Università degli Studi di Brescia
Dipartimento di Ingegneria dell'Informazione

Dottorato di ricerca in Ingegneria dell'Informazione - Ciclo XXXVI

Effective Approaches for Handling Plan Constraints and Temporally Extended Goals in Automated Planning

Luigi Bonassi

Supervisors

Alfonso E. Gerevini
Enrico Scala

Settore Scientifico Disciplinare ING-INF/05

Contents

List of Figures

List of Tables

Abstract (Italian)

Una delle più importanti sfide dell'Intelligenza Artificiale (IA) è progettare agenti autonomi in grado di agire in ambienti complessi per raggiungere un obiettivo finale rispettando molteplici vincoli. Questa sfida è attualmente affrontata dalla Pianificazione Automatica, un campo dell'IA che si occupa del problema della sintesi di una sequenza di azioni, ovvero un piano, per raggiungere un insieme di obiettivi. Una parte rilevante della ricerca scientifica si concentra sui modelli di pianificazione classica, in cui gli obiettivi sono definiti come condizioni che devono essere soddisfatte nello stato finale raggiunto dall'esecuzione del piano. Tuttavia, questo approccio presenta alcune limitazioni significative; molti obiettivi degli agenti autonomi possono coinvolgere il mantenimento di una condizione di safety durante l'intera esecuzione del piano, reazioni a determinati input entro un periodo di tempo limitato, ed obiettivi aggiuntivi da raggiungere. Inoltre, diversi vincoli possono influenzare il comportamento dell'agente e la struttura dei piani desiderati. Questa tesi affronta il problema della pianificazione con queste specifiche, ed esplora tre diverse modalità di formulazione di questi vincoli. In primo luogo, vengono considerati i vincoli sulla traiettoria degli stati definiti in PDDL3, uno dei linguaggi di pianificazione più diffusi. In secondo luogo, viene introdotto un nuovo formalismo per esprimere vincoli di traiettoria sulle azioni di un piano. Infine, vengono consideriati goal temporali espressi in Pure-Past Linear Temporal Logic. Per ciascun formalismo, viene proposto uno schema di compilazione per tradurre problemi di pianificazione con queste specifiche in rappresentazioni equivalenti gestibili da pianificatori già esistenti. Tutte le compilazioni proposte sono polinomiali, corrette, complete e preservano la dimensione della soluzione. Un'analisi sperimentale mostra che gli approcci proposti ottengono prestazioni migliori rispetto ad altre tecniche all'avanguardia nella maggior parte dei domini di pianificatione considerati.

Abstract

A significant challenge in Artificial Intelligence (AI) is to design autonomous systems that can act in complex environments to achieve a final objective while adhering to many constraints. This challenge is tackled by Automated Planning, which is a field of AI that deals with the problem of synthesizing a sequence of actions, i.e. a plan, to achieve a set of goals. A significant body of research focuses on classical planning models, where goals are defined as conditions that must hold in the final state reached by the plan execution. However, there are some major limitations to this approach; many goals of autonomous agents may involve the maintenance of a safety condition over the whole plan execution, reactions to certain inputs within a limited time frame, and intermediate objectives to achieve. Moreover, multiple constraints may dictate the agent's behavior and influence the structure of desired plans. This thesis addresses the problem of planning under these specifications and explores three different ways of formulating these constraints. Firstly, we consider state trajectory constraints as defined in PDDL3, one of the most popular planning languages. Secondly, we introduce a new formalism for expressing trajectory constraints over actions rather than traversed states. Lastly, we consider temporally extended goals expressed in Pure-Past Linear Temporal Logic. For each formalism, we propose a compilation schema to translate planning problems with these specifications into equivalent representations that can be handled by off-the-shelf classical planners. All proposed compilations are polynomial, sound, complete, and preserve the size of the solution. Experimental analysis shows that the proposed approaches perform better than other state-of-the-art techniques in the majority of the considered benchmark domains.

Chapter 1

Introduction

“Artificial Intelligence” (AI) is the discipline that studies rational agents, and an agent is rational if it does the “right thing”. Rational agents are capable of gathering and interpreting data to reach a complex objective. Moreover, by reasoning on the knowledge acquired from the environment, such systems can choose the correct actions to reach the final goal. Rational agents are capable of reasoning on symbolic rules or can learn numeric models and can adapt their behavior by analyzing how the environment responds to their actions. Moreover, rational agents must be able to act autonomously, that is, without human intervention [Russell and Norvig, 2010].

This Thesis focuses on the reasoning aspect of rational agents. In particular, we address the challenge of designing rational systems that can act autonomously in complex environments to achieve a final objective while adhering to many constraints. This challenge is tackled by Automated Planning, which is a field of AI that deals with the problem of synthesizing a sequence of actions, i.e. a plan, to achieve a set of goals. A significant body of research focuses on classical planning models, where goals are defined as conditions that must hold in the final state reached by the plan execution. Although this type of solution may be adequate to capture many theoretical problems, real-world scenarios often involve complex temporal constraints that must be taken into account when computing the plan. For example, many goals of autonomous agents may involve the maintenance of a safety condition over the whole plan execution, reactions to certain inputs within a limited time frame, and intermediate objectives to achieve [Bacchus and Kabanza, 1998].

To overcome these limitations of classical planning, many previous works address the problem of planning with goals that are *temporally extended* [Bacchus and Kabanza, 1998, 2000], with *trajectory constraints* [Gerevini et al., 2009] and with *temporal preferences* [Gerevini et al., 2009, Baier and McIlraith, 2008]. Temporally extended goals and trajectory constraints are temporal properties that every valid plan must satisfy, while preferences represent desirable properties that a good quality plan would satisfy.

From a representation point of view, many formalisms have been studied to express trajectory constraints and temporally extended goals. Linear Temporal Logic [Pnueli, 1977] over finite traces ($\textsc{ltl}_f$) [De Giacomo and Vardi, 2013] is the most popular language for expressing temporally extended goals, while PDDL3 is a planning language that standardized constraints and preferences over the state trajectories generated by a plan. Recently, Pure-Past Linear Temporal Logic [De Giacomo et al., 2020], a variation of $\textsc{ltl}_f$ that predicates on past events, has been considered for expressing temporally extended goals. Other formalisms, such as $\mathcal{LPP}$, focus on temporal preferences over states and action occurrences [Bienvenu et al., 2011].

From a computational perspective, handling these types of constraints requires the design of dedicated planning systems [Bacchus and Kabanza, 1998, Benton et al., 2012, Hsu et al., 2007] or compilation approaches [Baier and McIlraith, 2006a,b, Edelkamp et al., 2006a, Torres and Baier, 2015, Wright et al., 2018, Percassi and Gerevini, 2019]. Compilation is a technique that works by reformulating a planning problem with temporal constraints into a new equivalent problem without them. By adopting compilation, we can exploit existing classical planners for solving the compiled problems, and there is a clear separation between the module that handles the constraints from the search engine, making this approach more modular and extensible. However, compilation significantly increases the size of the resulting problem and this makes the overall planning process more challenging. Therefore, a critical challenge is to design compilation approaches that minimize the overhead introduced in the resulting planning problems.

This Thesis explores three different ways of formulating these constraints. Firstly, we consider PDDL3 state trajectory constraints. Secondly, we introduce a new formalism, called PAC, to express trajectory constraints on actions rather than traversed states. Lastly, we consider temporally extended goals expressed in Pure-Past Linear Temporal Logic. For each of these formalisms, we propose a compilation schema and various optimizations thereof to translate planning problems with these specifications into equivalent representations that can be handled by off-the-shelf classical planners. Moreover, we formally prove that the resulting compilations are polynomial, sound, complete, and preserve the size of the solution. Each compilation system is thoroughly tested in an in-depth experimental analysis involving novel benchmarks specifically designed to test the scalability and effectiveness of each approach. The results show that the proposed approaches perform better than other state-of-the-art techniques in the majority of the considered benchmark domains.

1.1 Thesis Outline and Contributions

This section details the structure of this Thesis and reports the contributions in terms of publications, tools developed, and the design of new benchmarks featuring trajectory constraints and temporally extended goals.

1.1.1 Outline

Chapter 2

This chapter provides preliminary notions on classical planning problems. In particular, Section 2.1 gives an overview of PDDL [McDermott et al., 1998], the standard language used to formalize classical planning problems. This section also provides an intuitive explanation of some advanced topics, such as planning with first-order specifications and how to obtain a ground representation of a planning problem. Finally, Section 2.2 formally defines classical planning problems.

Chapter 3

In this chapter, we formally review state-of-the-art languages for expressing temporally extended goals and trajectory constraints in planning problems. Section 3.1

formally defines LTL$_f$ and PPLTL, two of the most popular temporal logic formalisms to express temporally extended goals. Section 3.2 formalizes the class of PDDL3 state trajectory constraints that are called "qualitative" [Gerevini et al., 2009]. Section 3.3 formalizes PAC, which is a new language for expressing temporal trajectory constraints over action sequences rather than states. This chapter also provides known relations between LTL$_f$, PPLTL, PDDL3, and PAC.

Chapter 4

This chapter reviews the state-of-the-art approaches to handle temporally extended goals in LTL$_f$ and PDDL3 state trajectory constraints. Section 4.1 provides background notions on Nondeterministic Finite-State Automatons, which are the basis of many techniques discussed in this chapter. Section 4.2 reviews an automata-based compilation for handling LTL$_f$ goals which is worst-case exponential and solution size preserving, while Section 4.3 shows a compilation technique to handle LTL$_f$ goals which is polynomial but increases the size of solution plans. Lastly, Section 4.4 reviews native planning systems supporting PDDL3 state trajectory constraints.

Chapter 5

In this chapter, we propose a novel compilation approach, named TCORE, for handling classical planning problems with PDDL3 constraints. TCORE uses the notion of regression to determine the relationship between each action and each trajectory constraint. By doing so, and with the introduction of a few additional variables, TCORE efficiently compiles a PDDL3 planning problem into a classical planning problem that can be solved by any off-the-shelf classical planner supporting disjunctive preconditions and conditional effects. Section 5.1 details the TCORE compilation schema and presents formal results on the soundness and completeness of the approach. Section 5.2 presents an experimental analysis involving TCORE, state-of-the-art compilation approaches supporting LTL$_f$ goals, and native PDDL3 planners.

Chapter 6

This chapter presents a technique for planning with PAC constraints via compilation to classical planning. The compilation approach, named PAC-C, works by extending the model of actions to (I) prevent the execution of actions that would violate constraints encoding safety conditions, and (II) force the planner to schedule all actions that must be executed to satisfy PAC constraints. The result is an encoding that is polynomial, sound and complete, and that preserves the size of the solution plans. Section 6.1 presents the PAC-C encoding and details some theoretical results of the compilation. Section 6.2 experimentally shows the usefulness of PAC in the context of expressing control knowledge to improve solution coverage and plan quality of a classical planner. This Section illustrates how we designed our benchmark suite featuring control knowledge expressed as PAC action constraints. Moreover, we show how the same knowledge can be expressed in LTL_f and PDDL3 (when possible), and we report an experimental comparison between PAC-C, a state-of-the-art LTL_f compilation approach, and TCORE.

Chapter 7

This chapter presents an encoding to translate problems with temporally extended goals in PPLTL into classical planning problems. The encoding, called PLAN4PAST, exploits the observation that to evaluate a PPLTL formula it is sufficient to consider (I) only a relevant subset of temporal formulas, and (II) only the current step and the previous step. Leveraging these observations, we devise an efficient encoding that uses only a few additional variables, at most linear in the size of the PPLTL goal, is sound, complete, and does not add any spurious action. As a result, we obtain a classical planning problem that can be solved by any classical planner, such as LAMA Richter and Westphal [2010], supporting conditional effects and derived predicates. Section 7.2 details the PLAN4PAST encoding and formally proves the correctness of the approach. Section 7.3 empirically compares PLAN4PAST with the state-of-the-art encodings for LTL_f temporally extended goals on a set of benchmarks featuring semantically equivalent LTL_f and PPLTL formula.

1.1.2 Publications and Contributions

The research outlined in this Thesis stems from different publications that have been presented and published at prestigious AI conferences. In particular, the work related to PDDL3 constraints has been published in Bonassi et al. [2021] and Bonassi et al. [2022b]. The TCORE compilation system is publicly available at `https://github.com/LBonassi95/tcore`, while all benchmarks featuring PDDL3 and LTL$_f$ constraints employed for the experimental analysis are publicly available at `https://github.com/LBonassi95/Benchmarks-ICAPS-2021`. Moreover, the work on PDDL3 presented in this thesis has been further extended to other forms of planning in Bonassi et al. [2024].

The PAC language for expressing action trajectory constraints and PAC-C have been presented in Bonassi et al. [2022a]. The PAC-C compilation has been implemented in Python and is publicly available at `https://github.com/LBonassi95/PAC-C`, while the newly designed benchmarks featuring domain knowledge expressed in PAC, PDDL3 and LTL$_f$ are publicly available at `https://github.com/LBonassi95/PAC-benchmarks-IJCAI22`.

Lastly, the work on Pure-Past Temporal Logic has been done in collaboration with Francesco Fuggitti, Giuseppe de Giacomo, and Marco Favorito. In particular, the work on the PLAN4PAST (`https://github.com/whitemech/Plan4Past`) encoding is one of the main contributions of Francesco Fuggitti's dissertation [Fuggitti, 2023]. The publications related to this work are Bonassi et al. [2023a] and Bonassi et al. [2023b]. My Thesis complements Fuggitti's dissertation in the formalization of some theoretical results, in the design of the experimental analysis, and in the design of the temporal logic formulas employed in the benchmarks, which can be found at `https://github.com/whitemech/Plan4Past-data`.

Chapter 2

Background on Classical Planning

Automated planning is a discipline that, given an initial state of the world, addresses the problem of finding a sequence of actions (a plan) to achieve the specified goals. The action sequence is computed from a formal description of the planning model which is called the *planning problem*. The most popular and standard language used to define planning problems is the Planning Domain Definition Language (PDDL) [McDermott et al., 1998]. PDDL is a knowledge representation language designed to capture rich planning models in a natural and compact way. The first version of PDDL was presented during the first edition of the *International Planning Competition* (IPC) in 1998 [McDermott et al., 1998]. In this first iteration, the PDDL language is based on the notions of propositional and first-order logic and allows one to model the class of so-called *classical planning* problems. Over the next years, PDDL has evolved to support more complex planning models; in version 2.1 [Fox and Long, 2003], PDDL was extended with numeric variables and durative actions to capture numeric and temporal planning problems. Version 2.2 [Edelkamp and Hoffmann, 2004] introduced derived predicates and axioms, while version 3 of PDDL [Gerevini et al., 2009] formalized preferences and constraints as a subclass of temporal logic formulas [Pnueli, 1977, De Giacomo and Vardi, 2013] over state trajectories.

This Thesis heavily relies on the concepts of classical planning problems. As such, this Chapter is dedicated to the background notions of classical planning, and it is divided into two sections:

1. Section 2.1 provides an informal overview of the class of classical planning problems that can be specified in PDDL. The purpose of this section is to familiarize the reader with the foundational concepts of PDDL, such as actions, states, and the first-order constructs employed in PDDL. Moreover, this section introduces the distinction between first-order and ground representations of a planning problem.
2. Section 2.2 provides the formal definitions of a classical planning problem and its components.

2.1 Classical Planning in PDDL

PDDL uses many notions of first-order logic notions to compactly represent planning problems. We will illustrate how these notions are employed using a running example featuring the well-known Blocksworld domain. For a complete treatment of first-order logic and how it is used in planning, we refer the reader to Russell and Norvig [2010].

In Blocksworld, there is a set of blocks on a table, and the objective is to arrange the blocks in a particular configuration using a robotic arm. The arm can pick up a block, put it on the table, or stack it on top of another block. The goal is to create one or more towers of blocks. In PDDL, the main components of a planning problem are objects, predicates, action schemas, initial state, and goal.

Objects, predicates, and state representation. One of the main concepts of PDDL is the state, which is a structure that is meant to capture the current status of the world. A planning state is derived from the objects and the predicates of planning problems. Objects are a set of unique symbols. In Blocksworld, objects are used to represent the available blocks, which in our example are a, b, c. Predicates are first-order logic structures that define the relations between objects. For example, in Blocksworld, the predicate $On(?x, ?y)$ defines a relation between two generic objects $?x$ and $?y$. As the name suggests, $On(?x, ?y)$ captures whether block $?x$ is currently placed over block $?y$. In Blocksworld, we usually have the following predicates:

- $On(?x, ?y)$ - captures whether block $?x$ is on top of block $?y$.
- $Ontable(?x)$ - captures whether block $?x$ is on the table.
- $Clear(?x)$ - captures whether block $?x$ is clear (no other block is on top of $?x$).
- $Handempty$ - captures whether the arm is not holding any block.
- $Holding(?x)$ - captures whether the arm is holding block $?x$.

A predicate compactly represents a set of *ground atoms*. A ground atom is a propositional symbol obtained by substituting every variable of a predicate with an object and represents a Boolean variable that can be either true or false. For example, if we substitute $?x$ with a and $?y$ with b in the predicate $On(?x, ?y)$ we obtain the ground atom $On(a, b)$. In planning, these ground atoms are often called either atoms or fluents. Furthermore, we use the term *literal* to refer to an atom f (positive literal) or to a negated atom $\neg f$(negative literal).A state is a collection of atoms that are currently true. For example, the state

$$s_1 = \{On(a, b), Ontable(b), Holding(c), Clear(a)\}$$

tells us that: a is over b, no block is over a, b is on the table, and the arm is holding c. Figure 2.1 shows a graphical representation of this state. In planning, the closed-world assumption holds; that is, all atoms that do not belong to a state are considered false. In our example, $Ontable(c)$ is false in s_1. Every planning problem has an initial state, representing the initial status of the world. In this case, the initial state tells us the initial configuration of the blocks.

Action schemas and ground actions. Action schemas provide the definition of the actions that can be executed. In Blocksworld we have the following action schemas:

- $Pick\text{-}Up(?x)$ - to pick up the block $?x$ from the table.
- $Put\text{-}Down(?x)$ - to put down the block $?x$ on the table.

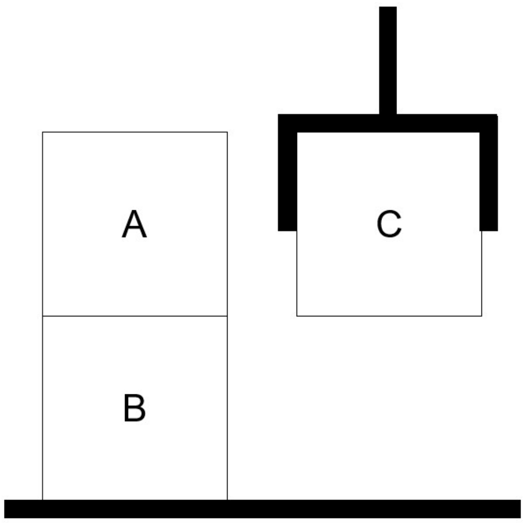


Figure 2.1: A graphical representation of a state of the Blocksworld domain.

- *Stack*$(?x, ?y)$ - to stack the block $?x$ on top of block $?y$.
- *Unstack*$(?x, ?y)$ - to un-stack the block $?x$ that was over $?y$.

Each action schema has a name, a set of variables, a precondition, and a set of effects. For example, the action *Stack*$(?x, ?y)$ can be formalized as follows:

Action: *Pick-Up*$(?x)$
Precondition: $Clear(?x) \wedge Ontable(?x) \wedge Handempty$
Effect:$\{\neg Handempty, \neg Clear(?x), \neg Ontable(?x), Holding(?x)\}$

The name of the action is *Pick-Up* and the only variable is $?x$. The precondition represents a property that must be met before executing an action. To pick up a block $?x$ from the table, we must have that the arm is not currently holding a block, $?x$ is on the table and no block is over $?x$. Effects describe how the action induces the successor state. Intuitively, after *Pick-Up*$(?x)$, the hand will be empty, $?x$ will not be clear (because in the arm), $?x$ will be removed from the table, and the arm will hold $?x$. The notion of action schema should not be confused with the concept of "action"; the schema is a compact representation of a group of *ground actions.*

Ground actions, or simply actions, are obtained by replacing all variables of an action schema with objects. For example, by replacing $?x$ with a we obtain the action:

> Action: $\mathit{Pick\text{-}Up}(a)$
>
> Precondition: $\mathit{Clear}(a) \land \mathit{Ontable}(a) \land \mathit{Handempty}$
>
> Effect:$\{\neg \mathit{Handempty}, \neg \mathit{Clear}(a), \neg \mathit{Ontable}(a), \mathit{Holding}(a)\}$.

Actions define *exactly* how the new successor state is generated and the preconditions that must be met. The precondition is a first-order formula, while the effect is a set of literals. An action with precondition ϕ can be executed in a state s if and only if $s \models \phi$. The symbol "$\models$" is the logical entailment operator, and we say that "s satisfies ϕ" to indicate $s \models \phi$. Since the states are complete (that is, a state assigns a truth value to each atom), it is always possible to determine whether a state s satisfies ($\models$) or does not satisfy ($\not\models$) a formula ϕ. Effects describe how a new state is generated in terms of what changes after the action. Specifically, executing an action in a state s induces a new state s', which is obtained starting from the set of atoms formed by s, removing the negative literals in the action's effects, and adding the positive literals mentioned in the action's effect. As we can see, $\mathit{Pick\text{-}Up}(a)$ removes (that is, makes false) $\mathit{Handempty}$, $\mathit{Clear}(a)$, $\mathit{Ontable}(a)$ from a state s, and adds (that is, makes true) $\mathit{Holding}(a)$ to a state s.

Goal. The goal is a first-order formula representing the objective to achieve. It could be a simple condition such as "Create the tower *a*-*b*-*c*" or a complex condition such as "Place any block on top of *c*". To represent such complex goals, PDDL allows the use of first-order quantifiers $\forall$ (for all) and $\exists$ (exists). For example, we could capture the complex goal with the formula $\exists\, ?x \cdot \mathit{On}(?x, c)$. First-order quantifiers can also be used to compactly specify the preconditions of actions.

Solutions. A solution to a planning problem is a sequence of actions (that is, a plan) that transforms the initial state s_I into a state s_G that satisfies the goal. Suppose that, initially, every block is on the table, that is, $s_I = \{\mathit{Ontable}(a)$, $\mathit{Ontable}(b)$, $\mathit{Ontable}(c)$, $\mathit{Clear}(a)$, $\mathit{Clear}(b)$, $\mathit{Clear}(c)$, $\mathit{Handempty}\}$. A solution π to achieve the goal $G = \exists\, ?x \cdot On(?x, c)$ could be:

$$\pi = \langle \textit{Pick-Up}(a), \textit{Stack}(a, c) \rangle$$

Indeed, after executing $\textit{Pick-Up}(a)$ and $\textit{Stack}(a, c)$ we will reach a state where $On(a, c)$ is true, satisfying the goal G.

This discussion covers the fundamental concepts of classical planning in PDDL. Moving forward, we delve into an analysis of two advanced features of PDDL known as *conditional effects* and *derived predicates*. Subsequently, we conclude by exploring the methodologies employed by modern planners to compute the ground representation of a planning problem.

2.1.1 Conditional Effects

Conditional effects are constructs that allow one to naturally model actions whose behavior depends on the execution state. A conditional effect is a pair $c \triangleright e$ where c is a first-order formula and e is a PDDL effect. When we execute an action in a state s with a conditional effect $c \triangleright e$, if the condition c is satisfied by s, then the effect e is considered in the generation of the successor state s'. Otherwise, when c is not satisfied, e is ignored. Conditional effects can be used to naturally capture more complex problems. For example, consider a version of Blocksworld where the arm has a battery and can function only when it is charged. In this alternative domain, some blocks are heavy, and picking up a heavy block consumes the battery. With conditional effects, we can easily express "If the arm picks up a heavy block, then the battery discharges" by rewriting the *Pick-Up* action schema as follows:

Action: $\textit{Pick-Up}(?x)$
    Precondition: $Clear(?x) \land Ontable(?x) \land Handempty \land Charged$
    Effect:$\{\neg Handempty, \neg Clear(?x), \neg Ontable(?x), Holding(?x),$
    $Heavy(?x) \triangleright \neg Charged\}$

This action schema says that when we pick up a block $?x$, if the condition $Heavy(?x)$ is satisfied, then the action has $\neg Charged$ as an effect. Otherwise, the arm remains charged. At the current state-of-the-art, many planning systems support conditional effects, making it a central feature of PDDL. Moreover, this Thesis heavily exploits conditional effects to deal with more complex planning specifications such as trajectory constraints.

2.1.2 Derived Predicates and Axioms

Derived predicates have been introduced in the 2.2 version of PDDL [Edelkamp and Hoffmann, 2004], and are a special type of first-order predicate representing a group of atoms whose truth is derived from the truth of other atoms in the current state via some rules called *axioms*. Derived predicates provide a natural way to capture complex formulas, and can be used to simplify the formulation of preconditions, conditions of conditional effects, and goals. However, unlike basic atoms, the truth of an atomic derived predicate in a state cannot be affected by the effects of an action, but is instead determined by axioms. Axioms are rules of the form $d \leftarrow \phi$, where d is a derived predicate and ϕ is a first-order formula, possibly involving other derived predicates. For example, we can naturally capture properties such as "block $?x$ is above block $?y$" with the $\mathit{Above}(?x, ?y)$ derived predicate, whose truth is derived by the following axiom:

$$\mathit{Above}(?x, ?y) \leftarrow \mathit{On}(?x, ?y) \vee \exists\, z \cdot \mathit{On}(?x, ?z) \wedge \mathit{Above}(?z, ?y).$$

Intuitively, this axiom defines that $?x$ is above $?y$ iff either $?x$ is over $?y$ or there exists a block $?z$ such that $?x$ is over $?z$ and $?z$ is above $?y$. This rather simple example shows that we can formulate very complex axioms. However, there are some limitations; namely, we must restrict ourselves to a set of axioms that are *stratified*. Without entering in detail, a stratified set of axioms 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 involving derived predicates is satisfied by a state s. We refer the reader to Thiébaux et al. [2005] for a complete discussion on this topic. We conclude by remarking that, like for basic predicates and actions, it is possible to obtain propositional derived predicates (i.e., derived atoms) and axioms by substituting with objects all variables of the first-order derived predicates and axioms.

2.1.3 Ground Representation

Before searching for a solution, many planning approaches transform the first-order representation of a planning problem into a *ground* representation. In the literature, a ground problem is obtained by substituting (or instantiating) all action schemas with ground actions and by substituting all predicates with atoms. This could be done by enumerating all possible substitutions of variables in actions and predicates with the objects. This very expensive process is often performed rather efficiently by using state-of-the-art grounding techniques [Helmert, 2009].

These approaches aim at determining the set of actions and atoms that are *reachable*. A reachable atom is a proposition that is true in some reachable state, while reachable actions are those that can be executed in some reachable state. A state s is said to be reachable if there exists a sequence of actions that lead to s starting from the initial state of a problem. Clearly, actions and atoms that are not reachable can be safely deleted from the planning model; no solution will ever contain a non-reachable action nor will make true a non-reachable atom. However, in planning, determining whether an action's precondition or an atom can be achieved in some reachable state is as difficult as solving the original planning problem. Therefore, state-of-the-art approaches perform a reachability analysis on a simplified representation of a planning problem, which is obtained with the so-called *delete-free* relaxation. With this relaxation, grounders can *directly* determine the set of reachable atoms and actions without considering those that are not (in the relaxed representation). For example, the FastDownward grounder [Helmert, 2009] implements this relaxation technique and can compute compact ground representations very efficiently. For example, in the well-known Logistics planning domain, the naive (brute-force) grounding approach generates $5.82 \cdot 10^{10}$ ground actions, while FastDownward cuts the number of possible actions to $1.51 \cdot 10^{5}$. A similar procedure is also employed to produce the set of reachable ground axioms and derived predicates. For the remainder of the Thesis, we will often adopt a ground representation of a planning problem, that is:

1. Action schemas are replaced with ground actions.

2. Basic predicates are substituted with atoms.

3. Derived predicates are substituted with ground derived predicates.

4. Axioms are substituted with propositional axioms.

5. First-order formulas are rewritten as propositional formulas.

Regarding the last point, we remark that in planning it is always possible to rewrite a first-order formula with quantifiers as a propositional formula. This is done by rewriting universal quantifiers as conjunctions and existential quantifiers as disjunctions. For example, if we have three blocks $\{a, b, c\}$ then the formula $\forall\, ?x \cdot \mathit{Ontable}(?x)$ can be rewritten as the equivalent $\mathit{Ontable}(a) \wedge \mathit{Ontable}(b) \wedge \mathit{Ontable}(c)$.

2.2 Classical Planning Problems

This section formally defines the components of a classical planning problem. Our definitions inherently assume a ground formulation.

Definition 1 (Classical planning problem)**.** *A classical planning problem is a tuple $\Pi = \langle F, D, X, A, I, G, \mathit{Pre}, \mathit{Eff} \rangle$ where F is a set of fluents (atomic propositions), D is a set of derived predicates, X is a set of axioms, A is a set of action labels, $I \subseteq F$ is the initial state, G is a formula over F representing the goal of the problem, and Pre and Eff are two functions mapping an action a to a precondition and a set of conditional effects, respectively.*

As usual, following a closed world assumption, a planning state s is represented as a set of fluents, with the meaning that an atomic proposition f holds true in s if $f \in s$; otherwise, f is false in s. Moreover, we use $\mathit{Lit}(F)$ to indicate the set of literals that can be obtained by F, that is, $\mathit{Lit}(F) = F \cup \{\neg f \mid f \in F\}$. Axioms have the form $d \leftarrow \psi$ where $d \in D$ and ψ is a formula over $F \cup D$. 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 X. We assume that the set of axioms X is *stratified* [Thiébaux et al., 2005]; as discussed in Section 2.1.2, a stratified set of axioms 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 $F \cup D$ is satisfied by a state s. Throughout this Thesis, for the sake of clarity and succinctness, we will write $\Pi = \langle F, A, I, G, \mathit{Pre}, \mathit{Eff} \rangle$ to indicate a planning problem Π without axioms and derived predicates, as these components are employed only in some cases.

