally correct technique able to evaluate PPLTL formulas only relying on the truth value of some key subformulas. Therefore, we introduced a novel approach to handle PPLTL formulas that exploits the native backward feature of PPLTL when interpreting traces. In the next two chapters, we are going to exploit the mathematical construction presented here to devise a particularly effective technique for planning in deterministic and nondeterministic domains for temporally extended goals expressed in PPLTL.

Chapter 5

Classical Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic

In this chapter, we study classical planning with temporally extended goals expressed in Pure-Past Linear Temporal Logic (PPLTL). As we have discussed before, PPLTL is as expressive as Linear-time Temporal Logic on finite traces (LTL$_f$), but as shown in this dissertation, it is computationally much better behaved for planning. Specifically, we show that planning for PPLTL goals can be encoded into classical planning with minimal overhead, introducing only a number of new fluents that is at most linear in the PPLTL goal and no spurious additional actions. Based on these results, we implemented a system called Plan4Past, which can be used along with state-of-the-art classical planners, such as LAMA (Richter and Westphal, 2010). An empirical analysis demonstrates the practical effectiveness of Plan4Past, showing that a classical planner generally performs better with our compilation than with other existing compilations for LTL$_f$ goals over the considered benchmarks.

The content of this chapter has been published in (Bonassi et al., 2023b) at the International Conference of Automated Planning and Scheduling 2023.

5.1   Context and Motivation

Planning with temporally extended goals has a long tradition in AI Planning, including pioneering work in the late '90s (Bacchus et al., 1996; Bacchus and Kabanza, 1996; Bacchus et al., 1997; Bacchus and

Kabanza, 2000), work on planning via Model Checking (Cimatti et al., 1997, 1998; De Giacomo and Vardi, 1999; Giunchiglia and Traverso, 1999; Pistore and Traverso, 2001), and work on declarative and procedural constraints (Baier and McIlraith, 2006a; Baier et al., 2008). Also, the presence of trajectory constraints in PDDL3 (Gerevini et al., 2009) witnesses the relevance of temporally extended goals.

In fact, formalisms such as LTL have been advocated by the Formal Methods community (Baier and Katoen, 2008) as excellent tools to express properties of processes. While these properties of processes often have an infinite horizon, in AI Planning, tasks need to terminate. Thus, the finite-trace variant of LTL, namely LTL$_f$, is usually more appropriate (Baier and McIlraith, 2006a; De Giacomo and Vardi, 2013).

Planning for LTL$_f$ goals in deterministic domains requires some properties to be achieved along the execution of a plan and has already been studied in, e.g., (Baier and McIlraith, 2006a; De Giacomo and Vardi, 2013; Torres and Baier, 2015). From a theoretical complexity standpoint, we have a clear picture. Deterministic planning for LTL$_f$ goals is PSPACE-complete, just like for classical reachability goals (Bylander, 1994). In fact, in deterministic domains, the added expressiveness of LTL$_f$ goals is paid in terms of algorithmic sophistication but not in worst-case complexity. However, existing encodings of LTL$_f$ to classical planning are either worst-case exponential (Baier and McIlraith, 2006a) or significantly increase the plan length (Torres and Baier, 2015).

As we have seen in the previous chapter, an interesting alternative to LTL$_f$ is PPLTL, which has been attractive in expressing temporal specifications in other areas of AI. Given an initial state, planning for PPLTL goals requires reaching, from a specified initial state, a certain state satisfying the PPLTL goal, i.e., the state-trace produced to reach such a state satisfies the goal formula. In this chapter, we present an approach to solve classical planning for PPLTL goals that sidesteps altogether the construction of the DFA for the PPLTL formula as done, e.g., in (Baier and McIlraith, 2006a; Torres and Baier, 2015) for LTL$_f$. Our novel approach shows that classical planning for PPLTL goals can be encoded into planning for reachability goals with *minimal overhead* by only introducing few new fluents, at most linear in the size of the PPLTL goal, and without adding any spurious action. These new fluents keep track of the satisfaction of *few* key subformulas of the temporal goal at planning time, reducing planning for temporally extended goals to classical planning for reachability goals. In this way, planners lazily build the relevant part of the DFA for the goal formula on the fly during the planning search. Our solution exploits the fixpoint characterization (Gabbay et al., 1980; Manna, 1982; Emerson, 1990) of PPLTL formulas that, similarly to the one of LTL (Barringer et al., 1989; Bacchus and Kabanza, 1996), recursively splits the formula into a propositional formula on the current instant and a temporal formula on the past to be checked at the *previous* instant. The solution to this recursion can be obtained by storing previous values of a small number of formulas (at most linear in the original formula), à la dynamic programming. Intuitively, we examine and take advantage of the key native

difference between LTL$_f$ and PPLTL. That is, given the prefix of the trace computed so far, while the LTL$_f$ formula has to consider all possible extensions, a PPLTL can simply be evaluated on the history (the prefix of the trace) produced so far. Therefore, the use of PPLTL is crucial to obtain such nice results since it avoids any form of guessing about the future.

We describe an implementation of our novel compilation approach, called Plan4Past, which can be used along with state-of-the-art classical planners to solve the task. Finally, we report an experimental analysis showing the practical effectiveness of Plan4Past by comparing it against existing techniques for LTL$_f$ goals.

5.2 Classical Planning for Pure-Past Linear Temporal Logic Goals

5.2.1 Encoding to Classical Planning

We devise a particularly effective technique for classical planning for PPLTL goals by exploiting the result in Theorem 4.4 in Chapter 4. The key idea of the approach is to keep track of the values of the subformulas of the PPLTL goal and update such values when actions are selected during the planning process by exploiting the PNF. This way, we sidestep altogether the standard construction based on computing the automaton for the PPLTL goal φ and then building the cross-product between such an automaton and the automaton corresponding to the domain, see (De Giacomo and Rubin, 2018) and cf. Section 2.5.3.

Similarly to other compilation-based approaches dealing with temporally extended goals, e.g., (Baier and McIlraith, 2006a; Torres and Baier, 2015), we address planning for temporally extended goals in three steps. First, encode the original planning problem Γ with the temporally extended goal into a planning problem Γ' with a reachability goal. Second, invoke any off-the-shelf sound and complete planner to compute a plan solving the encoded problem Γ'. Third, rework the computed plan to get the solution to the original problem Γ. In our approach, we exploit Theorem 4.4 to do the compilation in the first step, and since no extra control actions are introduced, step three trivializes.

Given a planning problem $\Gamma = \langle \mathcal{D}, s_0, \varphi \rangle$, where $\mathcal{D} = \langle \mathcal{F}, \mathcal{F}_{der}, \mathcal{X}, A, pre, eff \rangle$ is a deterministic domain, s_0 the initial state and φ a PPLTL goal, the encoded planning problem is $\Gamma' = \langle \mathcal{D}', s_0', G' \rangle$, where $\mathcal{D}' = \langle \mathcal{F}', \mathcal{F}'_{der}, \mathcal{X}', A, pre, eff' \rangle$ is the encoded planning domain, s_0' is the new initial state and G' is the new reachability goal. The domain components are modified as follows.

Fluents. $\mathcal{F}'$ contains the fluents of $\mathcal{F}$, as well as one fluent for each proposition "$\mathsf{Y}\phi$" in Σ_φ to keep track of propositional interpretations σ_i. Formally, $\mathcal{F}' := \mathcal{F} \cup \Sigma_\varphi$.

Derived Predicates. $\mathcal{F}'_{der}$ contains the derived predicates of the original domain model $\mathcal{F}_{der}$, and a predicate val_ϕ for every subformula $\phi \in \mathsf{sub}(\varphi)$. Formally, $\mathcal{F}'_{der} := \mathcal{F}_{der} \cup \{\mathsf{val}_\phi \mid \phi \in \mathsf{sub}(\varphi)\}$.

Axioms. We employ *axioms* (Hoffmann and Edelkamp, 2005), which have the form $d \leftarrow \psi$, where $d \in \mathcal{F}'_{der}$ is a positive literal called *derived predicate*, and ψ is a propositional formula over a set of predicates. Let s be a state, axiom $x = d \leftarrow \psi$ determines that the derived predicate d holds true in s if and only if $s \models \psi$.

We include an axiom $x_\phi = \mathsf{val}_\phi \leftarrow \psi$ for every subformula $\phi \in \mathsf{sub}(\varphi)$. These axioms are intended to be such that the current state $(\sigma_i, s_i) \models \mathsf{val}_\phi$ if and only if $\mathsf{val}(\phi, \sigma_i, s_i)$ – without loss of generality, in this section, we assume that σ and s represent sets of positive literals, and we use (σ_i, s_i) to denote the state $\sigma_i \cup s_i$. By mimicking rules in Definition 4.2, we get the following axioms:

- $\mathsf{val}_p \leftarrow p$;

- $\mathsf{val}_{\mathsf{Y}\phi} \leftarrow \text{“}\mathsf{Y}\phi\text{”}$;

- $\mathsf{val}_{\phi_1 \mathsf{S} \phi_2} \leftarrow (\mathsf{val}_{\phi_2} \vee (\mathsf{val}_{\phi_1} \wedge \text{“}\mathsf{Y}(\phi_1 \mathsf{S} \phi_2)\text{”}))$;

- $\mathsf{val}_{\phi_1 \wedge \phi_2} \leftarrow (\mathsf{val}_{\phi_1} \wedge \mathsf{val}_{\phi_2})$;

- $\mathsf{val}_{\neg\phi} \leftarrow \neg\mathsf{val}_\phi$.

It is easy to see that indeed we have that $(\sigma_i, s_i) \models \mathsf{val}_\phi$ if and only if $\mathsf{val}(\phi, \sigma_i, s_i)$. Hence, to define Γ', we build a set of axioms for every subformula ϕ in $\mathsf{sub}(\varphi)$, i.e., $\mathcal{X}' := \mathcal{X} \cup \{x_\phi \mid \phi \in \mathsf{sub}(\varphi)\}$. Axioms allow us to elegantly model the mathematics of Chapter 4 (i.e., the $\mathsf{val}(\phi, \sigma_i, s_i)$) and are often convenient when dealing with more sophisticated forms of planning (see, e.g., Borgwardt et al. (2022)). They also simplify the action schema and goal descriptions without adding control predicates among fluents, thus simplifying the search, as shown in Thiébaux et al. (2005).

Actions. Action labels A and precondition functions *pre* remain unchanged. In fact, every domain's action $a \in A$ is only modified on its effects $\mathit{eff}(a)$ by adding a way to update the assignments of propositions in Σ_φ. For each "$\mathsf{Y}\phi$" $\in \Sigma_\varphi$, we model assignments updates by a set of conditional effects of the form:

$$\mathsf{val}_\phi \triangleright \quad \text{“}\mathsf{Y}\phi\text{”}$$

$$\neg\mathsf{val}_\phi \triangleright \neg\text{“}\mathsf{Y}\phi\text{”}$$

These additional effects are exactly the same for every action $a \in A$. Formally, let $\mathsf{eff}_{\mathsf{val}} = \{\mathsf{val}_\phi \triangleright \{\text{“}\mathsf{Y}\phi\text{”}\}, \neg\mathsf{val}_\phi \triangleright \{\neg\text{“}\mathsf{Y}\phi\text{”}\} \mid \text{“}\mathsf{Y}\phi\text{”} \in \Sigma_\varphi\}$ be the additional effects, then $\mathit{eff}'(a) := \mathit{eff}(a) \cup \mathsf{eff}_{\mathsf{val}}$.

It is easy to see that since σ_i maintains values of "$\mathsf{Y}\phi$" in Σ_φ, action effects are *independent* of the effect of the action on the original fluents, which, instead, is maintained in the propositional interpretation s_i. This means that we can compute the next value of σ without knowing either which action has been executed or which effect such action has had on the original fluents. Note that the auxiliary part $\mathsf{eff}_{\mathsf{val}}$ in eff'_a *deterministically* updates subformulas values in Σ_φ, without affecting any fluent $f \in \mathcal{F}$ of the original domain model. This is crucial for the encoding's correctness.

Initial State. The initial state is the same as the original problem Γ for the original fluents in $\mathcal{F}$, whereas the new fluents "$\mathsf{Y}\phi$" $\in \Sigma_\varphi$ are assigned to the truth value given by σ_0. That is $s'_0 = \sigma_0 \cup s_0$ – without loss of generality, here we assume that σ_0 and s_0 represent sets of positive literals.

Goal. The goal in Γ' is encoded as $G' = \mathsf{val}_\varphi$. That is $\mathsf{val}(\varphi, \sigma_n, s_n)$, associated with the original PPLTL goal formula φ, has to hold true in the last instant, as per Theorem 4.4.

It is easy to see that our encoding is polynomially related to the original problem.

Theorem 5.1. *Let Γ be a classical planning problem. The size of Γ' obtained following the encoding above is polynomial in the size of Γ. In particular, it is linear in the size of the domain specification and linear in the size of the goal.*

Proof. By construction of the encoding. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

Example 5.2. *Assume we have a PPLTL goal $\varphi = b \wedge (\neg a \mathbin{\mathsf{S}} c)$. In the following, we describe in detail the encoding of the formula φ. First, we compute the set of subformulas of φ as $\mathsf{sub}(\varphi) = \{a, b, c, \neg a, (\neg a \mathbin{\mathsf{S}} c), b \wedge (\neg a \mathbin{\mathsf{S}} c)\}$. Given a planning problem $\Gamma = \langle \mathcal{D}, s_0, \varphi \rangle$, where $\mathcal{D} = \langle \mathcal{F}, \mathcal{F}_{der}, \mathcal{X}, A, pre, eff \rangle$, the compact representation of Γ' is defined as follows.*

Given that the $\mathsf{pnf}(\varphi) = b \wedge (c \vee (\neg a \wedge \mathsf{Y}(\neg a \mathbin{\mathsf{S}} c)))$, the only key subformula we need to keep track of is $\mathsf{Y}(\neg a \mathbin{\mathsf{S}} c)$, which we maintain in Σ_φ as "$\mathsf{Y}(\neg a \mathbin{\mathsf{S}} c)$". Therefore, the new set of fluents $\mathcal{F}'$ is simply the union of the original set of fluents $\mathcal{F}$ and the new additional fluent $\{\text{"}\mathsf{Y}(\neg a \mathbin{\mathsf{S}} c)\text{"}\}$.

At this point, based on the set of subformulas $\mathsf{sub}(\varphi)$, we compute the set of additional derived predicates $\{\mathsf{val}_a, \mathsf{val}_b, \mathsf{val}_c, \mathsf{val}_{\neg a}, \mathsf{val}_{\neg a \mathbin{\mathsf{S}} c}, \mathsf{val}_{b \wedge (\neg a \mathbin{\mathsf{S}} c)}\}$. This set of additional derived predicates is added to the original set of derived predicates $\mathcal{F}_{der}$. Formally, $\mathcal{F}'_{der} = \mathcal{F}_{der} \cup \{\mathsf{val}_a, \mathsf{val}_b, \mathsf{val}_c, \mathsf{val}_{\neg a}, \mathsf{val}_{\neg a \mathbin{\mathsf{S}} c}, \mathsf{val}_{b \wedge (\neg a \mathbin{\mathsf{S}} c)}\}$.

Then, following the rules illustrated in Definition 4.2, we get the following new axioms:

- $\mathsf{val}_a \leftarrow a$;

- $\mathsf{val}_b \leftarrow b$;

- $\text{val}_c \leftarrow c;$

- $\text{val}_{\neg a} \leftarrow \neg \text{val}_a.$

- $\text{val}_{\neg a\, \mathsf{S}\, c} \leftarrow (\text{val}_c \vee (\text{val}_{\neg a} \wedge \text{``}\mathsf{Y}(\neg a\, \mathsf{S}\, c)\text{''}));$

- $\text{val}_{b \wedge (\neg a\, \mathsf{S}\, c)} \leftarrow (\text{val}_b \wedge \text{val}_{\neg a\, \mathsf{S}\, c}).$

which will be added to the original set of axioms $\mathcal{X}$ to form $\mathcal{X}'$.

For each action $a \in A$, while preconditions remain unchanged, action effects $\text{eff}'(a)$ are extended as follows: $\text{eff}'(a) = \text{eff}(a) \cup \text{eff}_{\text{val}}$, where $\text{eff}_{\text{val}} = \{\text{val}_{\neg a\, \mathsf{S}\, c} \rhd \text{``}\mathsf{Y}(\neg a\, \mathsf{S}\, c)\text{''}, \neg\text{val}_{\neg a\, \mathsf{S}\, c} \rhd \neg\text{``}\mathsf{Y}(\neg a\, \mathsf{S}\, c)\text{''}\}$.

After that, the new initial state is $s_0' = \sigma_0 \cup s_0$, where $\sigma_0(\text{``}\mathsf{Y}(\neg a\, \mathsf{S}\, c)\text{''}) \doteq \bot$. Finally, the new reachability goal is $G' = \text{val}_{b \wedge (\neg a\, \mathsf{S}\, c)}$.

5.2.2 Correctness

Let $\Gamma = \langle \mathcal{D}, s_0, \varphi \rangle$ be a classical planning problem, where $\mathcal{D}$ is a deterministic domain, s_0 is the initial state, and φ is a PPLTL goal formula, and let $\Gamma' = \langle \mathcal{D}', s_0', G' \rangle$ be the corresponding encoded planning problem as previously defined.

Any trace $\tau' = s_0', \ldots, s_n'$ on $\mathcal{D}'$ can be seen as $\tau' = zip(\tau^{[\varphi]}, \tau)$, where $\tau = s_0, \ldots, s_n \in (2^{\mathcal{F}})^+$, $\tau^{[\varphi]} = \sigma_0, \ldots, \sigma_n \in (2^{\Sigma_\varphi})^+$, where each element of τ' is of the form $s_i' = (\sigma_i, s_i)$ for all $i \geq 0$. Here, we use the $zip(\cdot, \cdot)$ function[1] to represent the aggregation of the two traces $\tau^{[\varphi]}$ and τ. Given a trace $\tau' = s_0', \ldots, s_n'$ on the encoded planning domain $\mathcal{D}'$, there exists a single trace $\tau' \mid_{\mathcal{F}} = \tau = s_0, \ldots, s_n$ on the original planning domain $\mathcal{D}$. Conversely, given a trace $\tau = s_0, \ldots, s_n$ on the original planning domain $\mathcal{D}$, there exists a unique corresponding trace $\tau^{[\varphi]}$, and hence a single $\tau' = zip(\tau^{[\varphi]}, \tau)$ on the encoded domain $\mathcal{D}'$.

Theorem 5.3. *Let φ be a PPLTL formula over $\mathcal{P}$, $\phi \in \mathsf{sub}(\varphi)$ a subformula of φ, τ a trace over $\mathcal{P}$, and $\tau^{[\varphi]}$ the corresponding trace over Σ_φ. Then,*

$$\text{val}(\phi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau)) \;\; \textit{iff} \;\; \mathsf{last}(\tau') \models \text{val}_\phi$$

Proof. Since $\sigma_n = \mathsf{last}(\tau^{[\varphi]})$ and $s_n = \mathsf{last}(\tau)$, we can rewrite the thesis as $\text{val}(\phi, \sigma_n, s_n)$ iff $\mathsf{last}(\tau') \models \text{val}_\phi$. We prove the thesis by double induction on the length of the trace τ' and on the structure of the formula.

---

[1] The $zip(\cdot, \cdot)$'s name has been inspired by the common functional programming method zip, which takes two input sequences, and produce an output sequence in which every two elements from input sequences are combined at the same position.

- Base case: $\tau' = (\sigma_0, s_0)$. Immediate by structural induction on the formula as axiom rules follow the val($\cdot$) predicate (cf. Definition 4.2).

- Inductive step: $\tau' = \tau''_{n-1} \cdot (\sigma_n, s_n)$. To prove the claim we proceed by structural induction on the formula.

  - $\phi = p$. By definition of val($\cdot$), we have val(p, σ_n, s_n) iff $s_n \models p$. By construction of axioms, we have that p iff val_p. Thus, the thesis holds.

  - $\phi = \mathsf{Y}\phi'$. By definition of val($\cdot$), we have val($\mathsf{Y}\phi', \sigma_n, s_n$) iff $\sigma_n \models$ "$\mathsf{Y}\phi'$". By definition and construction of axioms, we have that "$\mathsf{Y}\phi'$" iff $\mathsf{val}_{\mathsf{Y}\phi'}$. Thus, the thesis holds.

  - $\phi = \phi_1 \mathsf{S} \phi_2$. By definition of val($\cdot$), we have val($\phi_1 \mathsf{S} \phi_2, \sigma_n, s_n$) iff val($\phi_2, \sigma_n, s_n$)$\vee$(val($\phi_1, \sigma_n, s_n$)$\wedge$ $\sigma_n \models$ "$\mathsf{Y}(\phi_1 \mathsf{S} \phi_2)$"). By structural induction, val_{ϕ_2} iff val(ϕ_2, σ_n, s_n) and val_{ϕ_1} iff val(ϕ_1, σ_n, s_n). Therefore, by definition and construction of axioms, we have that $s_n \models \phi_1 \mathsf{S} \phi_2$ iff $\mathsf{val}_{\phi_1 \mathsf{S} \phi_2}$. Thus, the thesis holds.

  - $\phi = \phi_1 \wedge \phi_2$ or $\phi = \neg \phi'$. The thesis holds by structural induction.

$\square$

We start by observing that every executable action sequence $a_0, \ldots, a_{n-1}$ in the deterministic planning problem Γ for the PPLTL goal φ is also executable in the compiled planning problem Γ' (and vice versa) since, by definition, the encoding does not have auxiliary actions, actions preconditions do not change, and additional conditional effects on the original actions are deterministic.

Theorem 5.4 (Correctness). *Let Γ be a planning problem with a PPLTL goal φ, and Γ' be the corresponding compiled planning problem with a reachability goal. Then, every action sequence $\pi = a_0, \ldots, a_{n-1}$ is a plan for Γ iff $\pi = a_0, \ldots, a_{n-1}$ is a plan for Γ'.*

Proof. The action sequence π is a plan if its induced state trace τ is such that $\tau \models \varphi$. By Theorem 4.4, we have that $\tau \models \varphi$ if and only if val($\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau)$). However, by construction of the encoding for Γ', we have that val($\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau)$) holds if and only if val_φ holds in the last state of the induced state trace for Γ', i.e., in $\tau' = zip(\tau^{[\varphi]}, \tau)$. In other words, by Theorem 5.3, we have that val($\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau)$) holds if and only if $\mathsf{last}(\tau') \models \mathsf{val}_\varphi$, as this holds when the encoded problem for reachability goal val_φ holds – note that only the last element of the new trace τ' has to satisfy the goal condition. Hence, the thesis holds. $\square$

A direct consequence of Theorem 5.4 is that every sound and complete classical planner returns a plan π for the encoded planning problem Γ' if a plan π for the original planning problem Γ for PPLTL goal exists. If no solution exists for Γ', then no solution exists for Γ.

5.3 Experimental Evaluation

We implemented the approach of Section 5.2.1 in a tool called Plan4Past[2] (P4P$_\mathcal{X}$ for short), written in Python (approximately 1000 LOC) and currently released under the GNU LGPL-3.0 license. P4P$_\mathcal{X}$ takes as input a PDDL description and a PPLTL formula and gives as output a new PDDL description, which can be processed by any classical planner supporting axioms and conditional effects. We also tested an alternative version of P4P$_\mathcal{X}$ where all axioms are compiled into conditional effects, i.e., two for each sub-formula ϕ encoding the truth value of ϕ after each action. However, the resulting compilation proved much less effective than that using axioms, so we do not consider it in our analysis.

The empirical analysis examines the effectiveness of temporally extended goals formulated in PPLTL and handled by P4P$_\mathcal{X}$ compared to *semantically* equivalent temporally extended goals formulated in LTL$_f$ and handled by either LTLExp (Baier and McIlraith, 2006a) or LTLPoly (Torres and Baier, 2015).

To our knowledge, LTLExp and LTLPoly are the best approaches for planning with LTL$_f$ goals. In particular, Baier and McIlraith (2006a) build an NFA for the LTL$_f$ formula and compute the Cartesian product with the planning domain (cf. (De Giacomo and Rubin, 2018)), incurring in a worst-case exponential increase in the number of states of the NFA. Similarly, Torres and Baier (2015) implicitly construct the NFA for the goal formula. While the approach in (Torres and Baier, 2015) is optimal with respect to the computational complexity, it significantly increases the plan length. Indeed, working with NFAs during planning requires choosing not only the next actions in the plan but also the right nondeterministic transition of the NFA. This translates into extra dummy (synchronization) actions to insert into the plan, making the overall search harder for the planner. Dealing with spurious actions has already been studied in (Nebel, 2000) theoretically and practically from a heuristic perspective in, e.g., (Haslum, 2013). Therefore, from a theoretical standpoint, the advantage of P4P$_\mathcal{X}$ is clear: under an automata-theoretic view, P4P$_\mathcal{X}$ exploits the fact that goals are expressed in PPLTL to implicitly and incrementally build a DFA (vs. an NFA) for the temporal formula while doing planning, keeping optimality with respect to computational complexity and preserving the plan length.

Next, we want to determine whether this theoretical advantage manifests itself in actual planning performance from a practical perspective. To this end, we tested the three considered systems over a set of benchmarks and analyzed the number of problems solved (Coverage), the time spent to find a solution (compilation plus search time), the number of expanded nodes, and the plan length. As a classical planner, we considered LAMA (Richter and Westphal, 2010), a planner built on top of FastDownward (Helmert, 2006), and FF$_\mathcal{X}$ (Thiébaux et al., 2005). LAMA is a satisfying planner based on a sophisticated search mechanism

---

[2]Available online at `https://github.com/whitemech/Plan4Past`.

that runs (in the first iteration) Lazy Greedy Best-First Search driven by the h_{ff} (Hoffmann and Nebel, 2001) and the landmarks counting heuristics. LAMA yields solution plans of decreasing plan cost incrementally; for our analysis, we take the first generated plan. FF$_\mathcal{X}$ is yet another satisficing planner based on heuristic search and enforced hill climbing, and is the one originally used with LTLPoly and LTLExp. Both systems handle the compiled problems but the remaining of this section will focus attention on LAMA as it was the system obtaining the highest overall coverage for all compilations. All experiments were run on an Intel Xeon Gold 6140M 2.3 GHz running Linux CentOS 7, with runtime and memory limits of 1800s and 8GB, respectively.

5.3.1 Benchmark Domains

Our benchmark suite includes four domains: BLOCKSWORLD, ELEVATOR, ROVERS, and OPENSTACKS. These domains were introduced in past International Planning Competitions[3] (IPC), and all except ELE-VATOR have also been used by Torres and Baier (2015). For BLOCKSWORLD, ROVERS, and OPENSTACKS, we have a set of instances with the same temporally extended goals defined by Torres and Baier (2015) (hereinafter referred to as TB15) and a second set of instances with temporally extended goals defined by us (hereinafter referred to as BF22). For ELEVATOR, we only have BF22 goals. TB15 goals were originally specified in LTL$_f$, and for P4P$_\mathcal{X}$ we manually translated them to PPLTL. We did so for all but one type of temporally extended goal used in Torres and Baier (2015), namely that of type "$h : \alpha \mathbin{\mathsf{U}} \beta$" where α or β have n nested U operators, for which we did not find an easy translation into PPLTL. Recall that, as mentioned in Chapter 3, LTL$_f$ and PPLTL have the same expressiveness, but the best-known systematic procedure to translate one into the other is 3EXPTIME (in both directions). However, for each LTL$_f$ formula that we translated in PPLTL, we formally and automatically proved their *semantic* equivalence by verifying that the two formulations yield the same minimal DFA. BF22 goals were instead designed in PPLTL directly, and analogously to what was done for TB15, we formulated an equivalent formulation in LTL$_f$. TB15 goals are based on predefined families of formulas that are independent of the domain. Instead, BF22 goals are specific for each domain and were designed to stress all compilations and understand the planner's scalability over non-trivial and large instances. Indeed, all instances with TB15 proved trivial for Plan4Past. For TB15, we have 15 instances for BLOCKSWORLD, 7 for ROVERS, and 10 for OPENSTACKS. Their definition is provided by Torres and Baier (2015). BF22 are instead described below.