Actions are split into a set of labels A representing the unique name of the available actions and into two functions that store the preconditions and effects of each action. The precondition function $\mathit{Pre} : A \rightarrow 2^{2^{F \cup D}}$[1] maps an action label a to a formula $\mathit{Pre}(a)$ representing the preconditions of A, whereas $\mathit{Eff} : A \rightarrow 2^{2^{F \cup D}} \times 2^{\mathit{Lit}(F)}$ is a function mapping an action label a to a set of conditional effects.

Definition 2 (Conditional effect)**.** *Let F be a set of fluents. A conditional effect is a pair $c \triangleright e$, where $c \in 2^{2^{F \cup D}}$ is a formula over $F \cup D$, and $e \in 2^{\mathit{Lit}(F)}$ is a set of literals from F. With e^- and e^+, we indicate the partition of e that features only negative and positive literals, respectively.*

An action a can be executed in a state s only if $s \models \mathit{Pre}(a)$, while 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' where $\forall f \in F$, f holds true in s^i if and only if either (I) f was true in s and no conditional effect $c \triangleright e$ triggered in s deletes it ($\neg f \in e^-$) or (II) there is a conditional effect $c \triangleright e$ triggered in s that adds it ($f \in e^+$).

Definition 3. *Let a be an action and s a state. a is applicable in s if $s \models \mathit{Pre}(a)$, and the application of a in s yields the state s' defined as follows:*

$$s' = (s \setminus \bigcup_{\substack{c \triangleright e \in \mathit{Eff}(a) \\ \text{with } s \models c}} e^-) \cup \bigcup_{\substack{c \triangleright e \in \mathit{Eff}(a) \\ \text{with } s \models c}} e^+.$$

We indicate with $s' = s[a]$ the state resulting from applying a in s, and, similarly to other works [Röger et al., 2014], we assume a delete-before-adding semantics. That

[1]Note that with $F \cup D$ propositional symbols we can define $2^{2^{F \cup D}}$ semantically different formulas.

is, we compute the successor state by applying all the negative partitions e^- of effects before the positive partitions e^+. Conditional effects $c \triangleright e$ without condition, that is, when $c = \top$, are called *simple effects* and are simply denoted by e.

The execution of an action in a state s induces a new state s'. Thus, the execution of a plan $\pi = \langle a_0, \ldots, a_{n-1} \rangle$ induces a sequence of states $\tau = \langle s_0 = I, \ldots, s_n \rangle$, which we call *state trajectory*. A solution is a plan π that induces a state trajectory τ such that the last state s_n of τ satisfies the goal G.

Definition 4 (Solution)**.** *Let* $\Pi = \langle F, D, X, A, I, G, Pre, Eff \rangle$ *be a classical planning problem. Then a plan* $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ *is a solution for* Π *iff there exists a sequence of states (state trajectory)* $\tau = \langle s_0, s_1, ..., s_n \rangle$ *such that:*

1. $s_0 = I$

2. $\forall\, i \in [1, \ldots, n]\ s_i \models Pre(a_i)$ *and* $s_{i+1} = s_i[a_i]$

3. $s_n \models G$

In many cases, we are interested in solutions of good quality, that is, in solutions with the minimum cost. In this Thesis, we assume that actions have unit cost. Therefore, the cost of a solution π, denoted with $c(\pi)$, is equal to the number of actions in π, that is, $c(\pi) = |\pi|$. A solution π is said to be optimal iff no solution π' with $c(\pi') < c(\pi)$ exists.

2.2.1 Regression for Classical Planning

Regression is a theorem-proving mechanism that has been introduced in the *situation calculus* [Levesque et al., 1998]. In classical planning, regression allows systematic reasoning about actions and provides a mechanism to compute the condition that a state must satisfy to guarantee the satisfaction of another condition in the successor state [Rintanen, 2008]. Inspired by this notion of regression, we introduce an operator R whose objective is to regress a formula over the effects of an action. In this Thesis, we define regression on formulas that are in *Negation Normal Form* (NNF), that is, propositional formulas where the operands of negation are fluents only. Transforming a formula into NNF is a linear operation that we assumed performed automatically when needed.

Definition 5 (Regression operator)**.** *The regression $R(\phi, a)$ of a NNF formula ϕ through the effects $\mathit{Eff}(a)$ of action a is the formula obtained from ϕ by replacing every atom f in ϕ with $\Gamma_f(a) \vee (f \wedge \neg\Gamma_{\neg f}(a))$, where $\Gamma_l(a)$ for a literal l is defined as*

$$\Gamma_l(a) = \bigvee_{\substack{c \rhd e \,\in\, \mathit{Eff}(a) \\ \text{with } l \in e}} c.$$

In Definition 5, the condition $\Gamma_f(a)$ captures when the action a makes the atom f true. Instead, $\Gamma_{\neg f}(a)$ represents the conditions under which a falsifies f. Therefore, the action a makes f true in the next state only if either $\Gamma_f(a)$ holds, or f is true in the current state and $\Gamma_{\neg f}(a)$ is false. By replacing every atom f in a formula ϕ with $\Gamma_f(a) \vee (f \wedge \neg\Gamma_{\neg f}(a))$ we obtain another formula $R(\phi, a)$ that holds in the current state if and only if ϕ holds in the successor state induced by a.

Theorem 1 (Rintanen [2008])**.** *Let F be a set of fluents, ϕ be a formula over F, and a be an action. Then $s[a] \models \phi$ iff $s \models R(\phi, a)$.*

Notice that the notion of regression we use does not include the action's preconditions, as we are only interested in capturing when the *execution* of the action makes a formula true.

Example 1. *Consider the the action act with:*

$$\mathit{Eff}(\mathit{act}) = \{b, c \rhd d, e \vee f \rhd h, g \rhd \neg h\}$$

Then we have that:

- *$R(b, \mathit{act}) = \Gamma_b(\mathit{act}) \vee (b \wedge \neg\Gamma_{\neg b}(\mathit{act})) = \top \vee (b \wedge \neg\bot) = \top$ (notice that b is an abbreviation for the simple effect $\top \rhd b$).*
- *$R(d, \mathit{act}) = \Gamma_d(\mathit{act}) \vee (d \wedge \neg\Gamma_{\neg d}(\mathit{act})) = c \vee (d \wedge \neg\bot) = c \vee d$.*

- $R(h, act) = \Gamma_h(act) \vee (h \wedge \neg\Gamma_{\neg h}(act)) = e \vee f \vee (h \wedge \neg g)$.
- $R(d \wedge h, act) = R(d, act) \wedge R(h, act)$.
- $R(i, act) = \Gamma_i(act) \vee (i \wedge \neg\Gamma_{\neg i}(act)) = \bot \vee (i \wedge \neg\bot) = i$.

Notice how, as in the last case, if a literal is unaffected by the effects of an action, then the regression leaves the literal unaltered.

Chapter 3

Planning with Trajectory Constraints

A significant body of research in planning focuses on solving classical planning problems, where goals are defined as conditions on a final state. In this setting, any plan that achieves the goal is deemed to be an acceptable solution. However, there are some major limitations to this approach. Firstly, not all goals are "final state" goals; some involve maintenance or reaction, where the agent must keep a certain condition or react within a limited time frame to a condition. These are significant goals that cannot be achieved in a particular final state. Second, we may have many constraints to control how the agent behaves and how the solution plans are structured that cannot be expressed using final state goals ([Bacchus and Kabanza, 1998, Gerevini et al., 2009]).

To overcome these limitations of classical planning, many previous works address the problem of planning with goals that are *temporally extended* [Bacchus and Kabanza, 1998, 2000, Baier and McIlraith, 2006b,a, Torres and Baier, 2015] or with *trajectory constraints* [Gerevini et al., 2009]. All of these works try to address the problem of finding solutions to satisfy conditions that are *Non-Markovian.* In general, we can say that a condition is Markovian when all the information needed to determine the truth of the condition is encoded within the state itself [Bacchus et al., 1996, 1997]. Indeed, the classical planning model is Markovian. The action's out-come is uniquely determined by the current state, and the goal can be evaluated only on the last state of the trajectory; all other states can be ignored. However, a non-Markovian condition must be evaluated taking into consideration the whole trajectory of stats or the whole plan.

In the current state of the art, non-Markovian goals and constraints are formulated using different temporal logic languages. The aim of this Thesis is to develop novel techniques to handle:

- The class of state trajectory constraints defined by PDDL3.
- Pure-Past Linear Temporal Logic (PPLTL) temporally extended goals.
- A new class of constraints over sequences of actions scheduled in a plan. These action constraints are formalized using a new language called PAC, which is a contribution of this Thesis.

Moreover, when possible, we compare the proposed techniques with state-of-the-art approaches supporting our class of constraints. When this is not possible, we perform a cross-language analysis, that is, we express knowledge in different ways and evaluate which strategy is more effective. A formalism we consider in all the comparisons is Linear Temporal Logic over finite traces (LTL$_f$), one of the most popular languages for expressing temporally extended goals. Therefore, this chapter provides the definitions of LTL$_f$, PPLTL, PDDL3, and PAC. For the rest of the Thesis, for the sake of consistency with the previous literature, we will use the term "Temporally extended goals" when dealing with LTL$_f$ and PPLTL specifications, while we will use "trajectory constraints" in the context of planning with PDDL3 state trajectory constraints and with PAC action trajectory constraints.

The chapter is structured as follows. The first section provides the syntax and semantics of LTL$_f$ and PPLTL. We then define the class of PDDL3 constraints that are called *qualitative* state trajectory constraints [Gerevini et al., 2009] and show the relationship between these constraints and temporally extended goals. Lastly, we introduce PAC, the new language that we designed for expressing action trajectory constraints. Moreover, we discuss how PAC can express compelling properties that cannot be captured by either PDDL3 constraints or temporally extended goals.

3.1 Linear Temporal Logic and Pure-Past Linear Temporal Logic Over Finite Traces

3.1.1 Linear Temporal Logic Over Finite Traces

Linear Temporal Logic over finite traces (LTL$_f$) [De Giacomo and Vardi, 2013] is a formalism for expressing useful temporal specifications for controlling the structure of solutions in planning problems [Bacchus and Kabanza, 1998, Baier and McIlraith, 2006b,a, Torres and Baier, 2015] and for expressing control knowledge to scale up planning performances [Bacchus and Kabanza, 2000].

Given a set $\mathcal{P}$ of atomic propositions, the syntax of LTL$_f$ is defined as:

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

where $p \in \mathcal{P}$, X is the *next* operator and U is the *until* operator. Intuitively, $\mathsf{X}(\varphi)$ specifies that the formula φ must hold in the next step, while $\varphi_1 \,\mathsf{U}\, \varphi_2$ requires φ_1 to hold until eventually φ_2 holds.

LTL$_f$ formulas are interpreted on *finite nonempty* state trajectories $\tau = s_0 \dots s_n$ where s_i at instant i is a propositional interpretation over the alphabet $2^{\mathcal{P}}$. We denote the length of τ by $\mathsf{length}(\tau) = n{+}1$ and the last element of τ by $\mathsf{last}(\tau) = s_n$.

Given a state trajectory $\tau = s_0 \dots s_n$, we denote by $\tau_{i,j}$, with $0 \leq i \leq j \leq n$, the subtrajectory $s_i \dots s_j$ obtained from τ starting from position i and ending in position j. We define the satisfaction relation $\tau, i \models \varphi$, stating that φ holds at instant i, as follows:

- $\tau, i \models p$ iff $\mathsf{length}(\tau) \geq 1$ and $p \in s_i$ (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 $i < \mathsf{length}(\tau)$ and $\tau, i+1 \models \varphi$;
- $\tau, i \models \varphi_1 \,\mathsf{U}\, \varphi_2$ iff there exists k, with $i \leq k \leq \mathsf{length}(\tau)$ such that $\tau, k \models \varphi_2$ and for all j, with $i \leq j < k$, we have that $\tau, j \models \varphi_1$.

We also define the following common abbreviations: $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, the *eventually* operator $\mathsf{F}\varphi \equiv \mathit{true}\,\mathsf{U}\,\varphi$, the *always* operator $\mathsf{G}\varphi \equiv \neg\mathsf{F}\neg\varphi$, the *weak-next* operator $\mathsf{WX}\phi \equiv \neg\mathsf{X}\neg\phi$, and the *release* operator $\varphi_1\,\mathsf{R}\,\varphi_2 \equiv \neg(\neg\varphi_1\,\mathsf{U}\,\neg\varphi_2)$. These new operators are also called *derived* operators since they are defined using the basic LTL$_f$ operators. Furthermore, the semantics of derived operators follow from the semantics of X and U. That is,

- $\mathsf{F}(\varphi)$ specifies that φ must become true at some point of the trajectory. Formally, $\tau, i \models \mathsf{F}(\varphi)$ iff there exists k, with $i \leq k \leq \mathsf{length}(\tau)$ such that $\tau, k \models \varphi_2$.
- $\mathsf{G}(\varphi)$ requires φ to always hold in the future. Formally, $\tau, i \models \mathsf{G}(\varphi)$ iff for every k with $i \leq k \leq \mathsf{length}(\tau)$, $\tau, k \models \varphi$.
- $\mathsf{WX}(\varphi)$ specifies that the formula φ must hold in the next step, but only when the current step is not the last one. Formally, $\tau, i \models \mathsf{WX}(\varphi)$ iff $i = \mathsf{length}(\tau)$ or $\tau, i+1 \models \varphi$.
- $\varphi_1\,\mathsf{R}\,\varphi_2$ specifies that the formula φ_2 can become false only when φ_1 becomes true; if φ_1 never holds, then φ_2 must hold at each step of the trajectory. Formally, $\tau, i \models \varphi_1\,\mathsf{R}\,\varphi_2$ iff for every k with $i \leq k \leq \mathsf{length}(\tau)$, either $\tau, k \models \varphi_2$ or there exists j, with $i \leq j < k$, such that $\tau, j \models \varphi_1$.

A LTL$_f$ formula φ is *true* in τ, if $\tau, 0 \models \varphi$. Moreover, the formula $\mathsf{end} \equiv \neg\mathsf{X}(\mathit{true})$ expresses that the state trajectory has ended.

We denote by $\mathsf{sub}(\varphi)$ the set of all subformulas of φ obtained from the abstract syntax tree of φ [De Giacomo and Vardi, 2013]. For example, 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, $|\mathsf{sub}(\varphi)|$ defines the size of a LTL$_f$ formula φ.

Lastly, we define the language of a LTL$_f$ formula φ, denoted by $\mathcal{L}(\varphi)$, as the set of all state trajectories that satisfy φ, that is,

$$\mathcal{L}(\varphi) = \{\tau \in (2^{\mathcal{P}})^* \mid \tau \models \varphi\}.$$

Note that $(2^{\mathcal{P}})^*$ indicates the infinite set of possible state trajectories over $2^{\mathcal{P}}$. We conclude by showing how LTL$_f$ can capture some desirable properties.

Example 2. *Compelling* LTL$_f$ *patterns are:*

- $\mathsf{G}(request \implies \mathsf{F}(response))$ *means "Every request is eventually met with a response" [Torres and Baier, 2015].*
- $Priority(r_1, r_2) \rightarrow (\neg Delivered(r_2)\, \mathsf{U}\, Delivered(r_1) \wedge \mathsf{F} Delivered(r_2))$ *means "If* r_1 *has priority over* r_2*, then* r_2 *must be delivered, but not before* r_1*" [Baier and McIlraith, 2006a].*
- $\mathsf{G}(Request \implies \mathsf{X}(Response))$ *means "Every request is immediately met with a response" [De Giacomo et al., 2014].*
- $\neg(\mathsf{F}(Task_1) \vee \mathsf{F}(Task_2))$ *"Only one of task 1 or task 2 can be executed" [van der Aalst et al., 2009].*
- $\mathsf{F}(Task_1) \wedge \mathsf{F}(Taks_2)$ *"Both task 1 and task 2 must be eventually executed" [Baier and McIlraith, 2006a].*

3.1.2 Pure Past Linear Temporal Logic

Pure-Past Linear Temporal Logic (PPLTL) is the variant of LTL$_f$ that talks about the past instead of the future. PPLTL has been recently surveyed in [De Giacomo et al., 2020], where it is denoted as PLTL$_f$. Given a set $\mathcal{P}$ of propositions, PPLTL is defined as:

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

where $p \in \mathcal{P}$, Y is the *yesterday* operator and S is the *since* operator. Intuitively, $\mathsf{Y}(\varphi)$ specifies that the formula φ must hold in the previous step, while $\varphi_1\, \mathsf{S}\, \varphi_2$ is satisfied when φ_1 held since φ_2 became satisfied.

PPLTL formulas are also interpreted on finite non-empty state trajectories $\tau = s_0 \cdots s_n$. Therefore, we define the satisfaction relation $\tau, i \models \varphi$, stating that φ holds at instant i, as follows:

- $\tau, i \models p$ iff $\mathsf{length}(\tau) \geq 1$ and $p \in s_i$ (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 $i \geq 1$ and $\tau, i-1 \models \varphi$;
- $\tau, i \models \varphi_1 \,\mathsf{S}\, \varphi_2$ iff there exists k, with $0 \leq k \leq i < \mathsf{length}(\tau)$ such that $\tau, k \models \varphi_2$ and for all j, with $k < j \leq i$, we have that $\tau, j \models \varphi_1$.

In addition, we define the following common abbreviations: $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, the *once* operator $\mathsf{O}\varphi \equiv \mathit{true}\,\mathsf{S}\,\varphi$, the *historically* operator $\mathsf{H}\varphi \equiv \neg\mathsf{O}\neg\varphi$, the *weak-yesterday* operator $\mathsf{WY}\phi \equiv \neg\mathsf{Y}\neg\phi$. The semantics of these derived temporal PPLTL operators are the following:

- $\mathsf{O}(\varphi)$ requires that φ held at some point in the past. Formally, $\tau, i \models \mathsf{O}(\varphi)$ iff there exists k, with $0 \leq k \leq i$, such that $\tau, k \models \varphi$.
- $\mathsf{H}(\varphi)$ is satisfied when φ always held in the past. Formally, $\tau, i \models \mathsf{H}(\varphi)$ iff for every k with $0 \leq k \leq i$, $\tau, k \models \varphi$.
- $\mathsf{WY}(\varphi)$ specifies that the formula φ must hold at the previous step, but only when the current step is not the first. Formally, $\tau, i \models \mathsf{WY}\varphi$ iff $i = 1$ or $\tau, i-1 \models \varphi$.

PPLTL formulas are evaluated by looking at the state trajectory from the last state back to the first. That is, a PPLTL formula φ is *true* in τ if $\tau, n \models \varphi$. Moreover, the formula $\mathsf{start} \equiv \neg\mathsf{Y}(\mathit{true})$ expresses that the state trajectory has started.

As done for LTL$_f$, we denote by $\mathsf{sub}(\varphi)$ the set of all subformulas of φ obtained from the abstract syntax tree of φ. Also in this case, $|\mathsf{sub}(\varphi)|$ defines the size of a PPLTL formula φ. Moreover, we define the language of a PPLTL formula φ, denoted by $\mathcal{L}(\varphi)$, as the set of all state trajectories that satisfy φ, that is,

$$\mathcal{L}(\varphi) = \{\tau \in (2^{\mathcal{P}})^* \mid \tau \models \varphi\}.$$

Example 3 reports some compelling properties that can be expressed in PPLTL.

Example 3. *Useful* PPLTL *formulas are:*

- $\mathsf{H}(Bus \implies \mathsf{Y}(\neg Bus \,\mathsf{S}\, Ticket))$ *means "Every time I took the bus, I bought a new ticket beforehand" [De Giacomo et al., 2020].*

- $G \wedge \neg\mathsf{Y}(\mathsf{O}(G))$ *means "The current state is the first that satisfies G" [Bacchus et al., 1996].*

- $\mathsf{O}(Task_1 \wedge \mathsf{Y}(\mathsf{O}(Task_2))$ *means "Task 1 was accomplished, and before that Task 2 was accomplished" [Bonassi et al., 2023b].*

- $\mathsf{H}(Task_1 \rightarrow \mathsf{Y}(\mathsf{O}(Task_2)))$ *"Task 1 has been accomplished strictly after task 2 has been accomplished" [Bonassi et al., 2023b].*

We conclude by remarking that, although PPLTL and LTL$_f$ have the same expressive power, translating a formula from one into the other (and vice versa) can be prohibitive since the best-known algorithm is 3EXPTIME [De Giacomo et al., 2020].

3.1.3 Planning For Temporally Extended Goals

Classical planning deals with the problem of finding a sequence of actions to achieve the goal in the last state, while the problem of finding a plan to satisfy a temporal formula is called planning for *temporally extended goals* [Bacchus and Kabanza, 1998].

Definition 6. *A planning problem with a temporally extended goal is a tuple $\Pi = \langle F, D, X, A, I, \varphi, Pre, Eff\rangle$, where F, D, X, A, I, Pre, Eff are defined as in a classical planning problem, and φ is a temporal formula over F expressed in either* LTL$_f$ *or* PPLTL*.*

In this setting, since the goal is a temporal formula defined over the fluents in F, we must take into consideration the whole sequence of states induced by a plan.

Definition 7. *A plan π for a problem $\Pi = \langle F, D, X, A, I, \varphi, Pre, Eff\rangle$ is a sequence of actions $\langle a_0, a_1, \ldots, a_n\rangle$ from A; π is valid for Π (a solution) iff there exists a state trajectory $\tau = \langle s_0, s_1, \ldots, s_n\rangle$ such that:*

- $s_0 = I$
- $\forall\, i \in [1, \dots, n]\ s_i \models \mathit{Pre}(a_i)\ \mathit{and}\, s_{i+1} = s_i[a_i]$
- $\tau \models \varphi$

If the temporally extended goal is in LTL_f, then we must search for a plan that induces a state trajectory τ such that $\tau, 0 \models \varphi$. Otherwise, when φ is in PPLTL, then the induced state trajectory τ must be such that $\tau, n \models \varphi$. For the remainder of this Thesis, we write $\tau \models \varphi$ to indicate that $\tau, 0 \models \varphi$ with φ in LTL_f and $\tau, n \models \varphi$ with φ in PPLTL.

3.2 PDDL3 State Trajectory Constraints

PDDL3 is a popular planning language that has been introduced in the context of the 5th International Planning Competition (`https://lpg.unibs.it/ipc-5/`). This formalism allows users to define the quality of solutions by controlling the structure of desired plans with soft and hard constraints. These constraints are formulated using a standard set of temporal operators that can be formalized as a subset of LTL_f formulae. Soft constraints, also known as preferences, are temporal specifications that a good-quality solution should satisfy, while hard constraints are necessary conditions that every solution plan must satisfy. In this work, we focus on hard constraints, particularly those that in PDDL3 are called "qualitative" [Gerevini et al., 2009] (hereinafter PDDL3 constraints), because they involve only non-numeric terms. The rest of this section analyzes the syntax and semantics of PDDL3 constraints. A complete coverage of the PDDL3 language can be found in Gerevini et al. [2009].

PDDL3 qualitative temporal operators are: `always` (A), `at-most-once` (AO), `sometime` (ST), `sometime-before` (SB) and `sometime-after` (SA). A PDDL3 constraint c is syntactically defined as follows:

$$c := \mathsf{A}(\phi) \mid \mathsf{ST}(\phi) \mid \mathsf{AO}(\phi) \mid \mathsf{SB}(\phi, \psi) \mid \mathsf{SA}(\phi, \psi)$$

where ϕ/ψ are two propositional formulas that we assume in NNF. Semantically:

- $\mathsf{A}(\phi)$ is satisfied if every state induced by the plan satisfies ϕ.
- $\mathsf{ST}(\phi)$ requires that there is at least one state traversed by the plan where ϕ is true.
- $\mathsf{AO}(\phi)$ specifies that ϕ is true in at most one contiguous subsequence of traversed states.
- $\mathsf{SB}(\phi, \psi)$ requires that if ϕ is true in a state induced by the plan, then also ψ is true in a previously traversed state;
- $\mathsf{SA}(\phi, \psi)$ specifies that if ϕ is true in a traversed state, then also ψ is true in that state or a later traversed state.

Definition 8 provides the formal semantics of each PDDL3 constraint.

Definition 8. *Let F be a set of fluents, $\tau = \langle s_0, \ldots, s_n \rangle$ be a sequence of states, ϕ and ψ be propositional formulae over F. A* PDDL3 *constraint c is satisfied by τ, written as $\tau \models c$, if the following conditions hold:*

$\tau \models \mathsf{A}(\phi)$ *iff* $\forall i : 0 \leq i \leq n \cdot s_i \models \phi$

$\tau \models \mathsf{ST}(\phi)$ *iff* $\exists i : 0 \leq i \leq n \cdot s_i \models \phi$

$\tau \models \mathsf{AO}(\phi)$ *iff* $\forall i : 0 \leq i \leq n \cdot$ *if* $s_i \models \phi$ *then*

$\quad \exists j : j \geq i \cdot \forall k : i \leq k \leq j \cdot s_k \models \phi$ *and* $\forall k : k > j \cdot s_k \models \neg\phi$

$\tau \models \mathsf{SA}(\phi, \psi)$ *iff* $\forall i : 0 \leq i \leq n \cdot$ *if* $s_i \models \phi$ *then* $\exists j : i \leq j \leq n \cdot s_j \models \psi$

$\tau \models \mathsf{SB}(\phi, \psi)$ *iff* $\forall i : 0 \leq i \leq n \cdot$ *if* $s_i \models \phi$ *then* $\exists j : 0 \leq j < i \cdot s_j \models \psi$.

We formalize a PDDL3 planning problem as a classical planning problem extended with a collection of PDDL3 constraints.

Definition 9. *A* PDDL3 *planning problem is a tuple $\langle \Pi, \mathcal{C}_3 \rangle$, where Π is a classical planning problem and $\mathcal{C}_3$ is a set of* PDDL3 *constraints.*

The task of solving a PDDL3 planning problem is the task of searching for a plan that is a solution in the classical sense and that satisfies all PDDL3 constraints.

Definition 10. *A plan π is a solution for $\langle \Pi, \mathcal{C}_3 \rangle$ iff it is a solution for Π and the state trajectory τ induced by π is such that $\forall c \in \mathcal{C}_3 \cdot \tau \models c$.*

3.2.1 First-Order Formulation of PDDL3 Constraints

PDDL3 state trajectory constraints can also be formulated using first-order quantifiers.

Example 4. *The constraint $\textbf{\textit{sometime}}(\exists\,?x \cdot On(?x, a))$ expresses that at some point in the state trajectory, there must be a block placed over a.*

Note that in Example 4 the quantification is used in a formula inside the temporal operator. In these cases, the usual semantics of first-order logic applies. Quantifiers can also be placed outside the temporal operator.

Example 5. *The property "At some point all blocks are laying on the table" can be represented with the PDDL3 constraint: A valid PDDL3 constraint is*

$$\forall\,?x \cdot \textbf{\textit{sometime}}(Ontable(?x)).$$

In this case, the universal quantifier specifies a group of $\textbf{\textit{sometime}}$ constraints, one for every block that can be substituted with $?x$. For example, with three blocks a, b, and c, the constraint above is equivalent to the set of constraints: $\{\textbf{\textit{sometime}}(Ontable(a),$ $\textbf{\textit{sometime}}(Ontable(b), \textbf{\textit{sometime}}(Ontable(c)\}$.

First-order quantifiers allow the user to compactly capture complex properties, as shown in Example 6.

Example 6. *The constraint "Only one truck can be in a location at every time" can be captured in PDDL3 with:*

$$\begin{aligned}&\forall\,?truck_1, ?truck_2, ?loc \cdot\\ &\qquad \textbf{\textit{always}}((At(?truck_1, ?loc) \wedge At(?truck_2, ?loc)) \rightarrow ?truck_1 \neq ?truck_2).\end{aligned}$$

There is, however, a limitation to the formulation of such first-order constraints in the PDDL3 language: existential quantification can not be applied to temporal operators (see the BNF of PDDL3 [Gerevini and Long, 2005]). This means that $\exists\, ?x \cdot$ `at-most-once`$(On(?x, a))$ is not a valid PDDL3 constraint. Intuitively, this is because this specification defines a group of constraints where only one should be satisfied, while PDDL3 problems require *every* constraint to be satisfied. In general, the PDDL3 language could be extended with existential quantification and disjunctions between trajectory constraints, but this is left for future work.

3.2.2 Relation Between PDDL3 Constraints and Temporally Extended Goals

This section shows the relation between the temporally extended goals and PDDL3. As previously discussed, PDDL3 qualitative constraints can be formalized as a subset of LTL$_f$ formulae. This means that every PDDL3 constraint can be represented as a semantically equivalent LTL$_f$ formula. Moreover, since PPLTL is as expressive as LTL$_f$, every PDDL3 qualitative operator can also be expressed in terms of a semantically equivalent PPLTL formula, too. These formulations are shown in Table 3.1[1].

PDDL3	PPLTL formulation	LTL$_f$ formulation
$\mathsf{A}(\phi)$	$\mathsf{H}(\phi)$	$\mathsf{G}(\phi)$
$\mathsf{ST}(\phi)$	$\mathsf{O}(\phi)$	$\mathsf{F}(\phi)$
$\mathsf{AO}(\phi)$	$\mathsf{H}(\phi \rightarrow (\phi \mathsf{S}(\mathsf{H}(\neg\phi) \vee \mathsf{start})))$	$\mathsf{G}(\phi \rightarrow (\phi \mathsf{U}(\mathsf{G}(\neg\phi) \vee \mathsf{end})))$
$\mathsf{SA}(\phi_1, \phi_2)$	$(\neg\phi_1 \,\mathsf{S}\, \phi_2) \vee \mathsf{H}(\neg\phi_1)$	$\mathsf{G}(\phi_1 \rightarrow \mathsf{F}\phi_2)$
$\mathsf{SB}(\phi_1, \phi_2)$	$\mathsf{H}(\phi_1 \rightarrow \mathsf{Y}(\mathsf{O}(\phi_2)))$	$\phi_2 \,\mathsf{R}\, \neg\phi_1$

Table 3.1: PDDL3 operators, their equivalent PPLTL and LTL$_f$ formulas. ϕ, ϕ_1 and ϕ_2 are propositional formulas.

Theorem 2 (Camacho et al. [2019], Bonassi et al. [2023a])**.** *Let c be a* PDDL3 *qualitative trajectory constraint, φ_l and φ_p be the corresponding* LTL$_f$ *and* PPLTL *formu-*

[1]Notice that these are *possible* translations of PDDL3 constraints. For example, $\mathsf{SB}(\phi_1, \phi_2)$ can also be translated into LTL$_f$ as $(\neg\phi_1 \,\mathsf{U}\, (\neg\phi_1 \wedge \phi_2)) \vee \mathsf{G}(\neg\phi_1)$ [De Giacomo et al., 2014].

lations according to Table 3.1, and τ be a state trajectory. Then $\tau \models c$ iff $\tau \models \varphi_l$ iff $\tau \models \varphi_p$.

By exploiting these equivalences, we can reformulate a PDDL3 problem as a planning problem with a temporally extended goal.

Theorem 3. *Let $\Pi_3 = \langle\langle F, A, I, G, Pre, Eff\rangle, \mathcal{C}_3\rangle$ be a* PDDL3 *planning problem and let $\Pi_l = \langle F, A, I, G, \varphi, Eff\rangle$ be a planning problem with a* LTL$_f$ *temporally extended goal where:*

$$\varphi = \bigwedge_{c \in \mathcal{C}_3} \phi_c \wedge \mathsf{F}(G \wedge \mathsf{end})$$

such that ϕ_c is the equivalent formulation of c in LTL$_f$ *according to Table 3.1. Then π is a solution for Π_3 iff π is a solution for Π_l.*

Proof. Let π be a sequence of actions from A. It is easy to see that π induces the same state trajectory $\tau = s_0, \ldots, s_n$ for both Π_3 and Π_l, as both problems share the same action labels and preconditions and effects functions. Then by Theorem 2 we have that for every $c \in \mathcal{C}_3$, $\tau \models c$ iff $\tau \models \phi_c$. Moreover, we also have $s_n \models G$ iff $\tau \models \mathsf{F}(G \wedge \mathsf{end})$ (Camacho et al. [2019]). Therefore, we have that the trace τ induced by π satisfies every trajectory constraint in $\mathcal{C}_3$ and reaches G iff $\tau \models \varphi$. $\square$

Theorem 4. *Let $\Pi_3 = \langle\langle F, A, I, G, Pre, Eff\rangle, \mathcal{C}_3\rangle$ be a* PDDL3 *planning problem and let $\Pi_p = \langle F, A, I, G, \varphi, Eff\rangle$ be a planning problem with an* PPLTL *temporally extended goal where:*

$$\varphi = \bigwedge_{c \in \mathcal{C}_3} \phi_c \wedge G$$

such that ϕ_c is the equivalent formulation of c in PPLTL *according to Table 3.1. Then π is a solution for Π_3 iff π is a solution for Π_p.*

Proof. Let π be a sequence of actions from A that induces the state trajectory $\tau = s_0, \ldots, s_n$ for Π_3 and Π_l. Theorem 2 implies that for every $c \in \mathcal{C}_3$, $\tau \models c$ iff $\tau \models \phi_c$. Moreover, $s_n \models G$ iff $\tau \models G$ (Bonassi et al. [2023a]). Therefore, we have that the trace τ induced by π satisfies every trajectory constraint in $\mathcal{C}_3$ and reaches the goal G iff $\tau \models \varphi$.

$\square$

As shown by Theorems 3 and 4, obtaining a planning problem with a temporally extended goal from a PDDL3 problem is quite straightforward. Instead, it is not clear how to perform the opposite operation. In particular, PDDL3 temporal operators cannot be nested, and all constraints of a PDDL3 planning problem must be satisfied in conjunction. For these reasons, it is not clear how to represent simple formulas such as $\mathsf{G}(a) \vee \mathsf{G}(b)$ or $\mathsf{H}(a \rightarrow \mathsf{Y}(b))$ in PDDL3. However, throughout this thesis, we will show how the lack of nesting and the restriction to conjunctive properties enable the design of efficient strategies to handle PDDL3 problems.

3.3 Action Trajectory Constraints in PAC

Planning with Action Constraints (PAC) [Bonassi et al., 2022a] is a new language introduced in this Thesis and focuses on a different way of expressing trajectory constraints. In particular, PAC defines a new set of constraints that are interpreted over sequences of actions rather than states. Therefore, we classify these constraints as action-trajectory constraints (hereinafter PAC constraints). A PAC constraint is defined by combining a PAC *temporal operator* with *actions formulae*.

3.3.1 Action Formulas

Definition 11. *Let A be a set of action labels. An action formula ϕ defined over A is a propositional formula defined over the set of atoms $\{a \mid$ "a" is an action label from $A\}$.*

The following definition formalizes the semantics of a formula ϕ in NNF.

Definition 12. *Let A be a set of action labels. Given a plan $\pi = \langle a_0, a_1, \dots a_{n-1}\rangle$, an action formula ϕ defined over A written in NNF is true at time t in π, i.e. $\pi(t)$ satisfies ϕ, iff:*

- *If $\phi = a$ then $\pi(t) = a$.*
- *If $\phi = \neg a$ then $\pi(t) \neq a$.*

- *If $\phi = \psi_1 \wedge \psi_2$ with ψ_1 and ψ_2 action formulae over A, then $\pi(t)$ satisfies ψ_1 and $\pi(t)$ satisfies ψ_2.*
- *If $\phi = \psi_1 \vee \psi_2$ with ψ_1 and ψ_2 action formulae over A, then $\pi(t)$ satisfies ψ_1 or $\pi(t)$ satisfies ψ_2.*

In a sequential plan, exactly one action is executed at each time step. That is, given a plan $\pi = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ defined over a set of actions labels A, the following formulas on the action atoms hold for $t = 0 \ldots n-1$:

$$\pi(t) = a_i \;\Rightarrow\; \pi(t) \neq a_j, \forall a_i, a_j \in A,\; a_i \neq a_j$$
$$\exists a_i \in A \cdot \pi(t) = a_i$$

Intuitively, the first property expresses that only an atomic action a_i can be satisfied by a plan π at time t, whereas the second property expresses that at every time t there exists at least one atomic action a_i satisfied by π. Due to these properties of a sequential plan, it is always possible to identify the set of actions that satisfy any action formula.

Example 7. *the action formula $a_1 \vee \neg a_2$ is satisfied by all action names (atoms) of a planning problem different from a_2. This is because, for every time step of a plan, the execution of any action except a_2 satisfies the formula. Another example is $\neg a_1 \wedge a_2$. In this case, a_2 is the only action satisfying $\neg a_1 \wedge a_2$.*

The next theorem formalizes how to compute the set of actions satisfying any action formula.

Theorem 5. *Let ϕ be an action formula in NNF defined over a set A of action labels. Then we can define a set of positive action literals $L^{pos}(\phi)$ such that $\forall\ a \in A \cdot a \models \phi$ iff $a \in L^{pos}(\phi)$.*

Proof. We define a function $L^{pos}(\cdot)$ that takes as input an action formula ϕ defined over A in NNF and outputs a set of positive action literals. The function $L^{pos}(\cdot)$ is recursively defined as follows:

$$L^{pos}(\phi) = \begin{cases} a & \text{if } \phi = a \\ A \setminus \{a\} & \text{if } \phi = \neg a \\ L^{pos}(\phi_1) \cup L^{pos}(\phi_2) & \text{if } \phi = \phi_1 \vee \phi_2 \\ L^{pos}(\phi_1) \cap L^{pos}(\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2 \end{cases}$$

We now prove that for an action $a \in A$, $a \models \phi$ iff $a \in L^{pos}(\phi)$ by structural induction on the syntax tree of ϕ. Let $P(\phi)$ be the property that we want to prove, i.e., $P(\phi) = a \models \phi$ iff $a \in L^{pos}(\phi)$.
Base case: ϕ is a node of the syntax tree that does not have any children. That is, ϕ is either a positive action literal a' or a negative action literal $\neg a'$.

- $P(a')$: the claim holds because $L^{pos}(a') = a'$ (the transformation L^{pos} leaves the formula unaltered).
- $P(\neg a')$: we have to prove $P(\neg a') = a \models \neg a'$ iff $a \in L^{pos}(\neg a')$
  - ($\Rightarrow$) If $a \models \neg a'$, then $a \neq a'$ by definition. Therefore, $a \in A \setminus \{a'\}$, and the claim holds as $L^{pos}(\neg a') = A \setminus \{a'\}$.
  - ($\Leftarrow$) If $a \in L^{pos}(\neg a')$, then $a \in A \setminus \{a'\}$. Therefore, we have that $a \neq a'$ and by definition $a \models \neg a'$.

Induction Step: Based on the induction hypothesis that the claim holds for ψ child of ϕ, i.e., we assume $P(\psi)$ to prove $P(\phi)$. ϕ has always two children ϕ_1 and ϕ_2.

Case $\phi = \phi_1 \wedge \phi_2$.

Hypothesis 1:	$P(\phi_1) = a \models \phi_1$ iff $a \in L^{pos}(\phi_1)$
Hypothesis 2:	$P(\phi_2) = a \models \phi_2$ iff $a \in L^{pos}(\phi_2)$
Claim:	$P(\phi_1 \wedge \phi_2) = a \models \phi_1 \wedge \phi_2$ iff $a \in L^{pos}(\phi_1 \wedge \phi_2)$

($\Rightarrow$) $a \models \phi_1 \wedge \phi_2$ implies $a \models \phi_1$ and $a \models \phi_2$. For $P(\phi_1)$ and $P(\phi_2)$, we have $a \in L^{pos}(\phi_1)$ and $a \in L^{pos}(\phi_2)$. Therefore $a \in L^{pos}(\phi_1) \cap L^{pos}(\phi_2) = L^{pos}(\phi_1 \wedge \phi_2)$ and the claim holds.

($\Leftarrow$) $a \in L^{pos}(\phi_1 \wedge \phi_2) = L^{pos}(\phi_1) \cap L^{pos}(\phi_2)$ implies $a \in L^{pos}(\phi_1)$ and $a \in L^{pos}(\phi_2)$. By induction hypothesis $P(\phi_1)$ and $P(\phi_2)$, we have $a \models \phi_1$ and $a \models \phi_2$. Therefore $a \models \phi_1 \wedge \phi_2$ and the claim holds.

Case $\phi = \phi_1 \vee \phi_2$

Hypothesis 1: $P(\phi_1) = a \models \phi_1$ iff $a \in L^{pos}(\phi_1)$

Hypothesis 2: $P(\phi_2) = a \models \phi_2$ iff $a \in L^{pos}(\phi_2)$

Claim: $P(\phi_1 \vee \phi_2) = a \models \phi_1 \vee \phi_2$ iff $a \in L^{pos}(\phi_1 \vee \phi_2)$

($\Rightarrow$) $a \models \phi_1 \vee \phi_2$ implies $a \models \phi_1$ or $a \models \phi_2$. If $a \models \phi_1$, then $a \in L^{pos}(\phi_1)$ by induction hypothesis. This implies $a \in L^{pos}(\phi_1) \cup L^{pos}(\phi_2) = L^{pos}(\phi_1 \vee \phi_2)$ and the claim holds. Otherwise, if $a \models \phi_2$, then $a \in L^{pos}(\phi_2)$ by induction hypothesis. Hence, $a \in L^{pos}(\phi_1) \cup L^{pos}(\phi_2) = L^{pos}(\phi_1 \vee \phi_2)$ and the claim holds.

($\Leftarrow$) $a \in L^{pos}(\phi_1 \vee \phi_2) = L^{pos}(\phi_1) \cup L^{pos}(\phi_2)$ implies either $a \in L^{pos}(\phi_1)$ or $a \in L^{pos}(\phi_2)$. In the former, by induction hypothesis $P(\phi_1)$ we have $a \models \phi_1$ and thus $a \models \phi_1 \vee \phi_2$. In the latter, analogously, by induction hypothesis $P(\phi_2)$ we have $a \models \phi_2$, $a \models \phi_1 \vee \phi_2$, and the claim holds.

□

In the rest of the thesis, we write $a \in \phi$ to denote $a \in L^{pos}(\phi)$, which implies that a satisfies ϕ.

3.3.2 PAC Constraints

In PAC, action constraints can be of the following types: `always` (A), `sometime` (ST), `at-most-once` (AO), `sometime-before` (SB), `sometime-after` (SA), `always-next` (AX) and `pattern` (PA). As we can see, PAC uses the same qualitative temporal operators of PDDL3 and defines the two new modalities `always-next` and `pattern`. A PAC constraint c is syntactically defined as follows:

$$c := \mathsf{A}(\phi) \mid \mathsf{ST}(\phi) \mid \mathsf{AO}(\phi) \mid \mathsf{SB}(\phi, \psi) \mid \mathsf{SA}(\phi, \psi) \mid \mathsf{AX}(\phi, \psi) \mid \mathsf{PA}(\phi_1 \ldots \phi_k)$$

where all $\phi/\psi/\phi_1/\ldots/\phi_k$ are action formulas. Semantically:

- $\mathsf{A}(\phi)$ requires that only actions that satisfy ϕ are in π.
- $\mathsf{ST}(\phi)$ is satisfied if at least one action that satisfies ϕ is executed in π.
- $\mathsf{AO}(\phi)$ requires that an action satisfying ϕ can appear in π only if no action satisfying ϕ is in π before.
- $\mathsf{SB}(\phi, \psi)$ requires that if an action satisfying ϕ appears in π at a time t, then an action satisfying ψ is in π at a time before t.
- $\mathsf{SA}(\phi, \psi)$ requires that if an action satisfying ϕ is in π at a time t, then an action that satisfies ψ is in π at a time after t.
- $\mathsf{AX}(\phi, \psi)$ requires that if an action satisfying ϕ is in π, then it is immediately followed by an action satisfying ψ.
- $\mathsf{PA}(\phi_1, \dots, \phi_k)$ requires that, for $i = 1 \dots k-1$, there exists an action in π satisfying ϕ_i followed at some later time by an action satisfying ϕ_{i+1}.

The formal semantics of each PAC constraint is provided in the following definition.

Definition 13. *Let A be a set of action labels, and let $\pi = \langle a_0, a_1, \dots, a_{n-1} \rangle$ be a plan where for every $t \in 0, \dots, n-1 \cdot \pi(t) \in A$. An action constraint c is satisfied by π, written as $\pi \models c$, if the following conditions hold:*

$$\pi \models \mathsf{A}(\phi) \textit{ iff } \forall t : 0 \leq t \leq n-1 \cdot \pi(t) \in \phi$$
$$\pi \models \mathsf{ST}(\phi) \textit{ iff } \exists t : 0 \leq t \leq n-1 \cdot \pi(t) \in \phi$$
$$\pi \models \mathsf{AO}(\phi) \textit{ iff } \forall t_1 : 0 \leq t_1 \leq n-1 \cdot \textit{if } \pi(t_1) \in \phi \textit{ then } \forall t_2 : t_1 < t_2 \leq n-1 \cdot \pi(t_2) \notin \phi$$
$$\pi \models \mathsf{SA}(\phi, \psi) \textit{ iff } \forall t_1 : 0 \leq t_1 \leq n-1 \cdot \textit{if } \pi(t_1) \in \phi \textit{ then } \exists t_2 : t_1 \leq t_2 \leq n-1 \cdot \pi(t_2) \in \psi$$
$$\pi \models \mathsf{SB}(\phi) \textit{ iff } \forall t_1 : 0 \leq t_1 \leq n-1 \cdot \textit{if } \pi(t_1) \in \phi \textit{ then } \exists t_2 : 0 \leq t_2 < t_1 \cdot \pi(t_2) \in \psi$$
$$\pi \models \mathsf{AX}(\phi, \psi) \textit{ iff } \forall t : 0 \leq t < n-1 \cdot if\ \pi(t) \in \phi\ \ \pi(t+1) \in \psi \textit{ and } \pi(n-1) \notin \phi$$
$$\pi \models \mathsf{PA}(\phi_1 \dots \phi_k) \textit{ iff } \exists \textit{ a sequence of actions } \langle a_1, \dots, a_k \rangle \textit{ from}$$
$$\pi \textit{ that are ordered as in } \pi, \textit{ such that } \forall i \in \{1, \dots, k\}\ a_i \in \phi_i.$$

We define a PAC problem as a classical planning problem enriched with PAC constraints.

Definition 14. *A* PAC *planning problem is a tuple* $\langle \Pi, \mathcal{C}_A \rangle$ *where* Π *is a classical planning problem and* $\mathcal{C}_A$ *is a set of* PAC *constraints.*

The solution plans of $\langle \Pi, \mathcal{C}_A \rangle$ are the solution plans of Π that satisfy all constraints in $\mathcal{C}_A$.

Definition 15. *A plan* π *is a solution for* $\langle \Pi, \mathcal{C}_A \rangle$ *iff it is a solution for* Π *and* $\forall c \in \mathcal{C}_A \cdot \pi \models c$.

3.3.3 First-Order Formulation of PAC Constraints

Similarly to PDDL3, we allow PAC constraints to be specified using a first-order representation. This feature is extremely important in the PAC language, as most actions involve many variables.

Example 8. *In the well-known Zenotravel domain [Penberthy and Weld, 1994, Long and Fox, 2003], we can specify "Plane$_1$ should fly to city$_1$ at least one time" with:*

$$\texttt{sometime}(\exists\, ?from, ?fl1, ?fl2 \cdot Fly(plane_1, ?from, city_1 ?fl1, ?fl2)).$$

In this example, $?fl1$ *and* $?fl2$ *represent the level of fuel of the plane. Since we are not interested in constraining these values, we use the* $\exists$ *quantifier to express that any of these values are acceptable.*

Like for PDDL3, we restrict the language to conjunctions of constraints, and we allow universal quantification of temporal operators (see Example 9).

Example 9. *With the constraint:*

$$\forall\, bus \cdot \texttt{sometime}(\exists\, city \cdot Drive(bus, city, city_a) \vee Drive(bus, city, city_b))$$

we are declaring an equivalent set of (instantiated) action constraints requiring all buses to drive to city-a or city-b at least once.

3.3.4 Relation Between PAC Constraints and Temporal Specifications Over the State Trajectory

Given the close relation between action constraints and temporal specifications over the state trajectory, such as PDDL3 constraints and LTL$_f$ temporally extended goals, one could wonder if there are some differences in terms of expressivity. That is, given a constraint c in PAC, is it possible to capture c using PDDL3 constraints or with a temporally extended goal? Example 4.1 shows that this is not the case.

Example 10. *Let Π be a planning problem where the set of actions A is $\{a_1, a_2, a_3\}$ and where:*

- $Pre(a_1) = p_0$, $Eff(a_1) = \{e_0\}$
- $Pre(a_2) = p_1$, $Eff(a_2) = \{e_0\}$
- $Pre(a_3) = e_0$, $Eff(a_3) = \{goal\}$.

Suppose that the initial state is $I = \{p_0, p_1\}$ and the goal is $G = goal$. It is easy to see that both plans:

$$\pi_1 = \langle a_1, a_3 \rangle$$

$$\pi_2 = \langle a_2, a_3 \rangle$$

solve the problem and induce the state trajectory

$$\tau = \langle \{p_0, p_1\}, \{p_0, p_1, e_0\}, \{p_0, p_1, e_0, goal\} \rangle.$$

Consider now the PAC constraint $\mathsf{ST}(a_2)$. While π_2 satisfies such action constraint, π_1 does not. However, it is not possible to distinguish π_1 from π_2 simply by looking at the (same) sequence of states induced by π_1 and π_2: no state trajectory constraint rules π_1 out. Similarly, no temporal goal in either LTL$_f$ or PPLTL can capture the PAC constraint.

Therefore, we must conclude that PAC is incompatible with both PDDL3 constraints and temporally extended goals. However, many PAC constraints can be formulated as temporally extended goals.

Example 11. *The constraint* $\texttt{sometime}(\mathit{Pick\text{-}Up}(a))$ *can be captured by the* LTL$_f$ *formula*

$$\mathsf{F}(\mathit{Ontable}(a) \wedge \mathsf{X}(\neg \mathit{Ontable}(a))).$$

Clearly, the only action that can achieve $(\mathit{Ontable}(a) \wedge \mathsf{X}(\neg \mathit{Ontable}(a)))$ *is* $\mathit{Pick\text{-}Up}(a)$*. Hence, the* F *operator ensures that the action is executed at least once.*

Determining a class of formulas where a systematic translation between PAC and LTL$_f$ (and PPLTL) is possible is an open research direction. Lastly, it is worth observing that PAC constraints could be seen as a subset of LTL$_f$ or PPLTL formulas interpreted over sequences of action propositions (see, e.g., De Giacomo and Vardi [2015] and Aminof et al. [2019]).

Chapter 4

State of the art

This chapter presents state-of-the-art techniques for solving automated planning problems with trajectory constraints and temporally extended goals. In the current state-of-the-art, there are two approaches for handling these temporal specifications, which can be categorized as native approaches and compilation approaches.

By native approaches, we mean planning systems that handle trajectory constraints and temporally extended goals directly into the heuristic search engine using specific algorithms and dedicated data structures. In contrast, compilation approaches take as input a planning problem with temporal constraints and goals and give as output a new planning problem written in a target formalism that does not contain them. The resulting problem is then solved by existing planners that support the target formalism. For example, the input formalism of a compilation system could be a planning problem with an LTL_f goal, and the target formalism could be classical planning. In general, compilation techniques change the structure of the input problem by adding new atoms, goals, actions, preconditions, and effects. There are several advantages to adopting compilation:

- The best planners that do not support the trajectory constraints and temporally extended goals can be used off-the-shelf.
- Technological advancements in classical planning, such as new search algorithms and heuristics, can be applied directly.

- The planning system is modular, with a clear separation between the module responsible for managing new operators and the module responsible for searching for a plan. Moreover, this separation makes the overall system more robust.

- Parallelization becomes possible, allowing multiple planners to run in parallel.

The main drawback of compilation is that the input problem increases in size, which negatively impacts the time a planner spends finding solutions. Additionally, certain types of search strategies cannot be introduced at the compilation level. On the contrary, native methods offer the possibility of designing new search algorithms and heuristics that are specific to trajectory constraints and temporal goals.

Many state-of-the-art compilation approaches exploit automatons to deal with temporally extended goals and trajectory constraints. Therefore, we introduce some background notions on Nondeterministic Finite-State Automatons. Then, we review two state-of-the-art compilation techniques to deal with LTL$_f$ temporally extended goals, and we briefly introduce some native planning systems supporting PDDL3 state trajectory constraints.

4.1 Automata Representation of Temporal Logic Formulas

One of the most common approaches to systematically deal with temporal goals in automated planning is to use automaton representation. An automaton is a structure made up of nodes (or automaton states) connected by transitions. Each automaton has an initial state and a set of final states. There are different classes of automatons, and we focus on those that can represent finite state properties.

Definition 16 (Non-Deterministic Finite-State Automata [Baier and Katoen, 2008])**.** *A nondeterministic finite-state automata (*NFA*) is a tuple $\mathcal{A} = \langle \Sigma, Q, q_I, \delta, Q_f \rangle$, where:*

- *Σ is a set of input symbols called "alphabet".*

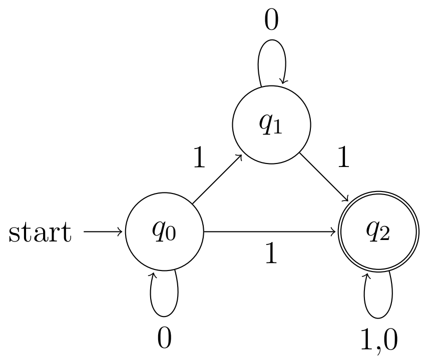


Figure 4.1: An example of a NFA.

- *Q is a set of labels representing the automaton states.*
- *$\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function mapping a state-symbol pair $\langle q, \sigma \rangle$ to a set of successor states $\delta(q, \sigma) \subseteq Q$.*
- *q_I is the initial automaton state.*
- *$Q_f \subseteq Q$ is a set of accepting automaton states.*

Example 12. *An example of* NFA *is depicted in Figure 4.1. Here, $Q = \{q_0, q_1, q_2\}$, the alphabet is $\Sigma = \{0, 1\}$, the initial state is q_0, and the transition function is defined as follows:*

$$\begin{aligned}
\delta(q_0, 1) &= \{q_1, q_2\} \\
\delta(q_0, 0) &= \{q_0\} \\
\delta(q_1, 0) &= \{q_1\} \\
\delta(q_1, 1) &= \{q_2\} \\
\delta(q_2, 0) &= \{q_2\} \\
\delta(q_2, 1) &= \{q_2\}
\end{aligned}$$

The intuitive behavior of an NFA is as follows. The automaton starts in the state q_I, and then is fed an input word $\tau = \langle \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \rangle \in \Sigma^*$[1]. The automaton reads this word symbol by symbol and updates the current state as defined by the

[1]Remark that $(\cdot)^*$ is the standard notation for "zero or more repetitions".

transition function δ. That is, if the current input symbol σ is read in the state q, the automaton chooses the successor state from $\delta(q, \sigma)$. If $\delta(q, \sigma)$ contains two or more states, then the decision for the next state is made at random. When the complete input word is read, the automaton halts. It accepts whenever the current state is accepting and rejects otherwise.

Definition 17. *Let $\mathcal{A} = \langle \Sigma, Q, q_I, \delta, Q_f \rangle$ be a non-deterministic automaton and let $\tau = \langle \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \rangle \in \Sigma^*$ be a sequence of symbols. A run for $\mathcal{A}$ is a sequence of states $q_0, q_1, \ldots q_n$ such that:*

- $q_0 = q_I$
- $q_{i+1} = \delta(q_i, \sigma_i)$ *for* $i = 0, 1 \ldots n-1$.

A run accepts τ if $q_n \in Q_f$, otherwise the run rejects τ. A fine trace τ is called accepted by $\mathcal{A}$ if there exists a run that accepts τ. Moreover, the accepting language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the (infinite) set of words accepted by $\mathcal{A}$, that is,

$$\mathcal{L}(\mathcal{A}) = \{\tau \in \Sigma^* \mid \textit{there exists an accepting run for } \tau \textit{ in } \mathcal{A}\}$$

If the transition function of an automaton is deterministic, that is, for every $q \in Q$ and for every $\sigma \in \Sigma$ we have $|\delta(q, \sigma)| = 1$, then we say that the automaton is a Deterministic Finite-State Automaton (DFA).

Automatons are widely used to capture the possible evolutions of a LTL$_f$ formula. That is, given a LTL$_f$ formula φ, it is always possible to build an NFA $\mathcal{A}_\varphi$ that accepts the same language of φ, that is, $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ [De Giacomo and Vardi, 2013].

Example 13. *An example of an automaton $\mathcal{A}_\varphi$ representing the LTL$_f$ formula $\varphi = \mathsf{G}(a \rightarrow \mathsf{F}(b))$ is depicted in Figure 4.2. Here, $Q = \{q_0, q_1\}$, the alphabet is $\Sigma = 2^{\mathcal{P}}$ with $\mathcal{P} = \{a, b\}$, the initial state is q_0, the final states are $Q_f = \{q_0\}$, and the transition function is defined as follows:*

$$\begin{array}{lcl}
\delta(q_0, b \wedge \neg a) & = & \{q_0\} \\
\delta(q_0, a \wedge \neg b) & = & \{q_1\} \\
\delta(q_1, \neg b) & = & \{q_1\} \\
\delta(q_1, b) & = & \{q_0\}.
\end{array}$$

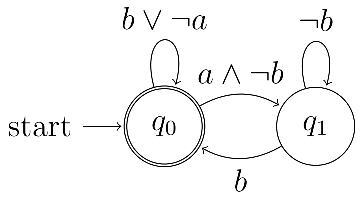


Figure 4.2: An example of a NFA (in this case, a DFA) representing the LTL$_f$ formula $\mathsf{G}(a \rightarrow \mathsf{F}(b))$.

Notice that, for the sake of succinctness, we use formulas in the transitions to capture the alphabet, which is the set of interpretations of the propositional symbols. Let $\tau_1 = \langle \{a, \neg b\}, \{\neg a, \neg b\}, \{a, b\} \rangle$ *and* $\tau_2 = \langle \{a, \neg b\}, \{\neg a, \neg b\}, \{\neg a, \neg b\} \rangle$ *be two sequences of interpretations of the symbols* a *and* b*. We have that* $\mathcal{A}_\varphi$ *accepts* τ_1 *with the run* q_0, q_1, q_0 *and rejects* τ_2 *with the run* q_0, q_1, q_1*. Consequently,* $\tau_1 \models \varphi$ *and* $\tau_2 \not\models \varphi$*.*

Automatons can also be used to capture all PDDL3 constraints, as they can be translated into an LTL$_f$ formula. We conclude by remarking that an NFA representing an LTL$_f$ formula has a worst-case exponential number of states [Baier and McIlraith, 2006a,b, De Giacomo and Vardi, 2013].

4.2 An Exponential NFA-Based Compilation for LTL$_f$ Goals

This section focuses on a NFA-based compilation technique for handling LTL$_f$ temporal goals. In the literature, two variants of this encoding have been presented, one supporting propositional LTL$_f$ goals [Baier and McIlraith, 2006b], and the other for handling first-order LTL$_f$ goals [Baier and McIlraith, 2006a]. In this section, for the sake of clarity, we describe the simpler propositional compilation proposed by Baier and McIlraith [2006b], while in our experiments we adopt the more advanced compilation [Baier and McIlraith, 2006a].

The compilation, which we call LTL-E, takes a problem $\Pi = \langle F, A, I, \varphi, \mathit{Pre}, \mathit{Eff} \rangle$ where φ is a temporally extended goal in LTL$_f$ and produces a new classical planning problem $\Pi' = \langle F', A, I', G', \mathit{Eff}', \mathit{Pre} \rangle$.