BlocksWorld. BF22 goals were formulated to study the reach of all compilations with complex temporally extended goals. Here, BF22 goals specify two intertwined goals, both requiring the existence in the state

---

[3]https://www.icaps-conference.org/competitions/

trajectory of the plan of a particular sequence of states. Consider a problem with n blocks, and let $on_{i,j}$ be the predicate modeling block i being on block j. The first goal in PPLTL is:

$$\mathsf{O}(on_{1,2} \wedge \mathsf{Y}(\mathsf{O}(on_{2,3} \wedge \mathsf{Y}(\mathsf{O}(\ldots \wedge \mathsf{Y}(\mathsf{O}(on_{n-1,n})))))))).$$

Its translation in LTL$_f$ is:

$$\mathsf{F}(on_{n-1,n} \wedge \mathsf{X}(\mathsf{F}(on_{n-2,n-1} \wedge \mathsf{X}(\mathsf{F}(\ldots \wedge \mathsf{X}(\mathsf{F}(on_{1,2})))))))).$$

The second goal (for an even number of blocks) in PPLTL is:

$$\bigwedge_{j \in \{6,8,\ldots,n\}} \mathsf{O}(on_{j,j-1} \wedge \mathsf{Y}(\phi)),$$

where ϕ is the formula $\mathsf{O}(on_{4,3} \wedge \mathsf{Y}(\mathsf{O}(on_{3,2} \wedge \mathsf{Y}(\mathsf{O}(on_{2,1})))))$, which encodes the construction of a bigger stack. The same constraint formulated in LTL$_f$ is:

$$\mathsf{F}(on_{2,1} \wedge \mathsf{X}(\mathsf{F}(on_{3,2} \wedge \mathsf{X}(\mathsf{F}(on_{4,3} \wedge \bigwedge_j \mathsf{X}(\mathsf{F}(on_{j,j-1})))))))).$$

The formulation for an odd number of blocks is analogous. We generate a temporally extended goal for each instance of the domain, starting from that with 10 up to 30 blocks.

OpenStacks. In this domain, BF22 goals require that a valid plan ships all specified requests following a specific production order. The PPLTL formula $\mathsf{H}(made_{p_3} \rightarrow \mathsf{Y}(\mathsf{O}(made_{p_2}))) \wedge \mathsf{H}(made_{p_2} \rightarrow \mathsf{Y}(\mathsf{O}(made_{p_1})))$ encodes that p_1 is made strictly before p_2, that in turn must be made before p_3. The equivalent LTL$_f$ formula is $(made_{p_2}) \mathsf{R} (\neg made_{p_3}) \wedge (made_{p_1}) \mathsf{R} (\neg made_{p_2})$. Every order must be shipped, and this is encoded with $\mathsf{O}(shipped_{order})$ in PPLTL, and with $\mathsf{F}(shipped_{order})$ in LTL$_f$.

Rovers. The goal of this domain is to gather and communicate data about soil, rock, and images to the Earth using a set of rovers. BF22 goals enforce a total order over the communications of the data. This temporally extended goal implicitly requires the data to be eventually communicated and is encoded in PPLTL as

$$\mathsf{O}(data_{soil} \wedge \mathsf{WY}(\mathsf{H}(\neg data_{rock}))) \wedge \mathsf{O}(data_{rock} \wedge \mathsf{WY}(\mathsf{H}(\neg data_{image}))) \wedge \mathsf{O}(data_{image}),$$

and in LTL$_f$ as

$$(\neg data_{rock}) \: \mathsf{U} \: (data_{soil}) \land (\neg data_{image}) \: \mathsf{U} \: (data_{rock}) \land \mathsf{F}(data_{image}).$$

Also, when the rover reaches the lander, that rover must re-calibrate all cameras (e.g., if the lander is at waypoint wl and the rover r has 2 cameras, $c1$ and $c2$, we have, in PPLTL, the formula

$$((\neg at_{r,wl} \: \mathsf{S} \: calibrated_{c1}) \land (\neg at_{r,wl} \: \mathsf{S} \: calibrated_{c2})) \lor \mathsf{H}(\neg at_{r,wl}),$$

and, in LTL$_f$, the formula

$$\mathsf{G}(at_{r,wl} \to (\mathsf{F}(calibrated_{c1}) \land \mathsf{F}(calibrated_{c2})))).$$

Elevator. This domain models the problem of scheduling passengers in the use of an elevator. In BF22, we split the passengers into half VIP passengers and half regular passengers, where VIP passengers must be served before every regular one. For instance, when there are four passengers p_0, p_1, p_2, p_3, with p_0, p_1 VIP and p_2, p_3 regular, we enforce this in PPLTL with:

$$\mathsf{O}(served_{p_2} \land served_{p_3}) \land \mathsf{O}(served_{p_0} \land served_{p_1} \land \mathsf{WY}(\mathsf{H}(\neg served_{p_2} \land \neg served_{p_3}))),$$

that can be expressed in LTL$_f$ with

$$\mathsf{F}(served_{p_2} \land served_{p_3}) \land (\neg served_{p_2} \land \neg served_{p_3}) \: \mathsf{U} \: (served_{p_0} \land served_{p_1}).$$

In the constraint, we also model that no passenger may share the elevator with another passenger and do so in PPLTL (resp. LTL$_f$) with $\mathsf{H}(boarded_{p_0} \to (\neg boarded_{p_1} \land \neg boarded_{p_2}))$ (resp. $\mathsf{G}(boarded_{p_0} \to (\neg boarded_{p_1} \land \neg boarded_{p_2})))$.

We include a total of 29 BF22 instances for the ELEVATOR domain.

5.3.2 Experimental Results

Table 5.1 reports on the overall performance of all compilations across all domains. Coverage-wise, P4P$_\mathcal{X}$ performs equally to or better than both LTLPoly and LTLExp over most instances. For the TB15 instances, P4P$_\mathcal{X}$ achieves the same coverage as LTLPoly (the best LTL$_f$-based compilation) but is much faster in terms of average run-time: P4P$_\mathcal{X}$ is roughly one order of magnitude faster than LTLPoly; this seems to be justified by a great reduction in the number of expanded nodes (up to two orders of magnitude in OPENSTACKS).

Domain	I	Coverage			Avg RT			Avg EN			Avg $|\pi|$		
		P4P$_\mathcal{X}$	LTLPoly	LTLExp	P4P$_\mathcal{X}$	LTLPoly	LTLExp	P4P$_\mathcal{X}$	LTLPoly	LTLExp	P4P$_\mathcal{X}$	LTLPoly	LTLExp
ROVERS	TB15 7	**7**	**7**	6	**1.43**	21.11	1.98	**12.67**	616.83	**12.67**	**5.33**	5.67 (74.50)	**5.33**
	BF22 40	**33**	6	22	35.36	–	**24.24**	**7665.36**	–	7826.32	43.68	–	**43.50**
BLOCKSWORLD	TB15 15	**15**	**15**	8	**1.41**	20.43	13.13	22.12	821.62	**21.75**	7.50	7.88 (132.88)	**7.25**
	BF22 21	**21**	1	1	–	–	–	–	–	–	–	–	–
OPENSTACKS	TB15 10	**10**	**10**	6	**6.11**	31.66	8.75	**52.67**	3863.33	52.83	22.00	**21.67** (349.00)	22.00
	BF22 30	7	5	**8**	**11.88**	68.86	19.50	99.80	1207764.80	**72.40**	**24.00**	**24.00** (841.00)	**24.00**
ELEVATOR	BF22 29	**29**	4	**29**	231.83	–	**228.09**	**1712076.48**	–	1712090.79	75.48	–	75.48
Total		**122**	48	80									

Table 5.1: Coverage, average Run-Time (Avg RT), Expanded Nodes (Avg EN) and Plan Length (Avg $|\pi|$) achieved by P4P$_\mathcal{X}$, LTLPoly and LTLExp. For LTLPoly, we report in parenthesis the average $|\pi|$ considering the actions added by the compilation. Averages are only among instances solved by those systems that obtain at least half of the coverage of the best performer. Column I is the number of instances in a domain. "–" indicates when a system is excluded by the comparison. In bold the best performers.

This is somehow expected. Indeed, for each planning action taken, LTLPoly needs to interleave quite a complex automaton synchronization phase, and this is done from the initial state all the way to the goal. On average, in TB15 instances, 94.3% of actions in plans obtained with LTLPoly come from the automaton's synchronization phases.

The situation is different if we look at the BF22 instances. Here, the best performing LTL$_f$ compilation is LTLExp, which is superior to LTLPoly over all instances. BF22 goals are of increasing dimensions and have been constructed to be computationally more challenging. For example, in BLOCKSWORLD, the hardest instance in TB15 requires a 22 actions plan, while BF22 instances require up to 652 actions. In the case of LTLPoly, the planner has to cope with too many synchronization phases and struggles to find solutions. In particular, in such a case, the only instance solved by LTLPoly of BLOCKSWORLD for BF22 goals has a plan of 6402 actions; 6298 of those are for the synchronization of the automaton. If we compare P4P$_\mathcal{X}$ and LTLExp, we observe that P4P$_\mathcal{X}$ is again the system performing generally better. The only exception is for one instance of OPENSTACKS. LTLExp solves this instance in roughly 739s while P4P$_\mathcal{X}$ times out. By looking at the average number of expanded nodes, LAMA's search turned out to be slightly less informed with P4P$_\mathcal{X}$ in this domain, which leads to timing out in that particular instance. For BLOCKSWORLD, P4P$_\mathcal{X}$ is instead much more effective than LTLExp; LTLExp only manages to compile 7 instances. Interestingly, the compilation time seems to be an issue for both LTL$_f$ compilations. Indeed, if we look at Figure 5.1 (right), P4P$_\mathcal{X}$ compiles 94.7% of the instances within 10s, whereas both LTLPoly and LTLExp converge much more slowly. Moreover, looking at Table 5.2, the time spent by the compilation of P4P$_\mathcal{X}$ is negligible compared to the entire planning process. In contrast, LTLPoly's compilation run-time is still two orders of magnitude larger.

Figure 5.1 (left) displays the number of benchmark instances solved with a given timeout. All systems achieve their maximum coverage quite quickly, with P4P$_\mathcal{X}$ leaving the others well behind right after the start.

Domain	P4P$_\mathcal{X}$	LTLPoly	LTLExp
BLOCKSWORLD	**0.42**	3.71	20.97
ELEVATOR	**0.50**	46.96	3.47
OPENSTACKS	**2.18**	115.32	22.68
ROVERS	**0.59**	15.15	4.93

Table 5.2: Average compilation time computed over instances compiled by all systems across all domains.
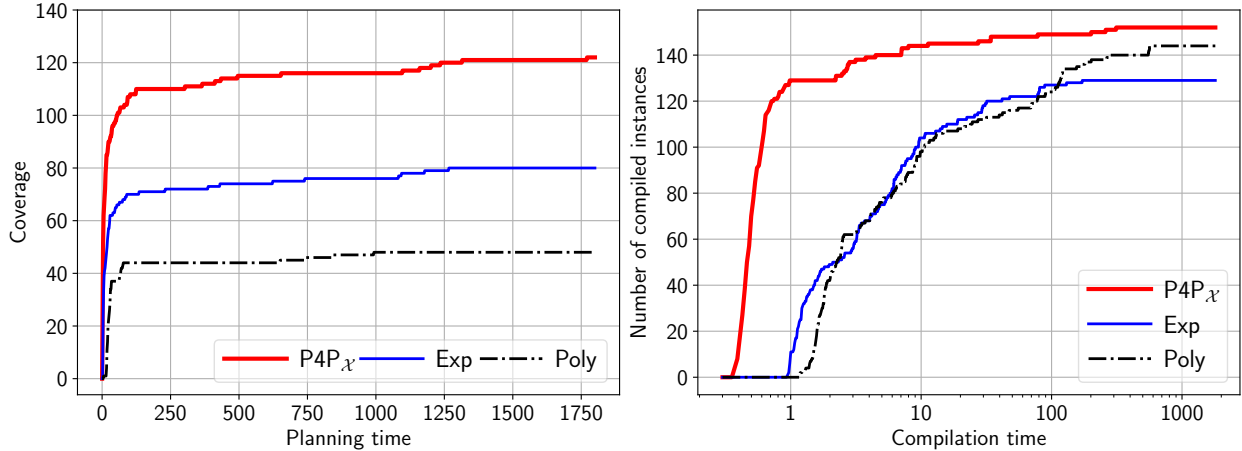
Figure 5.1: Number of solved instances (left) and compiled instances (right) versus computation time.

On the other hand, Figure 5.2 reports on a pairwise comparison P4P$_\mathcal{X}$ vs LTLExp and P4P$_\mathcal{X}$ vs LTLPoly over the number of expanded nodes and runtime, instance by instance. P4P$_\mathcal{X}$ is generally faster than LTLExp, apart from 15 instances. The number of expanded nodes between these two systems is surprisingly similar. Looking at our raw data, we observe that, for most of the instances, LTLExp spends much more time than P4P$_\mathcal{X}$ in compilation and slightly more time in evaluating a node of the search. The comparison P4P$_\mathcal{X}$ vs LTLPoly confirms our expectation on the number of expanded nodes. P4P$_\mathcal{X}$ expands nodes more slowly than LTLPoly, and therefore the runtime advantage of P4P$_\mathcal{X}$ is related to the fact that P4P$_\mathcal{X}$ leads LAMA to do much less search than LTLPoly.

Regarding plan quality, we observed that all compilations yield *solution plans* to the original problem (i.e., plans without spurious actions) of similar length, making their overall performance the same in these terms.

5.4   Summary and Discussion

In this chapter, we examined classical planning with PPLTL goals. PPLTL is a compelling formalism to express sophisticated planning goals and, compared to LTL$_f$-based approaches, allows for a polynomial-time compilation that is optimal with respect to the computational complexity and does not increase the plan

(a) LTLExp vs P4P$_\mathcal{X}$ (RT) (b) LTLPoly vs P4P$_\mathcal{X}$ (RT)

(c) LTLExp vs P4P$_\mathcal{X}$ (EN) (d) LTLPoly vs P4P$_\mathcal{X}$ (EN)
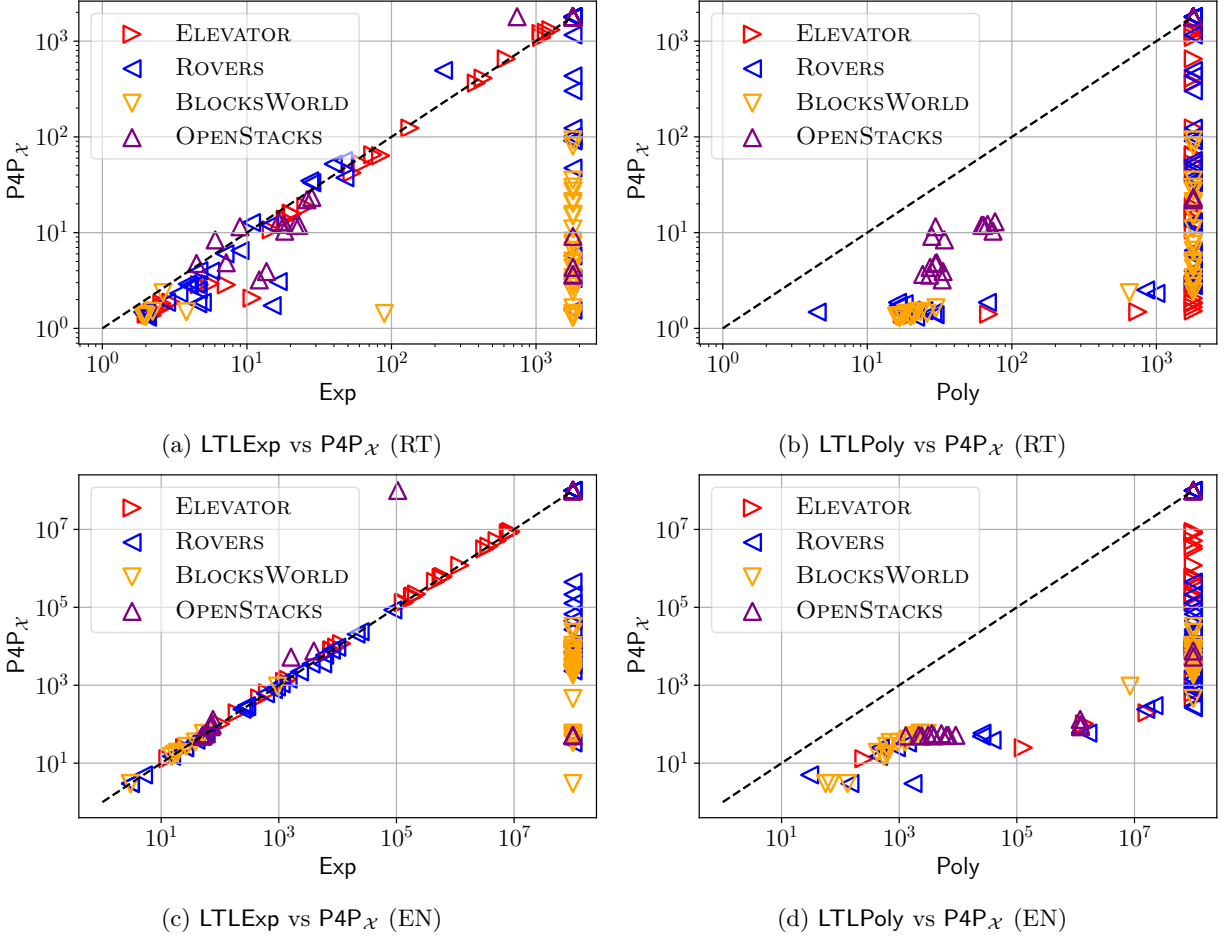
Figure 5.2: Pairwise comparison between P4P$_\mathcal{X}$ and LTLExp (left plots) and between P4P$_\mathcal{X}$ and LTLPoly (right plots) in terms of Run-Time (above) and Expanded Nodes (below).

length. These results suggest that PPLTL can be considered a "sweet spot" in expressing temporally extended goals for planning as it can specify planning goals while having, at the same time, a clear advantage in practice. Moreover, handling PPLTL goals is remarkably simple and elegant, given the direct mapping between the theoretical formulation (presented in Chapter 4) and the encoding to classical planning without sacrificing efficiency. We devised an encoding of planning with PPLTL goals to classical planning with reachability goals and demonstrated its practical effectiveness through extensive experiments. Here, we focused only on PPLTL. However, in principle, our approach can be extended to goals expressed in Pure-Past Linear Dynamic Logic (PPLDL) (De Giacomo et al., 2020a), a strictly more expressive variant of PPLTL involving regular expressions. Indeed, also PPLDL has a fixpoint characterization of the temporal operators. This extension remains open to future work. In the next chapter, we are going to see that although we focus on classical planning with PPLTL goals, the mathematical construction in Chapter 4 can also be applied to other forms of planning, including nondeterministic planning.

Chapter 6

FOND Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic

In this chapter, we study nondeterministic planning (FOND) for temporally extended goals expressed in Pure-Past Linear Temporal Logic (PPLTL). Although PPLTL is as expressive as Linear Temporal Logic on finite traces (LTL$_f$), FOND planning for PPLTL goals remains EXPTIME-complete, as for standard FOND planning, instead of becoming 2EXPTIME-complete as for LTL$_f$/LDL$_f$. Here, we present a notably simple encoding of FOND planning for PPLTL goals into standard FOND planning for reachability goals. As for classical planning examined in the previous chapter, the encoding only introduces a small number of new fluents (at most linear in the PPLTL goal) without adding any spurious action and allows planners to lazily build the relevant part of the DFA for the goal formula on-the-fly during the search.

We formally prove its correctness and implement this approach in the system Plan4Past, which can be used along with state-of-the-art FOND planners, such as PRP (Muise et al., 2012) or Paladinus (Fraga Pereira et al., 2022), and provide experimental evidence of its effectiveness.

The content of this chapter has been published in (Bonassi et al., 2023a) at the European Conference on Artificial Intelligence 2023.

6.1   Context and Motivation

Temporally extended goals are fundamental for the specification of a collection of real-world planning problems. Yet, many of these real-world planning problems have a *nondeterministic* behavior owing to unpredictable environmental conditions. Nondeterministic planning (FOND) for $\textsc{ltl}_f/\textsc{ldl}_f$ goals is a more challenging problem compared to classical planning and has been of increasing interest only in recent years with, e.g., (De Giacomo and Vardi, 2015; Camacho et al., 2017; De Giacomo and Rubin, 2018; Camacho and McIlraith, 2019). By now, the theoretical complexity of such a problem is well understood. FOND planning for $\textsc{ltl}_f$ goals is EXPTIME-complete in the domain as for classical reachability goals (Rintanen, 2004) and 2EXPTIME-complete in the goal formula (De Giacomo and Rubin, 2018). In nondeterministic domains, since $\textsc{ltl}_f$ goals can specify non-Markovian properties (Gabaldon, 2011), the added expressiveness of $\textsc{ltl}_f$ goals worsens the worst-case goal complexity to 2EXPTIME-complete, compared to the EXPTIME-complete of reachability goals. That is because, in FOND planning, it is required to (implicitly or explicitly) translate $\textsc{ltl}_f$ goal formulas into a *deterministic* automaton, which is double exponential in the worst case.

Along the same lines as the previous chapter, we study PPLTL as a valid alternative to $\textsc{ltl}_f$ for expressing temporally extended goals in nondeterministic planning. In fact, as mentioned in Section 3.7, employing PPLTL as a goal specification language has a theoretical advantage due to a property of reverse languages (Chandra et al., 1981) for which the DFA corresponding to a PPLTL formula can be computed in *single* exponential time directly from the formula (De Giacomo et al., 2020a). Obviously, the property of interest should be naturally expressible in PPLTL since translating $\textsc{ltl}_f$ into the PPLTL (and vice versa) is generally prohibitive given that the best-known algorithms are 3EXPTIME (De Giacomo et al., 2020a) (cf. Section 3.5.2).

As in classical planning, given an initial state, FOND planning for PPLTL goals requires reaching a certain state satisfying the PPLTL goal, i.e., the state-trace produced to reach such a state satisfies the goal formula. In this chapter, we present an approach to solve FOND planning for PPLTL goals that sidesteps altogether the construction of the DFA for the PPLTL formula as done, e.g., in (Camacho et al., 2017; De Giacomo and Rubin, 2018; Camacho and McIlraith, 2019) for $\textsc{ltl}_f$. Like for classical planning, the novel approach shows that nondeterministic planning for PPLTL goals can be encoded into planning for reachability goals with *minimal overhead* by only introducing few new fluents, at most linear in the size of the PPLTL goal, and without adding any spurious actions. These new fluents keep track of the satisfaction of *few* key subformulas of the temporal goal at planning time, reducing planning for temporally extended goals to standard FOND planning. In this way, planners lazily explore only the relevant part of the DFA for the goal formula on the fly during the planning search. Similar to the case of classical planning of Chapter 5, the solution for FOND

planning exploits the fixpoint characterization (Gabbay et al., 1980; Manna, 1982; Emerson, 1990) of PPLTL formulas and the mathematical construction seen in Chapter 4. Intuitively, we examine and take advantage of the key native difference between LTL$_f$ and PPLTL. That is, given the prefix of the trace computed so far, while the LTL$_f$ formula has to consider all possible extensions, a PPLTL can simply be evaluated on the history (the prefix of the trace) produced so far. Therefore, since PPLTL avoids any form of guessing about the future, it is essential to obtain such nice results.

To sum up, in this chapter, we make the following contributions. First, we devise an encoding of FOND planning for PPLTL goals into standard FOND planning for reachability goals, which is at most linear in the size of the formula, formally correct, and readily implementable in PDDL. Moreover, we devise an encoding variant without the use of derived predicates – not well supported by some FOND planners –, that is still polynomial in the size of the formula, correct, and readily implementable in PDDL. Both encodings have been implemented in the tool Plan4Past[1]. Finally, we empirically demonstrate the practical effectiveness of Plan4Past through an experimental analysis by using it along with state-of-the-art FOND planners and comparing it against existing techniques for FOND planning for LTL$_f$ goals.

6.2 FOND Planning for Pure-Past Linear Temporal Logic Goals

6.2.1 Encoding to FOND Planning

A great advantage of using PPLTL goals in expressing temporally extended goals is that the exact same encoding used in the classical planning setting works seamlessly for FOND planning too. In fact, so far, this has never been the case for LTL$_f$ goals since FOND planning requires working with DFAs to avoid any possible nondeterminism mismatch, whereas, in classical planning, the use of NFAs suffices. Here, instead, the encoding technique for FOND planning for PPLTL goals exploits the result in Theorem 4.4, where the idea is to keep track of the values of the subformulas of the PPLTL goal and update such values when actions are selected or added to the plan during the planning process by exploiting the PNF. This way, we sidestep altogether the standard construction based on computing the automaton for the PPLTL goal φ and then building the cross-product between such an automaton and the automaton corresponding to the domain, see (De Giacomo and Rubin, 2018) and cf. Section 2.5.3.

Given a planning problem $\Gamma = \langle \mathcal{D}, s_0, \varphi \rangle$, where $\mathcal{D} = \langle \mathcal{F}, \mathcal{F}_{der}, \mathcal{X}, A, pre, eff \rangle$ is a nondeterministic domain, s_0 the initial state and φ a PPLTL goal, the encoded planning problem is $\Gamma' = \langle \mathcal{D}', s_0', G' \rangle$, where $\mathcal{D}' = \langle \mathcal{F}', \mathcal{F}'_{der}, \mathcal{X}', A, pre, eff' \rangle$ is the encoded nondeterministic planning domain, s_0' is the new initial state and

---

[1]Available online at `https://github.com/whitemech/Plan4Past`

Components	Encoding
Fluents $\mathcal{F}'$	$\mathcal{F}' := \mathcal{F} \cup \Sigma_\varphi$
Derived Predicates $\mathcal{F}'_{der}$	$\mathcal{F}'_{der} := \mathcal{F}_{der} \cup \{\mathsf{val}_\phi \mid \phi \in \mathsf{sub}(\varphi)\}$
Axioms $\mathcal{X}'$	$\mathcal{X}' := \mathcal{X} \cup \{x_\phi \mid \phi \in \mathsf{sub}(\varphi)\}$ where x_ϕ is $$\begin{cases} \mathsf{val}_p \leftarrow p & (\phi = p) \\ \mathsf{val}_{\mathsf{Y}\phi'} \leftarrow \text{``}\mathsf{Y}\phi'\text{''} & (\phi = \mathsf{Y}\phi') \\ \mathsf{val}_{\phi_1 \,\mathsf{S}\, \phi_2} \leftarrow (\mathsf{val}_{\phi_2} \vee (\mathsf{val}_{\phi_1} \wedge \text{``}\mathsf{Y}(\phi_1 \,\mathsf{S}\, \phi_2)\text{''})) & (\phi = \phi_1 \,\mathsf{S}\, \phi_2) \\ \mathsf{val}_{\phi_1 \wedge \phi_2} \leftarrow (\mathsf{val}_{\phi_1} \wedge \mathsf{val}_{\phi_2}) & (\phi = \phi_1 \wedge \phi_2) \\ \mathsf{val}_{\neg\phi'} \leftarrow \neg\mathsf{val}_{\phi'} & (\phi = \neg\phi') \end{cases}$$
Action Labels A	$A := A$, i.e., unchanged
Preconditions pre	$pre(a) := pre(a)$ for every $a \in A$, i.e., unchanged
Effects eff'	$eff'(a) := \{\mathsf{eff}_i \cup \mathsf{eff}_{\mathsf{val}} \mid \mathsf{eff}_i \in eff(a)\}$, where $\mathsf{eff}_{\mathsf{val}} = \{\mathsf{val}_\phi \rhd \{\text{``}\mathsf{Y}\phi\text{''}\}, \neg\mathsf{val}_\phi \rhd \{\neg\text{``}\mathsf{Y}\phi\text{''}\} \mid \text{``}\mathsf{Y}\phi\text{''} \in \Sigma_\varphi\}$
Initial State s_0'	$s_0' := \sigma_0 \cup s_0$
Goal G'	$G' := \mathsf{val}_\varphi$

Table 6.1: Components of the encoded FOND planning problem $\Gamma' = \langle\langle\mathcal{F}', \mathcal{F}'_{der}, \mathcal{X}', A, pre, eff'\rangle, s_0', G'\rangle$ using axioms for a given FOND planning problem $\Gamma = \langle\langle\mathcal{F}, \mathcal{F}_{der}, \mathcal{X}, A, pre, eff\rangle, s_0, \varphi\rangle$.

G' is the new reachability goal. Table 6.1 compactly summarizes the formal construction of Γ' presented in Section 5.2.1. Here, we recall that σ_0 present in the initial state s_0' represents the propositional interpretation of Σ_φ at the first instant. Moreover, the only difference between the encoding described in Chapter 5 and the one here in Table 6.1 is that $\mathcal{D}$ being nondeterministic, the new effects $eff'(a)$ will also contain all the possible nondeterministic outcomes $\{\mathsf{eff}_1, \ldots, \mathsf{eff}_n\}$ of $eff(a)$. Clearly, as shown in Table 6.1, the additional part $\mathsf{eff}_{\mathsf{val}}$ in $eff'(a)$ deterministically updates the assignments of propositions in Σ_φ without affecting the nondeterminism of the original domain. This is the main reason why the exact same encoding is correct for both classical and FOND planning.

The encoding is polynomially related to the original problem.

Theorem 6.1. *Let Γ be a FOND planning problem. The size of Γ' obtained following the encoding of Table 6.1 is polynomial in the size of Γ. In particular, it is linear in the size of the domain specification and linear in the size of the goal.*

Proof. By construction of the encoding. □

6.2.2 Correctness

Let $\Gamma = \langle \mathcal{D}, s_0, \varphi \rangle$ be a classical or FOND planning problem, where $\mathcal{D}$ is a nondeterministic domain, s_0 is the initial state, and φ is a PPLTL goal formula, and let $\Gamma' = \langle \mathcal{D}', s_0', G' \rangle$ be the corresponding encoded planning problem as previously defined.

We recall that any trace $\tau' = s_0', \ldots, s_n'$ on $\mathcal{D}'$ can be seen as the aggregation of $\tau^{[\varphi]}$ and τ, namely $\tau' = zip(\tau^{[\varphi]}, \tau)$, where $\tau = s_0, \ldots, s_n \in (2^{\mathcal{F}})^+$, $\tau^{[\varphi]} = \sigma_0, \ldots, \sigma_n \in (2^{\Sigma_\varphi})^+$, where each element of τ' is of the form $s_i' = (\sigma_i, s_i)$ for all $i \geq 0$. Given a trace $\tau' = s_0', \ldots, s_n'$ on the encoded planning domain $\mathcal{D}'$, there exists a single trace $\tau' \mid_{\mathcal{F}} = \tau = s_0, \ldots, s_n$ on the original planning domain $\mathcal{D}$. Conversely, given a trace $\tau = s_0, \ldots, s_n$ on the original planning domain $\mathcal{D}$, there exists a unique corresponding trace $\tau^{[\varphi]}$, and hence a single $\tau' = zip(\tau^{[\varphi]}, \tau)$ on the encoded domain $\mathcal{D}'$.

Theorem 6.2. *Let φ be a PPLTL formula over $\mathcal{P}$, $\phi \in \mathsf{sub}(\varphi)$ a subformula of φ, τ a trace over $\mathcal{P}$, and $\tau^{[\varphi]}$ the corresponding trace over Σ_φ. Then,*

$$\mathsf{val}(\phi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau)) \quad \textit{iff} \quad \mathsf{last}(\tau') \models \mathsf{val}_\phi$$

Proof. Analogous to the one of Theorem 5.3. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

Unlike the case of classical planning, when dealing with nondeterministic domain models, we need to reason about the correspondence of strategies. For every strategy $\pi : (2^{\mathcal{F}})^+ \to A$ for the FOND planning problem Γ with PPLTL goal φ, we can build the strategy $\pi' : (2^{\mathcal{F}'})^+ \to A$ for Γ' defined as $\pi'(\tau') = \pi(\tau' \mid \mathcal{F})$, where:

$$\pi'(\tau') = a \qquad\qquad \text{iff} \quad \pi(\tau' \mid_{\mathcal{F}}) = a$$
$$\pi'(\tau') \text{ is undefined} \qquad \text{iff} \quad \pi(\tau' \mid_{\mathcal{F}}) \text{ is undefined.}$$

Lemma 6.3. *For every strategy $\pi : (2^{\mathcal{F}})^+ \to A$ that is a (strong or strong-cyclic) winning strategy for the FOND planning problem Γ with PPLTL goal φ, the strategy $\pi' : (2^{\mathcal{F}'})^+ \to A$, defined as above, is a (strong or strong-cyclic, resp.) winning strategy for the encoded planning problem Γ'.*

Proof. Strategy π is winning if every generated execution τ (that is stochastic-fair, for strong-cyclic solutions) is finite, i.e., $\pi(\tau)$ is undefined, and such that $\tau \models \varphi$. Correspondingly, the strategy π' induces the finite generated execution $\tau' = zip(\tau^{[\varphi]}, \tau)$. Then, $\mathsf{val}(\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau))$ holds by Theorem 4.4. By construction of the encoding and by Theorem 6.2, we have that $\mathsf{val}(\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau))$ if and only if val_φ holds in the last instant, so we have that $\mathsf{last}(\tau') \models \mathsf{val}_\varphi$.

On the other hand, if a generated execution τ' is finite, i.e., such that $\pi'(\tau')$ is undefined, then π induces a corresponding finite generated execution $\tau = \tau' \mid_{\mathcal{F}}$. The strategy π being winning, it must be the case that $\tau \models \varphi$. Hence, by Theorem 4.4, $\mathsf{last}(\tau') \models \mathsf{val}_\varphi$. Thus, if π is winning for Γ, then π' is winning for Γ'. $\qquad \square$

Now we consider the converse. For every strategy $\pi' : (2^{\mathcal{F}'})^+ \to A$ for the encoded planning problem Γ', we can build the strategy $\pi : (2^{\mathcal{F}})^+ \to A$ for the original problem Γ with PPLTL goal φ defined as $\pi(\tau) = \pi'(\tau')$ (where $\tau' = zip(\tau^{[\varphi]}, \tau)$) with:

$$\begin{aligned} \pi(\tau) = a & \qquad \text{iff} \quad \pi'(\tau') = a \\ \pi(\tau) \text{ is undefined} & \qquad \text{iff} \quad \pi'(\tau') \text{ is undefined.} \end{aligned}$$

Lemma 6.4. *For every $\pi' : (2^{\mathcal{F}'})^+ \to A$ that is a (strong or strong-cyclic) winning strategy for the encoded planning problem Γ', the $\pi : (2^{\mathcal{F}})^+ \to A$, defined as above, is a (strong or strong-cyclic, resp.) winning strategy for the FOND planning problem Γ with PPLTL goal φ.*

Proof. Strategy π' is winning if every generated execution τ' (that is stochastic fair, for strong-cyclic solutions) is finite, i.e., such that $\pi'(\tau')$ is undefined, and such that $\mathsf{last}(\tau') \models \mathsf{val}_\varphi$. Correspondingly, the strategy π induces the finite generated execution $\tau = \tau' \mid_{\mathcal{F}}$. Then, by Theorems 4.4 and 6.2, considering that $\mathsf{val}(\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau))$ holds, we have that $\tau \models \varphi$.

On the other hand, if a generated execution τ is finite, i.e., such that $\pi(\tau)$ is undefined then π' induces a corresponding finite generated execution $\tau' = zip(\tau^{[\varphi]}, \tau)$. The strategy π' being winning, we have that $\mathsf{last}(\tau') \models \mathsf{val}_\varphi$. Hence, by Theorems 4.4 and 6.2, $\tau \models \varphi$. Thus, if the strategy π' is winning for Γ', then the strategy π is winning for Γ. $\qquad \square$

By Lemma 6.3 and Lemma 6.4 we immediately get:

Theorem 6.5 (Correctness). *Let Γ be a FOND planning problem with a PPLTL goal φ, and Γ' be the corresponding encoded FOND planning problem with reachability goal G'. Then, Γ has a (strong or strong-cyclic) winning strategy iff Γ' has a (strong or strong-cyclic, resp.) winning strategy.*

As a result, let Γ be a FOND (strong or strong-cyclic) planning problem with a PPLTL goal φ, and Γ' be the corresponding encoded FOND (strong or strong-cyclic, resp.) planning problem with reachability goal G'. Then, every sound and complete planner (FOND strong or FOND strong-cyclic, resp.) returns a winning strategy π' for Γ' if a winning strategy π for Γ exists. If no solution exists for Γ', then there is no solution for Γ.

Particularly in the nondeterministic case, it is also important to observe that strategies returned by a FOND planner for Γ' are going to be "memory-less" policies of the form $\Pi'(s') = a$ or $\Pi'(s')$ undefined at

the goal. These can be immediately transformed into trace-based strategies by defining:

$$\pi(\tau') = a \qquad \text{iff} \quad \Pi'(\mathsf{last}(\tau')) = a$$

$$\pi(\tau') \text{ is undefined} \quad \text{iff} \quad \Pi'(\mathsf{last}(\tau')) \text{ is undefined.}$$

This possibility is crucial since strategies for the original problem Γ with a PPLTL goal φ may not be memory-less policies. Intuitively, to see why this is true, it is sufficient to think about a policy that visits the same state twice but chooses two different actions depending on the satisfaction of the temporally extended goal along the trace. When FOND planning for temporally extended goals is encoded into FOND planning for standard reachability goals, then planning states will encode additional fluents accounting for the temporal goal, and therefore, the policy can be memory-less. In other words, they might need to be finite-state controllers or transducers. We can use the component σ_i of $s_i' = (\sigma_i, s_i)$ as the state of the transducer, $\sigma_{i+1}(\text{``}\mathsf{Y}\phi\text{''}) = \mathsf{val}(\phi, \sigma_i, s_i)$ (for each "$\mathsf{Y}\phi$" $\in \Sigma_\varphi$) as the factorized transition function, and $\Pi'(s_i')$ as the output function of the transducer. In the case of deterministic domains, we do not need these general forms of strategies because sequences of actions suffice, and these are in direct correspondence between the two domains.

6.2.3   Encoding without Derived Predicates

Given that almost all current state-of-the-art FOND planners do not handle axioms well, we describe a variant of the encoding presented in the previous section that does not introduce additional derived predicates. Like the previous encoding, this variant introduces a fresh atom of the form "$\mathsf{Y}\phi$" for each temporal component of φ. However, instead of using the val_ϕ predicates, it combines quoted atoms with propositions of the original domain to explicitly represent the PNF of a formula and does so by representing the formula in PPNF (Propositional Previous Normal Form).

Definition 6.6. *Let φ be a PPLTL formula. $\mathsf{ppnf}(\varphi)$ is a propositional formula obtained by substituting every $\mathsf{Y}(\phi)$ with "$\mathsf{Y}(\phi)$" in $\mathsf{pnf}(\varphi)$.*

For instance, if $\varphi = a \mathbin{\mathsf{S}} b$ then $\mathsf{ppnf}(\varphi) = b \vee (a \wedge \text{``}\mathsf{Y}(a \mathbin{\mathsf{S}} b)\text{''})$. For any formula φ, the $\mathsf{ppnf}(\varphi)$ captures the truth of $\mathsf{pnf}(\varphi)$ without using temporal operators, provided that every "$\mathsf{Y}\phi$" $\in \Sigma_\varphi$ reflects the truth of $\mathsf{Y}\phi$. Most importantly, $\mathsf{ppnf}(\varphi)$ is linear in the size of φ.

Lemma 6.7. *Let φ be a PPLTL formula. The size of $\mathsf{ppnf}(\varphi)$ is linear in the size of φ.*

Proof. As for the $\mathsf{pnf}(\varphi)$ in Chapter 4, we observe that the definition of the $\mathsf{ppnf}(\varphi)$ does not recur on the subformulas of the form "$\mathsf{Y}\phi$" and "$\mathsf{Y}(\phi_1 \mathbin{\mathsf{S}} \phi_2)$". Moreover, for each operator in the formula, we recur on its

Components	Encoding
Fluents $\mathcal{F}''$	$\mathcal{F}'' := \mathcal{F} \cup \Sigma_\varphi$
Derived Predicates $\mathcal{F}_{der}$	$\mathcal{F}_{der} := \mathcal{F}_{der}$, i.e., unchanged
Axioms $\mathcal{X}$	$\mathcal{X} := \mathcal{X}$, i.e., unchanged
Action Labels A	$A := A$, i.e., unchanged
Preconditions pre	$pre(a) := pre(a)$ for every $a \in A$, i.e., unchanged
Effects eff''	$\mathit{eff}''(a) := \{\mathsf{eff}_i \cup \mathsf{eff}_{\mathsf{ppnf}} \mid \mathsf{eff}_i \in \mathit{eff}(a)\}$, where $\mathsf{eff}_{\mathsf{ppnf}} = \{\mathsf{ppnf}(\phi) \rhd \{\text{``}\mathsf{Y}\phi\text{''}\}, \neg\mathsf{ppnf}(\phi) \rhd \{\neg\text{``}\mathsf{Y}\phi\text{''}\} \mid \text{``}\mathsf{Y}\phi\text{''} \in \Sigma_\varphi\}$
Initial State s_0''	$s_0'' := \sigma_0 \cup s_0$
Goal G''	$G'' := \mathsf{ppnf}(\varphi)$

Table 6.2: Components of the encoded FOND planning problem $\Gamma'' = \langle\langle\mathcal{F}'', \mathcal{F}_{der}, \mathcal{X}, A, pre, \mathit{eff}''\rangle, s_0'', G''\rangle$ without additional derived predicates for a given FOND planning problem $\Gamma = \langle\langle\mathcal{F}, \mathcal{F}_{der}, \mathcal{X}, A, pre, \mathit{eff}\rangle, s_0, \varphi\rangle$.

components only once. Hence, the final size of the $\mathsf{ppnf}(\varphi)$ is linear in the size of φ. $\qquad\square$

Given a FOND planning problem $\Gamma = \langle\mathcal{D}, s_0, \varphi\rangle$ where $\mathcal{D} = \langle\mathcal{F}, \mathcal{F}_{der}, \mathcal{X}, A, pre, \mathit{eff}\rangle$ and φ is a PPLTL goal, Table 6.2 formalizes the encoding of an equivalent FOND problem $\Gamma'' = \langle\mathcal{D}'', s_0'', G''\rangle$ with $\mathcal{D}'' = \langle\mathcal{F}'', \mathcal{F}_{der}, \mathcal{X}, A, pre, \mathit{eff}''\rangle$ for a goal G''. Compared to the encoding of Table 6.1, the function eff'' uses the propositional formula $\mathsf{ppnf}(\phi)$ to update the truth of each quoted atom "$\mathsf{Y}(\phi)$" $\in \Sigma_\varphi$ after every action. Intuitively, if $\mathsf{ppnf}(\phi)$ holds in s, then $\mathsf{Y}(\phi)$ will hold in the successor state s', and we keep track of this by ensuring that "$\mathsf{Y}(\phi)$" holds true in s', formalized via conditional effect $\mathsf{ppnf}(\phi) \rhd \{\text{``}\mathsf{Y}\phi\text{''}\}$. Analogously, if a state $s$ does not satisfy $\mathsf{ppnf}(\phi)$, then the conditional effect $\neg\mathsf{ppnf}(\phi) \rhd \{\neg\text{``}\mathsf{Y}\phi\text{''}\}$ will make "$\mathsf{Y}\phi$" false in s'. These effects are added to each outcome of an action a. Finally, goal G'' asks to satisfy $\mathsf{ppnf}(\varphi)$.

By comparing the encoding shown in Table 6.1 and the one shown here in Table 6.2, it is easy to see that the latter encoding is formally correct too. First, by the construction of the ppnf and the derived predicates of Γ', for any state s, we have that $s \models \mathsf{ppnf}(\phi)$ if and only if $s \models \mathsf{val}_\phi$ for all $\phi \in \mathsf{sub}(\varphi)$. Consequently, the additional conditional effects $\mathsf{val}_\phi \rhd$ "$\mathsf{Y}(\phi)$" and $\neg\mathsf{val}_\phi \rhd \neg$"$\mathsf{Y}(\phi)$" of Table 6.1 are triggered by a state s if and only if the conditional effects of the encoding in Table 6.2, $\mathsf{ppnf}(\phi) \rhd$ "$\mathsf{Y}(\phi)$" and $\neg\mathsf{ppnf}(\phi) \rhd \neg$"$\mathsf{Y}(\phi)$", are triggered. And the same is trivially true for the goal conditions. Thus, the effects $\mathsf{eff}'(a)$ and $\mathsf{eff}''(a)$ will induce the same set of possible successor states. Now, given that Γ'' and Γ' share the same fluents, initial state, actions, and precondition function, we have that an action a is applicable in a state s for Γ' if and only if it is for Γ''. Hence, a winning strategy for Γ', which we previously proved to be winning also for Γ,

is a winning strategy for Γ'' as it will induce the same set of traces.

Furthermore, although this second encoding removes additional axioms that the first encoding generates, it remains polynomially related to the original problem as, intuitively, the biggest part of $\mathsf{eff}_{\mathsf{ppnf}}$ is represented by the conditions $\mathsf{ppnf}(\phi)$ and $\neg\mathsf{ppnf}(\phi)$ when $|\phi| = |\varphi|$ (i.e., when the subformula is the formula itself).

6.3 Experimental Evaluation

We implemented the approaches of Sections 6.2.1 and 6.2.3 in the Plan4Past tool, which can be run with either the intensive conditional effects compilation ($\mathsf{P4P}_{\mathsf{ce}}$ for short) – approximately 1300 LOC – or with axioms ($\mathsf{P4P}_{\mathcal{X}}$ for short). Both configurations take as input a FOND planning problem, written in PDDL, and a PPLTL formula giving as output a PDDL description of a new FOND problem.

Preliminary experiments revealed how current FOND planners struggle to preprocess most problems compiled by $\mathsf{P4P}_{\mathsf{ce}}$. To overcome this issue, we further optimize this encoding by aggregating the set $\mathsf{eff}_{\mathsf{ppnf}}$ of new conditional effects into a dummy action `check`. Then, we force the encoding to always execute one occurrence of `check` before any domain actions. Clearly, such a modification does not undermine our theoretical results and proves to be much more convenient with the current FOND planners. Thus, we will hereinafter refer to $\mathsf{P4P}_{\mathsf{ce}}$ as the compilation with such a modification.

We used two state-of-the-art FOND planners: PRP (Muise et al., 2012) and Paladinus (Fraga Pereira et al., 2022). Both engines support basic conditional effects. However, although Paladinus fully supports axioms, it does not support disjunctive conditional effects. Conversely, PRP supports conditional effects with disjunctive conditions but does not support axioms. Hence, since $\mathsf{P4P}_{\mathsf{ce}}$ requires disjunctive conditional effects and no axioms while $\mathsf{P4P}_{\mathcal{X}}$ requires conjunctive conditional effects and axioms, we use $\mathsf{P4P}_{\mathsf{ce}}$ with PRP ($\mathsf{P4P}_{\mathsf{ce}}^{\mathsf{PRP}}$ for short), and Paladinus with $\mathsf{P4P}_{\mathcal{X}}$ ($\mathsf{P4P}_{\mathcal{X}}^{\mathsf{Pal}}$ for short).

Our empirical analysis aims to evaluate the effectiveness of temporally extended goals formulated in PPLTL and handled by Plan4Past, and *semantically* equivalent (polynomially related) temporally extended goals formulated in LTL_f and handled by the state-of-the-art tool ltlfond2fond[2] (ltlf2f for short) (Camacho et al., 2017; Camacho and McIlraith, 2019). ltlf2f is a compilation approach that explicitly computes an automaton representing the LTL_f temporal goal. The advantage of Plan4Past seems clear: the encoding performed by ltlf2f is exponential, while both compilations performed by $\mathsf{P4P}_{\mathsf{ce}}$ and $\mathsf{P4P}_{\mathcal{X}}$ are polynomial in the size of the PPLTL goals. Yet, from a practical standpoint, the impact of this advantage in FOND planning is unclear. To this end, we tested the three systems over a set benchmark and analyzed the number

---

[2]ltlfond2fond is available online at `https://bitbucket.org/acamacho/ltlfond2fond` and can operate in two modes: one introduces conditional effects, while the other substitutes such effects with a cascade of actions. In our experiments, we used the latter, as it always performs better independently from the planner.

of problems solved (Coverage), the time spent to get a solution (compilation plus search), and the size of the policies (number of state-action pairs). Like for P4P$_{ce}$, the problems encoded by ltlf2f are supported by FOND planners with conditional effects, but in our findings, ltlf2f performs much better with PRP. Hence, in the following, we only present the ltlf2f results with PRP. All experiments ran up to 1800s on a Xeon-Gold 6140M 2.3 GHz with 8GB of memory.

6.3.1 Benchmark Domains

Our benchmark suite features the FOND domains ROVERS, BLOCKSWORLD, and ROBOT-COFFEE used by Camacho et al. (2017). We tested all compilations on the same instances tested by Camacho et al. (2017) (C17 for short). C17 temporally extended goals were originally specified in LTL$_f$, and therefore, for comparability reasons, we manually translated them to PPLTL. In particular, for each LTL$_f$ formula translated in PPLTL, we formally and automatically proved their semantic equivalence by verifying that the two formulations yield the same minimal DFA (modulo state renaming – cf. Section 3.5.2). We observed that all C17 instances were trivially solved by the three systems. Hence, to study the scalability of the compilations for each domain, we generated a new set of instances of increasing dimensions (BF23 for short). The C17 instances are publicly available, while our newly generated instances are described below, along with other information about the domain and the translations from PPLTL and LTL$_f$.

BlocksWorld. In BLOCKSWORLD, we aim to arrange blocks to a particular configuration. An arm can pick-up and move blocks on top of other blocks or on the table. In the nondeterministic version, a block can fall on the table while the arm is holding it. Starting from the 24 C17 instances, we generated bigger problems considering the four types of formula employed by Camacho et al. (2017):

$$\mathsf{F}(a) \cup (\mathsf{F}(b_1) \wedge \mathsf{F}(b_2) \wedge \ldots \wedge \mathsf{F}(b_n)) \tag{1}$$

$$(\mathsf{F}(b_1) \vee \mathsf{F}(b_2) \vee \ldots \vee \mathsf{F}(b_n)) \cup \mathsf{F}(a) \tag{2}$$

$$\mathsf{F}(\mathsf{F}(b_1) \wedge \mathsf{F}(b_2) \wedge \ldots \wedge \mathsf{F}(b_n) \wedge a) \tag{3}$$

$$(a \cup \mathsf{F}(b_1)) \wedge (a \cup \mathsf{F}(b_2)) \wedge \ldots \wedge (a \cup \mathsf{F}(b_n)) \tag{4}$$

where atoms a and b_i represent whether a block is on the table or on top of another block. Formulas (1) and (4) require each atom b_i to eventually be true in some state, while formula (2) expresses that a eventually becomes true. In all three formulas, the "Until" ($\cup$) does not add any semantic meaning (i.e., requirements to be satisfied by the trace) to the formula. For these three lucky cases, the equivalent PPLTL formulation is obtained by simply swapping the future temporal operators with the past ones. For instance, we write

formula (4) in PPLTL as $(a \, \mathsf{S} \, \mathsf{O}(b_1)) \wedge (a \, \mathsf{S} \, \mathsf{O}(b_2)) \wedge \ldots \wedge (a \, \mathsf{S} \, \mathsf{O}(b_n))$. On the other hand, the formula (3) expresses that "there exists a state where a is true, and each atom b_i is true in the same state or future states". In this case, the translation in PPLTL is not straightforward: we used the semantically equivalent formula $\mathsf{O}(b_1 \wedge \mathsf{O}(a)) \wedge \mathsf{O}(b_2 \wedge \mathsf{O}(a)) \wedge \ldots \wedge \mathsf{O}(b_n \wedge \mathsf{O}(a))$. Each temporal goal is used over 10 instances obtained by adding more blocks on the table, resulting in 40 new instances.

To challenge the different compilations for this domain, we also designed a new type of formula, i.e., $seq_{i,j}$ where i is the number of blocks, and j is the number of towers to build in a specific order. For example, formula $seq_{3,2}$ requires building two towers made of three blocks. In the end, such blocks must be on the table. In LTL$_f$, this formula is expressed as

$$\mathsf{F}(on_{a,c} \wedge on_{c,b} \wedge \mathsf{XF}(on_{c,b} \wedge on_{b,a} \wedge \mathsf{XF}(on_{a,table} \wedge on_{b,table} \wedge on_{c,table}))),$$

whereas in PPLTL it is expressed as

$$\mathsf{O}(on_{a,table} \wedge on_{b,table} \wedge on_{c,table} \wedge \mathsf{YO}(on_{c,b} \wedge on_{b,a} \wedge \mathsf{YO}(on_{a,c} \wedge on_{c,b}))).$$

We generated a new instance with a goal $seq_{i,j}$ for $i = 3, 4$ and with $j = 1, \ldots, i!$, for a total of 30 "seq" problems. Therefore, the total number of instances for BLOCKSWORLD sums up to 94 instances. These include the 24 original instances from (Camacho et al., 2017), 40 instances that we generated by scaling the temporal goals from (Camacho et al., 2017), and the 30 "seq" instances.

Rovers. In this domain, the goal is to gather data about soil, rocks, and images of a planet by planning the activities of one or more rovers. We used the 9 instances in C17 and generated other larger instances with 4 types of formulas:

$$\mathsf{F}(g_1) \wedge \mathsf{F}(g_2) \wedge \ldots \wedge \mathsf{F}(g_n) \tag{1}$$

$$\mathsf{F}(\mathsf{F}(g_1) \wedge \mathsf{F}(g_2) \wedge \ldots \wedge \mathsf{F}(g_n)) \tag{2}$$

$$\mathsf{F}(\mathsf{F}(g_1) \vee \mathsf{F}(g_2) \vee \ldots \vee \mathsf{F}(g_n)) \tag{3}$$

$$\mathsf{G}(\mathsf{F}(\mathsf{G}(g_1) \vee \mathsf{G}(g_2) \vee \ldots \vee \mathsf{G}(g_n))) \tag{4}$$

where every g_i is an atom representing the completion of a task, i.e., the communication of data about soil, rock, or image. The first two formulas require that each task is accomplished in some state of the trajectory induced by the execution of the policy. Formula (3) requires that at least one task is eventually completed,

whereas the last formula requires that all tasks are satisfied in the last state. The translation for formulas (1), (2), and (3) is luckily direct; we can simply swap future operators with their past counterparts. For instance, we write formula (2) in PPLTL as $O(O(g_1) \wedge O(g_2) \wedge \ldots \wedge O(g_n))$. The last formula requires one of the goals to be achieved in the last state; this could be represented in PPLTL without using any temporal operator. However, for the sake of a fair comparison, we translated such LTL_f formula into a PPLTL formula with the same number of nested temporal operators, i.e., $H(H(H(g_1) \vee H(g_2) \vee \ldots \vee H(g_n))) \vee (g_1 \vee g_2 \vee \ldots \vee g_n)$. These constraints are used over 40 propositional problems of ROVERS from the 5th IPC, giving us 160 instances, which, added to the 9 C17 instances, gives a total of 169 instances.

Robot-Coffee. In this domain, a robot has to deliver coffee to different offices and can move between adjacent offices; coffee can be prepared in a kitchen. We considered the 10 problems from C17 and generated further larger instances with the following types of temporally extended goals:

$$F(C_{o_1}) \wedge \ldots \wedge F(C_{o_n}) \tag{1}$$

$$F(F(C_{o_1}) \wedge \ldots \wedge F(C_{o_n})) \tag{2}$$

$$F(C_{o_1}) \wedge \ldots \wedge F(C_{o_n}) \wedge G(R_{o_i} \to X(\neg R_{o_j})) \wedge \ldots \tag{3}$$

$$X(X(\ldots X(R_{kitchen}))) \tag{4}$$

The first two formulas are satisfied by a policy delivering the coffee to all offices. Formula (3) is as formula (1) but with the requirement that if the robot is in the office o_i, then in the next state, it cannot be in office o_j. This in PPLTL can be easily expressed with $H(Y(R_{o_i}) \to \neg(R_{o_j})) \wedge \neg R_{o_i}$. The last temporally extended goal requires the robot to be in the kitchen in the k+1-th state, where k is equal to the number of nested X operators. In PPLTL, this can be captured by $O(R_{kitchen} \wedge Y(Y(\ldots Y(start))))$, where the number of Y is equal to k. We generated a new set of instances, 15 for each type of formula, increasing the number of offices. Formulas (1), (2), and (3) scale with the number of offices, whereas the formula of type (4) scales with k ranging from 7 to 21. This gives us a total of 70 instances.