The LTL$_f$ goal φ is translated into an NFA $\mathcal{A}_\varphi = \langle \Sigma, Q, q_I, \delta, Q_f \rangle$, which is then merged into the planning problem Π. In the planning domain, each state of the automaton is represented by a fluent. More formally, for each state q of the automaton, the compilation adds a new fluent E_q. Therefore, $F' = F \cup \{E_q \mid q \in Q\}$. The translation is such that if a sequence of actions $\pi = \langle a_0, a_1 \ldots, a_{n-1} \rangle$ generates the state trajectory $\tau = \langle s_0, s_1, \ldots, s_n \rangle$ then E_q is true in s_n if and only if there is a run of $\mathcal{A}_\varphi$ on τ that ends in the state q. This is done by updating the effect function with the dynamics of the NFA.

Let $Pred(q)$ be the set of states that have a transition to q in $\mathcal{A}_\varphi$, and let $\lambda_{p,q}$ be the condition that makes p transition to q, that is, $\lambda_{p,q}$ is the formula ϕ if $q \in \delta(p, \phi)$, $\perp$ otherwise. Then, for every action $a \in A$ and for every atom E_q, the compilation creates a new set of effects C defined as follows:

$$C = \{\theta^+_{a,E_q} \rhd E_q \mid a \in A \text{ and } q \in Q\} \cup \{\theta^-_{a,E_q} \rhd \neg E_q \mid a \in A \text{ and } q \in Q\}.$$

For an action a and a state q, the conditional effects $\theta^+_{a,E_q} \rhd E_q$ and $\theta^-_{a,E_q} \rhd \neg E_q$ are meant to keep q and E_q in sync; that is, whenever the automaton transitions to q, then E_q must be true, and when the automaton transition to a state different to q then E_q must be false. Intuitively, θ^+_{a,E_q} represents the condition under which the execution of a makes the automaton transition to q. This happens whenever (I) the current state is E_p, (II) p is a predecessor of q (with $p \neq q$, otherwise we would include an auto-transition), and (III) the action a makes $\lambda_{p,q}$ true. Therefore,

$$\theta^+_{a,E_q} = \bigvee_{p \in Pred(q) \setminus \{q\}} E_p \wedge R(\lambda_{p,q}, a).$$

Note that $R(\lambda_{p,q}, a)$ is the regression (see Definition 5) of $\lambda_{p,q}$ through the effects of a, which is a formula that captures when the execution of a makes $\lambda_{p,q}$ true.

On the contrary, θ^-_{a,E_q} encodes when a makes the automaton transition to a state different to q. In this case, the atom E_q must be deleted from the next state. This happens whenever the automaton's next transition is not q. Hence, when θ^+_{a,E_q} is false, and when there is no auto-transition to q. Therefore, θ^{-}_{a,E_q} is

$$\theta^{-}_{a,E_q} = \neg\theta^{+}_{a,E_q} \wedge \neg R(\lambda_{q,q}, a).$$

All conditional effects in C are added to the effects of every action, that is, $\mathit{Eff}'(a) = \mathit{Eff}(a) \cup C$ for every $a \in A$. The initial state must give an account of which atoms E_q are initially true. Therefore, the new initial state is $I' = \{E_q \mid q \in \delta(q_I, \phi) \text{ and } I \models \phi\}$. Lastly, the automaton $\mathcal{A}_\varphi$ accepts a state trajectory τ if and only if τ satisfies φ. Therefore, the new goal G' is defined according to the acceptance condition of the NFA, that is, $G' = \{E_q \mid q \in Q_f\}$.

We remark that, since the number of states of $\mathcal{A}_\varphi$ is worst-case exponential, the resulting planning problem Π' could be exponentially larger than Π.

4.3 A Polynomial Compilation for LTL$_f$ goals

In this section, we show a polynomial compilation approach for translating a planning problem with a LTL$_f$ temporally extended goal into a classical planning problem. This compilation, which has been presented by Torres and Baier [2015], heavily relies on Alternating Automatons, a symbolic variation of Non-Deterministic Automatons. For the sake of clarity, in this section, we reinterpret the automata-theoretic view of Torres and Baier [2015] under a perspective that only relies on the semantics of LTL$_f$ formulas.

4.3.1 Progression of Linear Temporal Logic Formulas on Finite Traces

The key theoretical technique that is employed in this compilation schema is the progression (or fixpoint characterization) of a LTL$_f$ formula [Gabbay et al., 1980, Emerson, 1990] in Negation Normal Form[2]. Intuitively, progression is a transfor-

[2]Notice that every LTL$_f$ formula can be put in NNF by pushing the negation inside temporal operators and by using the "Release" and "Weak next" operators (see section 3.1). For example, the formula $\neg(a \,\mathsf{U}\, \mathsf{X}(b))$ can be transformed in NNF as follows: $\neg(a \,\mathsf{U}\, \mathsf{X}(b)) \equiv (\neg a \,\mathsf{R}\, \neg\mathsf{X}(b)) \equiv (\neg a \,\mathsf{R}\, \mathsf{WX}(\neg b))$.

mation that splits a LTL$_f$ formula into conditions that must hold in the present and conditions that must hold in the future. Formally, a LTL$_f$ formula is progressed using the $\mathsf{xnf}(\cdot)$ transformation, which is recursively defined as follows:

- $\mathsf{xnf}(l) = l$;
- $\mathsf{xnf}(\mathsf{end}) = \mathsf{end}$;
- $\mathsf{xnf}(\mathsf{X}\phi) = \mathsf{X}\phi$;
- $\mathsf{xnf}(\mathsf{WX}\phi) = \mathsf{X}\phi \lor \mathsf{end}$;
- $\mathsf{xnf}(\phi_1 \,\mathsf{U}\, \phi_2) = \mathsf{xnf}(\phi_2) \lor (\mathsf{xnf}(\phi_1) \land \mathsf{X}(\phi_1 \,\mathsf{U}\, \phi_2))$;
- $\mathsf{xnf}(\phi_1 \land \phi_2) = \mathsf{xnf}(\phi_1) \land \mathsf{xnf}(\phi_2)$;
- $\mathsf{xnf}(\phi_1 \lor \phi_2) = \mathsf{xnf}(\phi_1) \lor \mathsf{xnf}(\phi_2)$;
- $\mathsf{xnf}(\phi_1 \,\mathsf{R}\, \phi_2) = \mathsf{xnf}(\phi_2) \land (\mathsf{end} \lor \mathsf{xnf}(\phi_1) \lor \mathsf{X}(\phi_1 \,\mathsf{R}\, \phi_2))$.

For convenience and clarity, we also introduce the $\mathsf{xnf}(\cdot)$ transformation for the two derived operators “F” and “G”:

- $\mathsf{xnf}(\mathsf{F}(\phi)) = \mathsf{xnf}(\phi) \lor \mathsf{X}(\mathsf{F}(\phi))$;
- $\mathsf{xnf}(\mathsf{G}(\phi)) = \mathsf{xnf}(\phi) \land (\mathsf{end} \lor \mathsf{X}(\mathsf{G}(\phi))$.

A formula resulting from the application of $\mathsf{xnf}(\cdot)$ is in Next Normal Form (XNF). A formula $\mathsf{xnf}(\varphi)$ in XNF has a particular structure: all proper temporal subformulas (that is, subformulas whose main construct is a temporal operator) of $\mathsf{xnf}(\varphi)$ appear only in the scope of the X operator. This means that given a state trajectory $\tau = \langle s_0, \dots s_n\rangle$ to determine the truth of a LTL$_f$ formula φ at instant i, that is, $\tau, i \models \varphi$, we just need the truth of the atomic proposition in state s_i plus the truth of some key formulas ϕ, with $\phi \in \mathsf{sub}(\varphi)$, in the trace $\langle s_{i+1}, \dots s_n\rangle$.

Example 14. *Consider the formula $\varphi = \mathsf{F}(a)$. The XNF of φ is:*

$$\mathsf{xnf}(\mathsf{F}(a)) = \mathsf{xnf}(a) \lor \mathsf{XF}(a) = a \lor \mathsf{XF}(a).$$

Given a state trajectory $\tau = \langle s_i, s_{i+1}, \dots s_n \rangle$, we have $\tau, i \models a \vee \mathsf{XF}(a)$ iff $s_i \models a$ and $\tau, i+1 \models \mathsf{F}(a)$. As we can see, to determine the truth of $\mathsf{F}(a)$, we only need the current state s_i and the extension of the trace $\tau, i+1$.

Observe that formulas of the form $\mathsf{X}(\phi)$ appearing in $\mathsf{xnf}(\varphi)$ are such that $\phi \in \mathsf{sub}(\varphi) \cup \{\mathsf{end}\}$. Moreover, it is easy to see that the XNF transformation can be performed in linear time and that the resulting formula is equivalent to the original. This well-known result has been recently recalled in, e.g., Li et al. [2019].

Theorem 6. *Every LTL$_f$ formula φ can be converted to its XNF form $\mathsf{xnf}(\varphi)$ in linear-time in the size of the formula (i.e., $|\mathsf{sub}(\varphi)|$). Moreover, $\mathsf{xnf}(\varphi) \equiv \varphi$.*

The XNF provides a systematic way of reasoning on what we need to do to satisfy a LTL$_f$ formula while planning. To evaluate a LTL$_f$ formula we need the current evaluation of the atomic proposition and the truth of the temporal subformulas in the next step. However, while the planning process goes on and the state trajectory is generated, we only have the information about the current state, while the future trajectory has yet to be computed. Furthermore, directly using the $\mathsf{xnf}(\cdot)$ transformation requires reasoning on arbitrary complex temporal formulas; note that $\mathsf{xnf}(\varphi)$ for a LTL$_f$ formula φ is linear only in the size of its subformulas $|\mathsf{sub}(\varphi)|$ (see Theorem 6). Hence, during planning, we would need to use complex data structures to compactly represent $\mathsf{xnf}(\varphi)$, and this makes the design of a compilation approach prohibitive.

Instead, following the recursive definition of $\mathsf{xnf}(\cdot)$, we introduce a new transformation $\delta(\cdot)$ that only uses propositional atoms to compactly represent the XNF of a LTL$_f$ formula. In particular, given a LTL$_f$ formula φ, we introduce the set of atomic propositions $Q_\varphi = \{\text{“}\phi\text{”} \mid \phi \in \mathsf{sub}(\phi) \cup \{\mathsf{end}\}\}$ to capture the truth of the subformulas of φ. Then, given a planning state s, we recursively define the function $\delta(\text{“}\varphi\text{”}, s)$ as follows:

- $\delta(\text{“}l\text{”}, s) = s \models l$ (for l literal);
- $\delta(\text{“}\mathsf{X}\phi\text{”}, s) = \text{“}\phi\text{”}$
- $\delta(\text{“}\mathsf{WX}\phi\text{”}, s) = \text{“}\phi\text{”} \vee \text{“}\mathsf{end}\text{”}$

- $\delta(\text{“}\mathsf{end}\text{”}, s) = \bot$
- $\delta(\text{“}\phi_1 \wedge \phi_2\text{”}, s) = \delta(\text{“}\phi_1\text{”}, s) \wedge \delta(\text{“}\phi_2\text{”}, s)$;
- $\delta(\text{“}\phi_1 \vee \phi_2\text{”}, s) = \delta(\text{“}\phi_1\text{”}, s) \vee \delta(\text{“}\phi_2\text{”}, s)$;
- $\delta(\text{“}\phi_1 \,\mathsf{U}\, \phi_2\text{”}, s) = \delta(\text{“}\phi_2\text{”}, s) \vee (\delta(\text{“}\phi_1\text{”}, s) \wedge \text{“}\phi_1 \,\mathsf{U}\, \phi_2\text{”})$;
- $\delta(\text{“}\phi_1 \,\mathsf{R}\, \phi_2\text{”}, s) = \delta(\text{“}\phi_2\text{”}, s) \wedge (\text{“}\mathsf{end}\text{”} \vee \delta(\text{“}\phi_1\text{”}, s) \vee \text{“}\phi_1 \,\mathsf{R}\, \phi_2\text{”})$.
- $\delta(\text{“}\mathsf{F}\phi\text{”}, s) = \delta(\text{“}\phi\text{”}, s) \vee \text{“}\mathsf{F}\phi\text{”}$;
- $\delta(\text{“}\mathsf{G}\phi\text{”}, s) = \delta(\text{“}\phi\text{”}, s) \wedge (\text{“}\mathsf{end}\text{”} \vee \text{“}\mathsf{G}\phi\text{”})$.

It is easy to see that $\delta(\text{“}\varphi\text{”}, s)$[3] is (I) linear in Q_φ and (II) captures the progression of a formula ϕ after a state s. In particular, given a proposition “ϕ” and a state s, $\delta(\text{“}\phi\text{”}, s)$ is a formula representing:

- The literal conditions that must hold in s to satisfy ϕ.
- The temporal subformulas that must hold after s to satisfy ϕ.
- Whether or not s must be the last state to satisfy ϕ.

Example 15. *Consider the formula $\varphi = \mathsf{F}(a) \vee (b \,\mathsf{U}\, c)$. Given a state s, we have:*

$$\begin{aligned}
&\delta(\text{“}\varphi\text{”}, s) = \\
&(\delta(\text{“}a\text{”}, s) \vee \text{“}\mathsf{F}(a)\text{”}) \vee (\delta(\text{“}c\text{”}, s) \vee (\delta(\text{“}b\text{”}, s) \wedge \text{“}b \,\mathsf{U}\, c\text{”})) = \\
&(s \models a \vee \text{“}\mathsf{F}(a)\text{”}) \vee (s \models c \vee (s \models b \wedge \text{“}b \,\mathsf{U}\, c\text{”}))
\end{aligned}$$

Depending on the value of the atomic propositions in s, we can determine different ways we can satisfy φ. For example:

- *Suppose $s \not\models a$, $s \models b$ and $s \not\models c$. In this case, $\delta(\text{“}\varphi\text{”}, s)$ can be simplified to “$\mathsf{F}(a)$” $\vee$ “$b \,\mathsf{U}\, c$”, meaning that $\mathsf{F}(a) \vee (b \,\mathsf{U}\, c)$ holds after s iff either $\mathsf{F}(a)$ or $b \,\mathsf{U}\, c$ holds after state s.*

[3]Observe that $\delta(\cdot)$ is also the transition function of the Alternating Automaton in De Giacomo and Vardi [2013] and Torres and Baier [2015].

- *Suppose $s \models a$. In this case, $\delta(``\varphi", s)$ simplifies to $\top$, meaning that the state trajectory $\tau = \langle s \rangle$ satisfies φ.*

- *Suppose $s \not\models a$, $s \not\models b$ and $s \not\models c$. In this case, $\delta(``\varphi", s)$ simplifies to $\bot$, meaning that after s, φ does not hold.*

As we can see from Example 15, given a state s and a formula φ, we can effectively use $\delta(``\varphi", s)$ to determine the temporal subformulas that must hold in the future to satisfy $\varphi$ depending on the truth of the atomic proposition in $s$. However, $\delta(``\varphi", s)$ is an arbitrary complex propositional formula, and to keep track of the truth of $\delta(``\varphi", s)$ we would need to introduce an exponential number of variables. For example, monitoring the truth of the LTL$_f$ formula $\mathsf{G}(l_1) \vee \mathsf{G}(l_2) \vee \mathsf{G}(l_3)$ using $\delta(``\mathsf{G}(l_1) \vee \mathsf{G}(l_2) \vee \mathsf{G}(l_3)", s)$ would require introducing 2^3 fluents, e.g., $holds_{``\mathsf{G}(l_1)"}, holds_{``\mathsf{G}(l_1)" \vee ``\mathsf{G}(l_2)"}, holds_{``\mathsf{G}(l_1)" \vee ``\mathsf{G}(l_3)"}$, etc. Hence, we cannot keep track of the truth of all possible models $\delta(``\varphi", s)$ at the same time. Instead, it is sufficient to monitor one of the models of $\delta(``\varphi", s)$ at a time. That is, we determine a set $Q_m \subseteq Q_\varphi$ such that $Q_m \models \delta(``\varphi", s)$. If we manage to satisfy all temporal subformulas ϕ such that $``\phi" \in Q_m$, then we will satisfy $\varphi$; otherwise, when it is not possible to satisfy all formulas in $Q_m$, we will consider a different model $Q'_m$ of $\delta(``\varphi", s)$.

4.3.2 The LTL-P Encoding

Intuitively, the compilation schema introduced by Torres and Baier [2015], which throughout this thesis is called LTL-P, works as follows. LTL-P uses the set of propositional variables $Q_\varphi = \{``\phi" \mid \phi \in \mathsf{sub}(\varphi) \cup \{\mathsf{end}\}\}$ to record all subformulas $\phi$ of $\varphi$ to satisfy at the current step. Initially, the variable $``\varphi"$ records that φ must be satisfied in the initial state.

After a new state s is generated, for every $``\phi"$ atom currently true, LTL-P determines a set $Q_m \subseteq Q_\varphi$ of $``\phi'"$ atoms to be satisfied at the next step, and such that $Q_m \models \delta(``\phi", s)$. If Q_m is empty, then we achieved the LTL$_f$ goal φ. Otherwise, this process is repeated until we arrive at a state where no formula has to be satisfied at the next step.

LTL-P realizes this schema by alternating two phases: the *world phase* and the *progression phase*. During the world phase, the planner can choose to execute only the actions of the original problem. In the progression phase, the planner is forced to execute a sequence of *progression* actions to compute one of the models of $\delta(\text{“}\phi\text{”}, s)$. Therefore, a plan for the compiled problem will have the following structure:

$$\langle p_{-1}, a_0, p_0, a_1, p_1, \ldots, a_n, p_n \rangle$$

where p_i indicates a sequence of progression actions. In particular, let s_i be a state such that "ϕ" $\in s$, and let s'_i be a state resulting from the execution of the progression actions p_i in s_i. Then, $s'_i \models \delta(\text{“}\phi\text{”}, s_i)$.

We now show how the encoding works in detail. Given a planning problem $\Pi = \langle F, A, I, \varphi, Pre, Eff \rangle$, the compiled problem is $\Pi = \langle F', A', I', G', Pre', Eff' \rangle$ where each component of Π' is detailed in the next paragraphs.

New fluents. The new set of fluents F' contains the original fluents in F plus:

- $Q_\varphi = \{\text{“}\phi\text{”} \mid \phi \in \mathsf{sub}(\varphi) \cup \{\mathsf{end}\}\}$.
- $F_\delta = \{eval_\phi \mid \phi \in \mathsf{sub}(\varphi)\}$. Intuitively, a fluent $eval_\phi$ signals that we need to compute $\delta(\text{“}\phi\text{”}, s)$ in the progression phase.
- The fluent *exec*, used to block the execution when "end" must be satisfied in the current step (that is, execution must end).
- The fluents *prog* and *world*, used to represent the world and progression phase.
- The fluent *setup*, used in the setup of the progression phase.

Formally, $F' = F \cup Q_\varphi \cup F_\delta \cup \{\text{“}\mathsf{end}\text{”}, prog, exec, world, setup\}$

World phase. During the world phase, only one of the actions $a \in A$ of the original problem can be executed. The preconditions and effects of these actions are changed to allow the execution only in the world phase and to switch to the progression phase afterward. Formally, for every $a \in A$:

$$\begin{aligned} \mathit{Pre}'(a) &= \mathit{Pre}(a) \land \mathit{exec} \land \mathit{world} \\ \mathit{Eff}'(a) &= \mathit{Eff}(a) \cup \{\mathit{setup}, \neg \mathit{world}\} \end{aligned}$$

Progression phase. At the start of the progression phase, we initialize the fluents that represent the formulas that have to be progressed. For example, consider a newly induced state s_i where the fluent "$(\phi_1 \mathsf{U} \phi_2)$" holds. In this case, we had previously committed to satisfying $\phi_1 \mathsf{U} \phi_2$, and we start the progression phase with the atom $\mathit{eval}_{\phi_1 \mathsf{U} \phi_2}$. This operation is performed by the new action *setup-act*, which is defined as follows:

$$\begin{aligned} \mathit{Pre}'(\mathit{setup\text{-}act}) &= \mathit{setup} \land \mathit{exec} \\ \mathit{Eff}'(\mathit{setup\text{-}act}) &= \{\text{“}\phi\text{”} \rhd \mathit{eval}_\phi, \text{“}\phi\text{”} \rhd \neg\text{“}\phi\text{”} \mid \phi \in \mathsf{sub}(\varphi) \cup \{\mathsf{end}\}\} \cup \\ &\quad\ \{\mathit{prog}, \neg \mathit{setup}\} \end{aligned}$$

Notice that the *setup-act* action also deletes all current "ϕ" fluents. This is because, after the progression phase, a new set of these fluents will replace the old one. We now get to the core of the progression phase: the evaluation actions, which are defined in Table 4.1.

As we can see, there is an action $\mathit{evaluate}_i(\phi)$ for every disjunct of $\delta(\text{“}\phi\text{”}, s)$, and the structure of these actions mimics the computation of $\delta(\cdot)$. Each $\mathit{evaluate}_i(\phi)$ action has as preconditions:

- *prog* and *exec* to ensure this action is executed only in the progression phase and when the execution is not blocked.
- eval_ϕ to ensure that this action is executed only when we need to compute $\delta(\text{“}\phi\text{”}, s)$.

Moreover, actions of the form $\mathit{evaluate}(l)$ have l as a precondition; indeed, since $\delta(\text{“}l\text{”}, s) = s \models l$, the literal l must be true in the current state.

Action	Preconditions	Effects
$evaluate_1(l)$	$prog \wedge exec \wedge eval_l \wedge l$	$\{\neg eval_l\}$
$evaluate_1(\mathsf{end})$	$prog \wedge exec \wedge eval_{\mathsf{end}}$	$\{\neg eval_{\mathsf{end}}, \neg exec\}$
$evaluate_1(\phi_1 \wedge \phi_2)$	$prog \wedge exec \wedge eval_{\phi_1 \wedge \phi_2}$	$\{eval_{\phi_1}, eval_{\phi_2}, \neg eval_{\phi_1 \wedge \phi_2}\}$
$evaluate_1(\mathsf{X}\phi)$	$prog \wedge exec \wedge eval_{\mathsf{X}\phi}$	$\{\text{“}\phi\text{”}, \neg eval_{\mathsf{X}\phi}\}$
$evaluate_1(\mathsf{WX}\phi)$	$prog \wedge exec \wedge eval_{\mathsf{WX}\phi}$	$\{\text{“}\phi\text{”}, \neg eval_{\mathsf{WX}\phi}\}$
$evaluate_2(\mathsf{WX}\phi)$	$prog \wedge exec \wedge eval_{\mathsf{WX}\phi}$	$\{\text{“}\mathsf{end}\text{”}, \neg eval_{\mathsf{WX}\phi}\}$
$evaluate_1(\phi_1 \vee \phi_2)$	$prog \wedge exec \wedge eval_{\phi_1 \vee \phi_2}$	$\{eval_{\phi_1}, \neg eval_{\phi_1 \vee \phi_2}\}$
$evaluate_2(\phi_1 \vee \phi_2)$	$prog \wedge exec \wedge eval_{\phi_1 \vee \phi_2}$	$\{eval_{\phi_2}, \neg eval_{\phi_1 \vee \phi_2}\}$
$evaluate_1(\phi_1 \,\mathsf{U}\, \phi_2)$	$prog \wedge exec \wedge eval_{\phi_1 \mathsf{U} \phi_2}$	$\{eval_{\phi_2}, \neg eval_{\phi_1 \mathsf{U} \phi_2}\}$
$evaluate_2(\phi_1 \,\mathsf{U}\, \phi_2)$	$prog \wedge exec \wedge eval_{\phi_1 \mathsf{U} \phi_2}$	$\{eval_{\phi_1}, \text{“}\phi_1 \,\mathsf{U}\, \phi_2\text{”}, \neg eval_{\phi_1 \mathsf{U} \phi_2}\}$
$evaluate_1(\phi_1 \,\mathsf{R}\, \phi_2)$	$prog \wedge exec \wedge eval_{\phi_1 \mathsf{R} \phi_2}$	$\{eval_{\phi_2}, \text{“}\mathsf{end}\text{”}, \neg eval_{\phi_1 \mathsf{R} \phi_2}\}$
$evaluate_2(\phi_1 \,\mathsf{R}\, \phi_2)$	$prog \wedge exec \wedge eval_{\phi_1 \mathsf{R} \phi_2}$	$\{eval_{\phi_2}, eval_{\phi_1}, \neg eval_{\phi_1 \mathsf{R} \phi_2}\}$
$evaluate_3(\phi_1 \,\mathsf{R}\, \phi_2)$	$prog \wedge exec \wedge eval_{\phi_1 \mathsf{R} \phi_2}$	$\{eval_{\phi_2}, \text{“}\phi_1 \,\mathsf{R}\, \phi_2\text{”}, \neg eval_{\phi_1 \mathsf{R} \phi_2}\}$

Table 4.1: Definition of the *evaluate* actions. Operators $\mathsf{F}\phi$ and $\mathsf{G}\phi$ are omitted, and assume to be rewritten as $\mathsf{F}\phi = \top \,\mathsf{U}\, \phi$ and $\mathsf{G}\phi = \bot \,\mathsf{R}\, \phi$.

The effects of *evaluate* actions make true atoms of the form “ϕ” to signal that ϕ will be achieved in the next step, and atoms of the form $eval_\psi$ to proceed with the computation of $\delta(\text{“}\psi\text{”}, s)$. The progression phase ends when all formulas are progressed, that is, there are no fluents of the form $eval_\phi$. It is easy to see that, let $eval_\psi$ be an atom true at the start of the progression phase, at the end of the progression phases we will have a set $Q_\psi \subseteq Q_\varphi$ of atoms such that $Q_\psi \models \delta(\text{“}\psi\text{”}, s)$.

Notice that the progression phase may have many dead ends. This happens when some literal implied by some $\delta(\text{“}\phi\text{”}, s)$ is not satisfied. For example, $\delta(\text{“}\mathsf{G}(b)\text{”}, s)$ requires $s \models b$, and this implies that the action $evaluate(b)$ with b as a precondition must be executed in the progression phase. In addition, the progression phase becomes a dead-end when $evaluate_1(\mathsf{end})$ has to be executed; this action has $\neg exec$ as an effect, blocking the execution of all other actions. Indeed, $\delta(\text{“}\mathsf{end}\text{”}, s) = \bot$ indicates that we cannot satisfy end after another state s. We remark that, at each progression phase, the planner can choose between different $evaluate_i(\phi)$ actions to obtain different models of $\delta(\text{“}\phi\text{”}, s)$. Therefore, the choices to be made in future progression phases are influenced by the $evaluate_i(\psi)$ actions selected in the present.

When a dead end is encountered, the planner will backtrack to choose different $evaluate_i(\psi)$ actions in a previous progression phase.

At the end of a progression phase, the only action that can be executed is the $world$ action, which reestablishes the world phase.

- $Pre'(world) = prog \wedge exec \wedge \bigwedge_{\phi \in \mathsf{sub}(\varphi) \cup \{\mathsf{end}\}} \neg eval_\phi$
- $E\!f\!f(world) = \{world, \neg prog\}$

The new set of actions A' is $A \cup \{setup\text{-}act, world\} \cup \{evaluate_i(\phi) \mid \phi \in \mathsf{sub}(\varphi) \cup \{\mathsf{end}\}\}$.

New initial state. Initially, we need to determine which subgoals we want to achieve given the truth of the fluents in the initial state. Therefore, initially, we need to perform a progression phase with the starting formula "φ". Hence, $I' = I \cup \{\text{“}\varphi\text{”}, setup, exec\}$.

New goal. Lastly, the goal is to reach a state in which all fluents of the form "ϕ" are false, except for "end", which may be true in the last state. Therefore, $G' = \{world, exec\} \wedge \bigwedge_{\phi \in \mathsf{sub}(\varphi)} \neg\text{“}\phi\text{”}$.

Discussion. It is easy to see that this encoding is linear in the size of the LTL$_f$ goal φ. However, this optimality regarding the size of the resulting problem comes at the cost of adding many actions to the solutions. In particular, following the complexity taxonomy of Nebel [2000], this encoding *preserves plan size polynomially.* Without going into much detail, this means that the size of the resulting plans grows by a polynomial factor; this is because, in this case, we have to interleave every original action with a sequence p of $evaluate_i(\phi)$ actions where $|p|$ is linear in $|\mathsf{sub}(\varphi)|$. Whether or not it would be possible to improve the encoding by Torres and Baier [2015] (e.g., preserving plan size only linearly) is an interesting future direction.

4.4 Approaches for PDDL3 State Trajectory Constraints

In this section, we briefly review the state-of-the-art approaches that can handle PDDL3 state trajectory constraints.

A baseline approach is to translate the PDDL3 constraints into LTL_f formulas and then adopt the NFA-based compilation described in Section 4.2. This is the idea adopted by the planner MIPS [Edelkamp et al., 2006a,b]. This planner reformulates every PDDL3 constraint c as a NFA $\mathcal{A}_c$, and then encodes the synchronized simulation of the automata by using atoms of the form $At(q, \mathcal{A}_c)$ to indicate that the current state of the automaton $\mathcal{A}_c$ is q. For detecting accepting states of an automaton $\mathcal{A}_c$, MIPS utilizes atoms of the form $Accepting(\mathcal{A}_c)$. The initial state includes the start state of the automaton and an additional atom if such a state is accepting. For all automatons, the goal specification includes automaton acceptance. Lastly, MIPS specifies the allowed automaton transitions as planning actions by declaring a ground action for each automaton transition. These actions are executed during a synchronization phase to keep the state of each automaton updated. As a solving engine, MIPS is tightly integrated with the FF planner [Hoffmann and Nebel, 2001] to efficiently solve problems resulting from the encoding.

OPTIC [Benton et al., 2012] is a native PDDL3 planner that exploits the automaton representation of PDDL3 constraints directly in the search engine and in the relaxed planning graph heuristic of the planner. In particular, PDDL3 constraints are embedded within the planning search by augmenting the states with the current position of the corresponding automaton. Each time an action is applied during the search, the positions of the automaton are updated to reflect the state reached directly.