6.3.2 Experimental Results

Table 6.3 reports the overall results for all systems. The last three rows are for C17, while BF23 is indicated by "domain-formula" (e.g., BLOCKSWORLD-1 refers to BLOCKSWORLD with goals of type (1)). $\text{P4P}^{\text{PRP}}_{\text{ce}}$ achieves the highest coverage, followed by ltlf2f and $\text{P4P}^{\text{Pal}}_{\mathcal{X}}$. C17 instances are small and trivially solved by all systems within a few seconds. In contrast, BF23 instances are larger and designed to challenge the compilations. Indeed, here we observe that $\text{P4P}^{\text{PRP}}_{\text{ce}}$ solves 80.2% more instances than $\text{P4P}^{\text{Pal}}_{\mathcal{X}}$ and 69.9%

Domain	I	Coverage			Avg RT			Avg $|\pi|$		
		$\mathsf{P4P_{ce}^{PRP}}$	$\mathsf{P4P_{\mathcal{X}}^{Pal}}$	ltlf2f	$\mathsf{P4P_{ce}^{PRP}}$	$\mathsf{P4P_{\mathcal{X}}^{Pal}}$	ltlf2f	$\mathsf{P4P_{ce}^{PRP}}$	$\mathsf{P4P_{\mathcal{X}}^{Pal}}$	ltlf2f
BLOCKSWORLD-1	10	**10**	**10**	1	69.56	**14.98**	–	4.00	**3.00**	–
BLOCKSWORLD-2	10	**10**	**10**	10	27.11	10.99	**2.72**	2.00	**1.00**	2.00
BLOCKSWORLD-3	10	6	**10**	2	85.48	**5.99**	–	5.00	**2.00**	–
BLOCKSWORLD-4	10	6	**10**	4	30.04	**11.47**	19.56	**3.00**	**3.00**	3.00
BLOCKSWORLD-seq	30	**30**	8	20	**2.16**	32.25	2.29	14.12	14.00	**13.75**
ROBOT-COFFEE-1	15	**12**	5	4	**5.14**	53.99	53.34	56.00	**50.50**	56.00
ROBOT-COFFEE-2	15	**11**	5	2	**14.96**	361.63	–	60.00	**54.00**	–
ROBOT-COFFEE-3	15	**2**	0	1	–	–	–	–	–	–
ROBOT-COFFEE-4	15	**15**	**15**	15	15.82	2.59	**2.04**	15.00	**14.47**	15.00
ROVERS-1	40	**24**	3	10	**8.99**	–	23.47	**28.40**	–	30.50
ROVERS-2	40	**23**	4	7	–	–	–	–	–	–
ROVERS-3	40	37	20	**40**	27.37	258.75	**2.81**	6.05	5.90	**5.30**
ROVERS-4	40	**23**	16	7	**32.97**	52.34	–	5.62	**5.06**	–
BLOCKSWORLD	24	**24**	**24**	**24**	2.66	1.85	**1.71**	7.71	**5.21**	6.75
ROBOT-COFFEE	10	**10**	**10**	**10**	2.46	10.03	**1.97**	14.22	**12.33**	13.44
ROVERS	9	**9**	**9**	**9**	3.42	7.41	**1.85**	9.11	12.56	**9.11**
Total		**252**	159	166						

Table 6.3: Coverage, average Run-Time (Avg RT), and Policy size (Avg $|\pi|$) achieved by $\mathsf{P4P_{ce}}$, $\mathsf{P4P_{\mathcal{X}}}$ and ltlf2f. Averages are computed among instances solved by all systems obtaining at least 25% of coverage compared to the best performer. The policy size is reported without counting spurious actions added by compilations. Column I is the number of instances in a domain. "–" indicates a system excluded by the comparison. In bold are the best performers.

more instances than ltlf2f, yet $\mathsf{P4P_{ce}^{PRP}}$ does not dominate the other compilations. $\mathsf{P4P_{\mathcal{X}}^{Pal}}$ is effective in handling BLOCKSWORLD-3 and BLOCKSWORLD-4, while $\mathsf{P4P_{ce}^{PRP}}$ times out. Instead, $\mathsf{P4P_{ce}^{PRP}}$ works better with goals of type "seq". ltlf2f is extremely effective at handling ROVERS-3. Here, temporal goals can be represented by a very compact automaton, enabling ltlf2f to quickly solve all problems. This advantage can neither be exploited by $\mathsf{P4P_{ce}}$ nor by $\mathsf{P4P_{\mathcal{X}}}$; such compilations work on the syntactic structure of formulas and are unable to exploit the semantics of temporal goals. Instead, in Rovers and Robot-Coffee, $\mathsf{P4P_{ce}^{PRP}}$ is more effective than ltlf2f when temporal goals are of type (1) and (2). The automaton blows up when such instances become larger, and ltlf2f either fails at compilation time or produces problems too complex for PRP to handle. In ROVERS-4, $\mathsf{P4P_{ce}^{PRP}}$ and $\mathsf{P4P_{\mathcal{X}}^{Pal}}$ perform well, while we observed that in many instances ltlf2f crashes during the automaton computation. ROBOT-COFFEE-4 is easily handled by all systems, while ROBOT-COFFEE-3 proved to be challenging for every compilation. Finally, all systems computed policies of comparable dimensions.

On average, ltlf2f has the lowest runtime in 6 domains (3 are C17 domains featuring small instances), whereas $\mathsf{P4P_{ce}^{PRP}}$ and $\mathsf{P4P_{\mathcal{X}}^{Pal}}$ combined have the lowest runtime in 8 domains. To shed some light on this, Figure 6.1 displays coverage over time for all systems. ltlf2f solves more instances than the other compilations only at the beginning until the planning time reaches 29.7 seconds (visible in the logarithmic scale plot on
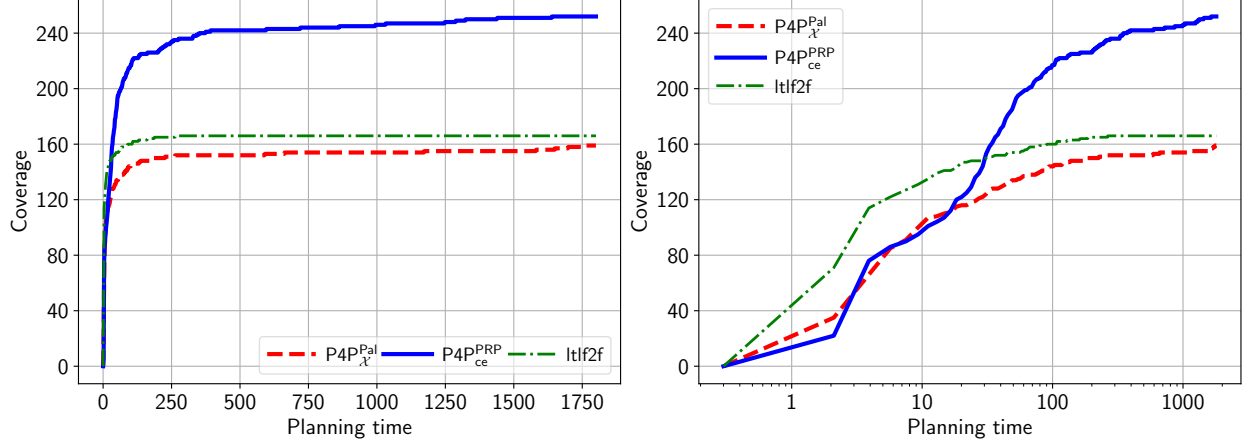
Figure 6.1: Survival plot in linear (left) and logarithmic (right) scale.
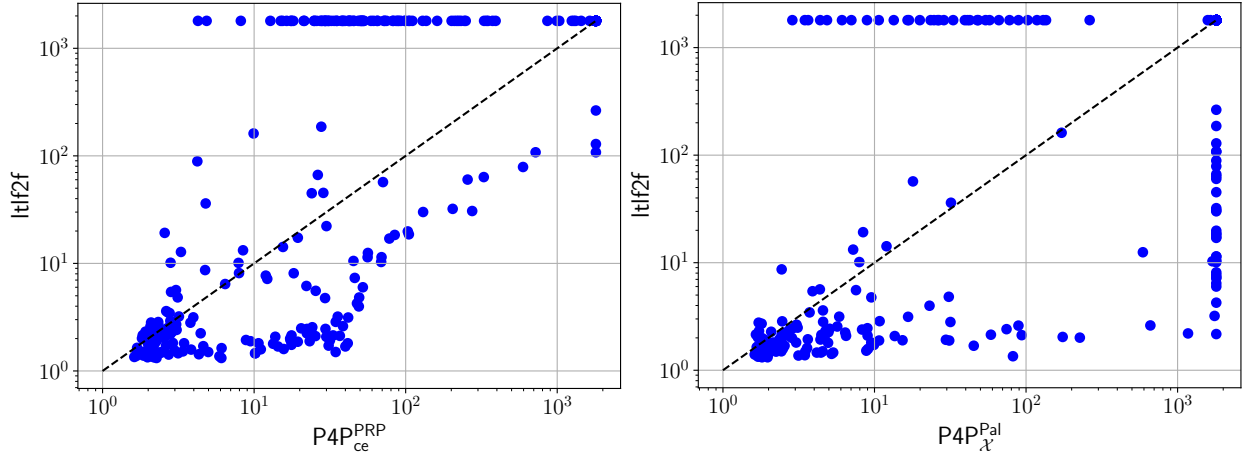
Figure 6.2: Run-Time comparison of ltlf2f vs $\mathsf{P4P}_{ce}^{PRP}$ and $\mathsf{P4P}_{\mathcal{X}}^{Pal}$.

the right). Instead, from that point on, $\mathsf{P4P}_{ce}^{PRP}$ visibly dominates ltlf2f (linear scale plot on the left). In an instance-by-instance comparison (Figure 6.2, left), we observe that ltlf2f solves many problems (130) before $\mathsf{P4P}_{ce}^{PRP}$ does. $\mathsf{P4P}_{ce}$ introduces complex formulas in conditional effects and goals, and we observed that the preprocessing of PRP often exceeds the compilation time by orders of magnitude. Overcoming this issue without introducing axioms is an open question for future work. The runtime pairwise comparison of ltlf2f with $\mathsf{P4P}_{\mathcal{X}}^{Pal}$, Figure 6.2 (right), shows that ltlf2f is faster than $\mathsf{P4P}_{\mathcal{X}}^{Pal}$ in most instances, and this behavior can be attributed to Paladinus being slower than PRP in the considered domains.

6.4   Summary and Discussion

In this chapter, we study FOND planning for PPLTL goals. PPLTL expresses temporal specifications that solution strategies must comply with. We formally prove that FOND planning for PPLTL goals can be poly-

nomially encoded to FOND planning for reachability goals, presenting two encodings that allow solving more problems than those solved by a state-of-the-art approach supporting the same equivalent goals expressed in LTL$_f$. The theoretical and practical advantages of PPLTL observed here may definitely make PPLTL a promising candidate to become a mainstream language to express temporal goals in planning. Future work concerns the development of a FOND planner that can *natively* handle PPLTL goals.

Chapter 7

Declarative Trace Alignment via Automated Planning

Analyzing traces of events produced by a process under execution is critical to many tasks in Business Process Management (BPM). However, modeling, implementation, or human-execution errors often make a trace non-compliant with respect to a BPM model. *Trace Alignment* is the problem of aligning process executions to a process model by "repairing" execution traces with a minimal number of modifications. In this dissertation, we consider *declarative* trace alignment, where the process model is a formal specification expressed in the linear-time temporal or dynamic logics on finite traces. We solve the problem by reducing it to cost-optimal planning and resorting to state-of-the-art automated planners. The resulting approach impressively outperforms existing ad-hoc solutions.

The work presented here will be partially included in a submission to the Journal of Artificial Intelligence Research. All new PDDL encodings have been implemented in a tool called TraceAligner, which currently represents the best-performing tool for Trace Alignment with declarative specifications. A separate article describing its structure and functioning has been published in (De Giacomo et al., 2023) on the Journal of Software Impacts.

7.1 Introduction

Business Process Management (BPM) is the research area concerned with discovering, modeling, analyzing, and managing business processes (BPs) to measure their productivity and improve their performance (Dumas et al., 2018). Usually, BPs are high-level processes involving automated and human-based activities such

that, when executed, generate sequences of *activities* (or events) called *traces*, typically collected in a *log* (i.e., a set of traces). When activities require manual intervention, it is not uncommon for log traces to be inconsistent with the expected process behavior. For instance, an insurance claim process where a human operator is responsible for collecting all the documents related to the claim, checking the information they contain, and, if correct, starting the claim process, is highly error-prone. Therefore, identifying and analyzing such traces to prevent errors is of paramount importance, and this is the main objective of what in BPM is known as *Trace Alignment* (Adriansyah et al., 2011; Carmona et al., 2018). Existing works from Process Mining have witnessed that trace alignment is a highly-relevant problem with practical value to uncover common and frequent deviation patterns in several domains.

An instance of trace alignment includes a log trace, a BP model, or *specification*, and a cost for each modification (insertion or deletion of activities) applicable to the input trace. Thus, trace alignment is the problem of checking whether an actual trace related to a BP execution conforms to the expected process behavior and, if not, finding a *minimal* set of changes that *aligns* the trace to the process. Such changes mainly consist in adding or deleting activities at some positions in the trace when necessary. To solve the trace alignment problem, existing approaches, e.g., (de Leoni et al., 2012; de Leoni et al., 2015), are based on ad-hoc implementations of the A^* search algorithm, which compromises scalability as the input complexity increases, namely when there are large specifications and long traces. In this dissertation, we reduce the trace alignment problem to deterministic cost-optimal planning (Geffner and Bonet, 2013) to exploit the efficiency, versatility, and customizability of state-of-the-art planners. The work has already been published in (De Giacomo et al., 2017), where the approach is validated through experimental analysis, and results show that it outperforms by far ad-hoc techniques included in the PROM toolkit (`promtools.org`). Such good results are confirmed and further improved in the recent extension that we present here, which not only extends the expressiveness of the specification formalism but also analyzes several reductions and evaluates their effectiveness. As for the encodings, we specifically tested several semantically equivalent variants, spanning from a very high-level and easily readable one up to grounded STRIPS-like versions. The encodings differ in many features, such as the presence of conjunctive goals or the way states are encoded.

Trace alignment is interesting for the AI community in two respects. First, the problem can be applied to executing agents: traces can model agent executions, whereas specifications can model properties that agent executions are expected to satisfy. In this way, solving the problem can be regarded as an approach to identify and possibly fix potential deviations of an agent's behavior from its nominal one. Note that this is somewhat related to existing works on verifying agent conformance to agent interaction protocols, e.g., (Baldoni et al., 2005; Ancona et al., 2013; El-Menshawy et al., 2013; Abushark et al., 2017). Second, the problem is an interesting application of planning, which turns out to be orders of magnitude more efficient

than state-of-the-art ad-hoc tools, thus witnessing once more the power and generality of planning. Finally, it is worth noticing that parts of the reduction and encodings devised here are applicable in general to any problem that includes temporal constraints expressible as finite-state automata and that the experimental study carried out here provides useful guidelines for an efficient representation of such constraints as part of a planning domain. Thus, while devised and presented for the specific case of trace alignment, the obtained results are of general applicability.

7.2 Model Specification Languages

A BP model defines the (possibly partial) execution order of the activities of interest and can be specified either *procedurally* or *declaratively*. A procedural specification consists of an executable model, such as a Petri net or a BPMN[1]/BPEL[2] specification, whereas a declarative specification consists of a set of formal constraints over the traces (BP executions). While a procedural specification prescribes the execution steps of the BP (i.e., when and under which conditions the various activities have to be executed), a declarative specification uses formal languages to define the set of requirements BP executions have to comply with. The former approach is well suited for actual execution, while the latter provides a process description amenable to various forms of analysis.

Previous approaches employing declarative models, such as (de Leoni et al., 2012; de Leoni et al., 2015; De Giacomo et al., 2016, 2017), are all based on a restricted set of predefined template modeling formulas called DECLARE (van der Aalst et al., 2009). Although DECLARE identifies useful patterns, it is limited in its expressiveness. For this reason, in this dissertation, we consider model specifications provided as formulas in full-fledged linear-time temporal logic and linear-time dynamic logic on finite traces, respectively LTL_f and LDL_f (De Giacomo and Vardi, 2013), which are strictly more expressive than DECLARE. But our approach could be seamlessly extended to consider also PPLTL or PPLDL formulas. In our case, temporal logics express properties that involve trace positions, and interpretations are events occurring at different positions of the trace. As mentioned in Chapter 2, LTL_f and LDL_f give great flexibility in constraints modeling. While LTL_f is usually sufficient to capture properties of interest without compromising readability, LDL_f augments LTL_f with regular expressions, keeping the same complexity properties. Also, in this context from an algorithmic point of view, the key aspect of using LTL_f and LDL_f formulas as specification models is being able to exploit their translation to equivalent finite-state automata (De Giacomo and Vardi, 2013) (cf. Section 2.4).

---

[1]`https://www.omg.org/spec/BPMN/2.0`
[2]`https://www.oasis-open.org/committees/download.php/12791`

7.3   Trace Alignment Problem

A *log trace* is a trace such that the propositional interpretation associated with each position contains only one proposition, i.e. a singleton. In this chapter, for simplicity and without loss of generality, we only refer to log traces. For notational convenience, we use single propositions (not singletons): we write $\tau = a, b, c$ instead of $\tau = \{a\}, \{b\}, \{c\}$. Consider a (log) trace $\tau = e_1, \ldots, e_n$ over the finite set of events, i.e., propositions, $\mathcal{E}$, and an $\text{LTL}_f/\text{LDL}_f$ formula φ such that $\tau \not\models \varphi$. We want to "repair" τ, i.e., changing it into a new trace $\hat{\tau}$ such that $\hat{\tau} \models \varphi$. We can repair traces by executing the following operations: *skip* an event, i.e., leave the event unchanged; *delete* an event from a position; *add* a new event at a certain position. We represent these operations through special events, extending the set of event $\mathcal{E}$ with fresh events del_e and add_e for every event $e \in \mathcal{E}$. We call such events *repair events*, denote the obtained set as $\mathcal{E}^+$, and call traces over $\mathcal{E}^+$ *repair traces*. Observe that although *skip* events are technically repair events, they are drawn directly from $\mathcal{E}$ and have no cost. Therefore, in a repair trace, $e \in \mathcal{E}$ stands for the skipping of event e, del_e stands for deletion of e, and add_e for the addition of e. In general, modifications of traces are not allowed but can be obtained as deletions followed by additions. Intuitively, given a trace $\tau = e_1, \ldots, e_k, \ldots, e_n$, an event p can be added to τ at position k when $1 \leq k \leq n$, resulting in $\hat{\tau} = e_1, \ldots, e_{k-1}, p, e_k, \ldots, e_n$. An event e_k can also be deleted from τ, resulting in $\hat{\tau}' = e_1, \ldots, e_{k-1}, e_{k+1}, \ldots, e_n$. A repair trace τ^+ represents a set of modifications that transform, when successfully executed, a trace τ into a new trace $\hat{\tau}$. We say that a repair trace τ^+ is *applicable* to a log trace τ, if τ^+ can be obtained from τ by: (i) inserting any arbitrarily long sequence of events of the form add_*, with $*$ being any event from $\mathcal{E}$, before the first event, after the last event, or between any two consecutive events of τ; (ii) replacing any event e of τ by either itself or del_*. We formally capture the result of performing the operations of a repair trace by defining the trace *induced* by τ^+, i.e., the trace $\hat{\tau}$ over $\mathcal{E}$ obtained from τ^+ by: (i) deleting every occurrence of del_e; and (ii) replacing every occurrence of add_e with the event e. For instance, the trace induced by the repair trace $\tau^+ = del_c, add_a, b, b, del_a, add_c$ is the trace $\hat{\tau} = a, b, b, c$, and the original trace is $\tau = c, b, b, a$. When τ^+ is applicable to a trace τ and induces the trace $\hat{\tau}$, we say that τ^+ *transforms* τ into $\hat{\tau}$. Furthermore, we associate the *cost of a repair trace* τ^+, denoted as $cost(\tau^+)$, as the number of add_* and del_* events occurring in τ^+.

Therefore, the *trace alignment* problem can be stated as follows: given a trace τ and an $\text{LTL}_f/\text{LDL}_f$ formula φ such that $\tau \not\models \varphi$, find a repair trace τ^+ of minimal cost that transforms τ into a trace $\hat{\tau}$ such that $\hat{\tau} \models \varphi$. Consequently, when φ is satisfiable, there always exists a solution to the problem as repair events always allow to obtain any log trace from the original one.

Given a log trace $\tau = e_1, \ldots, e_n$, we can associate the *trace automaton* $\mathcal{T} = \langle \mathcal{E}_\tau, Q_\tau, q_0^\tau, \rho_\tau, F_\tau \rangle$, where
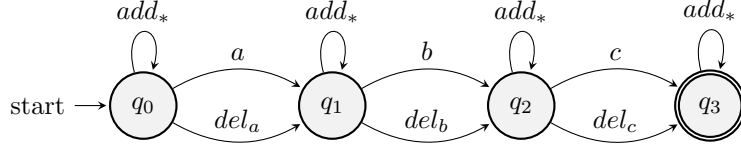
Figure 7.1: Repair automaton of trace $\tau = a, b, c$ over $\mathcal{E} = \{a, b, c, d\}$. The subscript $*$ is an abbreviation for every proposition.

$\mathcal{E}_\tau = \{e_1, \ldots, e_n\}$ is the input alphabet; $Q_\tau = \{q_0^\tau, \ldots, q_n^\tau\}$ is the set of $n+1$ states; q_0^τ is the initial state; $\rho^\tau = \{\langle q_i^\tau, e_{i+1}, q_{i+1}^\tau \rangle \mid 0 \leq i < n\}$ is the transition relation; and $F^\tau = \{q_n^\tau\}$ is the set of final states. It is easy to see that by construction $\mathcal{T}$ is deterministic and accepts only τ, i.e., $\mathcal{L}_\mathcal{T} = \{\tau\}$. Similarly, as we have seen throughout this dissertation, given a temporal formula φ to check τ against, we can obtain a corresponding NFA $\mathcal{A} = \langle \mathcal{E}, Q, q_0, \rho, F \rangle$, which, in this context, we call the *formula automaton* such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A})$.

We can augment $\mathcal{T}$ and $\mathcal{A}$ as follows. From $\mathcal{T}$, we define the *repair automaton* of τ denoted as $\mathcal{T}^+ = \langle \mathcal{E}_\tau^+, Q_\tau, q_0^\tau, \rho_\tau^+, F_\tau \rangle$, where ρ_τ^+ extends ρ_τ with the following fresh transitions: (i) $\langle q, del_p, q' \rangle$, for all $\langle q, p, q' \rangle \in \rho_\tau$; and (ii) $\langle q, add_e, q \rangle$, for all e in $\mathcal{E}$ and $q \in Q_\tau$. $\mathcal{T}^+$ accepts repair traces, denoted as $\mathcal{L}_{\mathcal{T}^+}$. For instance, the repair automaton of the trace $\tau = a, b, c$ over $\mathcal{E} = \{a, b, c, d\}$ is shown in Figure 7.1. By construction, $\mathcal{T}^+$ remains deterministic and accepts exactly all the repair traces τ^+ over $\mathcal{E}^+$ that can be derived from τ. From $\mathcal{A} = \langle \mathcal{E}, Q, q_0, \rho, F \rangle$ we derive the *augmented formula automaton* of $\mathcal{A}$ denoted as $\mathcal{A}^+ = \langle \mathcal{E}^+, Q, q_0, \rho^+, F \rangle$, where $\mathcal{E}^+ = \mathcal{E}_\tau^+$ and ρ^+ contains: (i) one transition $\langle q, e, q' \rangle$ for each transition $\langle q, \psi, q' \rangle \in \rho$ and $e \in \mathcal{E}$ such that $\{e\} \models \psi$; (ii) one transition $\langle q, add_e, q' \rangle$ for each transition $\langle q, \psi, q' \rangle \in \rho$ and $e \in \mathcal{E}$ such that $\{e\} \models \psi$; and (iii) one transition $\langle q, del_e, q \rangle$ for each $q \in Q$ and $e \in \mathcal{E}$. Observe that, from the definition, we have that $\mathcal{L}_\mathcal{A} \subseteq \mathcal{L}_{\mathcal{A}^+}$. Intuitively, $\mathcal{A}^+$ accepts all repair traces τ^+ (including those not applicable to τ) that induce a trace $\hat{\tau}$ satisfying φ. Figure 7.2a shows the automaton for $\varphi_1 = \mathsf{G}(a \to \mathsf{F}b)$, whereas Figure 7.2b shows the corresponding augmented formula automaton.

Theorem 7.1. *Given an* LTL$_f$/LDL$_f$ *formula φ over $\mathcal{E}$, let $\mathcal{A}$ and $\mathcal{A}^+$ be the corresponding formula automaton and augmented formula automaton, respectively. A repair trace τ^+ over $\mathcal{E}^+$ is accepted by $\mathcal{A}^+$ if and only if the trace $\hat{\tau}$ over $\mathcal{E}$ induced by τ^+ is accepted by $\mathcal{A}$.*

Proof. By definition, the induced trace $\hat{\tau}$ can be obtained from τ^+ by (i) deleting every occurrence of del_e, and (ii) replacing every occurrence of add_e with e. We prove that after applying any of these operations, while the trace τ and the computations τ^+ induced on $\mathcal{A}^+$ change, the last state of such computations is preserved. Since this implies preservation of acceptance (or rejection), it follows that $\hat{\tau} \in \mathcal{L}_{\mathcal{A}^+}$ if and only if $\tau^+ \in \mathcal{L}_{\mathcal{A}^+}$. This, together with the fact that $\hat{\tau} \in \mathcal{L}_\mathcal{A}$ if and only if $\hat{\tau} \in \mathcal{L}_{\mathcal{A}^+}$ – this is because $\mathcal{A}$ and $\mathcal{A}^+$ have the same initial, final, and non-final states, and transitions of $\mathcal{A}$ are a subset of those of $\mathcal{A}^+$ – implies
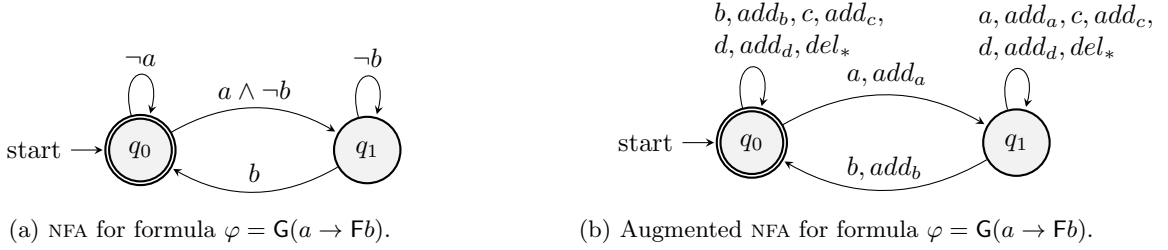
(a) NFA for formula $\varphi = \mathsf{G}(a \to \mathsf{F}b)$.

(b) Augmented NFA for formula $\varphi = \mathsf{G}(a \to \mathsf{F}b)$.

Figure 7.2: Automaton and augmented formula automaton of $\mathsf{G}(a \to \mathsf{F}b)$.

the thesis.

Consider a repair trace $\tau^+ = e_1^+, \cdots, e_m^+$ and let $q^0 \xrightarrow{e_1^+} q^1 \cdots q^{m-1} \xrightarrow{e_m^+} q^m$ (with $q^0 = q_0$) be the generic computation induced by τ^+ on $\mathcal{A}^+$ (recall that, in general, $\mathcal{A}^+$ is nondeterministic). By the definition of ρ^+, every state of $\mathcal{A}^+$ has a self-loop labeled by del_*, thus, for $del_{e_i^+}$ occurring in position $i \in [1, m]$ of τ^+, the generic computation has the following form:

$$q^0 \xrightarrow{e_1^+} q^1 \cdots q^{i-2} \xrightarrow{e_{i-1}^+} q^{i-1} \xrightarrow{del_{e_i^+}} q^i \xrightarrow{e_{i+1}^+} q^{i+1} \cdots q^{m-1} \xrightarrow{e_m^+} q^m, \text{ with } q^i = q^{i-1}.$$

Because $q^i = q^{i-1}$ and, from the computation, $\langle q^i, e_{i+1}^+, q^{i+1}\rangle \in \rho^+$, it follows that $\langle q^{i-1}, e_{i+1}^+, q^{i+1}\rangle \in \rho^+$. Consequently, deleting e_i^+ due to $del_{e_i^+}$ at position i transforms the computation into the following one, preserving the last state q^m: $q^0 \xrightarrow{e_1^+} q^1 \cdots q^{i-2} \xrightarrow{e_{i-1}^+} q^{i-1} \xrightarrow{e_{i+1}^+} q^{i+1} \cdots q^{m-1} \xrightarrow{e_m^+} q^m$.

As for the replacement of add_e with e, since by the definition of ρ^+, $\langle q, add_e, q'\rangle \in \rho$ if and only if $\langle q, e, q'\rangle \in \rho^+$, it follows that the operation does not affect the states of the generic computation, thus preserving the computation's last state. Hence the thesis holds. □

Theorem 7.2. *Consider a log trace τ and an LTL$_f$/LDL$_f$ formula φ, both over $\mathcal{E}$. Let $\mathcal{T}^+$ be the repair automaton of τ and $\mathcal{A}^+$ the augmented formula automaton obtained from a formula automaton $\mathcal{A}$ of φ. A repair trace τ^+ is a solution to the trace alignment problem for τ and φ if and only if τ^+ is a trace with minimal $cost(\tau^+)$ such that $\tau^+ \in \mathcal{L}_{\mathcal{T}^+} \cap \mathcal{L}_{\mathcal{A}^+}$.*

Proof. Knowing that $\mathcal{T}^+$ is deterministic and accepts all repair traces τ^+ over $\mathcal{E}^+$ and that $\mathcal{A}^+$ accepts all repair traces τ^+ inducing traces $\hat{\tau}$ satisfying φ, we obtain that, given τ and φ, trace alignment is equivalent to searching for a repair trace τ^+ (over $\mathcal{E}^+$) that is accepted by both $\mathcal{A}^+$ and $\mathcal{T}^+$, and has minimal cost, i.e., contains a minimal number of repair events. Indeed, acceptance by (*i*) $\mathcal{T}^+$ and (*ii*) $\mathcal{A}^+$ guarantees that we are considering all (and only) the repair traces τ^+ that (*i*) can be obtained from τ, and (*ii*) induce a log trace $\hat{\tau}$ satisfying φ. □

In other words, the search space of our problem is the language $\mathcal{L}_{\mathcal{T}^+} \cap \mathcal{L}_{\mathcal{A}^+}$. In the next section, we

show how the search can actually be performed by resorting to planning technology. Observe that the approach can be easily extended to the case of many $\text{LTL}_f/\text{LDL}_f$ formulas $\varphi_1, \ldots, \varphi_n$ just by either taking the conjunction of such formulas or by computing the augmented formula automaton for each formula, i.e., $\mathcal{A}_1^+, \ldots, \mathcal{A}_n^+$, and then searching for a trace that is accepted by $\mathcal{T}^+$ and all $\mathcal{A}_1^+, \ldots, \mathcal{A}_n^+$.

7.4 Declarative Trace Alignment as Optimal Planning

The idea behind the reduction to *deterministic cost-optimal planning* is to model events from $\mathcal{E}^+$ as actions whose execution triggers state changes in both $\mathcal{T}^+$ and $\mathcal{A}^+$, according to their respective transition function and relation. The problem then consists in finding a deterministic plan that takes both automata from their initial state to a final state at a minimum cost, where actions corresponding to *add* and *del* are assigned unitary cost, while skip actions have no cost. The obtained plan is a representation of the repair trace that solves the problem.

We encode trace alignment into cost-optimal planning as follows. Consider an instance of trace alignment, i.e., a trace τ and an $\text{LTL}_f/\text{LDL}_f$ formula φ, and let $\mathcal{T}^+ = \langle \mathcal{E}_\tau^+, Q_\tau, q_0^\tau, \rho_\tau^+, F_\tau \rangle$ and $\mathcal{A}^+ = \langle \mathcal{E}^+, Q, q_0, \rho^+, F \rangle$ be the corresponding repair and augmented formula automata, respectively. Without loss of generality, we assume disjoint Q_τ and Q. We define the planning domain $\mathcal{D} = \langle \mathcal{S}, A, cost, tr \rangle$ over $\mathcal{F} = Q_\tau \cup Q$ (automata states are used as fluents), where: (i) $\mathcal{S} \subseteq \{\{q_\tau\} \cup Q' \mid q_\tau \in Q_\tau \text{ and } Q' \subseteq Q\}$, i.e., the states of $\mathcal{D}$ are sets containing exactly one state of $\mathcal{T}^+$ and a subset of states of $\mathcal{A}^+$; (ii) $A = \mathcal{E}^+$, i.e., we use events from $\mathcal{E}^+$ as actions; in particular, an event $e \in \mathcal{E}$ models a skip action on e, whereas events add_e and del_e model the addition and deletion of event e, respectively; (iii) $cost(del_e) = cost(add_e) = 1$ and $cost(e) = 0$ for all $e \in \mathcal{E}$; and (iv) for $a \in A$, $q_\tau, q_\tau' \in Q_\tau$, and $Q_1, Q_2 \subseteq Q$, $tr(\{q_\tau\} \cup Q_1, a) = \{q_\tau'\} \cup Q_2$ if and only if:

1. $\langle q_\tau, a, q_\tau' \rangle \in \rho_\tau^+$;

2. there exists a transition $\langle q, a, q' \rangle \in \rho^+$ such that $q \in Q_1$;

3. for all $q \in Q_1$, if there exists a transition $\langle q, a, q' \rangle \in \rho^+$ then $q' \in Q_2$;

4. for all $q' \in Q_2$, there exists a transition $\langle q, a, q' \rangle \in \rho^+$ such that $q \in Q_1$.

$\mathcal{D}$ models the synchronous product of $\mathcal{T}^+$ and $\mathcal{A}^+$, where fluents represent the state(s) that each automaton is in. Since $\mathcal{T}^+$ is deterministic, every state of the planning domain contains exactly one state from Q_τ. Instead, with $\mathcal{A}^+$ possibly being nondeterministic, the domain states include, in general, many states from Q. Requirements 1 and 2 capture executability: action a can be executed only if the corresponding event is accepted by $\mathcal{T}^+$, in its current state, and by $\mathcal{A}^+$, in at least one of its current states. Requirements 3 and 4

define the successor state, which consists of the (unique) successor state of $\mathcal{T}^+$ union *all* the successor states of $\mathcal{A}^+$, with respect to action a and the current states of $\mathcal{T}^+$ and $\mathcal{A}^+$. Although the transition function tr of domain $\mathcal{D}$ is deterministic, the domain accounts for the nondeterministic transitions of $\mathcal{A}^+$. Moreover, since $A = \mathcal{E}^+$, every plan for $\mathcal{D}$ is also a repair trace.

The cost-optimal planning problem is defined as $\Gamma = \langle \mathcal{D}, s_0, G \rangle$, where (i) $s_0 = \{q_0, q_0^\tau\}$; and (ii) $G = (\bigvee_{q \in F} q) \wedge q_f^\tau$, with q_f^τ the (unique) final state of $\mathcal{T}^+$. In other words, s_0 models that $\mathcal{T}^+$ and $\mathcal{A}^+$ start in their initial state, and G models that $\mathcal{T}^+$ is in its final state and that at least one computation of $\mathcal{A}^+$, which is nondeterministic, ends up in a final state. We call Γ the *trace-alignment planning problem*. A solution to this problem is a minimal-cost plan inducing a domain trace that ends in a state satisfying G.

We can show the correspondence between optimal solutions of Γ and solutions to instances of trace alignment over which Γ is defined.

Theorem 7.3. *Consider a log trace τ and an* LTL$_f$/LDL$_f$ *formula φ, both over $\mathcal{E}$. If $\Gamma = \langle \mathcal{D}, s_0, G \rangle$ is the trace-alignment planning problem of τ with respect to φ, then a plan π is an optimal solution to $\mathcal{P}$ if and only if π is a solution to the trace-alignment problem instance defined by τ and φ.*

Proof. By construction, we have that: (i) if π is a plan that solves Γ then it is a repair trace accepted by both $\mathcal{T}^+$ and $\mathcal{A}^+$; (ii) if, vice versa, τ is a repair trace accepted by both $\mathcal{T}^+$ and $\mathcal{A}^+$, because the transition function of $\mathcal{D}$ accounts for all the transitions of $\mathcal{T}^+$ and $\mathcal{A}^+$ then π is also a solution to Γ. Since the cost of a plan π for Γ corresponds to that of the corresponding repair trace, by Theorem 7.2 the thesis holds. □

Hence, we can resort to planning technology to solve the trace alignment problem. Recall that, in general, we deal with many formulas instead of just one.

7.5 Encodings in PDDL

In the previous section, we have shown how trace alignment can be reduced to cost-optimal planning and building, in this way, the theoretical bases for a solution technique that exploits planning technology. In general, for a given planning problem, there exist many possible equivalent formulations, each having a different, possibly dramatic, impact on solution performance. Here, we present two classes of concrete PDDL encoding variants, along with some possible optimizations, which will be used in the experimental evaluation. Given an instance of trace alignment consisting of a set of *augmented formula automata* $\mathcal{A}_1^+, \ldots, \mathcal{A}_n^+$, corresponding to n LTL$_f$/LDL$_f$ formulas $\varphi_1, \ldots, \varphi_n$, and a *repair automaton* $\mathcal{T}^+$ obtained from a log trace τ, we show how to encode the problem in PDDL.

The first class of encodings includes a high-level general encoding with three other variants: one where conjunctive formulas are used to encode goal states, one where the number of fluents used to model the automata states is reduced, and one which combines the two. The encodings from this class have the advantage of increased readability and understandability at the expense of a slight performance loss, as shown by the experiments.

The second class of encodings is more low-level and, roughly speaking, can be understood as including "ground" versions of the variants of the first class. In this case, we propose two encodings: a basic one and one with conjunctive goals. The advantages of these encodings are dual with respect to those of the first class; namely, we obtain better performance at the expense of readability.

A further difference between the two classes is the fact that while the first class makes no assumptions on the input augmented formula automata and can deal directly with NFAs, the second class assumes they are DFAs. This difference, however, turns out to be only theoretical and irrelevant in practice since all available state-of-the-art tools for automata construction output a DFA. Nonetheless, the encoding for the NFAs is potentially useful anyway, as, for instance, one may decide to express the specifications $\varphi_1, \ldots, \varphi_n$ directly as NFAs or without taking advantage of the available tools, or in case efficient tools will be developed in the future, which output an NFA.

7.5.1 General Encoding

Planning Domain. We provide two abstract types: `event` and `state`. The former captures the events involved in a transition between two different states of a formula/repair automaton, while the latter is used to identify the states of each formula automaton uniquely (through sub-type `automaton_state`) and of the repair trace automaton (through sub-type `trace_state`). To capture the structure of the automata as well as their evolution stemming from actions executions, we include in $\mathcal{D}$ four (Boolean) *domain predicates*: (*i*) (`trace ?t1 - repair_state ?e - event ?t2 - repair_state`), meaning that (`trace t1 e t2`) holds if the repair automaton has a transition from state `t1` to `t2` under event `e`; (*ii*) (`automaton ?s1 - automaton_state ?e - event ?s2 - automaton_state`), meaning that (`automaton s1 e s2`) holds if there exists a formula automaton with a transition from state `s1` to `s2`, under event `e`; (*iii*) (`cur_state ?s - state`), with (`cur_state s`) holding if `s` is the current state of a formula/repair automaton; and *(iv)* (`final_state ?s - state`) with (`final_state s`) holding if `s` is a final accepting state of a formula/repair automaton.

Predicates `trace`, `automaton` and `final_state` capture the structure of the repair and the formula automata, as well as their final states. The values of these predicates are fixed when the planning problem is instantiated and do not change when actions are executed. Predicate `cur_state` accounts for the current states of both

```
                                        (:action add
                                         :parameters (?e - event)
                                         :effect
                                            (and (increase
                                              (total-cost) 1)
            (:action sync                 (forall
             :parameters                    (?s1 ?s2 - automaton_state)
                (?t1 - trace_state          (when
                 ?e - event                   (and (cur_state ?s1)
                 ?t2 - trace_state)           (automaton ?s1 ?e ?s2))
             :precondition                   (and (not (cur_state ?s1))
                (and                         (cur_state ?s2))))))
                  (cur_state ?t1)
                  (trace ?t1 ?e ?t2))
             :effect                       (:action del
                (and                        :parameters
                  (not (cur_state ?t1))        (?t1 - trace_state
                  (cur_state ?t2)             ?e - event
                (forall                       ?t2 - trace_state)
                  (?s1 ?s2 - automaton_state)  :precondition
                  (when                         (and
                    (and (cur_state ?s1)          (cur_state ?t1)
                    (automaton ?s1 ?e ?s2))       (trace ?t1 ?e ?t2))
                    (and (not (cur_state ?s1))  :effect (and (increase
                    (cur_state ?s2))))))          (total-cost) 1)
                                              (not (cur_state ?t1))
                                              (cur_state ?t2)))
```

Figure 7.3: PDDL `sync`, `add` and `del` actions.

the formula and the repair automata. Thus, essentially, only such current states contribute to defining the state space of the domain. Then, to model action costs, we introduce the *numeric fluent* `total-cost`, whose value is increased every time an action is executed by the cost associated with that action.

Planning actions model *alignments* on the input trace τ, and trigger transitions on the repair and formula automata. As usual, actions are characterized by *preconditions* and *effects*. In this encoding, we have three actions: `sync`, to *synchronously advance* the repair and the formula automata on an event originally occurring in the input trace, and `add` and `del`, to add and delete events to/from the trace, and consequently trigger a transition on the repair automaton and, as well as, when needed, on the formula automata (Figure 7.3). We model `sync` and `del` in such a way that they can be applied only if there exists a transition from the current state `t1` of the repair automaton to a subsequent state `t2`, with `e` the event involved in the transition. Unlike the reduction presented in Section 7.4, states occur as action parameters to avoid existential quantifiers in preconditions and effects. Action `del` yields a *single* move in the repair automaton only, as, on additions, the formula automata loop in their current states, thus, there is no need to account for these transitions. Action `sync` yields a transition in the repair automaton as well as one per formula automaton (all to be performed synchronously), all corresponding to the advancement of the respective automaton on event `e`. Finally, action `add` triggers a synchronous transition on the formula automata, depending on their current states and the event `e`, while leaves the repair automaton in its current state as, again, this loops in the case

111

```
(define (problem prob-trace)
(:domain alignment)
(:objects
    t1 t2 t3 t4 - trace_state S10 s11 s12 S20 s21 s22 - automaton_state act1 act2 act3 -
    event
)
(:init
    (= (total-cost) 0)
    ; repair automaton
    (cur_state t1) (trace t1 act1 t2) (trace t2 act2 t3) (trace t3 act3 t4) (final_state t4)

    ; formula automata
    ; DFA for F(act1) -> F(act2)
    (cur_state s10) (final_state s10) (final_state s11) (automaton s10 act1 s12) (automaton
    s10 act2 s11) (automaton s12 act2 s11)

    ; NFA for F(act1 and ((XG act1) or (XG act2)))
    (cur_state s20) (final_state s21) (final_state s22) (automaton s20 act1 s21) (automaton
    s20 act1 s22) (automaton s21 act2 s20) (automaton s22 act1 s20) (automaton s22 act2 s20)
)
(:goal (and
    (or (cur_state s10) (cur_state s11)) (or (cur_state s21) (cur_state s22)) (cur_state t4)
    )
)
(:metric minimize (total-cost))
)
```

Figure 7.4: Example of a PDDL problem instance. Transitions are specified by tuples of the form (x t e t'), with x replaced by trace for repair automaton transitions and by automaton for formula automata transitions. The single fluent automaton, which encodes transitions of many formula automata, does not introduce ambiguities, as the automata have disjoint sets of states. The first block of transitions refers to the repair automaton, while the other ones to the formula automata. The first formula automaton is deterministic and corresponds to the LTL$_f$ formula F(act1) $\rightarrow$ F(act2). The second formula automaton is nondeterministic and captures the LTL$_f$ formula F(act1 $\land$ ((XG(act1)) $\lor$ (XG(act2)))).

of an action and there is no need to account for this.

Planning Problem. First, we define the objects needed to model the states of the repair automaton (type trace_state), the states of the formula automata (automaton_state), and the involved events (event). Second, the initial state sets the total cost to zero and defines the exact structure of the repair automaton and of every formula automaton. Specifically, this includes the specification of all transitions connecting two distinct states for every automaton. The current state and accepting states are identified as well. Then, the goal condition is specified as a conjunction of (i) one formula expressing the fact that the repair automaton is in its (unique) final state; and (ii) one disjunctive formula for each formula automaton, expressing the fact that the automaton is in at least one of its final states. Finally, we include the specification (:metric minimize (total-cost)), to require that the returned plan yields minimal overall cost. An example of a problem instance is shown in Figure 7.4.

Conjunctive Goals

The presence of *disjunctive* goals is a drawback of the general encoding. Not all planners support such forms of goals, which typically yields a decrease in performance (Howe and Dahlman, 2002), as confirmed in our experiments later. Nevertheless, we can easily rewrite the goal condition as a simple conjunction by suitably manipulating the involved formula automata when they contain multiple final states, which is the situation where disjunctive goals arise. A similar technique has been already exploited in the literature, e.g., (Gazen and Knoblock, 1997). A straightforward way to rewrite the goal condition requires to (*i*) add a new dummy state with no outgoing transitions; (*ii*) add a new special action in the domain definition (Figure 7.5), executable only in the final states of the original automaton, which makes the automaton move to the dummy state; and (*iii*) include only the dummy state in the set of final states.

In detail, we introduce an auxiliary fluent (`dummy_trans ?s1 - automaton_state ?de - dummy_act ?s2 - automaton_state`), to represent the transitions from the multiple final states of the original formula automata to the new final dummy state, introduced for every formula automaton. Then, we use `dummy_trans` in the effect condition of the auxiliary action `goto-goal`, allowing each formula automaton with multiple final states to move to the dummy final state whenever at least one of its original final states is reached.

```
(:action goto-goal
    :parameters (?t1 - trace_state ?de - dummy_act)
    :precondition (and (cur_state ?t1) (final_state ?t1))
    :effect
          (forall (?s1 ?s2 - automaton_state)
              (when (and (cur_state ?s1) (dummy_trans ?s1 ?de ?s2))
                  (and (not (cur_state ?s1)) (cur_state ?s2)))
          )
)
```

Figure 7.5: Fragment of the additional `goto-goal` PDDL action.

In the problem definition, we add new objects to represent the dummy states and the single `dummy` event. Additionally, we provide dummy transitions (i.e., `dummy_trans`) only for those formula automata with multiple final states. With these changes, we easily obtain simple conjunctive goals only.

Shared States

Another drawback that we address is reducing the number of objects needed to represent the automata states. These, again, may yield an additional overhead in a planner's grounding phase as the number of automata increases. Recall that, in general, the NFA obtained from an LTL$_f$/LDL$_f$ formula contains up to an exponential number of states, each represented, in the general encoding, with a distinct object. Although most automata are often reasonably small, sharing their states to optimize the grounding phase can have a significant impact

```
(:action sync
    :parameters (?t1 - trace_state ?e - activity ?t2 - trace_state)
    :precondition (and (cur_state_trace ?t1) (trace_trans ?t1 ?e ?t2))
    :effect (and (not (cur_state_trace ?t1)) (cur_state_trace ?t2)
        (forall (?a - automaton ?s1 ?s2 - automaton_state)
        (when (and (cur_state ?a ?s1) (automaton_trans ?a ?s1 ?e ?s2))
            (and (not (cur_state ?a ?s1)) (cur_state ?a ?s2)))
        )
    )
)


(define (problem ...)
(:objects
    a1 a2 - automaton t1 t2 t3 t4 - trace_state s1 s2 s3 - automaton_state act1 act2 act3 -
    event
)
(:init (cur_state_trace t0) ...

    ; DFA for F(act1) -> F(act2)
    (cur_state a1 s1) (final_state a1 s1) (final_state a1 s2) (automaton a1 s1 act1 s2) (
    automaton a1 s1 act2 s3) (automaton a1 s2 act2 s3)

    (...)
)
(:goal (and (or (cur_state a1 s1) (cur_state a1 s2)) ...)) ...)
```

Figure 7.6: PDDL `sync` action in domain with shared states and fragment of a PDDL encoding instance with shared states.

on the approach's scalability. To share automata states, we slightly modify the domain and the problem definition. In the domain, predicates `cur_state`, `final_state` and `automaton` are parameterized with respect to the automaton and, consequently, the effects of actions `sync` and `add` are universally quantified on the newly introduced parameter. In addition, we add the new predicate `cur_state_trace` to distinguish between states of the repair automaton, not requiring quantification, and states of formula automata. Figure 7.6 reports how the `sync` action accounts for these changes.

As for the problem, we introduce one object per formula automaton, N objects for the automaton states, with N the maximum number of states among all the formula automata. Then, in the initial state, we associate the automata transitions with each automaton object previously defined. Finally, the goal condition is as in the general encoding except that predicates are also quantified over the specific formula automata. Figure 7.6 shows how the previous example is modified according to these modifications.

Conjunctive Goals & Shared States

The above-mentioned encoding optimizations are very effective, as we will see later, in increasing the scalability of the general encoding. Interestingly, they are not mutually exclusive and can be combined to produce a further variant. As the experiment section shows, such a combination results in one of the best-performing and most-scalable variants among those proposed here.

Clearly, in the encodings discussed so far, the domain is independent of the problem instance. That is, once written, the PDDL domain specification remains unchanged, and only the PDDL problem specification needs to be modified in order to solve different instances of trace alignment. This aspect is crucial in reducing the compilation time to transform a trace alignment instance into cost-optimal planning since the domain does not need to be generated every time a new problem instance is to be solved. In the next section, we adopt a different approach where this property does not hold anymore and a new domain must be generated for every different instance.

7.5.2 STRIPS-like Encoding

This encoding stems from the encoding employed in (De Giacomo et al., 2017) for their experiments. Theoretically, the encoding does not change, but in this dissertation, we provide a new implementation optimized and built from scratch.

The idea behind the STRIPS encoding development is to reduce the planner's grounding time so as to positively impact scalability as the number of PDDL objects increases. The adopted encoding is provided in a STRIPS-like language with negative preconditions and essentially corresponds to a grounded version of the general encoding presented before. As such, the PDDL domain is not independent of the problem instance, thus requiring one separate domain/problem pair for each considered log trace. Experiments show that this approach features the best scalability.

Disjunctive Goals

Planning Domain. To write a grounded version of the general encoding presented in Section 7.5.1, we need to simulate every possible move of the automaton constraints and of the repair trace automaton. The only PDDL predicate needed to model the planning domain is the `cur_state` predicate, which operates as in previous encodings.

Given the set of constraint automata, we start by grouping together transitions labeled with a certain log event linking any pair of distinct states. We do that to build *sync* and *add* actions that simultaneously execute a prescribed transition among all automata containing such a transition. We do not consider loop transitions because they do not contribute to changing the predicate `cur_state`, as is the case of all other encodings. Once we have all relevant transitions grouped by log event, we want to model every possible combination of them so that every possible move is allowed. Specifically, given M to be the number of distinct automata involved in at least a transition of a certain log event, we find all possible simple combinations of transitions of length k with $1 \leq k \leq M$. Among the combinations found, we can ignore all combinations in which the

same automaton is associated with two or more transitions present in the combination. We do so since we know that, at any given time instant, an automaton can execute one and only one transition. The following is the running example for the STRIPS-like encoding.

Example 7.4. *Suppose we have only two constraint models:* $\mathsf{F}a$ *and* $\mathsf{G}(a \leftrightarrow \mathsf{X}b) \wedge \neg b$. *The automata corresponding to these two formulas are as shown in Figure 7.7.*

(a) DFA for $\mathsf{F}(a)$ (b) DFA for $\mathsf{G}(a \leftrightarrow \mathsf{X}b) \wedge \neg b$
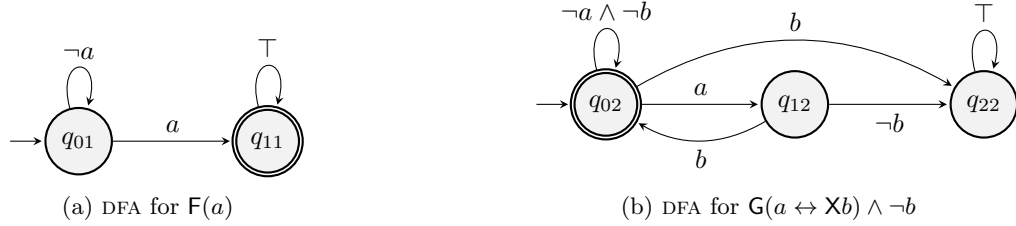
Figure 7.7: Automata for $\mathsf{F}(a)$ and $\mathsf{G}(a \leftrightarrow \mathsf{X}b) \wedge \neg b$.

In this case, we have just two automata constraints, i.e., $M = 2$. Transitions we are interested in, grouped by labels, are $a = \{q_{01} \rightsquigarrow q_{11}, q_{02} \rightsquigarrow q_{12}, q_{12} \rightsquigarrow q_{22}\}$ and $b = \{q_{02} \rightsquigarrow q_{22}, q_{12} \rightsquigarrow q_{02}\}$. Note that the transition $q_{12} \rightsquigarrow q_{22}$ appears with label a because it is part of b's complement, given the actual label of the transition (i.e., $\neg b$). At this point, to capture all possible automata moves, we just need to enumerate all possible simple combinations of transitions. In particular, Table 7.1 summarizes the combinations we can get for each label.

#	Label	k	Involved Transitions	Excluded Transitions
ct_1			$\{q_{01} \rightsquigarrow q_{11}\}$	$\{q_{02} \rightsquigarrow q_{12}, q_{12} \rightsquigarrow q_{22}\}$
ct_2		1	$\{q_{02} \rightsquigarrow q_{12}\}$	$\{q_{01} \rightsquigarrow q_{11}, \cancel{q_{12} \rightsquigarrow q_{22}}\}$
ct_3			$\{q_{12} \rightsquigarrow q_{22}\}$	$\{q_{01} \rightsquigarrow q_{11}, \cancel{q_{02} \rightsquigarrow q_{12}}\}$
ct_4	a		$\{q_{01} \rightsquigarrow q_{11}, q_{02} \rightsquigarrow q_{12}\}$	$\{\cancel{q_{12} \rightsquigarrow q_{22}}\}$
ct_5		2	$\{q_{01} \rightsquigarrow q_{11}, q_{12} \rightsquigarrow q_{22}\}$	$\{\cancel{q_{02} \rightsquigarrow q_{12}}\}$
ct_6			$\{\cancel{q_{02} \rightsquigarrow q_{12}}, \cancel{q_{12} \rightsquigarrow q_{22}}\}$	$\{q_{01} \rightsquigarrow q_{11}\}$
ct_7		1	$\{q_{02} \rightsquigarrow q_{22}\}$	$\{\cancel{q_{12} \rightsquigarrow q_{02}}\}$
ct_8	b		$\{q_{12} \rightsquigarrow q_{02}\}$	$\{\cancel{q_{02} \rightsquigarrow q_{22}}\}$
ct_9		2	$\{\cancel{q_{02} \rightsquigarrow q_{22}}, \cancel{q_{12} \rightsquigarrow q_{02}}\}$	$-$

Table 7.1: Enumeration of all simple k combinations of constraint automata transitions for each label appearing in constraint automata between distinct states. Involved transitions are the ones that we consider as a possible automaton move. Excluded transitions are the ones that do not take part in automata moves. Canceled transitions are not informative and can be discarded as they belong to an automaton for which another transition has already been selected for execution. Thus, they would not occur by definition.

The remaining combinations allow us to build the necessary PDDL `sync` and `add` actions. In particular, every combination will correspond to an `add` action. Preconditions include source states (expressed with `cur_state`) of transitions belonging to the combination and the negation of all other source states of transitions that do not belong to the specific combination but that have the same log activity label. Likewise, action

effects will negate the source states of transitions within the combination and add destination states of the same set of transitions. Here, we include the unitary action cost for the `add` action as for all other encodings.

Example 7.5. *Recalling the previous example, the* `add` *action for the combination* ct_1 *will be as in Figure 7.8.*

```
(:action add-a-ct1
 :parameters ()
 :precondition (and (cur_state q01) (not (cur_state q02)) (not (cur_state q12)))
 :effect (and (not (cur_state q01)) (cur_state q11) (increase (total-cost) 1))
)
```

Figure 7.8: PDDL `add` action for combination ct_1.

Analogously, we can build a corresponding PDDL `sync` action to cover the possibility of advancing both the trace automaton and the constraint automata, assuming that the trace automaton already has a transition with the same label. In such a case, the `sync` action would have the same components as the `add` action, but differently from the `add` action, the `sync` action would also include preconditions and effects associated to the specific trace automaton transition.

Example 7.6. *Assuming there exists a transition in the trace automaton from state* t_2 *to state* t_3, *then the* PDDL `sync` *action corresponding to the* ct_1 *combination will be as in Figure 7.9.*

```
(:action sync-a-ct1
 :parameters ()
 :precondition (and (cur_state q01) (not (cur_state q02)) (not (cur_state q12)) (cur_state
    t2))
 :effect (and (not (cur_state q01)) (not (cur_state t2)) (cur_state q11) (cur_state t3))
)
```

Figure 7.9: PDDL `sync` action for combination ct_1.

Moreover, for each label associated with at least one transition of the trace automaton, we need to give it the possibility to execute synchronous moves for any other state of constraint automata that does not appear in the combination of transitions above. In other words, for every label in the trace automaton, we need to include a PDDL `sync` action to be executed every time the `cur_state` does not hold constraint automata states from which a transition with that particular label is available. Most of the time, these additional `sync` actions take into account automata loops.

Example 7.7. *From previous examples, all constraint automata transitions associated with the label a are* $\{q_{01} \rightsquigarrow q_{11}, q_{02} \rightsquigarrow q_{12}, q_{12} \rightsquigarrow q_{22}\}$. *If we assume a trace automaton transition with label a as* $t_2 \rightsquigarrow t_3$, *then the additional* `sync` *action we would have is as shown in Figure 7.10.*

117

```
(:action sync-a-t2t3
 :parameters ()
 :precondition (and (cur_state t2) (not (cur_state q01)) (not (cur_state q02)) (not (
    cur_state q12)))
 :effect (and (not (cur_state t2)) (cur_state t3))
)
```

Figure 7.10: PDDL `sync` action for transition $t_2 \rightsquigarrow t_3$.