Lastly, the native planner SGPLAN Hsu et al. [2007] is based on a decomposition-based approach and handles trajectory constraints using the FF planning system as a sub-solver.

Chapter 5

Handling PDDL3 Constraints via Compilation

This chapter focuses on how to handle PDDL3 constraints. The approach we propose works via compilation, that is, we translate a PDDL3 planning problem into a classical planning problem that can be handled by any classical planner supporting conditional effects and disjunctive conditions. Our compilation substantially revisits the automata-less approach proposed by Percassi and Gerevini [2019] for soft trajectory constraints through the lens of *hard* trajectory constraints. The proposed compilation schema characterizes the state trajectory constraints into two classes. The former class encompasses constraints for which it is always possible to prune any plan prefix that violates them; these are safety conditions that every plan must satisfy. The latter class encompasses constraints that induce intermediate goals, landmarks [Richter and Westphal, 2010] that every plan needs to traverse. This characterization leads to the devising of a simple compilation that, with a minimal number of additional atoms, exploits the notion of regression to efficiently prune plan prefixes that do not comply with the first class of constraints, and ensures that all intermediate landmarks are reached. We formally prove the correctness of this technique and show that the novel compilation substantially extends the reach of planning over the considered class of planning problems.

5.1 TCORE: Trajectory constraints COmpilation via REgression

This section describes the compilation schema, called TCORE (Trajectory constraints COmpilation via REgression). TCORE makes extensive use of the operator R (Def. 5) and of a set of *monitoring atoms.* These are used to extend the action preconditions and effects to:

1. Block invalid extensions of the plan prefix generated during planning.
2. Keep track of the truth of the relevant formulae (w.r.t. PDDL3 constraints) in the states generated by the plan prefix.

Our technique uses regression to identify the actual influence an action has on the PDDL3 constraint of interest. Monitoring atoms serve the purpose of collecting relevant facts on the plan state trajectory and asserting their truth/falsity. Our approach uses two types of monitoring atoms:

- $hold_c$ atoms to reflect whether a constraint c has been satisfied
- $seen_\psi$ atoms to capture whether some formula ψ has ever held in some state generated by the plan prefix.

We show how the extended action preconditions and effects are constructed by distinguishing the trajectory constraints into two classes: *safety trajectory constraints* (*STCs*) and *liveness trajectory constraints* (*LTCs*). This categorization is inspired by the well-established classes of *safety* and *liveness* temporal properties used in model checking [Lamport, 1977]. Intuitively, STCs can be checked along any plan prefix and if they are violated, there is no way the planner can ever re-establish them; they are invariant conditions that must be maintained over the state trajectory of the plan.

Example 16. *Consider the state trajectory $\tau = \langle s_0, s_1, \ldots, s_i \rangle$. The constraint $c = \mathsf{A}(a \wedge b)$ requires that every state of τ satisfies the formula $a \wedge b$. Therefore, if there exists a state s of τ such that $s \not\models a \wedge b$, then c is violated. Furthermore, all* *state trajectories having τ as prefix violate c. Hence, $\mathsf{A}(a \wedge b)$ is a safety trajectory constraint.*

LTCs are constraints that require certain conditions to be true in *some* state of the state trajectory induced by the plan.

Example 17. *Consider the state trajectory $\tau = \langle s_0, s_1, \ldots, s_i \rangle$ and the constraint $c = \mathsf{ST}(a)$. By definition, $\tau \models c$ iff a is true in a state s of τ. Therefore, $\mathsf{ST}(a)$ is a liveness trajectory constraint.*

The STCs are $\mathsf{A}(\phi)$, $\mathsf{AO}(\phi)$, $\mathsf{SB}(\phi, \psi)$, while the LTCs are $\mathsf{ST}(\phi)$ and $\mathsf{SA}(\phi, \psi)$. For each $\mathsf{AO}(\phi)$ and $\mathsf{SB}(\phi, \psi)$, we add the fresh predicates $seen_\phi$ and $seen_\psi$ to record whether ϕ and ψ have ever held. In this way, we are adding to each state the necessary information to evaluate the truth of $\mathsf{AO}(\phi)$ and $\mathsf{SB}(\phi, \psi)$ without having to consider the entire state trajectory. Similarly, for each LTC, we add a predicate $hold_c$ to record whether the constraint c is already satisfied or not according to the current plan prefix. Note that no additional monitoring atom is necessary for $\mathsf{A}(\phi)$, since all the information (that is, whether ϕ is true or false) is already present in each state.

Algorithm 1 The TCORE algorithm

```
1: function TCORE($\langle\langle F, A, I, G, Pre, Eff\rangle, \mathcal{C}_3\rangle$)
2:     $F' = F \cup monitoringAtoms(\mathcal{C}_3)$                ▷ Additional atoms generation
3:     $I' = I \cup$
           $\bigcup_{c:\mathsf{ST}(\phi)\in\mathcal{C}_3} \{hold_c \mid I \models \phi\} \cup \bigcup_{c:\mathsf{SA}(\phi,\psi)\in\mathcal{C}_3} \{hold_c \mid I \models \psi \vee \neg\phi\} \cup$
           $\bigcup_{\mathsf{SB}(\phi,\psi)\in\mathcal{C}_3} \{seen_\psi \mid I \models \psi\} \cup \bigcup_{\mathsf{AO}(\phi)\in\mathcal{C}_3} \{seen_\phi \mid I \models \phi\}$
4:     if $\exists\mathsf{SB}(\phi,\psi) \in \mathcal{C}_3 . I \models \phi$ or $\exists\mathsf{A}(\phi) \in \mathcal{C}_3 . I \models \neg\phi$ then
5:         return Unsolvable Problem
6:     $Pre', Eff' = Pre, Eff$
7:     for all $a \in A$ do
8:         $P,\ E =$ PreAndEff $(a,\ \mathcal{C}_3)$
9:         $Pre'(a) = Pre(a) \wedge \bigwedge_{p\in P} p$
```

```
10:        $Eff'(a) = Eff(a) \cup E$
11:    for all LTC $c \in \mathcal{C}_3$ do
12:        $G' = G \wedge hold_c$
13:    return Classical Planning Problem $\langle F', A, I', G', Pre', Eff' \rangle$
14: function PreAndEff $(a, \mathcal{C}_3)$
15:    $P, E = \{\top\}, \{\}$
16:    for all STC$c \in \mathcal{C}_3$ do
17:        if $c$ is $\mathsf{A}(\phi)$ then $\rho = R(\neg\phi, a)$
18:        else if $c$ is $\mathsf{AO}(\phi)$ then
19:            $\rho = R(\phi, a) \wedge seen_\phi \wedge \neg\phi$
20:            $E = E \cup \{R(\phi, a) \triangleright \{seen_\phi\}\}$
21:        else if $c$ is $\mathsf{SB}(\phi, \psi)$ then
22:            $\rho = R(\phi, a) \wedge \neg seen_\psi$
23:            $E = E \cup \{R(\psi, a) \triangleright \{seen_\psi\}\}$
24:        $P = P \cup \neg\rho$
25:    for all LTC $c \in \mathcal{C}_3$ do
26:        if $c$ is $\mathsf{ST}(\phi)$ then $\rho = R(\phi, a)$
27:        else if $c$ is $\mathsf{SA}(\phi, \psi)$ then
28:            $E = E \cup \{R(\phi, a) \wedge \neg R(\psi, a) \triangleright \{\neg hold_c\}\}$
29:            $\rho = R(\psi, a)$
30:        $E = E \cup \{\rho \triangleright \{hold_c\}\}$
31:    return $P$, $E$
```

Algorithm 1 describes the full compilation. As a very first step, we create the necessary atoms (line 2) and set up the initial state to reflect the status of the trajectory constraints; in particular, we need to capture whether a LTC is already achieved in I, or if a formula (ϕ or ψ) that is necessary for the evaluation of an STC is already true in I. Depending on the kind of input constraint, we set the associated monitoring atom: for each $c = \mathsf{ST}(\phi)$ such that ϕ is already true in I[1], and for each

[1]Notice that a $\mathsf{ST}(\phi)$ constraint is always satisfied whenever $I \models \phi$, meaning that we could

$c = \mathsf{SA}(\phi, \psi)$ such that ψ is already true in I or ϕ is false in I, we set $hold_c$ true in I'. Analogously, for all constraints $\mathsf{SB}(\phi, \psi)$ ($\mathsf{AO}(\phi)$) we set $seen_\psi$ ($seen_\phi$) true in I' if ψ (ϕ) is true in I. Then we check whether any STC is already unsatisfied; if so, the problem is unsolvable.

Example 18. *Consider the constraint $c = \mathsf{SB}(a, b)$. By definition, c requires that a can become true only if b was true in some previous state. To record whether b has ever held, we introduce the monitoring atom $seen_b$. If the initial state I satisfies b, then $seen_b$ is set to true in the new initial state I'. Moreover, according to the definition of $\mathsf{SB}(a, b)$, b must hold strictly before a. Therefore, if a holds in I, then the* PDDL3 *problem is unsolvable and* TCORE *terminates.*

After the initialization phase, the algorithm iterates over all actions and constraints (lines 7-10) to modify each original action model (preconditions and effects) by considering the interactions between the constraints and the action model. In particular, TCORE uses the PreAndEff function (line 14) to determine:

1. A new precondition P that prevents the violation of any STC.
2. A new set of conditional effects E to capture the achievement of some LTC or some relevant formula.

If the constraint is a STC, the PreAndEff function determines, by regression, a condition ρ such that, if ρ holds in the state where the action is applied, the execution of such an action will violate the constraint. The condition ρ models whether the considered action:

- Makes formula ϕ false (in the case of $\mathsf{A}(\phi)$, line 17)
- Makes formula ϕ true for the second time (in the case of $\mathsf{AO}(\phi)$, line 18)
- Makes formula ϕ true while $seen_\psi$ is false (in the case of $\mathsf{SB}(\phi, \psi)$, line 21)

The regressed condition ρ is negated and then added to the new precondition P of the action. In this way, if the action execution violates the constraint in a given state, such an action is deemed inapplicable by the planner.

remove such constraint from the original problem in this case.

Example 19. *Consider the constraint* $c = \mathsf{SB}(a, b)$ *and the action act defined as follows:*

- $Pre(act) = \top$
- $Eff(act) = \{d \vee e \triangleright a\}$

To prevent the violation of c*, we must ensure that act does not make* a *true when* $seen_b$ *is false. Using regression, we determine that the execution of act in a state* s *makes* a *true only when* $s \models R(a, act)$*. By Definition 5, we have* $R(a, act) = d \vee e \vee a$*. Therefore, to prevent the violation of* $\mathsf{SB}(a, b)$*, we extend* $Pre(act)$ *with the condition* $\rho = seen_b \vee \neg(d \vee e \vee a)$*.*

In the case of STCs, conditional effects are added to keep track of whether relevant formulae have ever held in the state trajectory of the plan prefix. For instance, $\mathsf{SB}(\phi, \psi)$ requires to deal with the truth of $seen_\psi$: if an action makes ψ true, then the action must make $seen_\psi$ true as well. In this way, we can prevent applying an action a when it makes ϕ true and ψ has not held before in the current plan state trajectory (lines 21–23).

Example 20. *Consider the constraint* $c = \mathsf{SB}(a, b)$ *and the action act defined as follows:*

- $Pre(act) = e$
- $Eff(act) = \{b\}$

An occurrence of act might satisfy b*, and we need to update the atom* $seen_b$ *accordingly. We have that the execution of act in a state* s *makes* b *true when* $s \models R(b, act)$*, and* $R(b, act) = \top$*. Therefore, to keep track of the truth of* b*, we add to* $Eff(act)$ *the conditional effect* $R(b, act) \triangleright seen_b = \top \triangleright seen_b$*.*

For each LTC $c \in \mathcal{C}_3$, the algorithm yields a formula ρ that is true only in those states where the action achieves the targeted formula expressed in c. Note here the slightly different treatment for the two types of LTCs. While $\mathsf{ST}(\phi)$ only requires ϕ to be true, for $\mathsf{SA}(\phi, \psi)$ we need to signal the necessity of ψ only when ϕ becomes satisfied; we do so by introducing two conditional effects (lines 28 and 29) affecting the additional goal $hold_c$ of G' (line 11). Also observe that ϕ can become true multiple times, and each state satisfying ϕ needs to be followed by a state such that ψ is true again; this state can also be the same state in which ϕ holds, as prescribed by the semantics of PDDL3.

Example 21. *Consider the constraint $c = \mathsf{SA}(a \wedge b, d \vee e)$ and the action act defined as follows:*

- $Pre(act) = f$
- $Eff(act) = \{b, \neg d\}$

We have $R(a \wedge b, act) = a$ and $R(d \vee e, act) = e$. Therefore, the action act will be extended with the two conditional effects: $a \wedge \neg e \rhd \neg hold_c$ and $e \rhd hold_c$. The first effect captures when the constraint c is violated due to $a \wedge b$ becoming true (while $d \vee e$ is false), while the second effect captures when c is satisfied by $d \vee e$ becoming true after act.

Note that Algorithm 1 can add irrelevant preconditions and conditional effects that can easily be omitted by looking at whether regression leaves a formula unaltered. For example, for $\mathsf{A}(\phi)$, if $\rho = R(\neg\phi, a) = \neg\phi$, there is no need to extend $Pre(a)$ with $\neg\rho = \phi$ at line 18. Such optimizations are implemented but omitted here for clarity and compactness.

As trajectory constraints are monitored along the entire plan, and regression through effects provides sufficient conditions for ensuring that no STC is violated by an action and no LTC remains unsatisfied at the end of the plan, it is easy to see that the compiled problem always finds a solution that conforms with the trajectory constraints of the problem. Moreover, since the exploited regression also establishes necessary conditions, the existence of a solution in the compiled problem implies that the original problem is solvable.

Theorem 7. *Let $\Pi = \langle\langle F, A, I, G, Pre, Eff\rangle, \mathcal{C}_3\rangle$ be a PDDL3 planning problem, and let $\Pi' = \langle F', I', A, G', Pre', Eff'\rangle$ be the classical planning problem generated by Algorithm 1. A plan $\pi = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ is a solution for Π iff so does for Π'.*

Proof. Let $\pi = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ be a sequence of action labels from A, let $\tau = \langle s_0, s_1, \ldots s_n \rangle$ be the state trajectory induced by π for Π, and let $\tau' = \langle s'_0, s'_1, \ldots s'_n \rangle$ be the state trajectory induced by π for Π'. We prove that π is a solution for Π iff π is a solution for Π'. We assume that π has at least one action, as the empty plan case is straightforward. We proceed by analyzing each direction.

(π solution for $\Pi \Rightarrow \pi$ solution for Π'). **By contradiction**, π is a solution for Π but π is not a solution for Π'. If π is not a solution for Π', then at least one of the following holds:

1. $\exists\, a_i \in \pi$ such that $s'_i \not\models \mathit{Pre}'(a_i)$.

2. $s'_n \not\models G'$.

(1.) In this case, we have an action a_i in π with its preconditions not satisfied by s'_i. By definition, we have $\mathit{Pre}'(a_i) = \mathit{Pre}(a_i) \land p_1 \land p_2 \land \ldots \land p_k$. By hypothesis, π is a solution for Π, and therefore $s'_i \models \mathit{Pre}'(a_i)$[2]. Therefore, there exists a formula $p_c \in \{p_1, ..., p_k\}$ such that $s'_i \not\models p_c$. The new precondition p_c is introduced for a constraint $c \in \mathcal{C}_3$ that is either $\mathsf{A}(\phi)$, $\mathsf{AO}(\phi)$, or $\mathsf{SB}(\phi, \psi)$.

- ($c = \mathsf{A}(\phi)$) $s'_i \not\models p_c$ and p_c is $R(\phi, a_i)$. Therefore, $s_i[a_i] \not\models \phi$, and τ violates $\mathsf{A}(\phi)$ (**contradiction**).
- ($c = \mathsf{AO}(\phi)$). $s'_i \not\models p_c$ and $p_c = \neg(R(\phi, a_i) \land \mathit{seen}_\phi \land \neg\phi)$. Therefore, $s'_i \models R(\phi, a_i) \land \mathit{seen}_\phi \land \neg\phi$. We have that:
  - ϕ is false in s_i
  - $s'_i \models R(\phi, a_i)$ implies $s_{i+1} \models \phi$
  - $s'_i \models \mathit{seen}_\phi$ implies that ϕ has held in τ' before s'_i. If $I \models \phi$ then seen_ϕ has been set to true in I' by the compilation. Otherwise, a conditional effect $R(\phi, a) \triangleright \mathit{seen}_\phi$ has been triggered by some state before reaching s'_i.

[2]We remark that effects added by TCORE only affect the new variables introduced by the compilation. Therefore, $s_i \models \phi \iff s'_i \models \phi$ for any ϕ defined over F.

We have that ϕ became true at least twice in τ, and π is not a solution for π because $\mathsf{AO}(\phi)$ is violated (**contradiction**).

- ($c = \mathsf{SB}(\phi, \psi)$). $s'_i \not\models p_c$ and $p_c = \neg(R(\phi, a_i) \wedge \neg seen_\psi)$. This can be rewritten as $s'_i \models R(\phi, a_i) \wedge \neg seen_\psi$. We have $s_{i+1} \models \phi$ and $seen_\psi$ false in s'_i implies that ψ has never held:
  - No action can make $seen_\psi$ false. Hence, no conditional effect $R(\psi, a) \rhd seen_\psi$ was triggered and ψ was false in the initial state.

  We have ϕ true in s_{i+1} but ψ has never been true in the past, therefore τ violates $\mathsf{SB}(\phi, \psi)$ (**contradiction**).

(2.) $s'_n \not\models G'$. By definition, $G' = G \wedge \bigwedge_{LTC\ c \in \mathcal{C}_3} hold_c$. Since π is a solution for Π, we have that $s_n \models G$ which subsumes $s'_n \models G$. Thus, there exists a constraint $c = \mathsf{ST}(\phi)$ or $c = \mathsf{SA}(\phi, \psi)$ such that $s'_n \not\models hold_c$.

- ($c = \mathsf{ST}(\phi)$). $s'_n \not\models hold_c$ implies that ϕ has never been true in τ:
  - No action can make $hold_c$ false, so $I \not\models \phi$. Moreover, no conditional effect $R(\phi, a) \rhd hold_c$ has been triggered.

  Therefore, $\mathsf{ST}(\phi)$ is violated by τ (**contradiction**).

- ($c = \mathsf{SA}(\phi, \psi)$). $s'_n \not\models hold_c$ implies either:
  - $hold_c$ is false in the initial state and no action a_j makes $hold_c$ true. We have that $I \models \phi \wedge \neg\psi$ and, analogously to the sometime case, no action achieves ψ.
  - An action a_i makes $hold_c$ false and after that no action makes $hold_c$ true. a_i makes $hold_c$ false implies that $s'_i \models R(\phi \wedge \neg\psi, a_i)$ and therefore $s_i[a_i] \models \phi \wedge \neg\psi$. If no action makes $hold_c$ true after a_i, then no action achieves ψ after a_i, as no conditional effect $R(\psi, a) \rhd hold_c$ is triggered.

  In both cases, there exists a state s such that $s \models \phi \wedge \neg\psi$ and no future state satisfies ψ. By definition, $\mathsf{SA}(\phi, \psi)$ is violated by τ (**contradiction**).

(π solution for $\Pi' \implies \pi$ solution Π). **By contradiction**, π is a solution for Π' but π is not a solution for Π. TCORE changes the initial state, goals, preconditions, and effects of actions. We have that π is not a solution for Π for the following reasons:
1. $\exists\, i$ such that $s_i \not\models \mathit{Pre}(a_i)$. This cannot be the case, as $s'_i \models \mathit{Pre}'(a_i)$ subsumes $s_i \models \mathit{Pre}(a_i)$.
2. $s_n \not\models G$. This cannot be the case, as $s'_n \models G'$ subsumes that $s_n \models G$.
3. $\exists\, c \in \mathcal{C}_3$ such that π does not satisfy c. We proceed case by case.

- ($c = \mathsf{A}(\phi)$). If τ violates $\mathsf{A}(\phi)$ then there exists a state s such that $s \not\models \phi$. If $s = I$, TCORE detects that Π' does not admit a solution plan (**contradiction**). Otherwise, if $I \models \phi$, then there exists a state s_{i+1} such that $s_{i+1} \not\models \phi$. As $s_{i+1} \not\models \phi$, we have that $s'_i \not\models R(\phi, a_i)$. Since $\mathit{Pre}'(a_i)$ has $R(\phi, a_i)$ as a precondition, we have that a_i cannot be executed. This makes π not a solution for Π' (**contradiction**).

- ($c = \mathsf{AO}(\phi)$). τ violates $\mathsf{AO}(\phi)$ when there exists i such that ϕ held at some point in $\tau_{i-1} = \langle s_0, \ldots, s_{i-1}\rangle$, $s_i \not\models \phi$ and $s_i[a_i] \models \phi$. This implies $s_i \models R(\phi, a_i)$. TCORE add the clause $p_c = \neg(R(\phi, a_i) \wedge \mathit{seen}_\phi \wedge \neg\phi)$ to $\mathit{Pre}'(a_i)$. We have that p_c is violated by s'_i, because $s'_i \models R(\phi, a_i)$, $s'_i \not\models \phi$ and ϕ held at some point in τ_{i-1} implies seen_ϕ is true in s'_i:

  - If $I \models \phi$, then seen_ϕ holds in I' by definition.
  - Otherwise, there exists a state s_j ($j < i-1$) such that s'_j triggers the conditional effect $R(\phi, a_j) \rhd \mathit{seen}_\phi$ of $\mathit{Eff}'(a_j)$.

  Therefore, a_i cannot be executed and π is not a solution for Π'(**contradiction**).

- ($c = \mathsf{SB}(\phi, \psi)$). τ violates $\mathsf{SB}(\phi, \psi)$ when there exists a i such that ψ never held in $\tau_i = \langle s_0, \ldots, s_i\rangle$, and $s_{i+1} \models \phi$. W.l.o.g. assume that τ_i is the shortest prefix of τ that violates $\mathsf{SB}(\phi, \psi)$, that is, either ϕ is true in the initial state or s_{i+1} is the first state that satisfies ϕ when ψ has never been true. If ϕ is true in I, then TCORE detects unsolvability of Π' (**contradiction**). Otherwise, since ψ has never been true, no conditional effect $R(\psi, a) \rhd \mathit{seen}_\psi$ will ever be triggered before reaching s'_i, making seen_ψ false in s'_i. Moreover, as $s_{i+1} \models \phi$, we have

$s_i \models R(\phi, a)$. Since $Pre'(a_i)$ includes the clause $\neg(R(\phi, a_i) \wedge \neg seen_\psi)$, the action a_i cannot be executed in s'_i, and π is not a solution for Π'(**contradiction**).

- ($c = \mathsf{ST}(\phi)$). τ violates $\mathsf{ST}(\phi)$ when ϕ is never true in τ. This implies that $hold_c$ is false in I' and no conditional effect $R(\phi, a) \triangleright hold_c$ gets triggered. Therefore, $hold_c$ will be false in s'_n, and (as $hold_c$ is a goal of Π') π is not a solution for Π'(**contradiction**).

- ($c = \mathsf{SA}(\phi, \psi)$). τ violates $\mathsf{SA}(\phi, \psi)$ when $I \models \phi \wedge \neg\psi$ and ψ is never true, or when the last occurrence of $\phi \wedge \neg\psi$ is after the last occurrence of ψ. In the former case, $hold_c$ is false in I', and after that, there exists no state that triggers a conditional effect $R(\psi, a) \triangleright hold_c$. In the latter case, let s_i be the last state such that $s_i \models \psi$ and let s_j be the last state such that $s_j \models \phi \wedge \neg\psi$ ($j > i$). Then there exist a state s_k with $i \leq k < j$ such that $s_k \models R(\phi \wedge \neg\psi, a_k)$. Therefore, s'_k triggers the conditional effect $R(\phi \wedge \neg\psi, a_k) \triangleright hold_c$ of $Eff'(a_k)$ which makes $hold_c$ false. After s_k, no state that makes ψ true exists, and $hold_c$ will never be true again. Therefore, $hold_c$ is false in s'_n, making π not a solution for Π'(**contradiction**).

□

5.2 Experimental Analysis

Our experimental analysis compares TCORE[3] with five other state-of-the-art approaches that handle trajectory constraints. Specifically, we consider the last available version of three native PDDL3 planners, that is, SGPLAN [Hsu et al., 2007], OPTIC [Benton et al., 2012], MIPS [Edelkamp et al., 2006b], and two compilation-based approaches supporting LTL_f temporally extended goals, that is, the exponential (LTL-E) and polynomial (LTL-P) compilations by Baier and McIlraith [2006a] and Torres and Baier [2015], respectively. The compiled problems generated by TCORE, LTL-E, and LTL-P were solved using LAMA [Richter and Westphal, 2010].

[3]TCORE is implemented in Python, and it can be downloaded from `https://github.com/LBonassi95/tcore`.

5.2.1 Benchmark Suite

Our benchmark suite involves domains from the Fifth International Planning Competition (`https://lpg.unibs.it/ipc-5/`). Since the competition instances did not contain qualitative state trajectory constraints, but only preferences, we generated a new set of instances as follows. For each instance with preferences, we ran some planners supporting preferences, that is, LAMA and Mercury [Domshlak et al., 2015] with the compiler by Percassi and Gerevini [2019], and LPRPG-P by Coles and Coles [2011], and we collected all plans generated within 30 CPU minutes. Out of this set, we took up to five plans (those with the larger number of satisfied preferences) and, for each of them, we generated a new problem instance having all satisfied preferences converted in qualitative state trajectory constraints. This gave us a total of 416 instances: 79 for `Trucks`, 90 for `Openstack`, 55 for `Storage`, 94 for `Rover`, and 98 for `TPP`. For the LTL$_f$ approaches, we translated each PDDL3 problem into a planning problem with LTL$_f$ temporally extended goals. In particular, we used the most effective translation from PDDL3 to LTL$_f$ among those described in Section 3.2.2 and those provided by the LTL-E and LTL-P tools.

We measured the number of instances solved by each system in each domain (coverage) and the CPU time spent to find a solution, if any. For compilation-based approaches, CPU time is the compilation time plus the planning time. All experiments were performed on an Xeon Gold 6140M 2.3 GHz, with time and memory limits of 1800s and 8GB, respectively.

5.2.2 Results

Coverage Analysis. Table 5.1 gives an overall picture of the coverage obtained by all systems. The system that obtained the highest coverage is TCORE, which solved more instances than the competitors in three of the 5 domains and achieved a substantially higher total coverage overall. The performance of TCORE in `Rover` is remarkable; here TCORE solves three times the instances solved by LTL-E (the second best performer). However, there seems to be some complementary between LTL-E and TCORE. Indeed, in `TPP`, LTL-E achieves the best coverage, overcoming TCORE by a substantial margin. Since TCORE is tailored for PDDL3, one could ex-

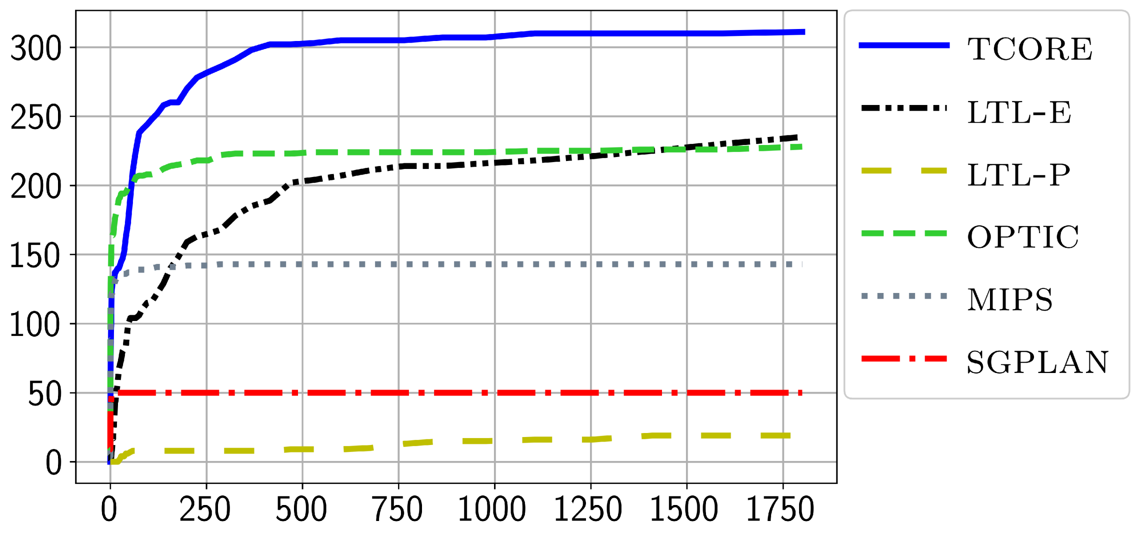


Figure 5.1: Coverage (y-axis) versus planning time (x-axis).