Here, note that, in action preconditions, the negation of the source states for which a transition in constraint automata exists is critical to get the action executed correctly.

Additionally, there could be symbols appearing in the trace automaton that do not appear in any constraint automata. To handle synchronous moves of that kind of symbols, we also generate a simple `sync` action to simulate the trace transition execution.

Example 7.8. *Recalling from the previous example, assuming that in the trace automaton the transition from t_3 to t_4 the symbol is c, which does not appear in any of the two constraint automata of our running example, the additional `sync` action would be like the one shown in Figure 7.11.*

```
(:action sync-c-t3t4
 :parameters ()
 :precondition (and (cur_state t3))
 :effect (and (not (cur_state t3)) (cur_state t4))
)
```

Figure 7.11: PDDL `sync` action for transition $t_3 \rightsquigarrow t_4$.

Finally, we need a way to handle `del` actions. In particular, we have as many `del` actions as transitions in the trace automaton, and their PDDL form is exactly the same of `sync` actions in Example 7.8, except that this time `del` actions include a unitary cost in their effects.

Planning Problem. First, we define all trace automaton states as well as all constraint automata states. After that, the *initial state* captures all initial states of both the trace automaton and constraint automata. This is done using the PDDL predicate `cur_state`. Also, in the initial state, the metric for total cost is initialized to zero.

The goal condition includes all automata accepting states. Specifically, the goal is the conjunction of the final state of the trace automaton with all accepting states of constraint automata. Note that if a constraint automaton has more than one accepting state, then a disjunction in the goal condition is required following the DFA acceptance condition definition. In other words, in general, the goal condition might be a conjunction of inner disjunctions. An example of a problem instance is reported in Figure 7.12.

```
(define (problem ...)
(:domain alignment)
(:objects
    t1 t2 t3 t4 - state q0q q11 q02 q12 q22 - state
)
(:init
    (= (total-cost) 0)
    ; repair and formula automata initial states
    (cur_state t1) (cur_state q01) (cur_state q02)
)
(:goal (and (cur_state q11) (cur_state q02) (cur_state t4))
)
(:metric minimize (total-cost))
)
```

Figure 7.12: Fragment of a PDDL STRIPS-like encoding instance.

Conjunctive Goals

In this section, we address the problem previously raised in Section 7.5.1 about dealing with possible multiple final accepting states of constraint automata that translate into disjunction in the goal condition.

Although the approach to avoid disjunctions is built upon the same idea of connecting accepting states to a new dummy state (cf. Section 7.5.1), the actual implementation of such an approach is different. Specifically, while changes in the planning problem are minimal (i.e., just add the necessary dummy states to the objects definition and add them in conjunction with other accepting states in the goal condition), significant changes are required in the corresponding planning domain. Indeed, in the planning domain, we need to model all possible ways of reaching these new dummy states starting from actual accepting states of constraint automata. To do so, we generate as many PDDL actions as all possible combinations of accepting states we may have based on the number of involved constraint automata. Given the number of constraint automata and fixing the single-accepting states (i.e., fixing those states of automata with a single accepting state), we compute all possible combinations among those states that are multiple-accepting states. Once we get all possible such combinations, we generate a PDDL action goto-goal for each combination, allowing the reaching of new dummy single-accepting states.

Example 7.9. *Consider the case where the constraint automata are $\mathsf{F}(a)$ and $\mathsf{G}(a \rightarrow \neg\mathsf{F}(b))$ with their deterministic automata (Figure 7.13).*

In this case, after fixing the state q_{11}, which is the only final state for $\mathsf{F}(a)$, the possible combinations are: $\{q_{11}, q_{02}\}$ and $\{q_{11}, q_{12}\}$. In other words, combinations will tell us from what set of states our automata are accepting. Assuming the final state of the trace automaton is t_{30}, the PDDL actions generated in this case will be as shown in Figure 7.14.
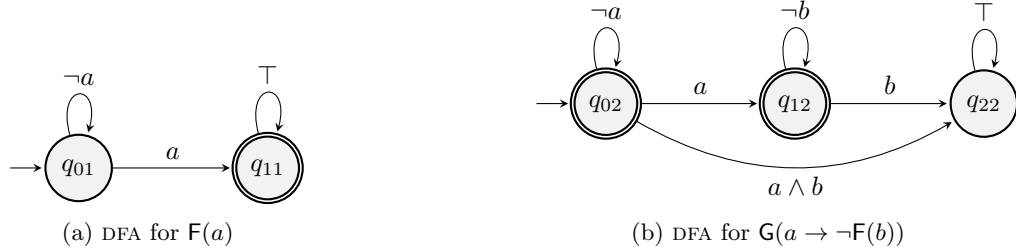
(a) DFA for $\mathsf{F}(a)$

(b) DFA for $\mathsf{G}(a \rightarrow \neg\mathsf{F}(b))$

Figure 7.13: Automata for $\mathsf{F}(a)$ and $\mathsf{G}(a \rightarrow \neg\mathsf{F}(b))$.

```
(:action goto-goal-cs1
 :precondition (and (cur_state t30) (cur_state q11) (cur_state q02))
 :effect (and (cur_state q2_dummy) (not (cur_state q02))))

(:action goto-goal-cs2
 :precondition (and (cur_state t30) (cur_state q11) (cur_state q12))
 :effect (and (cur_state q2_dummy) (not (cur_state q12))))
```

Figure 7.14: PDDL `goto-goal` for combinations of accepting states.

7.6 Experimental Evaluation

We devised a planning-based alignment tool implementing all PDDL encodings presented in Section 7.5 in a standard Java tool called TraceAligner[3], which can be run through a Command-line Interface. TraceAligner takes as input a process log, collected in either the XML or the XES (eXtensible Event Stream) format[4], and a set of LTL$_f$/LDL$_f$ declarative model formulas on log activities, and produces a set of cost-optimal planning instances, expressed in PDDL. After that, each planning task can be fed into a State-Of-The-Art (SOTA) optimal planner.

For the experiments, we use Lydia (De Giacomo and Favorito, 2021) to translate LTL$_f$/LDL$_f$ formulas into their corresponding DFAs, and FastDownward (Helmert, 2006) (FD for short) and SymBA*-2 (Torralba et al., 2014) as representative SOTA planners, which are sound, complete and optimal. We decided to employ FD and SymBA*-2 as optimal planners since, nowadays, almost all newly developed optimal planners for recent planning competitions are built upon them. Furthermore, we combined FD with all encodings described in Sections 7.5.1 and 7.5.2, thus obtaining the following tools: FD-Gen, FD-GenConj, FD-GenShare, FD-GenConjShare, and FD-Strips. We tested such tools with both the *blind* and the *max* heuristics (Bonet and Geffner, 2001). Instead, since SymBA*-2 does not support conditional effects and universal/existential quantifiers, we only combined it with the STRIPS-like encoding (Section 7.5.2), obtaining the SymBA*-2-Strips tool.

---

[3]TraceAligner is available online at `https://github.com/whitemech/TraceAligner`.
[4]`https://www.xes-standard.org/openxes/start`

Baselines. We evaluated every encoding implemented in our tool, combined with FD and SymBA*-2, against the existing ad-hoc trace-alignment solutions presented in (de Leoni et al., 2012) and in (De Giacomo et al., 2016), comparing time and scalability performance. Since the solutions proposed in (de Leoni et al., 2012) and in (De Giacomo et al., 2016) do not offer support for full-fledged $\text{LTL}_f/\text{LDL}_f$, we tested the approach using the translation of proper DECLARE models to LTL_f.

Experiment Types. We ran two types of experiments:

1. *Synthetic Logs.* In this experiment, we measured the scalability of the different planning-based compilations with respect to the *size* of the set of constraints (i.e., the model) and the *noise* in the log traces. In particular, we tested the approach on synthetic logs of different complexity. This experiment is an extended version of the one presented in (De Giacomo et al., 2017).

2. *Real Log.* In this experiment, we evaluated our tool on a real-life log pertaining to a process for handling loan requests using, as a model, a set of LTL_f constraints that should be satisfied by the process, according to domain experts. The aim of this experiment is to show that our tool can be successfully employed in an industrial context.

Experiment Setup. We ran the experiments on a machine equipped with a 12-core Intel Core i7-8700 processor running at 3.20 GHz with 32GB of memory and a timeout of 3600 seconds for each log. For convenience, although the time limit is per log, we report the average result per trace, namely, the sum of the average compilation time and average planning time per trace. We do that because, in the BPM community, aligning single traces or a small set of traces is usually more common than aligning the entire log at once. We used a standard cost function with unitary costs for any alignment step that adds/removes activities in/from the input trace and no cost for synchronous moves. The correctness of TraceAligner was also empirically verified by comparing the alignment cost among all the instances of all implemented encodings. No inconsistencies were found in all solved instances.

Results. For the experiment with synthetic logs, the logs were generated using the log generator presented in (Di Ciccio et al., 2015). We defined 3 DECLARE models with the same activity alphabet and containing 10, 15, and 20 LTL_f constraints, respectively. Then, to create noise in the logs, i.e., behaviors non-compliant with the original DECLARE models, we changed some of the constraints in these models and generated logs from them. In particular, we modified the original DECLARE models by replacing 3, 4, and 6 constraints in each model with their negative counterparts. Each modified model was used to generate 4 logs of 100 traces containing traces of different lengths, i.e., from 1 to 50 events, from 51 to 100 events, from 101 to 150 events,

10 Constraints	FD-Gen		FD-GenConj		FD-GenShare		FD-GenConjShare		FD-Strips		SymBA-Strips	de Leoni et al.	Alignment Cost
	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}			
log traces length	*3 constraints modified*												
1-50	0.47	0.08	0.08	0.03	0.43	0.07	0.07	**0.02**	0.03	0.13	0.22	0.34	1.77
51-100	2.46	0.3	0.33	0.11	1.94	0.25	0.29	0.08	**0.05**	0.3	0.48	1.37	2.11
101-150	11.72	0.93	1.38	0.34	7.81	0.7	1.13	0.25	**0.1**	0.68	1.09	5.9	3.03
151-200	23.78	1.7	2.57	0.66	13.83	1.19	1.99	0.48	**0.15**	1.16	2.4	12.98	3.79
log traces length	*4 constraints modified*												
1-50	1.81	0.19	0.26	0.05	1.62	0.17	0.24	**0.04**	**0.04**	0.23	0.22	-	2.74
51-100	16.65	1.18	2.12	0.36	12.83	0.97	1.85	0.28	**0.11**	1.04	0.45	-	5.86
101-150	-	3.25	5.82	1.11	33.2	2.43	4.82	0.83	**0.22**	2.63	0.99	-	9.68
151-200	-	5.8	10	2.14	-	4.03	7.86	1.54	**0.33**	4.62	1.9	-	13.4
log traces length	*6 constraints modified*												
1-50	3.82	0.3	0.54	0.08	3.46	0.27	0.51	0.07	**0.05**	0.33	0.21	-	4.23
51-100	34.61	2.08	4.44	0.61	26.9	1.72	3.91	0.47	**0.16**	1.8	0.43	-	9.73
101-150	-	4.98	9.81	1.64	-	3.73	8.18	1.21	**0.3**	4.25	0.93	-	16.23
151-200	-	8.36	15.12	3	-	5.79	11.9	2.14	**0.45**	6.96	1.94	-	21.63

Table 7.2: Experimental results for the *synthetic* case study with 10 constraints. The time (in *seconds*) is the total time (average compilation time per trace + average planning time per trace). The timeout is set to 3600 seconds.

15 Constraints	FD-Gen		FD-GenConj		FD-GenShare		FD-GenConjShare		FD-Strips		SymBA-Strips	de Leoni et al.	Alignment Cost
	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}			
log traces length	*3 constraints modified*												
1-50	6.09	0.63	0.43	0.12	5.74	0.6	0.4	0.1	**0.09**	0.13	0.34	1.08	1.71
51-100	-	2.98	4.42	0.63	-	2.71	3.9	0.52	**0.18**	0.3	0.68	6.64	2.23
101-150	-	8.35	15.08	2.05	-	7.32	12.84	1.65	**0.37**	0.68	1.42	24.05	3.07
151-200	-	20.46	-	5.5	-	17.37	-	4.38	**0.73**	1.16	3.14	-	4.2
log traces length	*4 constraints modified*												
1-50	-	4.21	7.56	0.67	-	3.97	7.19	0.59	**0.21**	0.23	0.35	-	3.8
51-100	-	16.8	-	3.2	-	15.3	33.7	2.64	**0.57**	1.04	0.63	-	5.94
101-150	-	-	-	9.9	-	-	-	8.02	**1.28**	2.63	1.34	-	9.49
151-200	-	-	-	18.22	-	-	-	14.25	**1.93**	4.62	2.38	-	12.3
log traces length	*6 constraints modified*												
1-50	-	11.37	30.91	1.68	-	10.75	29.56	1.45	0.44	**0.33**	0.36	-	6.24
51-100	-	32.82	-	5.89	-	29.92	-	4.73	0.99	1.8	**0.65**	-	9.51
101-150	-	-	-	14.49	-	-	-	11.4	1.88	4.25	**1.19**	-	14.44
151-200	-	-	-	-	-	-	-	23.2	3.09	6.96	**2.33**	-	20.61

Table 7.3: Experimental results for the *synthetic* case study with 15 constraints. The time (in *seconds*) is the total time (average compilation time per trace + average planning time per trace). The timeout is set to 3600 seconds.

and from 151 to 200 events, respectively. The experimental results for the experiment with synthetic logs are summarized in Tables 7.2, 7.3, and 7.4. The results for the experiment with the real-life log are, instead, reported in Table 7.5.

In general, in the experiment with synthetic logs, the ad-hoc approach of de Leoni et al. (2012) performs well only for short traces with the smallest amount of noise. For all other cases, i.e., for logs containing traces with more than 50 events or generated by altering more than 3 constraints in the original model, the FD-Strips tool associated with the encoding presented in Section 7.5.2, using the *blind* heuristic, outperforms every other tool included in the benchmarks. In particular, both the FD-Strips with the *blind* heuristic and the SymBA*-2-Strips tools show particularly good results for more complex problems. Here, observe that the FD-Strips tool with the *blind* heuristic usually outperforms all other tools except when the number of constraints and the amount of noise increase. Indeed, in such cases, while the SymBA*-2-Strips planning time remains steady, the FD-Strips's time worsens. In fact, results in Table 7.4 indicate that the algorithmic

20 Constraints	FD-Gen		FD-GenConj		FD-GenShare		FD-GenConjShare		FD-Strips		SymBA-Strips	de Leoni et al.	Alignment Cost
	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}			
log traces length	*3 constraints modified*												
1-50	-	0.78	5.32	0.55	-	21.55	4.84	0.47	**0.39**	-	0.76	3.99	1.86
51-100	-	6.22	-	4.19	-	-	-	3.41	**0.84**	-	1.33	34.91	2.61
101-150	-	20.07	-	13.12	-	-	-	10.5	**1.78**	-	2.6	-	3.31
151-200	-	-	-	-	-	-	-	-	**5.07**	-	5.67	-	4.44
log traces length	*4 constraints modified*												
1-50	-	5.91	-	4.01	-	-	-	3.31	1.02	-	**0.76**	-	3.87
51-100	-	-	-	-	-	-	-	31.81	5.2	-	**1.35**	-	7.15
101-150	-	-	-	-	-	-	-	-	11.37	-	**2.55**	-	10.16
151-200	-	-	-	-	-	-	-	-	19.82	-	**4.71**	-	14.12
log traces length	*6 constraints modified*												
1-50	-	25.34	-	16.28	-	-	-	14	3.51	-	**0.84**	-	6.93
51-100	-	-	-	-	-	-	-	-	10.14	-	**1.41**	-	10.96
101-150	-	-	-	-	-	-	-	-	21.05	-	**2.48**	-	16.23
151-200	-	-	-	-	-	-	-	-	-	-	**4.11**	-	21.9

Table 7.4: Experimental results for the *synthetic* case study with 20 constraints. The time (in *seconds*) is the total time (average compilation time per trace + average planning time per trace). The timeout is set to 3600 seconds.

Financial log		FD-Gen		FD-GenConj		FD-GenShare		FD-GenConjShare		FD-Strips		SymBA-Strips	de Leoni et al.	Alignment Cost
		h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}	h_{blind}	h_{max}			
log trace length	# traces	16 constraints												
3-50	607	-	0.43	0.05	**0.01**	-	0.43	0.05	**0.01**	-	-	-	0.15	0.5
51-75	38	-	4.24	1.68	0.09	-	4.19	1.49	**0.07**	0.3	7.46	0.66	0.45	2.18
76-100	5	-	6.27	2.56	0.19	-	6.17	2.19	**0.15**	0.61	10.87	1.12	2.78	2.4
101-128	4	-	5.85	3.4	0.22	-	5.74	2.82	**0.17**	0.68	12.51	1.49	5.88	2.5

Table 7.5: Experimental results for the *real* case study. The time (in *seconds*) is the total time (average compilation time per trace + average planning time per trace). The timeout is set to 3600 seconds.

choice for the search in the state space is crucial when aligning long traces with relevant noise. In particular, in this case, the bidirectional A* search employed in SymBA*-2 scales better than the blind A* search of FD.

It is worth noting that the FD-GenConjShare tool with the *max* heuristic, has good overall performance too, especially in the case of 10 and 15 constraints. The FD-GenConj tool follows with fairly good results with the *max* heuristic, but it scales well up until 15 constraints. On the other hand, both the FD-Gen and the FD-GenShare tools have similar poor results compared to other tools. Notably, the conjunctive goal optimization (FD-GenConj) performs better than the optimization with shared states (FD-GenShare).

In addition, the FD-Strips tool is the only encoding showing better results with the *blind* heuristic than with the *max* heuristic. This suggests that when the planner is not leveraged to ground the planning task, i.e., when the grounding is already performed at compilation time, the *max* heuristic is not as informative as it is in all the other encodings and, in some cases, is even negatively affecting the search process.

In the real-life log, the approach in (de Leoni et al., 2012) is always faster than the FD-Strips tool using the *max* heuristic, whereas it is faster than the FD-Strips (*blind* heuristic) and SymBA*-2-Strips tools only for very short traces, namely those with length 3-50. Moreover, the approach in (de Leoni et al., 2012) shows better performances than the FD-Gen and the FD-GenShare tools (using both *blind* and *max* heuristic), which confirm their poor results seen on synthetic logs. However, (de Leoni et al., 2012) is not as fast as the other general encodings, i.e., FD-GenConj and FD-GenConjShare (with both heuristics).

In general, the FD-GenConj and FD-GenConjShare with *max* heuristic, corresponding to optimized high-level encodings, show the fastest alignment time with respect to all other tools. They even perform better than the STRIPS-based tools. Also, like in the experiments with synthetic logs, the *max* heuristic improves the performance of the general encodings, whereas it is not so effective for the FD-Strips tool, which performs better with the *blind* heuristic.

Notably, the STRIPS-based tools were timed out on the longest log (607 traces) with short traces (traces length between 3 and 50). The reason behind this result is due to the high compilation time of such tools. Indeed, although the STRIPS-based tools scale well even when the alignment problems become significantly complex (especially in the synthetic case), the results shown are averages per trace. However, when aligning an entire log at once, the results change because of a higher compilation time that no longer becomes negligible. Specifically, when the log contains a high number of traces, the compilation time of the STRIPS-based tools has shown to be two orders of magnitude higher than the one of the high-level encodings. In such a case, the FD-GenConjShare tool combined with the *max* heuristic is typically the best choice.

To sum up, we can conclude that, in general, no tool outperforms the others, and the best tool for trace alignment has to be carefully chosen based on the user's specific needs.

7.7 Related Work

A number of research works exist on the use of planning techniques in the context of BPM, covering the various stages of the process life cycle.

For the design-time phase, the existing literature focuses on exploiting planning techniques to automatically generate candidate process models that can achieve some business goals starting from a complete (Ferreira and Ferreira, 2006) or an incomplete (Marrella and Lespérance, 2017) description of the process domain. Some research works also exist that use planning techniques to deal with problems for the run-time phase. The works in (Marrella et al., 2016, 2014; van Beest et al., 2014) report on approaches to adapt the running process instances to cope with anomalous situations, including connection anomalies, exogenous events, and task faults. All these approaches do not automatically use historical information from event logs and use planning techniques for completely different purposes.

The scientific literature reports various works that use planning and AI-based techniques for conformance checking. Model checking is used in (Regis et al., 2012; Montali, 2010) to verify properties in process models. However, these methods only provide a true/false answer without identifying deviations and their severity. In (López et al., 2016), the authors propose a technique to find optimal alignments using constraint-satisfaction problems, but their approach is limited to acyclic process models and may not always provide

optimal solutions.

Some research focuses on verifying process model compliance with business rules encoded as formulas, e.g., (Belardinelli et al., 2012; Montali, 2010). Existing techniques check trace compliance by representing it as a set of formulas. However, they only highlight non-compliant formulas and not where deviations occur, such as when an activity is not executed. This is an easier task than pinpointing deviations since there are, in general, multiple ways to repair a trace to satisfy a formula. Instead, when multiple formulas are unsatisfied, finding the least expensive changes to ensure all formulas are satisfied becomes a planning problem.

Recent work has used Process Mining techniques, including conformance checking, to solve AI problems like goal recognition. Specifically, Polyvyanyy et al. (2020) use off-the-shelf process discovery and conformance checking techniques to solve probabilistic goal recognition when there is incomplete knowledge about the world the agent operates in. However, their approach is limited to matching an agent's observed behavior to Petri nets, thus basing their modeling discovery on a procedural paradigm.

7.8 Summary and Discussion

In this chapter, we addressed the problem of declarative trace alignment, an important problem in the business process management area that involves aligning process models with formal specifications expressed in LTL$_f$ or LDL$_f$ formulas. We proposed a formally correct reduction to cost-optimal planning and developed two PDDL encodings with optimizations to improve performance and scalability. Our approach has been implemented in a tool called TraceAligner, which currently stands as the top-performing solution for declarative trace alignment. We presented an empirical evaluation of TraceAligner on extensive benchmarks, and the results showed that our approach dramatically outperforms existing ad-hoc solutions in terms of scalability.

Chapter 8

Natural Language to Flow Construction via Automated Planning

In this chapter, we describe an industry application that combines some of the topics we have previously discussed in this dissertation. In particular, we give an overview of our newly released Python package NL2LTL that leverages the latest techniques in natural language understanding (NLU) and large language models (LLMs) to translate English inputs to linear temporal logic (LTL) formulas. Such an interface allows direct translation to formal languages that a reasoning system can use, while at the same time, allowing the end user to provide inputs in natural language without having to understand the details of the underlying formal language in a system. The package comes with support for a set of default LTL patterns, corresponding to popular DECLARE templates, but is also fully extensible to new domains so adopters of the package can configure it to their needs. The package has been open-sourced and is free to use for the AI community under the MIT license.

Part of the practical outcome of this research has been published in (Fuggitti and Chakraborti, 2023) at the System Demonstration track of the AAAI Conference on Artificial Intelligence 2023, and in (Fuggitti and Chakraborti, 2023) at the System Demonstration track of the International Conference of Automated Planning and Scheduling 2023.

8.1 Constructing Workflows for Automation

A vast number of automation tools require the user to construct workflows that embody some form of automation or business process. This user is not the end user but rather the developer or administrator

of that process. Such applications range from goal-oriented conversational agents such as Dialogflow[1] or Watson Assistant[2], and data processing flows such as AutoAI/ML[3,4], to web service composition tools such as in App Connect[5] or Zapier[6]. The examples are many and diverse, but in essence, they take the form of a workflow or a decision tree. They model a mixture of actions that determine or sense user intent and compose one or more units of automated and manual processes, realized as steps in that workflow that are able to satisfy the requirements of that intent.

This is, of course, not new to the BPM community, which has decades of work to show for representing and reasoning about such processes, including sophisticated specification languages like BPMN[7] as well as workflow construction tools and languages, both commercial and academic, such as YAWL (van Der Aalst and Ter Hofstede, 2005), FLOWer (van der Aalst et al., 2005), DECLARE (van der Aalst et al., 2009), and others. One natural outcome of this is that the ability of process administrators to write sophisticated constraints has significantly grown. However, this has come at the cost of a much higher barrier to entry in terms of the expected expertise of users who are able to write such specifications.

This limitation is not due to a lack of subject matter expertise on the user's part but rather due to how different the modeling paradigms are. For instance, a declarative modeling approach offers an exponential increase in the complexity of specification relative to the complexity of the realized model but requires a completely different way of thinking from imperative modeling – not unlike the thought process that goes into the basics of programming. Increased sophistication of the specification language further increases the required programming knowledge of the user. As such, the full capabilities of these advanced modeling tools remain out of reach for non-expert users who have neither training nor experience in declarative modeling and programming (Reijers et al., 2013). Recent advances in natural language processing offer an intriguing way out of this conundrum.

8.1.1 Natural Language to Workflow Construction

A host of enterprise applications revolve around the management of workflows – this includes data processing pipelines in AutoML (He et al., 2021), web service composition (Lemos et al., 2015), dialogue trees in conversational systems (Muise et al., 2020), and so on. An emerging theme in this area is the adoption of natural language as a desired input modality (Chakraborti et al., 2022), aimed at reducing the barrier of entry and expertise required for users looking to adopt workflow management tools. In fact, one of the

---

[1] https://developers.google.com/learn/pathways/chatbots-dialogflow
[2] https://www.ibm.com/products/watson-assistant
[3] https://aws.amazon.com/machine-learning/automl
[4] https://cloud.google.com/automl
[5] https://www.ibm.com/cloud/app-connect
[6] https://zapier.com/how-it-works
[7] https://www.omg.org/spec/BPMN/2.0

fascinating outcomes of recent advances in natural language processing is the emergence of generic language models that can be instantiated for specific domains to perform non-trivial information processing using only an interface to natural language instruction from the user (Bommasani et al., 2021).

In rudimentary form, this form of interaction can manifest itself in constructing pipelines in Bash syntax (command line interface) through natural language (Agarwal et al., 2021) or composing web services through IFTTT (if-this-then-that) instructions[8]. In general, this no-code/low-code approach (Hirzel, 2022) applies to the space of programming through natural language, such as in (Li et al., 2022).[9] There are also recent examples of workflow construction starting from natural language/document or multi-document (Shing et al., 2018; Feng et al., 2020, 2021; Chambers et al., 2020). However, these flow construction systems do not feature an active user experience. In fact, among them, only Doc2Dial[10] (Feng et al., 2020, 2021) provides a user-agent dialogue flow, though with an associated grounding-document helping with user utterances semantic parsing – the documents, and not user specifications, remain the source of knowledge on the workflow.

Linear Temporal Logic as a Substrate for Workflow Specification

As we have seen in Chapter 7, *declarative specifications* represent one of the scientific advances towards easier authoring tools for business processes and their management. An important specification language in this paradigm is LTL, which can be used to address many well-known problems in process management, e.g., our contribution of Chapter 7 is one example, while also admitting translations to specifications of standard reasoning engines such as automated planners (more on this later in Section 8.4).

Natural Language to Linear Temporal Logic

As highlighted throughout the dissertation, while LTL allows for a declarative paradigm, it also has a secondary benefit: LTL formulas can be readily described in an easy-to-understand natural language format. For instance, DECLARE templates (van der Aalst et al., 2009) are translatable to LTL_f (De Giacomo et al., 2014) and to PPLTL formulas (this dissertation, Table 3.2); moreover, there exists a variety of tools to generate a human-readable construct given an LTL formula (Cherukuri et al., 2022). In this work, we aim to facilitate making the journey in the opposite direction – from natural language to LTL. This has two main advantages:

1. unstructured inputs to a system can be translated to a form that reasoning engines can consume;

2. the interface to the end user remains as accessible as possible.

---

[8] https://ifttt.com

[9] Interestingly, although not strictly code, programming languages can be used as an intermediate representation in the task of converting natural language instruction to workflow representations. This allows the use of off-the-shelf language models trained on generic code, for which there exists plenty of data. (Groth and Gil, 2009)

[10] https://doc2dial.github.io

8.2 Related Work

Existing works in this category fall under two classes: one which provides support for a range of LTL formulas but compromises on the expressiveness of the input, and the other which admits a range of natural language inputs but is built for a particular domain and is not readily useful or usable as a general purpose package for practitioners.

Constrained Natural Language

These works rely on some form of constrained natural language (CNL) to ease the translation – e.g., in (Hahn et al., 2022; Schmitt, 2022) authors use grammar-based English sentences or structured English patterns for LTL translation, while (Narizzano et al., 2018; Narizzano and Vuotto, 2017) depends on a different structure called property specification patterns. While the former is somewhat domain-agnostic, a structured input means that the end user still has to become accustomed to the input language to use a system relying on the translator.

Domain-Dependent Natural Language

There are a few approaches that have attempted to parse unstructured natural language but have only done so in very limited domains. Perhaps, the best examples of this are (Wang et al., 2020; Wang, 2020), which implement a parser specifically for a robotics domain, and (Nikora and Balcom, 2009), which implements a translator for jet propulsion systems using a set of predetermined patterns from (Dwyer et al., 1998). Authors in (Kim et al., 2017; Lignos et al., 2015) also explored LTL translations from user instruction in robotics, but those translations were done manually in the former, while the latter is more automation-friendly but is limited to a specific parsing technique.

Reusable Assets

Among these works, other than (Wang, 2020; Narizzano and Vuotto, 2017; Schmitt, 2022), none have publicly available code and they are therefore not readily usable for practitioners. On the other hand, there are a couple of code bases that have also attempted natural language to LTL translation (Head, 2015; Zheng, 2020), but they are at a very rudimentary stage with little to no support or documentation. To the extent of our knowledge, our package is the first one going public with support for a significant breadth of LTL patterns and an extensible API to make it usable in different domain-specific contexts. In the next section, we will briefly describe the key features of the package.

8.3 NL2LTL: Converting Natural Language Instructions to Linear Temporal Logic Formulas

NL2LTL[11] is a Python package for translating natural language instructions to LTL formulas, with a unique focus on extensibility: (*i*) the inputs and outputs are domain agnostic, so it can be used and adapted for any domain of choice; and (*ii*) any and all components – be it the natural language understanding module or the scope of supported LTL formulas – are extendable or modifiable. Before going into the details of the architecture, we present a few examples of interactions with the package that illustrate these principles.

```python
from nl2ltl import translate
from nl2ltl.engines.rasa.core import RasaEngine
from nl2ltl.engines.utils import pretty

engine = RasaEngine()
utterance = "Send me a Slack after receiving a Gmail"

ltlf_formulas = translate(utterance, engine)
pretty(ltlf_formulas)
```

Figure 8.1: The basic NL2LTL API is shown above – the user simply imports their NLU engine of choice and requests for a translation. The output is shown below in Figure 8.2.

```
DECLARE Template: (ChainResponse Slack Gmail)
English meaning:  Every time activity Slack happens, it must be directly followed by
                  activity Gmail (activity Gmail can also follow other activities).
Confidence:       0.9999997615814209

DECLARE Template: (ExistenceTwo Slack)
English meaning:  Slack will happen at least twice.
Confidence:       1.0441302578101386e-07

DECLARE Template: (RespondedExistence Slack Gmail)
English meaning:  If Slack happens at least once, then Gmail has to happen or happened before Slack.
Confidence:       6.342362723898987e-08

DECLARE Template: (Response Slack Gmail)
English meaning:  Whenever activity Slack happens, activity Gmail has to happen
                  eventually afterward.
Confidence:       4.357292482382036e-08

DECLARE Template: (Existence Slack)
English meaning:  Eventually, Slack will happen.
Confidence:       4.138687170751609e-09
```

Figure 8.2: Sample output (pretty print) of NL2LTL, illustrating candidate DECLARE templates suggested by the package, using the Rasa NLU Engine, for the request: *"Send me a Slack after receiving a Gmail"*, along with translations back to English to communicate to the user what the interpretations made by the system mean (with associated confidence).

---

[11]NL2LTL is available online at https://github.com/ibm/nl2ltl.

```
DECLARE Template:    (ExistenceTwo Slack)
English meaning:     Slack will happen at least twice.
LTLf translation:    (eventually (and Slack (next (eventually Slack))))
PPLTL translation:   (once (and Slack (before (once Slack))))
Confidence:          1.0441302578101386e-07
```

Figure 8.3: It is possible to translate between different formal representations based on how the output will be consumed further downstream in an application. In this example, a DECLARE template has been translated to its corresponding LTL$_f$ and PPLTL formulas (recall that this conversion may not be easily available for all formulas).

```
DECLARE Template: (RespondedExistence Slack Gmail)
English meaning:  If Slack happens at least once, then Gmail has to happen or happened before Slack.
Confidence:       1.0
```

Figure 8.4: An illustration of swapping out the Rasa NLU engine with GPT-4 on the example from Figure 8.1 – note that the interfacing API is exactly the same. However, there will be some changes in behavior based on the selection of components. For example, GPT-4 always commits to a single formula (compare this with the output of Figure 8.2).

8.3.1   Sample Interactions

Figure 8.1 illustrates the basic NL2LTL API. The developer loads their desired engine of choice – in this example, Rasa (Bocklisch et al., 2017) – and requests for a translation. The output (Figure 8.2) is a list of possible translations, ranked by NL2LTL's confidence in that translation. Each candidate translation has a couple of interesting properties:

- **Explanations:** Each formula is translated back to English to illustrate how the formal representation interprets it. Depending on the downstream application, the developer can use this to explain to the end user to what extent the translation matches their original request.

- **Alternative Representations:** While this particular translation is to DECLARE templates, each candidate can also be translated to other equivalent representations, such as LTL$_f$ and PPLTL, as shown in Figure 8.3.

Finally, note how the NLU engine can be swapped out to get translations from a different service. Figure 8.4 illustrates this for the GPT-4 model from Open AI API (Brown et al., 2020; OpenAI, 2023). Interestingly, different NLU engines demonstrate different behaviors – Rasa was found to be better at identifying the right formulas but grounding them to the correct parameters is much more robust in GPT-4. For instance, the latter is much more proficient in picking among two candidate formulas $\mathsf{F}(a, b)$ and $\mathsf{F}(b, a)$ where the desired ordering can be produced by just swapping keywords such as "before" to "after" in an utterance. While details of the empirical differences between engines are outside the scope of this dissertation, we hope to report on them in the future.
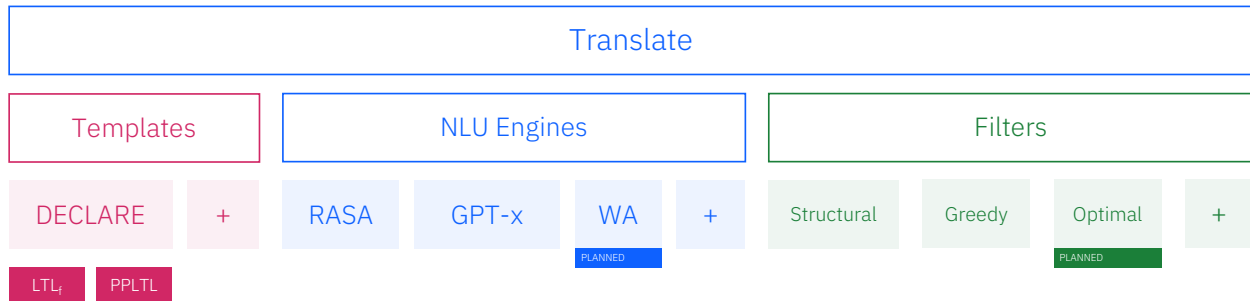
Figure 8.5: Overview of the NL2LTL package illustrating extensible components: (*i*) declare templates to translate to; (*ii*) NLU engines to use as translators; and (*iii*) Processors for conflicting and subsuming translations for an optimal parse.

8.3.2 Architecture

As we previously discussed in Figure 8.1, the primary function of NL2LTL is a translation method. This is assisted by three different components, shown in Figure 8.5.

Templates

Perhaps, the most important component of the package is the supported patterns, or "templates", to be identified. In this dissertation, we focused on DECLARE templates (van der Aalst et al., 2009) that are particularly suited for business process management tasks. As we mentioned before, while this covers a wide range of LTL formulas, a developer can also add their own templates specific to their application, such as ones suited for robotics domains where LTL is often used as a means of instruction (Lignos et al., 2015; Kim et al., 2017). Note that patterns available off-the-shelf already cover many common patterns (and can be translated among different representations, as shown in Figure 8.3, and so the new patterns are only required for very specific domain-dependent use cases.

NLU Engines

As demonstrated in Figures 8.2 and 8.4, NL2LTL can be configured with different NLU engines – while the API remains exactly the same, the characteristics of the results may vary. At the moment, NL2LTL comes with two engines pre-configured – one based on the intent-entity paradigm from Rasa (Bocklisch et al., 2017) that is traditionally used for natural language understanding, while the other is a language model-based extraction, tapping into the Open AI API (Brown et al., 2020). The configuration process for each engine is different - for instance, the former requires a set of training examples for each pattern to be identified (and an extra call to the corresponding training method in the API before it can be used, if not using the prepackaged patterns and model in NL2LTL), while the latter requires a prompt containing the training data.

As always, newer engines can be added by the user. We are currently experimenting with a Watson NLU[12] addition as a third off-the-shelf option.

Filter Functions

Interestingly, the patterns to be identified are not independent: (*i*) two LTL formulas can conflict with each other when both cannot be true at the same time, or (*ii*) one LTL formula can be subsumed by another when the latter always implies the former. As an example, in Figure 8.2, the first formula implies the third, i.e. if Gmail is to immediately follow Slack (required by ChainResponse), then Gmail following Slack at some point (required by RespondedExistence) is redundant. While in the example in Figure 8.2, the choice for which candidate to pick is pretty clear because of the markedly higher confidence, this need not always be the case.

To alleviate extra human effort in post-processing the set of translations, NL2LTL allows for a set of "filter functions" that can impose a desired selection policy on the candidate set. For example, a greedy policy might aggressively select detected patterns by decreasing the order of confidence while rejecting any that conflicts or subsumes with the already selected set. There can also be structural considerations in a filter – e.g., preferring binary formulas over unary ones, even if with lower confidence, if two parameters are identified in the input. In general, finding the optimal filter is a hard problem, but the developer is again allowed to use the NL2LTL API to implement any filter of choice.

8.4 Envisioned Product Impact: Workflow Construction via Automated Planning

We are currently working on an illustration of how the package can be adopted for use in real scenarios. For this, we will build on a system demonstration at AAAI 2022 (Brachman et al., 2022) on a web service composition task using natural language. The previous system was dependent on code that is specific to the parser in order to look for specific patterns requested by the user – this means that there is a significant developer overhead in investigating the particular parser used (in this case, parse trees from Abstract Syntax Representations (Astudillo et al., 2020) of user utterances), writing code for it which is limited in scope (limited by human effort), while eventually, that code is not reusable once the system upgrades to a different parser.

By augmenting the processing pipeline with the NL2LTL package, we are able to remove all parser-specific code and instead generate the required patterns with zero overhead. The patterns detected using the package,

---

[12]Watson Natural Language Understanding: `https://www.ibm.com/cloud/watson-natural-language-understanding`
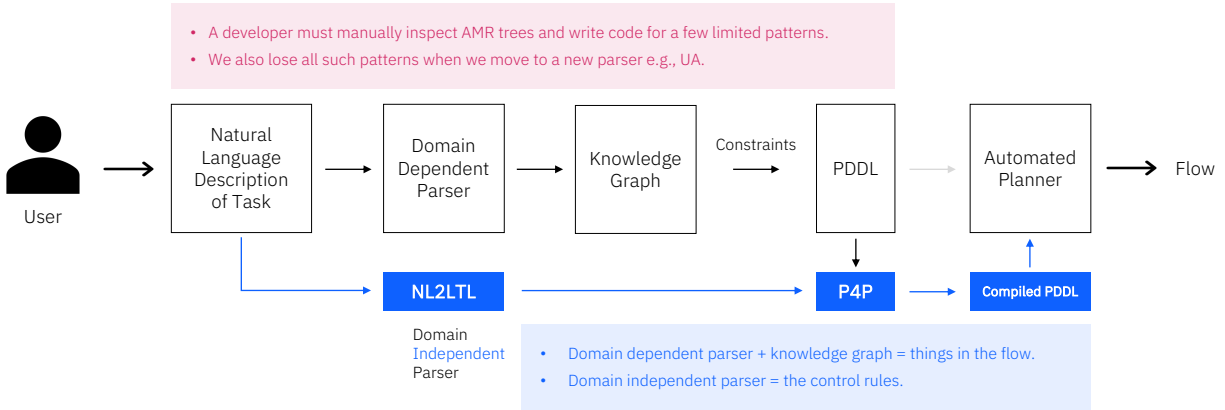
Figure 8.6: An application of NL2LTL to a real industrial application (Brachman et al., 2022).

parallel to the original processing pipeline, are eventually merged into one PDDL input to the automated planner in the end, using the compilation which receives as input the PDDL specification generated by the original pipeline along with the LTL patterns detected by the NL2LTL package and produces a compiled PDDL for the planner where the LTL patterns are enforced as control rules. For the LTL to PDDL compilation, we encourage the use of the one presented in this dissertation (cf. Chapters 5 and 6) but any existing approach (Baier and McIlraith, 2006a; Torres and Baier, 2015; Camacho et al., 2017; Camacho and McIlraith, 2019) will suffice. The modified processing pipeline is illustrated in Figure 8.6. We hope to report on the findings from this integration, as well as empirical studies on the contrastive translation capabilities (c.f. Section 8.3.1) of different NLU engines, in the near future.

Chapter 9

Conclusions

This dissertation represents a humble endeavor to advance and expand the vast body of human scientific knowledge that tries to formalize and integrate the art of human decision-making into artificial systems, empowering them with deliberating capabilities and autonomy while, at the same time, balancing the risks that come from wielding such power. As discussed in the introduction, one prominent approach to ensuring the safety of autonomous systems is to incorporate human-guided specifications. To address this challenge, formal languages and logics have been employed.

Within the specific scope of this dissertation, among the numerous applications of linear temporal logics on finite traces in AI, we particularly focused on developing efficient techniques for automated planning for goals expressed in pure-past linear temporal logics and on expanding the application of automated planning to two business domains, namely business process management and business automation, which have a significant impact on day-to-day lives of human beings.

9.1   Summary of Contributions

We have made a number of contributions to improve existing techniques in automated planning for temporally extended goals expressed using linear temporal logics on finite traces and to expand the use of automated planning in other closely related research fields, such as business process management. This section summarizes the specific contributions made in each chapter of this dissertation.

Chapter 3: Pure-Past Linear Temporal Logics

We considered the pure-past versions of the finite trace logics LTL_f and LDL_f, namely PPLTL and PPLDL, as first-class citizens of our research and described their main properties and characteristics. By exploiting a

well-known foundational result on reverse languages from (Chandra et al., 1981), we provided an algorithm to translate PPLTL and PPLDL formulas into their corresponding DFA that has an exponential improvement if compared to the algorithm to translate LTL$_f$ and LDL$_f$ formulas into DFA. Then, we reviewed the relationship between PPLTL/PPLDL and other formal languages, including FOL, RE$_f$, and MSO. These known results allowed us to expand the research by establishing that PPLTL and PPLDL have the same expressive power as LTL$_f$ and LDL$_f$, respectively, but transforming a PPLTL or PPLDL formula into its equivalent LTL$_f$ or LDL$_f$ formula and vice versa is computationally prohibitive (i.e., worst-case 3EXPTIME). Moving beyond considering only theoretical results for PPLTL and PPLDL, we have demonstrated their utility in a range of applications, providing translations in PPLTL of PDDL3 modal operators and of DECLARE patterns. Finally, we have discussed the impact of using pure-past temporal logics on well-known sequential decision-making problems, such as planning and decision problems in nondeterministic and non-Markovian domains, in comparison with LTL$_f$ and LDL$_f$.

Chapter 4: Handling Pure-Past Linear Temporal Logic Formulas

This chapter established the theoretical mathematical foundations to efficiently handle and evaluate PPLTL formulas. The fixpoint characterization of temporal logic formulas on finite traces is used to characterize the truth value of PPLTL formulas. Based on this characterization, we made three essential observations: (i) for evaluating a PPLTL formula, only the truth values of its subformulas are required; (ii) a PPLTL formula can be put in a form that depends on the current propositional evaluation and the evaluation of a small set of PPLTL subformulas at the previous time; and (iii) the value of this small set of subformulas can be recursively computed and stored as additional propositional variables in the domain of application. We exploited these observations to develop a novel and effective evaluation technique, which we proved to be correct. Additionally, we provided comprehensive examples to illustrate the effectiveness of the technique and its relationship with automata. Finally, the novel evaluation technique facilitated the development of more efficient algorithms, including those we examined in Chapters 5 and 6 for planning with temporally extended goals.

Chapter 5: Classical Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic

Chapter 5 explored the problem of classical planning for temporally extended goals expressed in pure-past linear temporal logic. The results have shown that despite the expressiveness of PPLTL being equivalent to LTL$_f$, deterministic planning for PPLTL goals is computationally more efficient than for LTL$_f$ goals. By exploiting the novel evaluation technique presented in Chapter 4, planning for PPLTL goals can be trans-

formed into classical planning with minimal overhead and without introducing spurious additional actions. Furthermore, the effectiveness of the proposed approach was demonstrated through the implementation of the open-source system Plan4Past, which was shown to outperform other existing compilations for LTL$_f$ goals on a variety of benchmarks.

Chapter 6: FOND Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic

This chapter investigated FOND planning for PPLTL goals. In contrast to previous work, we propose a direct technique for FOND planning for PPLTL goals that is optimal in terms of computational complexity and is also effective in practice. The technique involves a simple encoding of FOND planning for PPLTL goals into standard FOND planning for reachability goals, leveraging the observations made in Chapter 4. The encoding introduces only a few fluents (at most linear in the PPLTL goal), does not add any spurious actions, and allows planners to explore the relevant part of the deterministic automaton for the goal formula lazily during the planning search. We also provided an encoding variant that avoids introducing derived predicates, which are not well supported by existing FOND planners. We proved the correctness of both encodings and implemented them in the open-source system Plan4Past, which can be used with state-of-the-art FOND planners. Experimental results demonstrated the effectiveness of our approach compared to existing methods for handling LTL$_f$ temporally extended goals.

Chapter 7: Declarative Trace Alignment via Automated Planning

We delved into the challenge of declarative trace alignment, a problem that arises in business process management. In this context, the process model is defined by formal specifications expressed in either LTL$_f$ or LDL$_f$ formulas. We began by outlining the problem and its significance in the field. To tackle this problem, we then proposed a formally correct reduction to cost-optimal planning that allows us to use state-of-the-art automated planners. From a practical perspective, we developed two PDDL encodings to solve the trace alignment problem, each with various optimizations to enhance performance and scalability. Such encodings have been implemented in a tool called TraceAligner, which currently stands as the top-performing solution for declarative trace alignment. We concluded by providing an extensive empirical evaluation of TraceAligner on a range of benchmarks. The results demonstrated that our approach outperforms existing ad-hoc solutions in terms of performance and efficiency.

Chapter 8: Natural Language to Flow Construction via Automated Planning

In this chapter, we presented a novel application that combines several themes addressed in this dissertation and has practical industry relevance in the field of business automation. We proposed a new Python package, called NL2LTL, that employs the latest advancements in natural language understanding (NLU) and large language models (LLMs) to translate English inputs into LTL formulas. This interface provides an efficient way to translate natural language inputs to formal languages that reasoning systems can consume. Thus, end users can provide instructions in natural language without needing to understand the underlying formal notation used by reasoning systems. We featured the package with support for a predefined set of LTL patterns corresponding to popular DECLARE templates. These patterns can be easily customized to suit the specific needs of business processes. We have open-sourced the package and made it freely available for the AI community under the MIT license. Moreover, we provided an envisioned real product impact where NL2LTL can be used in a business automation context. This integration would help organizations save time and resources by reducing manual effort and errors that might occur due to misunderstandings.

9.2   Impact of Our Work

Recent publications have cited our works presented in this dissertation and built upon them in various ways. This witnesses the relevance and potential impact of our work in the field and highlights the importance of continuing research in this area.

By developing a novel approach capable of recognizing temporally extended goals in FOND planning domain models, Fraga Pereira et al. (2021) leverage the compilation of FOND planning for LTL_f/PPLTL goals into standard FOND planning for reachability goals presented in (De Giacomo and Fuggitti, 2021) to solve the problem goal recognition. To our knowledge, this is the first attempt to address the recognition of pure-past formulas. Along the same line of research, Aineto et al. (2021) tackled the Temporal Inference Problem to reason about the past, present, or future state of some observed agent. While their approach is restricted to a subset of LTL_f formulas that is sufficiently expressive to reason about past, present, and future in linear trajectories, our findings would allow a generalization of their technique when reasoning about past observations.

From a theoretical standpoint, a clear formalization of the semantics of PPLTL formulas and of the algorithm translating PPLTL formulas into corresponding DFAs (cf. Chapter 3) have paved the way for obtaining new theoretical results. De Giacomo et al. (2021a) use the semantics of PPLTL to prove that all possible safety conditions expressible in LTL, i.e., all first-order (logic) safety properties (Lichtenstein

et al., 1985), can be specified using LTL$_f$ on prefixes. Next, on the one hand, Artale et al. (2022) use such results to study and characterize the complexity of two novel safety fragments of LTL, namely LTL[$\widetilde{\mathsf{X}}$, G] and G(pLTL), for which they show that the finite-trace semantics can significantly decrease the complexity of both satisfiability and realizability. On the other hand, Cecconi et al. (2022) exploits our translation algorithm for PPLTL formulas to compute the DFA for formulas prescribing behavioral rules of processes.

Finally, our findings related to the clever handling and evaluation of pure-past linear temporal logics have led to new ongoing research. For instance, De Giacomo and Favorito (2021) aim at exploring a bottom-up transformation of PPLTL/PPLDL formulas into DFAs, whereas Geatti et al. (2022) have lately been motivated by our work on automated planning for PPLTL goals to develop an effective technique to solve the problem of reactive synthesis for DECLARE patterns via symbolic automata.

9.3   Future Work

Throughout the dissertation, we have already discussed potential directions for future research that relate to each of the main contributions. However, in this section, we take a more comprehensive approach and propose research directions that involve a wider perspective, potentially combining or extending multiple aspects of the work presented in this dissertation.

Investigating on the Impact of Using PPLTL and PPLDL

In this dissertation, when studying automated planning for temporally extended goals, we focused on goals expressed in PPLTL only. However, in principle, the whole theoretical approach we presented here (cf. Chapter 4) could be easily extended to goals expressed in PPLDL, that we recall to be a strictly more expressive variant of PPLTL. Indeed, PPLDL also has its own fixpoint characterization of temporal operators. Consequently, the contributions of Chapters 5 and 6 could be naturally extended to goals expressed in PPLDL. On the other hand, given that our contributions rely on a clever study of the syntactic structure of temporal formulas, it would be worth exploring if such studies might be advantageous to the following AI areas:

- Markov Decision Processes with non-Markovian rewards (Bacchus et al., 1996; Thiébaux et al., 2006; Brafman et al., 2018)

- Reinforcement Learning where rewards are based on traces (De Giacomo et al., 2019; Camacho et al., 2019b)

- Planning in non-Markovian domains (Brafman and De Giacomo, 2019a)

- Non-Markovian decision processes (Brafman and De Giacomo, 2019b)

PDDL4.0, Heuristics and Planners

This dissertation shows that deterministic and nondeterministic planning for goals expressed using pure-past linear temporal logics are much more well-behaved than goals expressed in LTL$_f$/LDL$_f$ since they allow for compilations that are optimal with respect to the computational complexity while having, at the same time, a clear advantage in practice. Given the advantages of PPLTL/PPLDL, these formalisms may definitely become promising candidates to be the mainstream languages to express temporal goals in planning. In this respect, we firmly believe that, as already happened for LTL$_f$, we could potentially introduce some of the most common and interesting PPLTL/PPLDL patterns within PDDL, giving rise to a novel version, i.e., the PDDL4.0.

Additionally, although our empirical results for planning with temporally extended goals are encouraging, the current classical and FOND planning technologies are faced with the difficulty of finding effective heuristics that are somewhat "aware" of temporal logics on finite traces and automata structures. The lack of suitable heuristics for classical and FOND planning for temporally extended goals has been identified as one of the major challenges to improve the search. Therefore, we envision the development of such PPLTL/PPLDL-aware heuristics that could dramatically improve the performance of planning, especially in the FOND setting.

Finally, the combination of novel PPLTL/PPLDL-aware heuristics and algorithms able to reason about such complex temporal specifications ultimately means designing and developing classical and FOND planners that can *natively* support goals expressed in PPLTL/PPLDL.

Enhancing Declarative Trace Alignment

Future directions concerning the declarative trace alignment problem addressed in Chapter 7 might consider a natural extension to the case of *data-aware* processes, i.e., where events are not just propositions but carry a payload in the form of a record, or even a first-order interpretation. In this way, one could fully exploit the potentialities of our approach. Although oriented more towards verification rather than alignment, a body of work exists on this class of processes/systems, e.g., see (Calvanese et al., 2018), which could provide a solid starting point to approach the problem.

A further extension could consider the *time-metric* dimension, where the process specification requires events to occur within, before, after, etc., specific time instances. For instance, one might require that whenever an event a occurs, it must not occur again before 3 time units, while event b must occur no later than 6 time units. A theoretical approach has already been developed in (De Giacomo et al., 2021b). However, such a solution has a non-elementary complexity that limits its practical applicability.

Combining these two extensions is currently a major challenge for the BPM community (Montali et al., 2013; Burattin et al., 2016; Maggi and Westergaard, 2014), that AI, and in particular planning, could provide an essential contribution to address.

Alternative NLP Processing Pipeline

As previously mentioned in Chapter 8, by building upon a web service composition task demonstrated at AAAI 2022 (Brachman et al., 2022), we are currently developing a practical application of the NL2LTL package to demonstrate its usability in real-world scenarios. To this end, we envision an augmented processing pipeline able to remove all parser-specific code and instead generate the required patterns with zero overhead. Then, we could leverage the work of Chapters 5 and 6 to enforce detected patterns as control rules. We hope to report on the findings from this integration, as well as empirical studies on the contrastive translation capabilities (c.f. Section 8.3.1) of different NLU engines, in the near future.

Finally, in recent years, there has been a growing popularity in the use of temporal logics such as LTL and LTL$_f$, prompting researchers to investigate how humans can effectively comprehend instructions given in these logics. For instance, Greenman et al. (2023) investigated this issue in depth with a two-year empirical study. This line of research could inspire further analyses evaluating the effectiveness of the NL2LTL package in comparison to human understanding, building on the insights from (Greenman et al., 2023).

Bibliography

Abushark, Y. B., Thangarajah, J., Harland, J., and Miller, T. (2017). A framework for automatically ensuring the conformance of agent designs. *J. Syst. Softw.*, 131:266–310.

Adriansyah, A., Sidorova, N., and van Dongen, B. F. (2011). Cost-Based Fitness in Conformance Checking. In *ACSD 2011*. IEEE.

Agarwal, M., Chakraborti, T., Fu, Q., Gros, D., Lin, X. V., Maene, J., Talamadupula, K., Teng, Z., and White, J. (2021). Neurips 2020 NLC2CMD Competition: Translating Natural Language to Bash Commands. In *NeurIPS 2020 Competition and Demonstration Track*. PMLR.

Aineto, D., Jiménez, S., and Onaindia, E. (2021). Generalized temporal inference via planning. In *KR*, pages 22–31.

Alechina, N., Logan, B., and Dastani, M. (2018). Modeling norm specification and verification in multiagent systems. *FLAP*, 5(2):457–490.

Aminof, B., De Giacomo, G., and Rubin, S. (2020). Stochastic fairness and language-theoretic fairness in planning in nondeterministic domains. In *ICAPS*, pages 20–28. AAAI Press.

Aminof, B., De Giacomo, G., Rubin, S., and Zuleger, F. (2022). Beyond Strong-Cyclic: Doing Your Best in Stochastic Environments. In *IJCAI*, pages 2525–2531. ijcai.org.

Ancona, D., Barbieri, M., and Mascardi, V. (2013). Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In *SAC*, pages 1377–1379. ACM.

Artale, A., Geatti, L., Gigante, N., Mazzullo, A., and Montanari, A. (2022). Complexity of safety and cosafety fragments of linear temporal logic. *CoRR*, abs/2211.14913.

Astudillo, R. F., Ballesteros, M., Naseem, T., Blodgett, A., and Florian, R. (2020). Transition-Based Parsing with Stack-Transformers. *arXiv:2010.10669*.

Bacchus, F., Boutilier, C., and Grove, A. (1996). Rewarding behaviors. In *AAAI*, pages 1160–1167.

Bacchus, F., Boutilier, C., and Grove, A. (1997). Structured solution methods for non-Markovian decision processes. In *AAAI*, pages 112–117.

Bacchus, F. and Kabanza, F. (1996). Planning for temporally extended goals. In *AAAI*, pages 1215–1222. AAAI Press.

Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial intelligence*, 116(1-2):123–191.

Bäckström, C. and Nebel, B. (1995). Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656.

Baier, C. and Katoen, J. (2008). *Principles of Model Checking*. MIT Press.

Baier, J. A., Fritz, C., Bienvenu, M., and McIlraith, S. A. (2008). Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In *AAAI*, pages 1509–1512. AAAI.