Domain	TCORE	LTL-E	LTL-P	OPTIC	MIPS	SGPLAN
`Rover` (94)	**92**	35	0	33	11	0
`TPP` (98)	22	**58**	1	9	1	1
`Trucks` (79)	**78**	41	0	67	29	35
`Openstack` (90)	88	78	10	**89**	**89**	0
`Storage` (55)	**31**	23	8	30	13	14
Total (416)	**311**	235	19	228	143	50

Table 5.1: Coverage achieved by TCORE, LTL-E, LTL-P, OPTIC, MIPS and SGPLAN. In parentheses, is the number of benchmark instances for a given domain.

pect that it performs always better against both LTL-E and LTL-P. TCORE works on the ground representation of the problem, and this requires grounding the planning problem upfront before compiling it. In `TPP`, TCORE did not manage to even trespass the grounding phase for several instances, causing a substantial drop in coverage. Instead, LTL-E works directly on the first-order representation and employs sophisticated decomposition techniques to obtain compact automatons, and this turned out to be very prolific. To gain more insight into the difference in performance between LTL-E and TCORE, we measured the number of atoms and effects

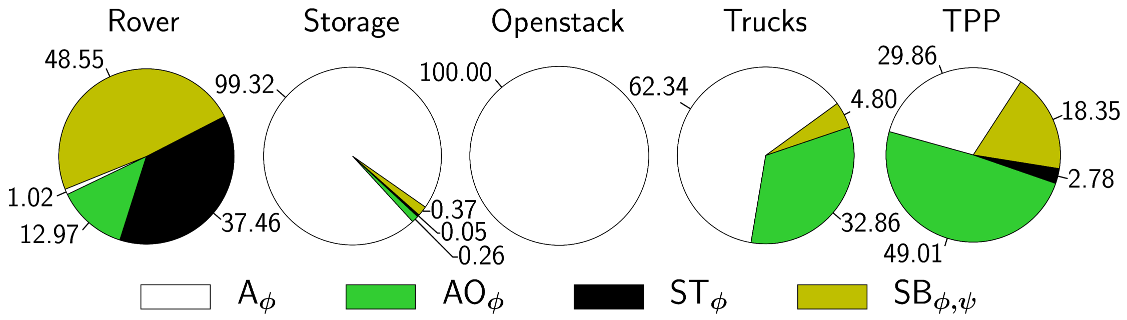


Figure 5.2: Constraints grouped by their type across all domains.

generated by these two compilations. Our findings show that the grounded instances generated by TCORE always contain fewer atoms and effects than those generated by LTL-E; TCORE generates 20.6% to 86.9% fewer atoms than LTL-E, and the difference of effects is from 1 to 3 orders of magnitude. Finally, we related performance with the number and types of constraints (Fig. 5.2); the PDDL3 native systems (OPTIC, SGPLAN) provide poor performances over instances involving LTCs, but they perform better when the constraints are only STCs (e.g., in `Openstack` for OPTIC).

CPU-time Analysis. Figure 5.1 analyzes the relationship between coverage and time allotted to each planner. The compilation-based approaches tend to increase coverage more slowly than the other approaches; this is due to the time that they spent on compilation. Native PDDL3 systems achieve high coverage in a handful of seconds, with SGPLAN exacerbating this trend to the point that its highest coverage is reached only after 4 seconds. Figure 5.3 shows a pairwise CPU-time comparison between the two best-performing compilation-based methods and between TCORE and OPTIC. TCORE is generally faster than LTL-E (which in turn dominates LTL-P), while it is slower than OPTIC due to the overhead caused by the compilation. Yet, OPTIC exceeds the time limit for many instances (in `Rover`, `TPP`, and `Trucks`) that are solved by TCORE.

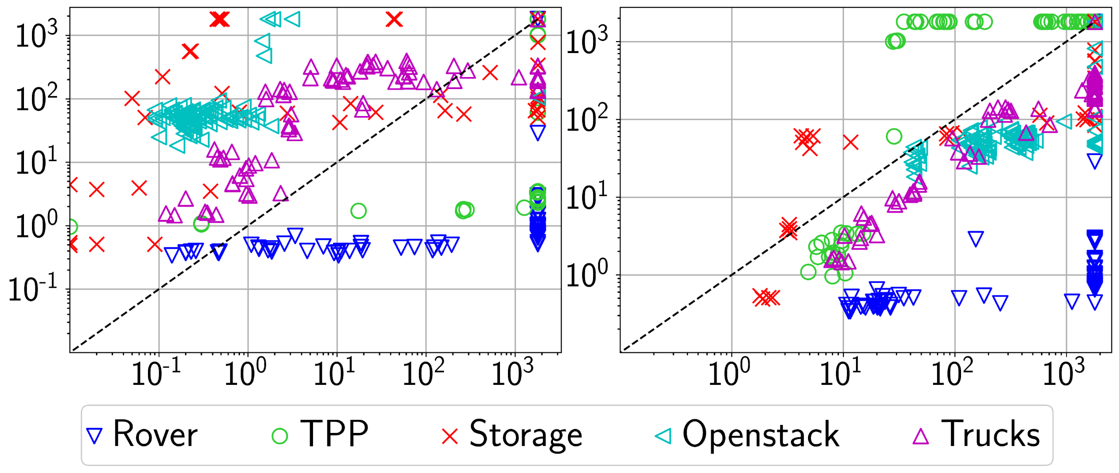


Figure 5.3: Left: TCORE (y-axis) vs OPTIC (x-axis); Right: TCORE (y-axis) vs LTL-E (x-axis). Each point (x, y) represents the CPU time spent to find a solution by TCORE (y-component) and by the compared system (x-component). A point falling under the bisector line indicates a faster time achieved by TCORE.

5.3 Discussion and Conclusions

The TCORE schema proves to be very efficient on the considered domains, as it exploits the structure of the problem actions to obtain an encoding that is more compact compared to the compilation for LTL$_f$ temporally extended goals presented in section 4.2 that relies upon explicit constructions of automatons. Moreover, the TCORE schema does not introduce any additional spurious action, and this results in an encoding that is more efficient than the polynomial compilation reviewed in Section 4.3. Compared to native approaches, TCORE can be effectively combined with state-of-the-art heuristic search planners, such as LAMA, to solve more instances overall. However, in TPP, TCORE falls behind LTL-E, which can decompose complex first-order formulas to build smaller automatons. Understanding how TCORE can employ a similar technique, possibly using a first-order schema, remains a future work.

Chapter 6

Handling PAC Constraints via Compilation

In this chapter, we report on a compilation-based technique to effectively deal with PAC action constraints. The proposed approach is polynomial, generates minimal overhead in terms of additional atoms, and preserves the size of solutions (no additional spurious action is required).

We then experimentally study the usefulness of PAC constraints as a tool to express control knowledge. The experimental results show that, for a class of problems, the performance of a classical planner can be significantly improved by exploiting the knowledge expressed by action constraints and handled by our compilation, while the same knowledge turns out to be less beneficial when specified as state trajectory constraints and handled by TCORE and LTL-E [Baier and McIlraith, 2006a].

6.1 Compiling PAC Constraints Away

In this section, we propose a compilation schema, called PAC-C (PAC compiler), that translates a PAC problem into an equivalent classical planning problem. As done for PDDL3 constraints, our approach for PAC distinguishes action constraints into two classes: *Safety PAc Constraints* (SPAC) and *Liveness PAc Constraints* (LPAC). Intuitively, SPACs are constraints used to express properties that must not be violated at any time in the plan, while LPACs enforce that certain actions must occur in every solution plan. SPACs are: $\mathsf{A}(\phi)$, $\mathsf{SB}(\phi, \psi)$, $\mathsf{AO}(\phi)$ and $\mathsf{AX}(\phi, \psi)$; LPACs are: $\mathsf{ST}(\phi)$, $\mathsf{SA}(\phi, \psi)$ and $\mathsf{PA}(\phi_1, \ldots, \phi_k)$. PAC-C works by preventing the execution of actions that would violate some SPACs and forcing the planner to include in the plan the actions necessary to satisfy all LPACs on a state-dependent basis.

Actions that cannot appear in the plan at some time step t depend on actions scheduled before t.

Example 22. *Consider the constraint* $\mathsf{AO}(a)$*. Having* a *in the plan at a time* t *should be prevented if* a *has already been scheduled in the plan prefix preceding* t*.*

The same logic applies to the actions that still need to be included in the plan to satisfy some LPAC. Therefore, similarly to what was done in Section 5.1, we extend the planning states with the necessary information about the presence in the plan under construction of the actions relevant to the constraints. We do so by introducing a set of fresh atoms, built by taking into account the constraint at hand, as described below.

SPAC Atoms. For every $\mathsf{AO}(\phi)$ and $\mathsf{SB}(\phi, \psi)$, atoms $done_\phi$ and $done_\psi$ are used to record whether ϕ and ψ have ever held. For every $\mathsf{AX}(\phi, \psi)$, atom $request_\psi$ is used to signal that the formula ϕ is satisfied at a plan step t, and the planner has to schedule an action $a \in \psi$ immediately after t.

LPAC Atoms. For every $\mathsf{ST}(\phi)$ and $\mathsf{SA}(\phi, \psi)$, atoms got_ϕ and $got_{\phi,\psi}$ are used to record whether or not the constraint is satisfied by the prefix plan. For every $\mathsf{PA}(\phi_1, \ldots, \phi_k)$, we add a set of atoms called *stage atoms* to keep track of the progress of the pattern in the plan. The set of stage atoms is defined as follows:

$$StageAtoms(\mathcal{C}_A) = \bigcup_{c=\mathsf{PA}(\phi_1,\ldots,\phi_k)\in\mathcal{C}_A} \{stage_c^1, \ldots, stage_c^k\}$$

Atoms $stage_c^i$ ($i \in \{1, \ldots, k\}$) will hold in a plan state s *iff* $\mathsf{PA}(\phi_1 \ldots \phi_i)$ is satisfied by the plan prefix up to s.

Example 23. *Consider the constraint $c = \mathsf{PA}(a_1, a_2, a_3 \vee a_4)$. The stage atoms relative to c are $\{stage_c^1, stage_c^2, stage_c^3\}$. Each atom represents the stage of satisfaction of c. For example, $stage_c^2$ indicates that $\mathsf{PA}(a_1, a_2)$ has been satisfied. Therefore, the stage atom $stage_c^3$ signals that the whole pattern has been satisfied.*

Algorithm 2 The PAC-C algorithm

```
 1: function PAC-C($\langle\langle F, A, I, G, Pre, Eff\rangle, \mathcal{C}_A\rangle$)
 2:     ▷ Phase (I)
 3:     $F' = F \cup SPAC\text{-}atoms \cup LPAC\text{-}atoms$
 4:     $I' = I \cup \bigcup_{\mathsf{SA}(\phi,\psi)} got_{\phi,\psi}$
 5:     ▷ Phase (II)
 6:     $Pre', Eff' = Pre, Eff$
 7:     $A' = \{a \mid a \in A$ and for each $\mathsf{A}(\phi) \in \mathcal{C}_A,\ a \in \phi\}$
 8:     for all $a \in A$ do
 9:         $P,\ E$ = PreAndEff $(a,\ \mathcal{C}_A)$
10:         $Pre'(a) = Pre(a) \wedge \bigwedge_{p \in P} p$
11:         $Eff'(a) = Eff(a) \cup E$
12:     ▷ Phase (III)
13:     $G' = G \ \wedge \bigwedge_{\mathsf{SA}(\phi,\psi)\in\mathcal{C}_A} got_{\phi,\psi} \wedge \bigwedge_{\mathsf{ST}(\phi)\in\mathcal{C}_A} got_{\phi} \wedge$
            $\bigwedge_{\mathsf{AX}(\phi,\psi)\in\mathcal{C}_A} \neg request_{\psi} \wedge \bigwedge_{c=\mathsf{PA}(\phi_1,\ldots,\phi_k)\in\mathcal{C}_A} stage_c^k$
14:     return $\langle F', A', I', G', Pre', Eff'\rangle$
15: function PREANDEFF $(a, \mathcal{C}_A)$
16:     $P, E = \{\top\}, \{\}$
17:     for all $c \in SPAC(\mathcal{C}_A)$ do
18:         if $c = \mathsf{AO}(\phi)$ and $a \in \phi$ then
19:             $P = P \cup \neg done_{\phi}$
20:             $E = E \cup \{done_{\phi}\}$
21:         if $c = \mathsf{SB}(\phi, \psi)$ then
22:             if $a \in \phi$ then $P = P \cup done_{\psi}$
```

```
23:            if $a \in \psi$ then $E = E \cup \{done_\psi\}$
24:        if $c = \mathsf{AX}(\phi, \psi)$ then
25:            if $a \in \phi$ then $E = E \cup \{request_\psi\}$
26:            else if $a \in \psi$ then $E = E \cup \{\neg request_\psi\}$
27:            if $a \notin \psi$ then $P = P \cup \neg request_\psi$
28:    for all $c \in LPAC(\mathcal{C}_A)$ do
29:        if $c = \mathsf{ST}(\phi)$ and $a \in \phi$ then $E = E \cup \{got_\phi\}$
30:        if $c = \mathsf{SA}(\phi, \psi)$ then
31:            if $a \in \psi$ then $E = E \cup \{got_{\phi,\psi}\}$
32:            if $a \in \phi$ and $a \notin \psi$ then $E = E \cup \{\neg got_{\phi,\psi}\}$
33:        if $c = \mathsf{PA}(\phi_1, \ldots, \phi_k)$ then
34:            for all $\phi_i \in \langle \phi_1 \ldots \phi_k \rangle \cdot a \in \phi_i$ do
35:                $E = E \cup \begin{cases} \{stage_c^{i-1} \triangleright stage_c^i\} & \textbf{if } i > 1 \\ \{stage_c^1\} & \textbf{otherwise} \end{cases}$
36:    return $P$, $E$
```

Compilation schema. Algorithm 2 specifies the full compilation schema, called PAC-C. There are three different phases: (I) creation of necessary atoms and setup of the initial state to reflect the status of the constraints; (II) revision of the preconditions and effects of relevant actions; (III) setup of the goal to enforce the satisfaction of all LPACs and AX constraints.

Phase (I). The necessary SPAC and LPAC atoms are created and initialized (lines 3-4). When a plan has no actions, all $\mathsf{SA}(\phi, \psi)$ constraints are satisfied, and so the corresponding got_ϕ atoms are set to true in the initial state.

Phase (II). The algorithm prunes all actions that do not satisfy the always constraints. Then it modifies the actions to keep all constraints in check. For a SPAC,

PAC-C determines new preconditions that must be fulfilled for the actions that interact with the constraint. In particular, PAC-C prevents having in the plan actions that (a) make ϕ true a second time (in the case of an $\mathsf{AO}(\phi)$), (b) make ϕ true if $done_\psi$ is false (in the case of a $\mathsf{SB}(\phi, \psi)$), and (c) cannot make ψ true when there is a request for it triggered by the previous action (in the case of an $\mathsf{AX}(\phi, \psi)$). For SPACs, new effects are added to keep track of the execution of relevant actions.

Example 24. *An $\mathsf{AX}(\phi, \psi)$ constraint requires restricting the possible actions in the plan at the next time step when ϕ becomes true. If an action a in the plan satisfies ϕ at some time t, the triggered request for some action satisfying ψ at time $t + 1$ is encoded by disallowing the occurrence of any action not satisfying ψ (lines 25-27). If an action does not trigger the constraint and satisfies ψ instead, then $\neg request_\psi$ is added to its effects (lines 25-26), disabling the request of ψ demanded by the constraint. Consider the $\mathsf{AX}(a_1, a_2 \vee a_3)$ constraint. By following the flow of the compilation for every action, we observe that:*

- *$request_{a_2 \vee a_3}$ is added to $Eff(a_1)$. This effect captures that if a_1 is executed, then either a_2 or a_3 has to be executed at the next step.*
- *For every action a_i with $a_i \neq a_2$ and $a_i \neq a_3$, $Pre(a_i)$ is extended with $request_{a_2 \vee a_3}$. This is done to prevent the execution of any action different from a_2 and a_3 when $request_{a_2 \vee a_3}$ holds.*
- *The effects of a_2 and a_3 are extended with the effect $\neg request_{a_2 \vee a_3}$. These new effects are meant to signal that the request has been fulfilled.*

For LPACs, PAC-C adds a set of effects to keep track of the relevant actions that appear in the plan. A $\mathsf{ST}(\phi)$ constraint requires that at least one action that satisfies ϕ appears sometime in the plan. The atom got_ϕ is then added to all actions that make ϕ true. For a $\mathsf{SA}(\phi, \psi)$ constraint, PAC-C adds effects to signal the necessity of ψ whenever ϕ becomes true (lines 31-32). For a $\mathsf{PA}(\phi_1, \ldots, \phi_k)$ constraint, the algorithm checks if an action satisfies any formula in $\{\phi_1 \ldots \phi_k\}$. For example, if an action a in the plan makes formula ϕ_i true and $\mathsf{PA}(\phi_1 \ldots \phi_{i-1})$ is already satisfied by the plan prefix up to a, then $\mathsf{PA}(\phi_1 \ldots \phi_i)$ will become satisfied. PAC-C keeps track of this information by adding the conditional effect $stage_c^{i-1} \triangleright stage_c^i$ to all actions satisfying ϕ_i (line 35).

Example 25. *Consider the constraint* $c = \mathsf{PA}(a_1, a_2, a_3 \vee a_4)$*. The compilation extends the effects* $\mathit{Eff}(a_1)$ *with* $\{stage_c^1\}$*, the effects* $\mathit{Eff}(a_2)$ *with* $\{stage_c^1 \triangleright stage_c^2\}$*, and both the effects* $\mathit{Eff}(a_3)$ *and* $\mathit{Eff}(a_4)$ *with* $\{stage_c^2 \triangleright stage_c^3\}$*. If, for example,* a_1*,* a_2*, and* a_4 *are executed (sometimes) in the plan in this order, then it is easy to see that the newly added conditional effects will make* $stage_c^1$ *true after* a_1*,* $stage_c^2$ *true after* a_2*, and* $stage_c^3$ *after* a_4*.*

Phase (III). The last step consists in setting up the new goals of the problem: all the $\mathsf{ST}(\phi)$, $\mathsf{SA}(\phi, \psi)$, $\mathsf{AX}(\phi, \psi)$ and $\mathsf{PA}(\phi_1, \ldots, \phi_k)$ constraints must be satisfied. This means that in the final state, all got_ϕ, $got_{\phi,\psi}$ and $stage_c^k$ atoms have to hold, and there is no pending request of an action to satisfy some $\mathsf{AX}(\phi, \psi)$ (line 13).

Example 26. *Consider the* $\mathsf{AX}(a_1, a_2 \vee a_3)$ *constraint. If* a_1 *is executed, then we must schedule either* a_2 *or* a_3 *in the next step. Therefore, the plan cannot terminate after* a_1*. Hence, we add* $\neg request_{a_2 \vee a_3}$ *to* G' *to prevent the plan from ending when there is a pending request. Consider the constraint* $c = \mathsf{PA}(a_1, a_2, a_3 \vee a_4)$*. To force the completion of the pattern, we add* $stage_c^3$ *to* G'*.*

The additional preconditions and effects of the compilation prevent the planner from generating any sequence of actions that violates one or more SPACs, while the additional goals force the planner to satisfy all LPACs. The following theorem states that any plan of the original problem Π is a solution of Π if and only if the same plan with its actions modified by PAC-C is a solution for the translated problem Π'. Note that the original and modified plan have exactly the same lengths.

Theorem 8. *Let* $\Pi = \langle\langle F, A, I, G, Pre, \mathit{Eff}\rangle, \mathcal{C}_A\rangle$ *be a* PAC *problem and* $\Pi' = \langle F', A', I', G', Pre', \mathit{Eff}'\rangle$ *the problem obtained by compiling* Π *through Algorithm 2. A plan* $\pi = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ *is a solution for* Π *iff so does for* Π'*.*

Proof. We focus on the case in which the plan has at least one action. We prove the two directions of the implication in the claim separately. We denote with

$\tau = \langle s_0, s_i, \ldots s_n \rangle$ the state trajectory induced by π for Π, and with $\tau' = \langle s'_0, s'_i, \ldots s'_n \rangle$ the state trajectory induced by π for Π'.

(π solution for $\Pi \implies \pi$ solution for Π'). **By contradiction**, assuming that π is a solution for Π but π is not a solution for Π'. If π is not a solution, at least one of the following three cases has to hold:

(I) $\exists\, t$ such that $\pi[t] \notin A'$.

(II) $\exists\, t$ such that $s'_t \not\models \mathit{Pre}(\pi[t])$.

(III) $s'_n \not\models G'$.

(I) If an action $\pi[t] \notin A'$ then (line 7 of pseudocode) $\exists\ \mathsf{A}(\phi) \cdot \pi[t] \not\models \phi$. $\pi[t]$ violates an always constraints, therefore π is not a solution for Π (**contradiction**).
(II) The precondition of $\pi[t]$ is as follows: $\mathit{Pre}'(\pi[t]) = \mathit{Pre}(\pi[t]) \wedge p_1 \wedge \ldots \wedge p_k$. If $s'_t \not\models \mathit{Pre}(\pi[t])$ then also $s_t \not\models \mathit{Pre}(\pi[t])$ and π would not be solution for Π (**contradiction**). Therefore $\exists\ p_c \in \{p_1, \ldots, p_k\}$ such that $s'_t \not\models p_c$. p_c is a new precondition added by Algorithm 2 due to the compilation of either an $\mathsf{AO}(\phi)$, $\mathsf{SB}(\phi, \psi)$ or $\mathsf{AX}(\phi, \psi)$ constraint.

- ($\mathsf{AO}(\phi)$). $p_c = \neg \mathit{done}_\phi$, which implies that $\mathit{done}_\phi \in s'_t$. $\mathit{done}_\phi \in s'_t$ plus $\mathit{done}_\phi \notin I'$ imply the existence of an action $\pi'[j]$ ($j < t$) with done_ϕ as an effect. This also means that $\pi[j] \in \phi$. As $p_c = \neg \mathit{done}_\phi$, $\pi[t] \in \phi$, too; therefore both $\pi[j] \in \phi$ and $\pi[t] \in \phi$ making π violate the $\mathsf{AO}(\phi)$ (**contradiction**).

- ($\mathsf{SB}(\phi, \psi)$). $p_c = \mathit{done}_\psi$, therefore $\mathit{done}_\psi \notin s'_t$. Since no action can delete done_ψ, it follows that there exists no index j with $j < t$ such that $\pi'[j]$ achieves done_ψ, which in turn implies that there is no index j with $j < t$ such that $\pi[j] \in \psi$ either (see line 23 of pseudocode). Since $p_c = \mathit{done}_\psi$, we also know that $\pi[t] \in \phi$. Therefore, we have that plan π has the action $\pi[t] \in \phi$, but before t there is no action that makes ψ in π; hence π violates $\mathsf{SB}(\phi, \psi)$ (**contradiction**).

- ($\mathsf{AX}(\phi, \psi)$). $p_c = \neg \mathit{request}_\psi$, which implies that $\mathit{request}_\psi \in s'_t$. $\mathit{request}_\psi \in s'_t$ implies that $\pi[t-1] \in \phi$ (line 25 of pseudocode). $p_c = \neg \mathit{request}_\psi$, therefore

$\pi[t] \notin \psi$. We have that π violates $\mathsf{AX}(\phi, \psi)$ because $\pi[t-1] \in \phi$ and $\pi[t] \notin \psi$ (**contradiction**).

(III) $G' = G \wedge \bigwedge_{\mathsf{SA}_{\phi,\psi} \in \mathcal{C}_A} got_\psi \wedge \bigwedge_{\mathsf{ST}_\phi \in \mathcal{C}_A} got_\phi \wedge \bigwedge_{\mathsf{AX}_{\phi,\psi} \in \mathcal{C}_A} \neg request_\psi \wedge \bigwedge_{c = \mathsf{PA}_{\phi_1 \dots \phi_k} \in \mathcal{C}_A} stage^k_c$. If $s'_n \not\models G$ then also $s_n \not\models G$; thus π is not a solution for Π (**contradiction**). Therefore, the only way $s'_n \not\models G'$ holds is when one among got_ϕ, got_ψ, $stage^k_c$ does not hold in s'_n or $request_\psi \in s'_n$. We proceed case by case:

- (got_ϕ). $got_\phi \notin s'_n$ implies that there is no action that achieves got_ϕ in π. Indeed got_ϕ can never be deleted by any action. This means that there is no action satisfying ϕ in π, too. Therefore, π violates $\mathsf{ST}(\phi)$ (**contradiction**).

- (got_ψ). $got_\psi \in I'$ (by definition) and $got_\psi \notin s'_n$ (by absurd assumption) imply that there exists an action $\pi[t_1]$ that deletes got_ψ, and no action after t_1 that adds got_ψ. This means that there exists an index t_1 with $1 \leq t_1 \leq |\pi|$ such that $\pi[t_1] \in \phi$, and $\forall t_2$ with $t_1 \leq t_2 \leq |\pi|$ we have that $\pi[t_2] \notin \psi$. By definition of $\mathsf{SA}(\phi, \psi)$, π violates such a constraint (**contradiction**).

- ($stage^k_c$). $stage^k_c$ is added by the algorithm for a $c = \mathsf{PA}(\phi_1, \dots \phi_k)$ constraint. Due to the structure of the compiled problem, atom $stage^k_c$ is true in the last state only if there exists a sequence of actions $\langle a_1, \dots, a_k \rangle$ from π ordered as in π such that $stage^1_c \in \mathit{Eff}'(a_1)$, and for each $i > 1$ $stage^{i-1}_c \rhd stage^i_c \in \mathit{Eff}'(a_i)$. Since we are assuming $stage^k_c \notin s'_n$, such a sequence of actions $\langle a_1, \dots, a_k \rangle$ does not exist. From the algorithm (line 35 of pseudocode), it follows that an action a_1 has $stage^1_c$ as an effect only if $a_1 \in \phi_1$, and an action a_i has $stage^{i-1}_c \rhd stage^i_c$ as an effect only if $a_i \in \phi_i$. This implies that there is no sequence of actions $\langle a_1, \dots, a_k \rangle$ from π, ordered as in π, such that $a_i \models \phi_i$ for all $i \in \{1, \dots, k\}$. Therefore, π violates $\mathsf{PA}(\phi_1, \dots \phi_k)$ (**contradiction**).

- ($request_\psi$). $request_\psi \in s'_n$ implies that there exists an action $\pi[t]$ that adds $request_\psi$ and each other subsequent action does not delete $request_\psi$. Since $request_\psi \in \mathit{Eff}'(\pi[t])$, and hence $\pi[t] \in \phi$ too, π violates $\mathsf{AX}(\phi, \psi)$ (**contradiction**) since, by construction, there will be no action a after $\pi[t]$ such that $a \in \psi$ and $a \notin \phi$.

(π solution for Π' $\implies$ π solution for Π). **By contradiction**, assuming π is a solution for Π' but π is not a solution for Π. Algorithm 2 changes the initial state, goals, preconditions, and effects of actions. We can have that π is not a solution for Π for the following reasons:
(I) $\exists\, t$ such that $s_t \not\models \mathit{Pre}(\pi[t])$. This cannot be the case, as $s'_t \models \mathit{Pre}'(\pi[t])$ subsumes $s_t \models \mathit{Pre}(\pi[t])$.
(II) $s_n \not\models G$. This cannot be the case, as $s'_n \models G'$ subsumes $s_n \models G$.
(III) $\exists\, c \in \mathcal{C}_A$ such that π does not satisfy c:

- ($c = \mathsf{A}(\phi)$). π that does not satisfy $\mathsf{A}(\phi)$ implies that $\exists\, t$ such that $\pi[t] \notin \phi$. It follows that action $\pi[t] \notin A'$, line 7 in Algorithm 2 (**contradiction**).

- ($c = \mathsf{AO}(\phi)$). π that does not satisfy $\mathsf{AO}(\phi)$ implies that there exist indexes t_1 and t_2 with $1 \leq t_1 < t_2 \leq |\pi|$ such that $\pi[t_1], \pi[t_2] \in \phi$. By construction, $\mathit{done}_\phi \in \mathit{Eff}'(\pi[t_1])$, and $\pi[t_2]$ will not be applicable since $\neg \mathit{done}_\phi$ is a conjunct of $\mathit{Pre}'(\pi[t_2])$, and there is no action that can ever make done_ϕ false (**contradiction**).

- ($c = \mathsf{SB}(\phi, \psi)$). π does not satisfy $\mathsf{SB}(\phi, \psi)$ implies that there exists an index t_1 with $1 \leq t_1 \leq |\pi|$ such that $\pi[t_1] \in \phi$ and for every index t_2 with $1 \leq t_2 < t_1$ $\pi[t_2] \notin \psi$. Note that done_ψ is a conjunct of $\mathit{Pre}'(\pi[t_1])$ by construction, and that no action executed before t_1 has done_ψ as an effect (lines 22 and 23 of pseudocode). Since $\mathit{done}_\psi \notin I'$, we have that $\mathit{done}_\psi \notin s'_{t_1}$ and $\pi[t_1]$ cannot be executed in π(**contradiction**).

- ($c = \mathsf{ST}(\phi)$). π that does not satisfy $\mathsf{ST}(\phi)$ implies that there is no action $a = \pi[t]$ such that $a \in \phi$. By construction, no action with got_ϕ as an effect is executed in π. Since got_ϕ is false in the initial state I', got_ϕ will be false in the last state reached by π. Therefore, π is not a solution as G' requires got_ϕ true (**contradiction**).

- ($c = \mathsf{SA}(\phi, \psi)$). π that does not satisfy $\mathsf{SA}(\phi, \psi)$ implies that there exists an index t_1 with $1 \leq t_1 \leq |\pi|$ such that $\pi[t_1] \in \phi$ and for every index t_2 with $t_1 \leq t_2 \leq |\pi|$ we have that $\pi[t_2] \notin \psi$. This implies that $\mathit{Eff}'(\pi[t_1])$ deletes got_ψ

(line 32 of pseudocode) and that no action executed after t_1 adds got_ψ. Since got_ψ is a conjunct of G', π does not achieve the goal (**contradiction**).

- $(c = \mathsf{AX}(\phi, \psi))$. π does not satisfy $\mathsf{AX}(\phi, \psi)$ implies that either the last action of π (a_{n-1}) makes ϕ true or there are two subsequent actions $\pi[t]$ and $\pi[t+1]$ such that $\pi[t] \in \phi$ and $\pi[t+1] \notin \psi$. In the first case, by construction, we have that a'_{n-1} adds $request_\psi$. Since $request_\psi$ is a conjunct of G', π does not achieve the goal (**contradiction**). In the other case, we have that $Eff'(\pi[t])$ achieve $request_\psi$, and that $\neg request_\psi$ is a conjunct of $Pre'(\pi[t+1])$ (lines 25 and 27 of pseudocode). Therefore $\pi[t+1]$ is not applicable (**contradiction**).

- $(c = \mathsf{PA}(\phi_1, \dots \phi_k))$. π does not satisfy $\mathsf{PA}(\phi_1, \dots \phi_k)$ implies that it does not exist a sequence $\langle a_1, \dots, a_k \rangle$ of actions from π, ordered as in π, such that for every $i \in \{1, \dots, k\}\, a_i \in \phi_i$. From the algorithm (line 35 of pseudocode) it follows that an action a'_1 has $stage_c^1$ as an effect only if $a_1 \in \phi_1$, and that an action a'_i has $stage_c^{i-1} \triangleright stage_c^i$ as an effect only if $a_i \in \phi_i$. This implies that there exists no sequence of actions $\langle a_1, \dots, a_k \rangle$ from π, ordered as in π, such that $stage_c^1 \in Eff'(a_1)$ and for every other action $stage_c^{i-1} \triangleright stage_c^i \in Eff(a'_i)$. Therefore, by construction, $stage_c^k$ will not hold in the last state reached by π (**contradiction**).

$\square$

6.2 Experimental Analysis

Our experiments are aimed at evaluating the usefulness of action constraints as knowledge that can be effectively exploited to improve solution coverage and solution quality (expressed as plan length). We evaluate the behavior of a classical planner with/without this (compiled) knowledge. For comparison, we also investigate how the classical planner can be enhanced by using the same control knowledge expressed as (compiled) state trajectory constraints formulated in LTL$_f$ or PDDL3. As a classical planner, we used LAMA [Richter and Westphal, 2010], which was run on the original benchmark problems and the corresponding problems extended with control knowledge. This knowledge was compiled by three different methods: PAC-C[1] (for action constraints), TCORE (for PDDL3 constraints), and LTL-E (for LTL$_f$ constraints) [Baier and McIlraith, 2006a]. To our knowledge, TCORE and LTL-E are the most effective approaches for dealing with the considered class of constraints.

We measured performance in terms of the number of solved instances (coverage), the CPU time of the planner, and the length of the solution plan (when found). For the compilation-based approaches, CPU time includes compilation time. All experiments ran on an Xeon Gold 6140M 2.3 GHz, with time and memory limits of 1800s and 8GB, respectively.

6.2.1 Benchmark Design

Since there are no available benchmarks that feature action constraints, we generated a new benchmark suite starting from the problems of the 5th International Planning Competition. We considered the following domains: `Trucks`, `Storage`, `TPP`, `Openstack` and `Rover`. All original instances of `Rover`, `TPP`, and `Openstack` are easily solved by LAMA, while the planner struggles to find solutions for some instances of `Trucks` and `Storage`. For this reason, we designed our action constraints with different objectives for the two groups of domains: in `Trucks` and `Storage`, constraints were designed to boost problem coverage, while in the other domains, the constraints were designed to improve solution quality. Our benchmark suite involves 160 instances: 30 in `Trucks`, `Storage`, `TPP`, and `Openstack`, and 40 in `Rover`. To evaluate the use of LTL$_f$ and PDDL3, for each instance, we generated two additional instances: one encoding the action constraints into an equivalent formulation in LTL$_f$, and the other encoding an equivalent instance using PDDL3 constraints. Such instances were not formulated starting from the action constraints specification; that is, the constraint knowledge was directly formulated into action constraints or state constraints (PDDL3 or LTL$_f$), without going through action constraints first. Note that the conversion to PDDL3 has been possible only for a subset of the considered domains. In what follows, we describe the constraints introduced in each domain.

[1]PAC-C can be downloaded at `https://github.com/LBonassi95/PAC-C`.

`TPP`. This domain encodes the Traveling Purchaser Problem (TPP) Ramesh [1981]. We have a set of markets and a set of products. Each market sells different products in different quantities, and the objective is to collect and deliver at the depot the required quantity of products by using trucks. Each truck can drive to different locations, buy, load, and unload products. While a single truck is sufficient to visit all markets, we observed that a greedy planner tends to move all trucks back and forth from depots to markets producing very bad quality plans. To overcome this problem we forced the planner to use only a single truck driving in a subset of the roads via the following `always` constraints:

$$\begin{aligned}
&\texttt{always}(\forall\, ?from,\ ?to \cdot\ \neg Drive(truck_2, ?from, ?to))\\
&\texttt{always}(\neg Drive(truck_1, market_2, depot_1) \wedge\\
&\qquad\quad \neg Drive(truck_1, market_1, market_2))
\end{aligned}$$

Moreover, we forced the truck to visit markets in a precise order through the following `pattern` constraint:

$$\begin{aligned}
\texttt{pattern}(&Drive(truck_1, depot_1, market_2),\\
&Drive(truck_1, market_2, market_1),\\
&Drive(truck_1, market_1, depot_1))
\end{aligned}$$

In `TPP` a planner is allowed to move a truck multiple times from depots to markets to deliver a product; a better strategy is gathering the required quantity of a product and then going back to unload the product. We enforced this by constraint

$$\texttt{sometime}(\exists\, ?market \cdot Load(product_1, truck_1, ?market, level_x))$$

This constraint is repeated for every product, and in each case, it is satisfiable as trucks have unlimited storage space. Finally, we require that after buying a product, that product is immediately loaded in the truck:

$$\begin{array}{l}\texttt{always-next(}\\ \qquad \exists\, ?product, \exists\, ?market \cdot\ Buy(truck_1, ?product, ?market),\\ \qquad \exists\, ?product, \exists\, ?market, \exists\, ?level \cdot Load(?product, truck_1, ?market, ?level))\end{array}$$

For `TPP`, we also designed an equivalent formulation using LTL$_f$ constraints by transferring the constraints over actions to constraints over states. This can be done by inspecting the action structure and enforcing to traverse only those states that would be traversed by the actions. E.g., we formulate pattern constraints in LTL$_f$ as follows:

$$\begin{array}{l}\mathsf{F}(At(truck_1, depot_1) \wedge \mathsf{X}(At(truck_1, market_2) \wedge\\ \quad \mathsf{X}(\mathsf{F}(At(truck_1, market_2) \wedge \mathsf{X}(At(truck_1, market_1) \wedge\\ \qquad \mathsf{X}(\mathsf{F}(At(truck_1, market_1) \wedge \mathsf{X}(At(truck_1, depot_1)))))))))\end{array}$$

Overall, we have one constraint in the smallest benchmark instance and 22 constraints in the largest one.

`Storage` In this domain the goal is to move some crates inside depots with a set of hoists. Hoists can operate inside and outside depots, lift crates, and drop them into depots or containers. Finding a solution in `Storage` can be a difficult task because leaving a crate right at the entrance of a depot will prevent hoists from moving into that depot in the future. To aid the planner, we forced crates to be positioned starting from the storage areas further away from the entrance. This was encoded using a `pattern` constraint. We also prevented the unnecessary lifting of a crate via the following constraint:

$$\forall\, ?crate \cdot\ \texttt{at-most-once}(\exists\, ?hoist \cdot\ Lift(?hoist, ?crate)).$$

All constraints were also translated into PDDL3 and LTL$_f$. E.g., the previous `at-most-once` constraint was translated in LTL$_f$ by the following formula, for each crate c, where $\phi = \exists\, ?hoist \cdot\ Lifting(?hoist, c)$:

$$\mathsf{G}(\phi \Rightarrow (\phi\, \mathsf{U}\, (\mathsf{G}(\neg\phi) \vee \mathsf{last})))$$

Each benchmark instance has from 2 to 21 constraints.

`Trucks.` This is a logistics domain concerning delivering packages to different locations by some trucks. The space inside the trucks is partitioned into areas, and a package can be loaded in an area only if all areas in between the door and the area in consideration are free. This requirement must also hold when a package is unloaded. In addition, some packages must be delivered by a deadline. To improve problem coverage, we used `at-most-once` constraints to impose that every package is loaded inside a truck at most one time. Overall, each benchmark instance has from 3 to 20 constraints.

`Rover.` The objective is to acquire data about soil, rocks, and images of a planet. Data are gathered by a set of rovers that can move across waypoints. Each rover has different equipment to either sample the soil/rocks or take images. The acquired data must be communicated to the lander. In this domain, many actions are unnecessary to achieve the goal. E.g., if the goal does not require the data of the rock located at some waypoint, there is no need to sample it. Such actions can be forbidden through `always` constraints. Moreover, by a set of `sometime-before` constraints, we required that the rovers communicate data only after *all* the needed data have been gathered. Finally, we broke some symmetrical solutions by forcing an order to the communications:

$$\begin{array}{l}\texttt{sometime-before}(\\ \quad \exists\, ?rover \cdot\ Send\_soil\_data(?rover, waypoint_x),\\ \quad \exists\, ?rover \cdot\ Send\_rock\_data(?rover, waypoint_y))\end{array}$$

These constraints were also formulated in LTL$_f$ and PDDL3. E.g., the previous action constraint in LTL$_f$ is:

$$\begin{aligned}&(\neg\phi \wedge \psi)\,\mathsf{R}\,(\neg\phi)\\ &\text{with } \phi = Communicated\_soil\_data(waypoint_x)\\ &\qquad \psi = Communicated\_rock\_data(waypoint_y)\end{aligned}$$

Each benchmark instance has from 6 to 138 constraints.

`Openstack`. This domain models a combinatorial optimization problem where a set of orders must be shipped. To start the production of an order, a new "stack" must be opened. Each order can be shipped only if a given set of products associated with that order has been produced. Once an order is shipped, the previously occupied stack can be used for new orders. To make a product, all orders that include it must be in a stack. The objective is to find a production that minimizes the number of opened stacks. The domain actions can open a new stack, start a new order, ship a finished order, set up the machine for production, and make a product. An optimal solution plan has the fewest open-new-stack actions. To aid the planner in finding a good quality solution, we used two `always-next` constraints: the first requires that after opening a new stack an order is immediately started; the second requires that after setting up the machine, the product is immediately made. Every instance of `Openstack` features these two constraints.

6.2.2 Experimental Results

Coverage. Table 6.1 shows the overall coverage achieved using BASELINE (LAMA run on the original instances without constraints), PAC-C, LTL-E and TCORE. We first comment on the results obtained for `Storage` and `Trucks`, the two domains featuring constraints formulated to improve coverage. The action constraints in `Trucks` help the planner by pruning the search space, and this led PAC-C to solve 7 more instances w.r.t. BASELINE. For `Storage`, the BASELINE fails to solve 10 instances, and we advocate this to the fact that hoists can leave crates in areas that

Domain	BASELINE	PAC-C	LTL-E	TCORE
`Trucks` (30)	15	**22**	11	18
`Storage` (30)	20	**30**	13	28
`Rover` (40)	**40**	**40**	20	**40**
`TPP` (30)	**30**	**30**	17	–
`Openstack` (30)	**30**	**30**	5	–
Total	135	**152**	66	86

Table 6.1: Coverage of the BASELINE, PAC-C, TCORE and LTL-E. In parenthesis, the number of benchmark instances for a given domain.

will obstruct future movements inside the depot, and this is not captured by LAMA's heuristic. This cannot happen for the instances with action constraints: a hoist can lift a crate at most once, and crates must be positioned starting from areas that are far away from the door. With these constraints, PAC-C manages to solve all instances of `Storage`. These results confirm that action constraints can improve the performance of a state-of-the-art classical planner. Also, using TCORE coverage is increased, but not as much as with PAC-C: 3 and 8 more instances are solved in `Trucks` and `Storage`, respectively. By encoding the same knowledge as state constraints in LTL$_f$ and using LTL-E, we did not obtain any improvement. Rather, the performance of LAMA was even worsened (the coverage was reduced for all considered domains). PAC-C turned out to be (much) more effective, coverage-wise, than LTL-E and TCORE.

Solution quality. Table 6.2 and Figures 6.1a, 6.1b and 6.1c give an overall picture of the quality of the solutions (in terms of plan length) obtained by the compilation-based systems with respect to the baseline across all domains. From Figure 6.1a it is possible to see that PAC-C performs really well in `Rover`, `TPP`, and `Openstack`. In `TPP`, the BASELINE moves trucks in a very suboptimal way to buy all products, while PAC-C substantially reduces the number of drive actions. Plan length is improved for 25 instances, and on average, plans have 40 actions less than the BASELINE. Also in `Rover` and `Openstack` PAC-C performs well, improving quality in most cases. These results confirm that control knowledge expressed as action constraints can effectively

Domain	Improved instances			Avg improvement		
	PAC-C	LTL-E	TCORE	PAC-C	LTL-E	TCORE
`Trucks`	3	1	**4**	-1.00	-1.27	**-0.80**
`Storage`	**3**	1	2	-6.30	**-4.46**	-5.70
`Rover`	22	5	**24**	**7.33**	0.55	5.28
`TPP`	**25**	12	–	**40.13**	15.82	–
`Openstack`	**24**	5	–	**3.03**	1.80	–

Table 6.2: Pairwise comparison of PAC-C/LTL-E/TCORE vs the BASELINE in terms of plan length over instances solved by both the two compared systems. The first 3 columns show the instances number with improved solutions, the others the average improvement.

lead to better-quality solutions. With LTL-E the improvement is limited. TCORE shows good performance in `Rover`, while for `Openstack` and `TPP` it was not possible to reformulate our action constraints in PDDL3 (in the tables indicated with "–"). For `Trucks` and `Storage`, solution quality is worsened by all the compared systems. In our benchmarks, coverage and quality are not improved at the same time; this is not surprising, since they were designed with constraints aimed at improving coverage or solution quality, but not both.

CPU time. Figure 6.1d shows how coverage and CPU time are related. As expected, all compilation-based approaches tend to increase their coverage over time more slowly w.r.t. the BASELINE, since performing the compilation takes some time that we do not have in BASELINE. While the coverage of the BASELINE tails off after around 26 secs (coverage gets to 132 solved instances), PAC-C continues to increase coverage, outperforming BASELINE after about 90 CPU seconds.

6.3 Discussion and Conclusions

Imposing constraints on the action trajectory of a plan is useful to guide the planner search, as well as to generate plans that have some desired properties. In this chapter, we presented a compilation-based approach to plan with PAC action trajectory

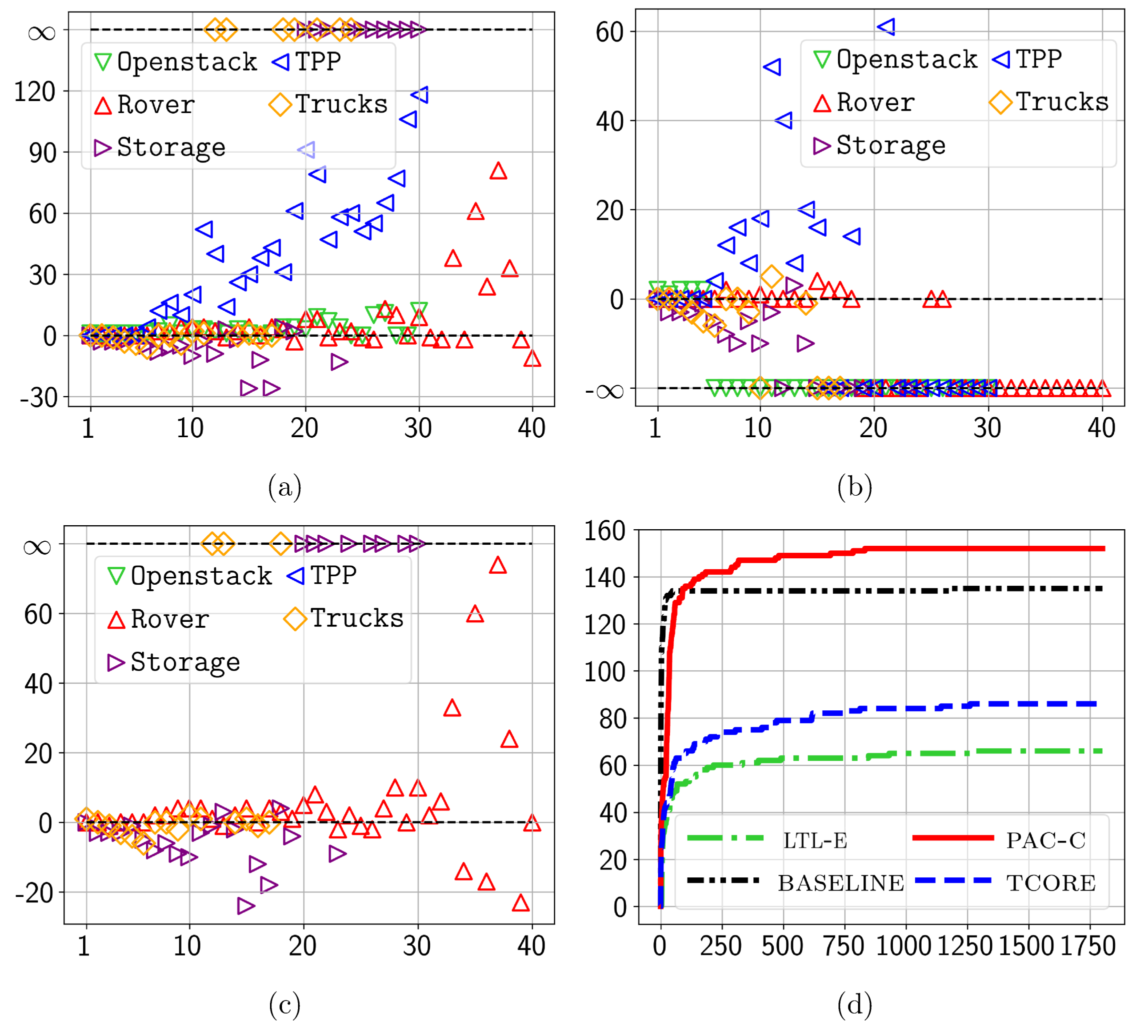


Figure 6.1: Quality improvement measured as the difference of plan length (y-axis) between PAC-C and the BASELINE (a), LTL-E and the BASELINE (b), TCORE and the BASELINE (c), over instances solved by at least one of the two compared systems (x-axis). (d) Coverage (y-axis) versus planning time (x-axis).

constraints. Our compilation scheme can be used alongside any classical planner supporting conditional effects and is relatively simple, computationally efficient, and compact. As experimentally shown, action constraints and our compilation provide an effective tool to express and use useful control knowledge. A comparison with other approaches shows that, for the considered benchmarks, a classical planner can exploit knowledge expressed as (compiled) action constraints more effectively than equivalent formulations using state-trajectory constraints.

Chapter 7

Handling Pure-Past Linear Temporal Logic Trajectory Constraints

In this chapter, we study the problem of planning with temporally extended goals expressed in PPLTL. As discussed in Section 3.1, PPLTL has the same expressive power as LTL_f. However, as we show in the following sections, PPLTL can be handled more efficiently than LTL_f. Specifically, we demonstrate that planning for PPLTL goals can be translated into classical planning with minimal additional complexity, introducing a number of new fluents that are at most linear in the size of the PPLTL goal, and preserving plan size exactly [Nebel, 2000].

Leveraging this encoding, we developed a system named Plan4Past, which can be used alongside any classical planner, such as LAMA [Richter and Westphal, 2010]. Our empirical analysis studies the effectiveness of Plan4Past, showing that a state-of-the-art classical planner performs better when utilizing our compilation method compared to existing LTL_f compilation over the considered benchmark instances.

7.1 Handling PPLTL Goals

In this section, we develop the basis for our technique. First, we observe that any sequence of actions produces a state trajectory on which PPLTL formulas can be evaluated. Therefore, while the planning process goes on, prefixes are produced, state trajectories are generated, and over them, PPLTL goals can be evaluated. The difficulty is that evaluating PPLTL formulas requires a state trajectory, and searching through state trajectories is quite demanding. Instead, our technique does not consider state trajectories at all. In particular, it exploits the following intuitions: (*i*) to evaluate the PPLTL goal formula, we only need the truth value of its subformulas; (*ii*) 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; (*iii*) 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. Next, we detail these intuitions.

As discussed in Section 4.3, temporal operators in LTL_f can be decomposed into present and future components using the fixpoint characterization of a LTL_f formula. 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 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 hold:

Proposition 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, $\mathsf{pnf}(\varphi) \equiv \varphi$.*

Proof. Immediate from the definition of $\mathsf{pnf}(\cdot)$ and the semantics of PPLTL formulas. □

Example 27. *Consider the formula $\varphi = a \,\mathsf{S}\, \mathsf{Y}(b)$. The* PNF *of φ is:*

$$\mathsf{pnf}(a \,\mathsf{S}\, \mathsf{Y}(b)) = \mathsf{pnf}(\mathsf{Y}(b)) \vee (\mathsf{pnf}(a) \wedge \mathsf{Y}(a \,\mathsf{S}\, \mathsf{Y}(b))) = \mathsf{Y}(b) \vee (a \wedge \mathsf{Y}(a \,\mathsf{S}\, \mathsf{Y}(b))).$$

Given a state trajectory $\tau = \langle s_0, s_1, \dots s_n \rangle$, we have that $\tau, n \models \mathsf{pnf}(a \,\mathsf{S}\, b)$ iff either $\tau, n-1 \models b$ (which is equivalent to $s_{n-1} \models b$) or $s_n \models a$ and $\tau, n-1 \models a \,\mathsf{S}\, b$. As we can see, to determine the truth of $a \,\mathsf{S}\, \mathsf{Y}(b)$, we only need the last state s_n and the prefix $\langle s_0, s_1, \dots s_{n-1} \rangle$ of τ.

Now, we show that to evaluate a PPLTL formula φ, we only need to keep track of the truth values of some key subformulas of φ. To do so, we introduce Σ_φ as the set of *propositions* of the form "$\mathsf{Y}\phi$" containing:

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

Example 28. *Consider $\varphi = a \,\mathsf{S}\, \mathsf{Y}(b)$. In this case, $\Sigma_\varphi = \{$"$\mathsf{Y}(a \,\mathsf{S}\, \mathsf{Y}(b))$", "$\mathsf{Y}(b)$"$\}$. This means that, to evaluate φ, we only need to keep track of $\mathsf{Y}(a \,\mathsf{S}\, \mathsf{Y}(b))$ and $\mathsf{Y}(b)$ using the propositional variables of 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 in Σ_φ are true at instant i. By suitably maintaining the value of propositions in Σ_φ in σ_i, we can evaluate a PPLTL formula just 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 18. *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* $\sigma_i \models$ "$\mathsf{Y}\phi'$";
- $\mathsf{val}(\phi_1 \mathbin{\mathsf{S}} \phi_2, \sigma_i, s_i)$ *iff* $\mathsf{val}(\phi_2, \sigma_i, s_i) \lor (\mathsf{val}(\phi_1, \sigma_i, s_i) \land \sigma_i \models$ "$\mathsf{Y}(\phi_1 \mathbin{\mathsf{S}} \phi_2)$");
- $\mathsf{val}(\phi_1 \land \phi_2, \sigma_i, s_i)$ *iff* $\mathsf{val}(\phi_1, \sigma_i, s_i) \land \mathsf{val}(\phi_2, \sigma_i, s_i)$;
- $\mathsf{val}(\neg\phi', \sigma_i, s_i)$ *iff* $\neg\mathsf{val}(\phi', \sigma_i, s_i)$.

Intuitively, the $\mathsf{val}(\phi, \sigma_i, s_i)$ predicate allows us to determine the truth value of any PPLTL formula $\phi \in \mathsf{sub}(\varphi)$ by reading a propositional interpretation s_i from the state trajectory τ and keeping track of the truth value of the propositions in Σ_φ by means of σ_i.

Example 29. *Consider the formula $\varphi = a \mathbin{\mathsf{S}} \mathsf{Y}(b)$. Following Definition 18, we can obtain the following rules:*

- $\mathsf{val}(a \mathbin{\mathsf{S}} \mathsf{Y}(b), \sigma_i, s_i)$ *iff* $\mathsf{val}(\mathsf{Y}(b), \sigma_i, s_i) \lor (\mathsf{val}(a, \sigma_i, s_i) \land \sigma_i \models$ "$\mathsf{Y}(a \mathbin{\mathsf{S}} b)$");
- $\mathsf{val}(\mathsf{Y}(b), \sigma_i, s_i)$ *iff* $\sigma_i \models$ "$\mathsf{Y}(b)$";
- $\mathsf{val}(a, \sigma_i, s_i)$ *iff* $s_i \models a$

This means $\mathsf{val}(a \mathbin{\mathsf{S}} \mathsf{Y}(b), \sigma_i, s_i)$*, which is meant to capture the truth of* $a \mathbin{\mathsf{S}} \mathsf{Y}(b)$*, holds iff either* $\sigma_i \models$ "$\mathsf{Y}(b)$"*, or* $s_i \models a$ *and* $\sigma_i \models$ "$\mathsf{Y}(a \mathbin{\mathsf{S}} b)$".

Observe that the rules in Definition 18 basically follow the PNF transformation rules where subformulas within the Y-scope are interpreted as propositions.

Now, given a state trajectory $\tau = \langle s_0 \cdots s_n \rangle$ over $\mathcal{P}$, we compute a corresponding state trajectory $\tau^{[\varphi]} = \sigma_0 \cdots \sigma_n$ over Σ_φ, where:

- $\sigma_0(\text{“}\mathsf{Y}\phi\text{”}) \doteq \bot$ for each “$\mathsf{Y}\phi$” $\in \Sigma_\varphi$;
- $\sigma_i(\text{“}\mathsf{Y}\phi\text{”}) \doteq \mathsf{val}(\phi, \sigma_{i-1}, s_{i-1})$, for all i with $0 < i \leq n$.

By constructing the trajectory $\tau^{[\varphi]}$ in this way, we can determine the truth of every $\phi \in \mathsf{sub}(\varphi)$ at each step i by using $\mathsf{val}(\phi, \sigma_i, s_i)$. This simple operation is shown in Example 30.

Example 30. *Consider the formula $\varphi = a \,\mathsf{S}\, \mathsf{Y}(b)$. Let $\tau = \langle \{b\}, \{\} \rangle$. In this case, using the rules shown in Example 29, we have:*

- *For $i = 0$*
  - $\sigma_0(\text{“}\mathsf{Y}b\text{”}) \doteq \bot$
  - $\sigma_0(\text{“}\mathsf{Y}(a \,\mathsf{S}\, \mathsf{Y}(b)))\text{”}) \doteq \bot$
- *For $i = 1$*
  - $\sigma_1(\text{“}\mathsf{Y}b\text{”}) \doteq \mathsf{val}(b, \sigma_0, s_0) = \top$
  - $\sigma_1(\text{“}\mathsf{Y}(a \,\mathsf{S}\, \mathsf{Y}(b)))\text{”}) \doteq \mathsf{val}(a \,\mathsf{S}\, \mathsf{Y}(b)), \sigma_0, s_0) = \bot$

Using a state-based representation, $\tau^{[\varphi]}$ can be seen as $\tau^{[\varphi]} = \langle \{\}, \{\text{“}\mathsf{Y}b\text{”}\} \rangle$. Given $\tau^{[\varphi]}$, we can determine that $\mathsf{val}(a \,\mathsf{S}\, \mathsf{Y}(b)), \sigma_1, s_1)$ holds. Therefore, $\tau, 1 \models a \,\mathsf{S}\, \mathsf{Y}(b)$.

We now formally prove this result. First, this property holds for state trajectories of length 1.

Lemma 1. *Let φ be PPLTL formula over $\mathcal{P}$, $\phi \in \mathsf{sub}(\varphi)$ a subformula of φ, and $\tau = s_0$ a state trajectory 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 σ_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 hypotesis the thesis holds.

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

□

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

Theorem 9. *Let φ be a* PPLTL *formula over $\mathcal{P}$, $\phi \in \mathsf{sub}(\varphi)$ a subformula of φ, τ a state trajectory over $\mathcal{P}$, and $\tau^{[\varphi]}$ the corresponding state trajectory over Σ_φ. Then,*

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

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

Base case. $\tau = s_0$. By Lemma 1, the thesis holds.

Inductive step. Let $\tau = \tau_{n-1} \cdot s_n$. By inductive hypothesis, the thesis holds for the state trajectory 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 holds also 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 now 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 $\tau_{n-1}{\cdot}s_n \models p$ iff $s_n \models p$. For the $\mathsf{val}(\cdot)$ predicate, we have $s_n \models p$ iff $\mathsf{val}(p, \sigma_n, s_n)$. Therefore, the thesis holds.

- $\phi = \mathsf{Y}\phi'$. We have $\tau_{n-1}{\cdot}s_n \models \mathsf{Y}\phi'$ 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 $\mathsf{val}(\phi_2, \sigma_n, s_n)$. Furthermore, $\tau_{n-1}{\cdot}s_n \models \mathsf{Y}(\phi_1 \,\mathsf{S}\, \phi_2)$ 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 state trajectory.

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

$\square$

Theorem 9 gives us the basis of our technique as it guarantees that by keeping a suitably updated state trajectory 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 state trajectory.

7.2 Compiling PPLTL Goals Away

We devise a new encoding for planning for PPLTL goals by exploiting Theorem 9. The key idea behind our approach is that, given a PPLTL formula and a planning problem, we keep track of values of formulas in σ as actions are applied.

Similarly to other encoding-based approaches dealing with temporally extended goals, for example, [Baier and McIlraith, 2006a,b, Torres and Baier, 2015], we address planning for temporally extended goals in three steps. First, compile the original

Components	Encoding
Fluents F'	$F' := F \cup \Sigma_\varphi$
Derived Predicates D'	$D' := D \cup \{\mathsf{val}_\phi \mid \phi \in \mathsf{sub}(\varphi)\}$
Axioms X'	$X' := X \cup \{x_\phi \mid \phi \in \mathsf{sub}(\varphi)\}$ 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'	$Eff'(a) := Eff(a) \cup \{\mathsf{val}_\phi \rhd \{\text{“}\mathsf{Y}\phi\text{”}\}, \neg\mathsf{val}_\phi \rhd \{\neg\text{“}\mathsf{Y}\phi\text{”}\} \mid \text{“}\mathsf{Y}\phi\text{”} \in \Sigma_\varphi\}$
Initial State I'	$I' := \sigma_0 \cup I$
Goal G'	$G' := \mathsf{val}_\varphi$

Table 7.1: Components of the compiled classical planning problem $\Pi' = \langle F', D', X', A, I', G', Pre, Eff' \rangle$ for a given planning problem $\Pi = \langle F, D, X, A, I, \varphi, Pre, Eff \rangle$.

planning problem Π with the temporally extended goal into a planning problem Π' with a final-state goal. Second, invoke any off-the-shelf sound and complete planner to compute a plan solving the compiled problem Π'. Third, rework the computed plan to get the solution to the original problem Π. In our approach, we exploit Theorem 9 to do the encoding in the first step, and since no extra control actions are introduced, step three trivializes.