Baier, J. A. and McIlraith, S. A. (2006a). Planning with First-Order Temporally Extended Goals using Heuristic Search. In *AAAI*, pages 788–795. AAAI.

Baier, J. A. and McIlraith, S. A. (2006b). Planning with Temporally Extended Goals Using Heuristic Search. In *ICAPS*, pages 342–345. AAAI.

Baldoni, M., Baroglio, C., Martelli, A., and Patti, V. (2005). Verification of protocol conformance and agent interoperability. In *CLIMA*, volume 3900 of *Lecture Notes in Computer Science*, pages 265–283. Springer.

Bansal, S., Li, Y., Tabajara, L. M., and Vardi, M. Y. (2020). Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In *AAAI*, pages 9766–9774.

Barringer, H., Fisher, M., Gabbay, D. M., Gough, G., and Owens, R. (1989). METATEM: A framework for programming in temporal logic. In *REX Workshop*, volume 430 of *LNCS*, pages 94–129. Springer.

Belardinelli, F., Lomuscio, A., and Patrizi, F. (2012). Verification of GSM-Based Artifact-Centric Systems through Finite Abstraction. In *Proceedings of the 10th International Conference on Service-Oriented Computing, ICSOC'12*, volume 7636 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg.

Blythe, J. (1999). Decision-theoretic planning. *AI Mag.*, 20(2):37–54.

Bocklisch, T., Faulkner, J., Pawlowski, N., and Nichol, A. (2017). Rasa: Open Source Language Understanding and Dialogue Management. *arXiv:1712.05181*.

Bolander, T. and Andersen, M. B. (2011). Epistemic planning for single and multi-agent systems. *J. Appl. Non Class. Logics*, 21(1):9–34.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. (2021). On the opportunities and risks of foundation models. *arXiv:2108.07258*.

Bonassi, L., De Giacomo, G., Favorito, M., Fuggitti, F., Gerevini, A., and Scala, E. (2023a). FOND Planning for Pure-Past Linear Temporal Logic Goals. In *ECAI*.

Bonassi, L., De Giacomo, G., Favorito, M., Fuggitti, F., Gerevini, A., and Scala, E. (2023b). Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic. In *ICAPS*, volume 33, pages 61–69.

Bonassi, L., Gerevini, A., and Scala, E. (2022). Planning with qualitative action-trajectory constraints in PDDL. In *IJCAI*, pages 4606–4613. ijcai.org.

Bonet, B. and Geffner, H. (2001). Planning as Heuristic Search. *Artificial Intelligence*, 129(1-2):5–33.

Borgwardt, S., Hoffmann, J., Kovtunova, A., Krötzsch, M., Nebel, B., and Steinmetz, M. (2022). Expressivity of planning with horn description logic ontologies. In *AAAI*, pages 5503–5511. AAAI Press.

Brachman, M., Bygrave, C., Chakraborti, T., Chaudhary, A., Ding, Z., Dugan, C., Gros, D., Gschwind, T., Johnson, J., Laredo, J., Czasch, C. M., Pan, Q., Rai, P., Ramalingam, R., Scotton, P., Surabathina, N., and Talamadupula, K. (2022). A Goal-driven Natural Language Interface for Creating Application Integration Workflows. In *AAAI Demonstration*.

Brafman, R. I. and De Giacomo, G. (2019a). Planning for LTLf/LDLf goals in non-Markovian fully observable nondeterministic domains. In *IJCAI*, pages 1602–1608. ijcai.org.

Brafman, R. I. and De Giacomo, G. (2019b). Regular decision processes: A model for non-Markovian domains. In *IJCAI*, pages 5516–5522. ijcai.org.

Brafman, R. I., De Giacomo, G., and Patrizi, F. (2018). LTLf/LDLf non-Markovian rewards. In *AAAI*, pages 1771–1778. AAAI Press.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language Models are Few-Shot Learners. *NeurIPS*.

Brzozowski, J. A. and Leiss, E. (1980). On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35.

Büchi, J. R. (1990). Weak second-order arithmetic and finite automata. In *The Collected Works of J. Richard Büchi*, pages 398–424. Springer.

Burattin, A., Maggi, F. M., and Sperduti, A. (2016). Conformance Checking Based on Multi-Perspective Declarative Process Models. *Expert Syst. Appl.*, 65:194–211.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204.

Calvanese, D., De Giacomo, G., Montali, M., and Patrizi, F. (2018). First-order μ-calculus over generic transition systems and applications to the situation calculus. *Inf. Comput.*, 259(3):328–347.

Camacho, A., Baier, J. A., Muise, C. J., and McIlraith, S. A. (2018). Finite LTL synthesis as planning. In *ICAPS*, pages 29–38. AAAI Press.

Camacho, A., Bienvenu, M., and McIlraith, S. A. (2019a). Towards a unified view of AI planning and reactive synthesis. In *ICAPS*, volume 29, pages 58–67.

Camacho, A. and McIlraith, S. A. (2019). Strong fully observable non-deterministic planning with LTL and LTLf goals. In *IJCAI*, pages 5523–5531.

Camacho, A., Toro Icarte, R., Klassen, T. Q., Valenzano, R. A., and McIlraith, S. A. (2019b). LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI*, pages 6065–6073. ijcai.org.

Camacho, A., Triantafillou, E., Muise, C. J., Baier, J. A., and McIlraith, S. A. (2017). Non-deterministic planning with temporally extended goals: LTL over finite and infinite traces. In *AAAI*, pages 3716–3724. AAAI Press.

Carmona, J., van Dongen, B. F., Solti, A., and Weidlich, M. (2018). *Conformance Checking - Relating Processes and Models*. Springer.

Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtós, N., and Carreras, M. (2015). ROSPlan: Planning in the robot operating system. In *ICAPS*, pages 333–341. AAAI Press.

Cecconi, A., De Giacomo, G., Di Ciccio, C., Maggi, F. M., and Mendling, J. (2022). Measuring the interestingness of temporal logic behavioral specifications in process mining. *Inf. Syst.*, 107:101920.

Chakraborti, T., Rizk, Y., Isahagian, V., Aksar, B., and Fuggitti, F. (2022). From Natural Language to Workflows: Towards Emergent Intelligence in Robotic Process Automation. In *BPM*.

Chambers, A. J., Stringfellow, A. M., Luo, B. B., Underwood, S. J., Allard, T. G., Johnston, I. A., Brockman, S., Shing, L., Wollaber, A., and VanDam, C. (2020). Automated Business Process Discovery from Unstructured Natural-Language Documents. In *BPM*.

Chandra, A., Kozen, D., and Stockmeyer, L. (1981). Alternation. *J. of the ACM*, 28(1):114–133.

Cherukuri, H., Ferrari, A., and Spoletini, P. (2022). Towards Explainable Formal Methods: From LTL to Natural Language with Neural Machine Translation. In *REFSQ*.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.

Church, A. (1957). Applications of recursive arithmetic to the problem of circuit synthesis–summaries of talks. *Institute for Symbolic Logic, Cornell University*.

Cimatti, A., Geatti, L., Gigante, N., Montanari, A., and Tonetta, S. (2020). Reactive synthesis from extended bounded response LTL specifications. In *FMCAD*, pages 83–92. IEEE.

Cimatti, A., Giunchiglia, F., Giunchiglia, E., and Traverso, P. (1997). Planning via model checking: A decision procedure for *AR*. In *ECP*, volume 1348 of *LNCS*, pages 130–142. Springer.

Cimatti, A., Pistore, M., Roveri, M., and Traverso, P. (2003). Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2):35–84.

Cimatti, A., Roveri, M., and Sheridan, D. (2004). Bounded verification of Past LTL. In *FMCAD*, pages 245–259.

Cimatti, A., Roveri, M., and Traverso, P. (1998). Strong planning in non-deterministic domains via model checking. In *AIPS*, pages 36–43. AAAI.

Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer.

Daniele, M., Traverso, P., and Vardi, M. Y. (1999). Strong cyclic planning revisited. In *ECP*, volume 1809 of *Lecture Notes in Computer Science*, pages 35–48. Springer.

De Giacomo, G., De Masellis, R., Maggi, F., and Montali, M. (2022). Monitoring constraints and metaconstraints with temporal logics on finite traces. *ACM Trans. Softw. Eng. Methodol.*, 31(4):68:1–68:44.

De Giacomo, G., De Masellis, R., and Montali, M. (2014). Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, pages 1027–1033.

De Giacomo, G., Di Stasio, A., Fuggitti, F., and Rubin, S. (2020a). Pure-past linear temporal and dynamic logic on finite traces. In *IJCAI*, pages 4959–4965. ijcai.org.

De Giacomo, G., Di Stasio, A., Tabajara, L. M., Vardi, M. Y., and Zhu, S. (2021a). Finite-trace and generalized-reactivity specifications in temporal synthesis. In *IJCAI*, pages 1852–1858. ijcai.org.

De Giacomo, G. and Favorito, M. (2021). Compositional approach to translate LTLf/LDLf into deterministic finite automata. In *ICAPS*, pages 122–130. AAAI Press.

De Giacomo, G., Favorito, M., Iocchi, L., Patrizi, F., and Ronca, A. (2020b). Temporal logic monitoring rewards via transducers. In *KR*, pages 860–870.

De Giacomo, G. and Fuggitti, F. (2021). FOND4LTLf: FOND planning for LTLf/PLTLf goals as a service. In *ICAPS*. System Demonstration.

De Giacomo, G., Fuggitti, F., Maggi, F. M., Marrella, A., and Patrizi, F. (2023). A Tool for Declarative Trace Alignment via Automated Planning. *Software Impacts*, 16:100505.

De Giacomo, G., Iocchi, L., Favorito, M., and Patrizi, F. (2019). Foundations for restraining bolts: Reinforcement learning with LTLf/LDLf restraining specifications. In *ICAPS*, pages 128–136. AAAI Press.

De Giacomo, G., Maggi, F., Marrella, A., and Patrizi, F. (2017). On the disruptive effectiveness of automated planning for LTLf-based trace alignment. In *AAAI*, pages 3555–3561. AAAI Press.

De Giacomo, G., Maggi, F. M., Marrella, A., and Sardiña, S. (2016). Computing Trace Alignment against Declarative Process Models through Planning. In *26th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

De Giacomo, G., Murano, A., Patrizi, F., and Perelli, G. (2021b). Timed trace alignment with metric temporal logic over finite traces. In *KR*, pages 227–236.

De Giacomo, G. and Rubin, S. (2018). Automata-theoretic foundations of FOND planning for LTLf and LDLf goals. In *IJCAI*, pages 4729–4735.

De Giacomo, G. and Vardi, M. Y. (1999). Automata-theoretic approach to planning for temporally extended goals. In *ECP*, volume 1809 of *LNCS*, pages 226–238. Springer.

De Giacomo, G. and Vardi, M. Y. (2013). Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pages 854–860. IJCAI/AAAI.

De Giacomo, G. and Vardi, M. Y. (2015). Synthesis for LTL and LDL on finite traces. In *IJCAI*.

De Giacomo, G. and Vardi, M. Y. (2016). LTLf and LDLf synthesis under partial observability. In *IJCAI*, volume 2016, pages 1044–1050.

de Leoni, M., Maggi, F. M., and van der Aalst, W. M. (2012). Aligning Event Logs and Declarative Process Models for Conformance Checking. In *10th Int. Conf. on Business Process Management (BPM 2012)*.

de Leoni, M., Maggi, F. M., and van der Aalst, W. M. (2015). An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. *Inf. Syst.*, 47:258–277.

Di Ciccio, C., Bernardi, M. L., Cimitile, M., and Maggi, F. M. (2015). Generating event logs through the simulation of DECLARE models. In *11th Int. Workshop on Ent. & Org. Modeling and Simulation (EOMAS 2015)*.

D'Ippolito, N., Rodríguez, N., and Sardiña, S. (2018). Fully observable non-deterministic planning as assumption-based reactive synthesis. *J. Artif. Intell. Res.*, 61:593–621.

Dumas, M., La Rosa, M., Mendling, J., Reijers, H. A., et al. (2018). *Fundamentals of Business Process Management*, volume 2. Springer.

Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., and Xu, L. (2016). Spot 2.0—a framework for LTL and ω-automata manipulation. In *ATVA*, pages 122–129.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1998). Property Specification Patterns for Finite-State Verification. In *Workshop on Formal Methods in Software Practice*.

Eisner, C. and Fisman, D. (2006). *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer.

El-Menshawy, M., Bentahar, J., Kholy, W. E., and Dssouli, R. (2013). Verifying conformance of multi-agent commitment-based protocols. *Expert Syst. Appl.*, 40(1):122–138.

Emerson, E. A. (1990). Temporal and modal logic. In *Handbook of Theoretical Computer Science, Chapter 16*.

Fellah, A., Jürgensen, H., and Yu, S. (1990). Constructions for alternating finite automata. *International journal of computer mathematics*, 35(1-4):117–132.

Feng, S., Patel, S. S., Wan, H., and Joshi, S. (2021). MultiDoc2Dial: Modeling Dialogues Grounded in Multiple Documents. In *EMNLP*.

Feng, S., Wan, H., Gunasekara, R. C., Patel, S. S., Joshi, S., and Lastras, L. A. (2020). doc2dial: A Goal-Oriented Document-Grounded Dialogue Dataset. In *EMNLP*.

Ferreira, H. M. and Ferreira, D. R. (2006). An Integrated Life Cycle for Workflow Management Based on Learning and Planning. *Int. J. Cooperative Inf. Syst.*, 15(4):485–505.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208.

Finkbeiner, B. and Sipma, H. (2004). Checking finite traces using alternating automata. *Formal Methods Syst. Des.*, 24(2):101–127.

Fischer, M. J. and Ladner, R. E. (1979). Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211.

Fisher, M. and Wooldridge, M. (2005). Temporal reasoning in agent-based systems. In *FAI*, volume 1, pages 469–495. Elsevier.

Fogarty, S., Kupferman, O., Vardi, M. Y., and Wilke, T. (2015). Profile trees for Büchi word automata, with application to determinization. *Information and Computation*, 245:136–151.

Fraga Pereira, R., Fuggitti, F., and De Giacomo, G. (2021). Recognizing LTLf/PLTLf goals in fully observable non-deterministic domain models. In *CoRR*, volume abs/2103.11692.

Fraga Pereira, R., Grahl Pereira, A., Messa, F., and De Giacomo, G. (2022). Iterative depth-first search for FOND planning. In *ICAPS*, pages 90–99. AAAI Press.

Fuggitti, F. (2019). FOND planning for LTLf and PLTLf goals. In *ICAPS*, Berkeley.

Fuggitti, F. and Chakraborti, T. (2023). NL2LTL – A Python Package for Converting Natural Language (NL) Instructions to Linear Temporal Logic (LTL) Formulas. In *AAAI*. System Demonstration.

Gabaldon, A. (2011). Non-Markovian control in the situation calculus. *AIJ*, 175(1):25–48.

Gabbay, D. (1987). The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*.

Gabbay, D. M., Pnueli, A., Shelah, S., and Stavi, J. (1980). On the temporal analysis of fairness. In *POPL*, pages 163–173. ACM Press.

Gazen, B. C. and Knoblock, C. A. (1997). Combining the expressivity of UCPOP with the efficiency of graphplan. In *ECP*, volume 1348 of *Lecture Notes in Computer Science*, pages 221–233. Springer.

Geatti, L., Montali, M., and Rivkin, A. (2022). Reactive synthesis for DECLARE via symbolic automata. *CoRR*, abs/2212.10875.

Geffert, V. and Okhotin, A. (2014). Transforming two-way alternating finite automata to one-way nondeterministic automata. In *Mathematical Foundations of Computer Science 2014*, pages 291–302, Berlin, Heidelberg. Springer Berlin Heidelberg.

Geffner, H. and Bonet, B. (2013). *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668.

Ghallab, M., Nau, D. S., and Traverso, P. (2004). *Automated planning - theory and practice*. Elsevier.

Giunchiglia, F. and Traverso, P. (1999). Planning as model checking. In *ECP*, volume 1809 of *Lecture Notes in Computer Science*, pages 1–20. Springer.

Greenman, B., Saarinen, S., Nelson, T., and Krishnamurthi, S. (2023). Little tricky logic: Misconceptions in the understanding of LTL. *Art Sci. Eng. Program.*, 7(2).

Groth, P. and Gil, Y. (2009). Analyzing the gap between workflows and their natural language descriptions. In *IEEE World Congress on Services*.

Hahn, C., Schmitt, F., Tillman, J. J., Metzger, N., Siber, J., and Finkbeiner, B. (2022). Formal Specifications from Natural Language. *arXiv:2206.01962*.

Haslum, P. (2013). Optimal delete-relaxed (and semi-relaxed) planning with conditional effects. In *IJCAI*, pages 2291–2297. IJCAI.

He, X., Zhao, K., and Chu, X. (2021). AutoML: A Survey of the State-of-the-Art. *Knowledge-Based Systems*.

Head, A. (2015). LTLTrans. `https://github.com/andrewhead/LTLTrans`. Accessed: 2016-02-09.

Helmert, M. (2006). The fast downward planning system. *JAIR*, 26(1):191–246.

Henriksen, J. G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., and Sandholm, A. (1995). MONA: Monadic second-order logic in practice. In *TACAS*, pages 89–110. Springer.

Hirzel, M. (2022). Low-code programming models. *arXiv:2205.02282*.

Hoffmann, J. and Edelkamp, S. (2005). The deterministic part of IPC-4: An overview. *JAIR*, 24:519–579.

Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

Howe, A. E. and Dahlman, E. (2002). A critical assessment of benchmark comparison in planning. *Journal of Artificial Intelligence Research*, 17:1–33.

Kamp, H. (1968). *Tense Logic and the Theory of Linear Order*. PhD thesis, Ucla.

Khoussainov, B. and Nerode, A. (2001). *Automata Theory and Its Applications*. Birkhauser Boston, Inc., Secaucus, NJ, USA.

Kim, J., Banks, C. J., and Shah, J. A. (2017). Collaborative Planning with Encoding of Users' High-Level Strategies. In *AAAI*.

Klarlund, N., Møller, A., and Schwartzbach, M. I. (2002). MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586. World Scientific Publishing Company. Earlier version in Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00, Springer-Verlag LNCS vol. 2088.

Knobbout, M., Dastani, M., and Meyer, J. C. (2016). A dynamic logic of norm change. In *ECAI*, volume 285 of *FAIA*, pages 886–894. IOS Press.

Lacerda, B., Parker, D., and Hawes, N. (2015). Optimal policy generation for partially satisfiable co-safe LTL specifications. In *IJCAI*, pages 1587–1593. AAAI Press.

Leiss, E. (1981). Succinct representation of regular languages by boolean automata. *Theoretical computer science*, 13(3):323–330.

Lemos, A. L., Daniel, F., and Benatallah, B. (2015). Web Service Composition: A Survey of Techniques and Tools. *ACM Computing Surveys*.

Li, J., Rozier, K. Y., Pu, G., Zhang, Y., and Vardi, M. Y. (2019). SAT-based explicit LTL_f satisfiability checking. In *AAAI*, pages 2946–2953. AAAI Press.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., et al. (2022). Competition-level code generation with alphacode. *arXiv:2203.07814*.

Lichtenstein, O., Pnueli, A., and Zuck, L. D. (1985). The glory of the past. In *Logic of Programs*, volume 193 of *LNCS*, pages 196–218. Springer.

Lignos, C., Raman, V., Finucane, C., Marcus, M. P., and Kress-Gazit, H. (2015). Provably correct reactive control from natural language. *Autonomous Robots*, 38(1):89–105.

López, M. T. G., Borrego, D., Carmona, J., and Gasca, R. M. (2016). Computing alignments with constraint programming: The acyclic case. In *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data, ATAED 2016*, volume 1592 of *CEUR Workshop Proceedings*, pages 96–110. CEUR-WS.org.

Maggi, F. M. and Westergaard, M. (2014). Using Timed Automata for *a Priori* Warnings and Planning for Timed Declarative Process Models. *Int. J. Coop. Inf. Syst.*, 23(1).

Maler, O. and Pnueli, A. (1990). Tight bounds on the complexity of cascaded decomposition of automata. In *FOCS*.

Manna, Z. (1982). *Verification of Sequential Programs: Temporal Axiomatization*, pages 53–102. Springer Netherlands.

Marrella, A. and Lespérance, Y. (2017). A planning approach to the automated synthesis of template-based process models. *Service Oriented Computing and Applications*, 11(4):367–392.

Marrella, A., Mecella, M., and Sardiña, S. (2014). SmartPM: An Adaptive Process Management System through Situation Calculus, IndiGolog, and Classical Planning. In *Knowledge Representation and Reasoning (KR 2014)*.

Marrella, A., Mecella, M., and Sardiña, S. (2016). Intelligent Process Adaptation in the SmartPM System. *ACM Trans. Intell. Syst. Technol.*, 8(2):25:1–25:43.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL – the planning domain definition language. Technical report, ICAPS.

McNaughton, R. and Papert, S. A. (1971). *Counter-Free Automata (MIT research monograph no. 65)*. The MIT Press.

Mealy, G. H. (1955). A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079.

Montali, M. (2010). *Specification and Verification of Declarative Open Interaction Models - A Logic-Based Approach*, volume 56 of *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg.

Montali, M., Chesani, F., Mello, P., and Maggi, F. M. (2013). Towards data-aware constraints in DECLARE. In *SAC*, pages 1391–1396. ACM.

Muise, C., Chakraborti, T., Agarwal, S., Bajgar, O., Chaudhary, A., Lastras-Montano, L. A., Ondrej, J., Vodolan, M., and Wiecha, C. (2020). Planning for Goal-Oriented Dialogue Systems. *arXiv:1910.08137*.

Muise, C., McIlraith, S., and Beck, C. (2012). Improved non-deterministic planning by exploiting state relevance. In *ICAPS*, volume 22.

Narizzano, M., Pulina, L., Tacchella, A., and Vuotto, S. (2018). Consistency of Property Specification Patterns with Boolean and Constrained Numerical Signals. In *NASA Formal Methods Symposium*.

Narizzano, M. and Vuotto, S. (2017). Structured Natural Language To Formal Language. `https://github.com/SAGE-Lab/snl2fl`. Accessed: 2018-07-12.

Nebel, B. (2000). On the compilability and expressive power of propositional planning formalisms. *JAIR*, 12:271–315.

Nikora, A. P. and Balcom, G. (2009). Automated Identification of LTL Patterns in Natural Language Requirements. In *Symposium on Software Reliability Engineering*.

OpenAI (2023). GPT-4 Technical Report. *arXiv:2303.08774*.

Patrizi, F., Lipovetzky, N., De Giacomo, G., and Geffner, H. (2011). Computing Infinite Plans for LTL Goals Using a Classical Planner. In *IJCAI*, pages 2003–2008.

Pešić, M., Bošnački, D., and van der Aalst, W. M. (2010). Enacting declarative languages using LTL: avoiding errors and improving performance. In *SPIN Workshop*, pages 146–161. Springer.

Pistore, M., Bettin, R., and Traverso, P. (2001). Symbolic techniques for planning with extended goals in non-deterministic domains. In *ECP*, pages 253–264.

Pistore, M. and Traverso, P. (2001). Planning as model checking for extended goals in non-deterministic domains. In *IJCAI*, pages 479–486. Morgan Kaufmann.

Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA. IEEE Computer Society.

Pnueli, A. and Rosner, R. (1989). On the synthesis of a reactive module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages - POPL*. ACM Press.

Polyvyanyy, A., Su, Z., Lipovetzky, N., and Sardiña, S. (2020). Goal recognition using off-the-shelf process mining techniques. In *AAMAS*, pages 1072–1080. International Foundation for Autonomous Agents and Multiagent Systems.

Quielle, J. and Sifakis, J. (1981). Specification and verification of concurrent programs in CESAR. In *LNCS*, volume 137, pages 337–351.

Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125.

Regis, G., Ricci, N., Aguirre, N. M., and Maibaum, T. (2012). Specifying and Verifying Declarative Fluent Temporal Logic Properties of Workflows. In *Proceedings of the 15th Brazilian Conference on Formal Methods: Foundations and Applications, SBMF'12*, volume 7498 of *Lecture Notes of Computer Science*. Springer-Verlag.

Reijers, H. A., Slaats, T., and Stahl, C. (2013). Declarative modeling – an academic dream or the future for BPM? In *BPM*.

Richter, S. and Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR*, 39:127–177.

Rintanen, J. (2004). Complexity of planning with partial observability. In *ICAPS*, pages 345–354.

Röger, G., Pommerening, F., and Helmert, M. (2014). Optimal planning in the presence of conditional effects: Extending LM-Cut with context splitting. In *ECAI*, volume 263 of *FAIA*, pages 765–770. IOS.

Schmitt, F. (2022). ML2: Machine Learning for Mathematics and Logics. `https://github.com/reactive-systems/ml2`. Accessed: 2022-02-03.

Shing, L., Wollaber, A., Chikkagoudar, S., Yuen, J., Alvino, P., Chambers, A., and Allard, T. (2018). Extracting workflows from natural language documents: A first step. In *BPM*.

Shmaryahu, D., Shani, G., Hoffmann, J., and Steinmetz, M. (2018). Simulated penetration testing as contingent planning. In *ICAPS*, pages 241–249. AAAI Press.

Sohrabi, S., Baier, J., and McIlraith, S. (2011). Preferred explanations: Theory and generation via planning. In *AAAI*, volume 25, pages 261–267.

Tabajara, L. M. and Vardi, M. Y. (2020). LTLf synthesis under partial observability: From theory to practice. In *GandALF*, volume 326 of *EPTCS*, pages 1–17.

Tabakov, D. and Vardi, M. Y. (2005). Experimental evaluation of classical automata constructions. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 396–411. Springer.

Thiébaux, S., Gretton, C., Slaney, J., Price, D., and Kabanza, F. (2006). Decision-theoretic planning with non-Markovian rewards. *Journal of Artificial Intelligence Research*, 25(1):17–74.

Thiébaux, S., Hoffmann, J., and Nebel, B. (2005). In defense of PDDL axioms. *AIJ*, 168(1-2):38–69.

Torralba, A., Alcazar, V., Borrajo, D., Kissmann, P., and Edelkamp, S. (2014). Symba: A symbolic bidirectional a planner. In *International Planning Competition*, pages 105–108.

Torres, J. and Baier, J. A. (2015). Polynomial-time reformulations of LTL temporally extended goals into final-state goals. In *IJCAI*, pages 1696–1703. AAAI Press.

van Beest, N. R. T. P., Kaldeli, E., Bulanov, P., Wortmann, J. C., and Lazovik, A. (2014). Automated runtime repair of business processes. *Inf. Syst.*, 39:45–79.

van der Aalst, W. M., Pesic, M., and Schonenberg, H. (2009). Declarative Workflows: Balancing Between Flexibility and Support. *Computer Science - R&D*, 23(2):99–113.

van Der Aalst, W. M. and Ter Hofstede, A. H. (2005). YAWL: Yet another workflow language. *Information systems*.

van der Aalst, W. M., Weske, M., and Grünbauer, D. (2005). Case Handling: A New Paradigm for Business Process Support. *Data & knowledge engineering*.

Vardi, M. Y. (1996). An automata-theoretic approach to linear temporal logic. In *LNCS 1043*, pages 238–266. Springer.

Vardi, M. Y. (1998). Reasoning about the past with two-way automata. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer.

Vukovic, M., Gerard, S., Hull, R., Katz, M., Shwartz, L., Sohrabi, S., Muise, C., Rofrano, J., Kalia, A., Hwang, J., Yabin, D., Jie, M., and Zhuoxuan, J. (2019). Towards automated planning for enterprise services: Opportunities and challenges. In *Service-Oriented Computing*, pages 64–68, Cham. Springer International Publishing.

Wang, C. (2020). Grounded LTL Parser. `https://github.com/czlwang/grounded_LTL_parser`. Accessed: 2021-04-27.

Wang, C., Ross, C., Kuo, Y., Katz, B., and Barbu, A. (2020). Learning a Natural-Language to LTL Executable Semantic Parser for Grounded Robotics. In *Proceedings of Machine Learning Research*.

Wolper, P. (1981). Temporal logic can be more expressive. *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, pages 340–348.

Zheng, S. (2020). Translation from natural language to Linear Temporal Logic. `https://github.com/suchzheng2/Lang2LTL`. Accessed: 2020-12-12.

Zhu, S., Pu, G., and Vardi, M. Y. (2019). First-order vs. second-order encodings for LTLf-to-automata translation. In *TAMC*, pages 684–705.

Zhu, S., Tabajara, L. M., Li, J., Pu, G., and Vardi, M. Y. (2017). Symbolic LTLf synthesis. In *IJCAI*, pages 1362–1369.

Zhu, S., Tabajara, L. M., Pu, G., and Vardi, M. Y. (2021). On the power of automata minimization in temporal synthesis. In *GandALF*, volume 346 of *EPTCS*, pages 117–134.