Given a planning problem $\Pi = \langle F, D, X, A, I, \varphi, Pre, Eff \rangle$, the compiled classical planning problem is $\Pi' = \langle F', D', X', A, I, G', Pre, Eff' \rangle$. Table 7.1 shows the formal construction of Π'.

In this encoding, we employ axioms to determine which subformula ϕ of the goal

φ is true in a planning state s. In particular, the new classical planning problem includes an axiom $\mathsf{val}_\phi \leftarrow \psi$ for every subformula $\phi \in \mathsf{sub}(\varphi)$. Given a sequence of states $(\sigma_0, I), \ldots, (\sigma_n, s_n)$, these axioms mimic the rules in Definition 18 and are intended to be such that the current state $(\sigma_i, s_i) \models \mathsf{val}_\phi$ iff $\mathsf{val}(\phi, \sigma_i, s_i)$ (in this section, without loss of generality, we assume that the interpretation σ_i is represented as a set of atoms, and we use (σ_i, s_i) to denote the state $\sigma_i \cup s_i$). Axioms not only elegantly model the mathematics behind Theorem 9 (that is, $\mathsf{val}(\phi, \sigma_i, s_i)$), but also simplify the action schema and goal descriptions without adding control predicates among fluents. From Table 7.1, it is also easy to see that no new action is added to the encoded problem and that the precondition function *Pre* remains unchanged. In fact, every problem'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. These additional effects are exactly the same for every action in A. Moreover, since σ_i maintains values of "$\mathsf{Y}\phi$" in Σ_φ, they are *independent* from 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 which action has been executed or which effect such action has had on the original fluents. Observe that the auxiliary part $\mathit{eff}_{\mathsf{val}}$ in $\mathit{Eff}'(a)$ updates the subformula values in Σ_φ without affecting any fluent $f \in F$ of the original problem. This is crucial for the encoding's correctness.

Example 31 (Compilation Example). *We describe in detail the compilation of the formula $\varphi = t \wedge (\neg a \,\mathsf{S}\, c)$ that enforces scenarios such as the one in which the agent must achieve the goal t while a was always false since c became true. The set of subformulas of φ is $\mathsf{sub}(\varphi) = \{a, c, t, \neg a, (\neg a \,\mathsf{S}\, c), t \wedge (\neg a \,\mathsf{S}\, c)\}$. Given a planning problem $\Pi = \langle F, D, X, A, I, \varphi, \mathit{Pre}, \mathit{Eff} \rangle$, the compiled classical planning problem $\Pi' = \langle F', D', X', A, I, G', \mathit{Pre}, \mathit{Eff}' \rangle$ is defined as follows;*

- *(F'). The new set of fluents is the union of the original set of fluents F with $\Sigma_\varphi = \{\text{“}\mathsf{Y}(\neg a \,\mathsf{S}\, c)\text{”}\}$. Therefore, $F' = F \cup \{\text{“}\mathsf{Y}(\neg a \,\mathsf{S}\, c)\text{”}\}$.*
- *(D'). The new set of derived predicates is obtained by adding one fluent of the form val_ϕ for each $\phi \in \mathsf{sub}(\varphi)$. Therefore, $D' = D \cup \{\mathsf{val}_a$, val_c, val_t, $\mathsf{val}_{\neg a}$, $\mathsf{val}_{\neg a \,\mathsf{S}\, c}$, $\mathsf{val}_{t \wedge (\neg a \,\mathsf{S}\, c)}\}$.*

- *(X'). Following the rules illustrated in Table 7.1, we have that X' is X plus the following new axioms:*
  - $\mathsf{val}_a \leftarrow a$*;*
  - $\mathsf{val}_c \leftarrow c$*;*
  - $\mathsf{val}_t \leftarrow t$*;*
  - $\mathsf{val}_{\neg a} \leftarrow \neg \mathsf{val}_a$*.*
  - $\mathsf{val}_{\neg a \,\mathsf{S}\, c} \leftarrow (\mathsf{val}_c \vee (\mathsf{val}_{\neg a} \wedge \text{“}\mathsf{Y}(\neg a \,\mathsf{S}\, c)\text{”}))$*;*
  - $\mathsf{val}_{t \wedge (\neg a \,\mathsf{S}\, c)} \leftarrow (\mathsf{val}_t \wedge \mathsf{val}_{\neg a \,\mathsf{S}\, c})$*.*
- *(Eff'). We extend the effects of each action act $\in A$ as follows:*

$$\mathit{Eff}(act) = \mathit{Eff}(act) \cup \{\mathsf{val}_{\neg a \,\mathsf{S}\, c} \rhd \text{“}\mathsf{Y}(\neg a \,\mathsf{S}\, c)\text{”}, \neg \mathsf{val}_{\neg a \,\mathsf{S}\, c} \rhd \neg \text{“}\mathsf{Y}(\neg a \,\mathsf{S}\, c)\text{”}\}.$$

- *(I'). The new initial state is $I' = I \cup \sigma_0$ where $\sigma_0 = \emptyset$, meaning that “$\mathsf{Y}(\neg a \,\mathsf{S}\, c)$” is false in I'.*
- *(G'). The new final-state goal is $G' = \{\mathsf{val}_{t \wedge (\neg a \,\mathsf{S}\, c)}\}$.*

Consider the plan $\pi = \langle a_0, a_1 \rangle$ for Π that induces the state trajectory

$$\tau = \langle \{a\}, \{c\}, \{t\} \rangle.$$

It is easy to see that $\tau \models t \wedge (\neg a \,\mathsf{S}\, c)$; in fact, a is false since c became true in the second state and t is true at the end of the trajectory. Now we comment on the state trajectory τ' induced by π for the compiled problem Π'. By analyzing the structure of the compiled problem Π', and in particular the new effect function Eff', we can see that π induces a state trajectory τ' defined as follows:

$$\tau' = \langle \{a\}, \{c\}, \{t, \text{“}\mathsf{Y}(\neg a \,\mathsf{S}\, c)\text{”}\} \rangle.$$

By analyzing the state trajectory, we can see that the derived predicate $\mathsf{val}_{\neg a \,\mathsf{S}\, c}$ becomes true in the second state because c holds. Therefore, the conditional effect $\mathsf{val}_{\neg a \,\mathsf{S}\, c} \rhd$

"$\mathsf{Y}(\neg a \,\mathsf{S}\, c)$" of action a_1 makes "$\mathsf{Y}(\neg a \,\mathsf{S}\, c)$" true in the last state. In such a state, $\mathsf{val}_{\neg a \mathsf{S} c}$ holds because "$\mathsf{Y}(\neg a \,\mathsf{S}\, c)$" $\wedge \neg a$ holds. Furthermore, since t is true in the last state, we have that $\mathsf{val}_{t \wedge (\neg a \mathsf{S} c)}$ (the goal of Π') holds in the last state.

It is easy to see that our encoding is polynomially related to the original problem and the satisfaction of the PPLTL goal φ on the trace τ' corresponds to the satisfaction of val_{φ} in the last instant of τ'.

Theorem 10. *The size of the encoded planning problem Π' is polynomial in the size of the original problem Π. In particular, the additional fluents introduced are linear in the size of the temporally extended* PPLTL *goal φ of Π.*

Proof. Immediate by analyzing the construction. □

Next, we turn to correctness. Let $\Pi = \langle F, D, X, A, I, \varphi, Pre, Eff \rangle$ be a planning problem, and let Π' be the corresponding compiled planning problem as previously defined. Any trace $\tau' = s'_0, \ldots, s'_n$ on Π' can be seen as $\tau' = zip(\tau^{[\varphi]}, \tau)$, with $\tau^{[\varphi]} = \sigma_0, \ldots, \sigma_n \in (2^{\Sigma_\varphi})^+$ and $\tau = I, \ldots, s_n \in (2^F)^+$, 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 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 problem Π', there exists a single trace $\tau' \mid_F = \tau = I, \ldots, s_n$ on the original planning problem Π. Conversely, given a trace $\tau = I, \ldots, s_n$ on Π, there exists a unique corresponding trace $\tau^{[\varphi]}$, and hence a single $\tau' = zip(\tau^{[\varphi]}, \tau)$ on the encoded problem Π'. Finally, we observe that every executable action sequence $a_0, \ldots, a_{n-1}$ in the planning problem Π with PPLTL goal φ is also executable in the encoded planning problem Π' (and vice versa) since the encoding does not have auxiliary actions, actions preconditions do not change, and additional conditional effects only affect the new fluents in Σ_φ.

Theorem 11 (Soundness and Completeness)**.** *Let Π be a planning problem with a* PPLTL *goal φ, and Π' be the corresponding encoded planning problem with a reachability goal. Then, every action sequence $\pi = \langle a_0, \ldots, a_{n-1} \rangle$ is a solution for Π iff π is a solution for Π'.*

Proof. Every executable action sequence $a_0, \ldots, a_{n-1}$ in the planning problem Π with PPLTL goal φ is also executable in the encoded planning problem Π' (and vice versa) since, by definition, the encoding does not have auxiliary actions, actions preconditions do not change, and additional conditional effects only affect the new fluents in Σ_φ.

The action sequence π is a plan if its induced state trace τ is such that $\tau \models \varphi$. By Theorem 9, we have $\tau \models \varphi$ iff $\mathsf{val}(\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau))$. However, by construction of the encoding for Π', we have that $\mathsf{val}(\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau))$ holds iff val_φ holds in the last state of the induced state trace for Π', that is, in $\tau' = zip(\tau^{[\varphi]}, \tau)$. In other words, $\mathsf{val}(\varphi, \mathsf{last}(\tau^{[\varphi]}), \mathsf{last}(\tau))$ iff $\mathsf{last}(\tau') \models \mathsf{val}_\varphi$. Hence, the thesis holds. $\square$

A direct consequence of Theorem 11 is that every sound and complete planner returns a plan π for the encoded planning problem Π' if a plan π for the original planning problem Π with PPLTL goal exists. If Π' has no solution, then so does Π.

7.3 Experiments

We implemented the approach of the previous section in a tool called PLAN4PAST[1] (P4P). P4P 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 where all axioms are compiled into conditional effects, 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.

Our analysis aims to shed some light on the effectiveness of temporally extended goals formulated in PPLTL and handled by P4P, and *semantically* equivalent temporally extended goals formulated in LTL$_f$ and handled by either LTL-E [Baier and McIlraith, 2006a] or LTL-P [Torres and Baier, 2015].

[1]Source code, benchmarks, and supplementary material are publicly available at `https://github.com/whitemech/Plan4Past`.

To our knowledge, LTL-E and LTL-P are the best approaches for planning for LTL_f goals[2]. In particular, as discussed in Section 4.2, LTL-E builds a NFA for the LTL_f formula and computes 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. On the other hand, LTL-P uses the fixpoint characterization of LTL_f formulas to devise an encoding that is polynomial but that significantly increases the length of solution plans. Dealing with additional spurious actions has already been studied in Nebel [2000] theoretically and practically from a heuristic perspective, for example, in Haslum [2013]. Therefore, the theoretical advantage of P4P is clear: the encoding is polynomial and preserving plan size exactly Nebel [2000].

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_x [Thiébaux et al., 2005]. LAMA is a satisficing planner based on a sophisticated search mechanism 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_x is yet another satisficing planner based on heuristic search and enforced hill climbing and is the one originally used with LTL-P and LTL-E. Both systems handle the compiled problems, but in the rest of this section, we will focus on LAMA as it was the system with the highest overall coverage for all compilations. All experiments were run on an Intel Xeon Gold 6140M 2.3 GHz, with runtime and memory limits of 1800s and 8GB, respectively.

[2]We could also include TCORE by limiting benchmark formulas to properties that can be captured by PDDL3. However, our empirical evaluation aims to test the scalability of all compilations on formulas with many nested operators that cannot be expressed in PDDL3.

7.3.1 Benchmark Domains

Our benchmark suite includes `Blocks`, `Elevator`, `Rover`, and `Openstack` domains. These domains were introduced in previous International Planning Competitions, and all except `Elevator` have also been used by Torres and Baier [2015]. For `Blocks`, `Rover`, and `Openstack` 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 BF23). For `Elevator`, we only have BF23. TB15 were originally specified in LTL$_f$, and for P4P 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 "$h : \alpha \,\mathsf{U}\, \beta$" where α or β have n nested U operators, for which we did not find an easy translation into PPLTL. For each LTL$_f$ formula that we translated into PPLTL, we formally and automatically proved their *semantic* equivalence by verifying that the two formulations yield the same minimal DFA. BF23 were designed in PPLTL directly, and analogously to what was done for TB15, we formulated an equivalent formulation in LTL$_f$. TB15 are based on predefined families of formulas that are independent of the domain. Instead, BF23 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 `Blocks`, 7 for `Rover`, and 10 for `Openstack`. Their definition is provided by Torres and Baier [2015]. BF23 are instead described below.

`Blocks.` BF23 were formulated to study the performance of all compilations with complex temporally extended goals. Here, BF23 specify two intertwined goals, both requiring the existence in the state trajectory of the plan of a particular sequence of states. Consider a problem with n blocks. The first goal in PPLTL is

$$\mathsf{O}(On(b_1, b_2) \wedge \mathsf{Y}(\mathsf{O}(On(b_2, b_3) \wedge \mathsf{Y}(\mathsf{O}(... \wedge \mathsf{Y}(\mathsf{O}(On(b_{n-1}, b_n)))))))).$$

Its translation in LTL$_f$ is

$$\mathsf{F}(On(b_{n-1}, b_n) \wedge \mathsf{X}(\mathsf{F}(On(b_{n-2}, b_{n-1}) \wedge \mathsf{X}(\mathsf{F}(\ldots \wedge \mathsf{X}(\mathsf{F}(On(b_1, b_2)))))))).$$

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

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

where $\phi = \mathsf{O}(On(b_4, b_3) \wedge \mathsf{Y}(\mathsf{O}(On(b_3, b_2) \wedge \mathsf{Y}(\mathsf{O}(On(b_2, b_1))))))$ encoding the construction of a bigger stack. The same constraint formulated in LTL$_f$ is

$$\mathsf{F}(On(b_2, b_1) \wedge \mathsf{X}(\mathsf{F}(On(b_3, b_2) \wedge \mathsf{X}(\mathsf{F}(On(b_4, b_3) \wedge \bigwedge_{j \in \{6,8,\ldots,n\}} \mathsf{X}(\mathsf{F}(On(b_j, b_{j-1})))))))).$$

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.

`Openstack`. Here, the BF23 instances require a valid plan to ship 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, which 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$_f$.

`Rover`. The goal of this domain is to gather and communicate data about soil, rock, and images to the Earth using a set of rovers. BF23 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

$$\begin{aligned}&\mathsf{O}(\mathit{Data}(\mathit{soil}) \land \mathsf{WY}(\mathsf{H}(\neg \mathit{Data}(\mathit{rock}))))\land\\ &\mathsf{O}(\mathit{Data}(\mathit{rock}) \land \mathsf{WY}(\mathsf{H}(\neg \mathit{Data}(\mathit{image}))))\land\\ &\mathsf{O}(\mathit{Data}(\mathit{image})).\end{aligned}$$

The semantically equivalent formula in LTL$_f$ is

$$(\neg \mathit{Data}(\mathit{rock}))\,\mathsf{U}\,(\mathit{Data}(\mathit{soil})) \land (\neg \mathit{Data}(\mathit{image}))\,\mathsf{U}\,(\mathit{Data}(\mathit{rock})) \land \mathsf{F}(\mathit{Data}(\mathit{image})).$$

Also, when the rover reaches the lander, that rover must re-calibrate all cameras. For instance, if the lander is at waypoint w_l and the rover r has 2 cameras, c_1 and c_2, we have, in PPLTL, the formula

$$((\neg \mathit{At}(r, w_l)\,\mathsf{S}\,\mathit{Calibrated}(c_1)) \land (\neg \mathit{At}(r, w_l)\,\mathsf{S}\,\mathit{Calibrated}(c_2))) \lor \mathsf{H}(\neg \mathit{At}(r, w_l))$$

and, in LTL$_f$, the formula

$$\mathsf{G}(\mathit{At}(r, w_l) \rightarrow (\mathsf{F}(\mathit{Calibrated}(c_1)) \land \mathsf{F}(\mathit{Calibrated}(c_2)))).$$

`Elevator.` This domain models the problem of scheduling passengers in the use of an elevator. In BF23, we split the passengers into half VIP and half regular passengers, where VIP passengers must be served before every regular one. For instance, we enforce this in PPLTL with

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

and in LTL$_f$ with

$$\mathsf{F}(\mathit{Served}(p_2)\land \mathit{Served}(p_3))\land(\neg \mathit{Served}(p_2)\land\neg \mathit{Served}(p_3))\,\mathsf{U}\,(\mathit{Served}(p_0)\land \mathit{Served}(p_1)).$$

Also, we model that no passenger may share the elevator with another passenger, and do so in PPLTL with

$$\mathsf{H}(\mathit{Boarded}(p_0) \rightarrow (\neg \mathit{Boarded}(p_1) \wedge \neg \mathit{Boarded}(p_2)))$$

and in LTL_f with

$$\mathsf{G}(\mathit{Boarded}(p_0) \rightarrow (\neg \mathit{Boarded}(p_1) \wedge \neg \mathit{Boarded}(p_2))).$$

7.3.2 Experimental Results

Domain		I	Coverage			Avg RT		
			P4P	LTL-P	LTL-E	P4P	LTL-P	LTL-E
`Rover`	TB15	7	**7**	**7**	6	**1.43**	21.11	1.98
	BF23	40	**33**	6	22	35.36	–	**24.24**
`Blocks`	TB15	15	**15**	**15**	8	**1.41**	20.43	13.13
	BF23	21	**21**	1	1	–	–	–
`Openstack`	TB15	10	**10**	**10**	6	**6.11**	31.66	8.75
	BF23	30	7	5	**8**	**11.88**	68.86	19.50
`Elevator`	BF23	29	**29**	4	**29**	231.83	–	**228.09**
Total			**122**	48	80			

Table 7.2: Coverage and average Run-Time (Avg RT) achieved by P4P, LTL-P and LTL-E. 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. Bolds are for best performers.

Tables 7.2 and 7.3 report on the overall performance of all compilations across all domains. Coverage-wise, P4P performs equally to or better than both LTL-P and LTL-E over most instances. For the TB15 instances, P4P achieves the same coverage as LTL-P (the best LTL_f-based compilation) but is much faster in terms of average runtime: P4P is roughly one order of magnitude faster than LTL-P; this seems to be

Domain		Avg EN			Avg PL		
		P4P	LTL-P	LTL-E	P4P	LTL-P	LTL-E
`Rover`	TB15	**12.67**	616.83	**12.67**	**5.33**	5.67 (74.50)	**5.33**
	BF23	**7665.36**	–	7826.32	43.68	–	**43.50**
`Blocks`	TB15	22.12	821.62	**21.75**	7.50	7.88 (132.88)	**7.25**
	BF23	–	–	–	–	–	–
`Openstack`	TB15	**52.67**	3863.33	52.83	22.00	**21.67** (349.00)	22.00
	BF23	99.80	1207764.80	**72.40**	**24.00**	**24.00** (841.00)	**24.00**
`Elevator`	BF23	**1712076.48**	–	1712090.79	**75.48**	–	**75.48**

Table 7.3: Average Expanded Nodes (Avg EN) and Plan Length (Avg PL) achieved by P4P, LTL-P and LTL-E. For LTL-P, we report in parenthesis the average PL 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. "–" indicates when a system is excluded by the comparison. Bolds are for best performers.

justified by a great reduction in the number of expanded nodes (up to two orders of magnitude in `Openstack`). This is somehow expected. Indeed, for each planning action taken, LTL-P interleaves quite a complex automaton synchronization phase, from the initial state all the way to the goal. On average, in TB15 instances, 94.3% of actions in plans obtained with LTL-P come from the automaton's synchronization phases.

The situation is different if we look at the BF23 instances. Here, the best performing LTL$_f$ compilation is LTL-E, which is superior to LTL-P over all instances. BF23 are of increasing dimensions and have been constructed to be computationally more challenging. For example, in `Blocks`, the hardest instance in TB15 requires a 22-action plan, while BF23 instances require up to 652 actions. In the case of LTL-P, the planner has to cope with too many synchronization phases and struggles to find solutions. If we compare P4P and LTL-E, we observe that P4P is again the system performing generally better. The only exception is for one instance of `Openstack`. LTL-E solves this instance in roughly 739s while P4P times out. By looking at the average number of expanded nodes, LAMA's search turned out to be slightly less informed with P4P in this domain, which leads to timing out in that particular instance. For `Blocks`, P4P is instead much more effective than LTL-E, which manages

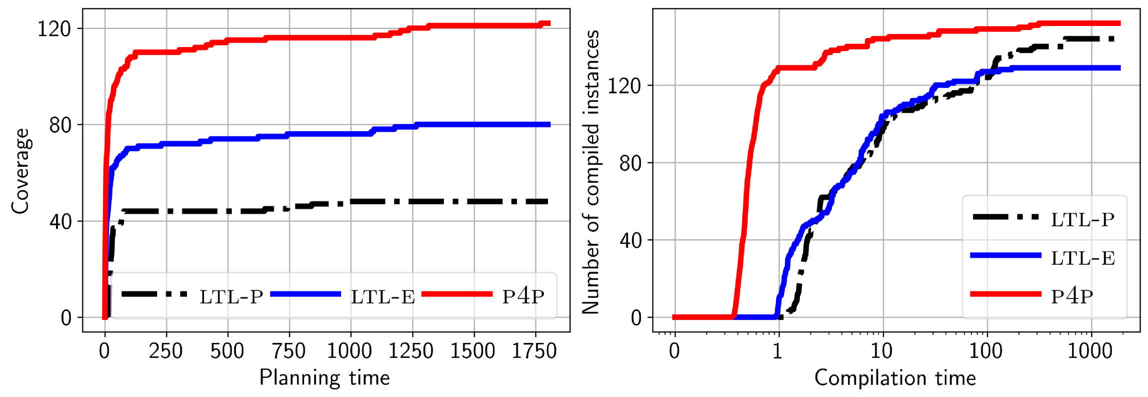


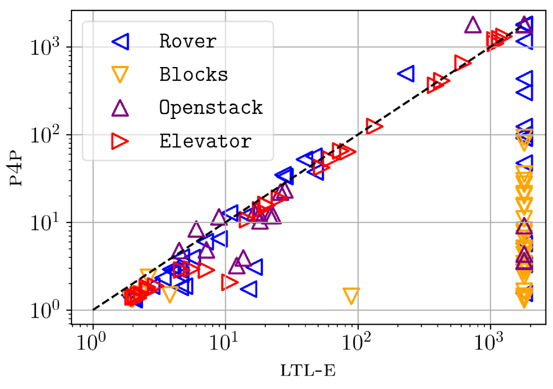
Figure 7.1: Number of solved instances (left) and compiled instances (right) versus computation time.

to compile only 7 instances. The compilation time seems to be an issue for both LTL$_f$ compilations.

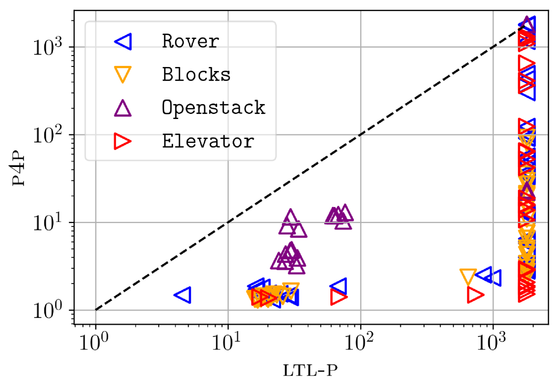
Indeed, if we look at Figure 7.1 (right), P4P compiles 94.7% of the instances within 10s, while both LTL-P and LTL-E converge much more slowly. Figure 7.1 (left) displays the number of benchmark instances solved with a given timeout. All systems achieve their maximum coverage quite quickly, with P4P leaving the others well behind right after the start.

Figure 7.2 reports on a pairwise comparison P4P vs LTL-E and P4P vs LTL-P over the number of expanded nodes and runtime, instance by instance. P4P is generally faster than LTL-E, 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, LTL-E spends much more time than P4P in compilation and slightly more time in evaluating a node of the search. The comparison P4P vs LTL-P confirms our expectation on the number of expanded nodes. P4P expands nodes more slowly than LTL-P, and therefore the runtime advantage of P4P is related to the fact that P4P leads LAMA to do much less search than LTL-P.

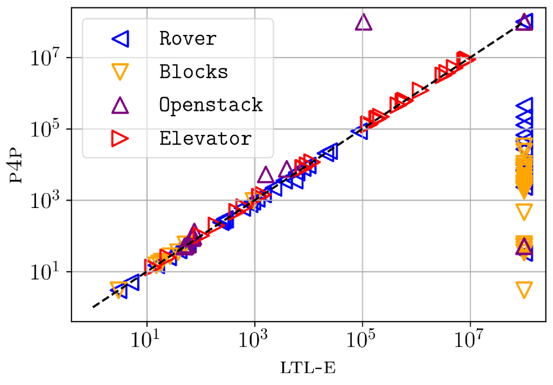
Regarding plan quality, we observed that all compilations yield *solution plans* to the original problem of similar length, making their overall performance the same in these terms.



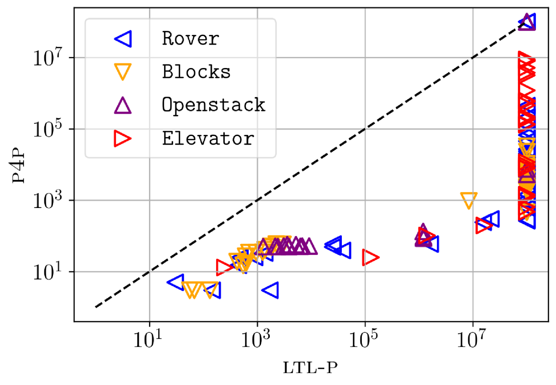
(a) LTL-E vs P4P (RT)



(b) LTL-P vs P4P (RT)



(c) LTL-E vs P4P (EN)



(d) LTL-P vs P4P (EN)

Figure 7.2: Pairwise comparison between P4P and LTL-E (left plots) and between P4P and LTL-P (right plots) in terms of Run-Time (above) and Expanded Nodes (below).

7.4 Discussion and Conclusions

In this chapter, we studied an efficient encoding for 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 encoding that preserves plan size exactly [Nebel, 2000]. Moreover, handling PPLTL goals is remarkably simple and elegant, given the direct mapping between the theoretical formulation and the encoding compilation without sacrificing efficiency. We devised an encoding of planning for PPLTL goals into classical planning for final-state 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., 2020], a strictly more expressive variant of PPLTL involving regular expressions. Indeed, also PPLDL has a fixpoint characterization of the temporal operators. This extension remains for future work. Finally, although our focus is on classical planning with PPLTL goals, the theoretical and practical advantages observed in this paper suggest that PPLTL could become a promising candidate for expressing temporally extended properties in other forms of planning, such as nondeterministic planning.

Chapter 8

Conclusions and Future Directions

8.1 Conclusions

This thesis gathers most of the research work done during my doctorate program and presents the main theoretical and empirical results. In particular, this dissertation focuses on planning to achieve complex temporal specifications represented as trajectory constraints and temporally extended goals.

We considered the well-known PDDL3 language to express trajectory constraints over state sequences, while we formalized a new language, called PAC, to express trajectory constraints on actions in a plan. Both languages can be formalized using a subset of Linear Temporal Logic formulas, over state trajectories for PDDL3 and over action sequences for PAC, and are simple, effective, and can be handled efficiently; indeed, although we could reformulate PDDL3 and PAC constraints in LTL_f, directly handling these languages results to be extremely effective. In particular, we designed two compilation approaches, TCORE for PDDL3 and PAC-C for PAC, which introduce a minimal amount of atoms, do not add spurious actions, and are sound and complete.

Regarding temporally extended goals, we considered Pure-Past Linear Temporal Logic, which is a temporal logic formalism that has been recently reviewed in De Giacomo et al. [2020], is as expressive as LTL_f, and only uses temporal operators that predicate on the past. While LTL_f requires dealing with possible extensions of a state trajectory, as remarked in Section 4.3, PPLTL formulas can be evaluated only using the past state trajectory that has already been computed. We exploit this observation to derive an encoding for planning with PPLTL goals to classical planning, which is polynomial, sound, complete, and that, differently from the polynomial compilation for LTL$_f$ goals, does not introduce any additional action.

These techniques for PDDL3, PAC, and PPLTL allow modern classical planners, such as LAMA, to solve planning problems with such specifications. A crucial aspect of the compilation techniques we propose is that they share the same theoretical advantages regarding complexity: the encodings are polynomial and *preserving plan size exactly* Nebel [2000]. To assess the practical effectiveness of these theoretical advantages, we performed a thorough experimental analysis involving novel benchmark cases.

For PDDL3, we used a sophisticated technique to generate PDDL3 planning problems starting from existing domains featuring only preferences. Moreover, we translated every PDDL3 problem as a planning problem with LTL$_f$ temporally extended goals, enabling the comparison between TCORE and the two state-of-the-art LTL$_f$ compilations LTL-E and LTL-P. For each compilation approach, we used LAMA as a classical planner. We also considered state-of-the-art planning systems that natively support PDDL3 constraints for these experiments. The results show that TCORE is the system that achieves the highest coverage overall.

Regarding PAC, we carefully designed new planning instances featuring control knowledge, expressed as PAC constraints, to help a classical planner solve more instances and improve the quality of solutions. For comparison, we expressed the same knowledge as PDDL3 constraints (when possible) and as LTL$_f$ goals, enabling an empirical comparison between PAC-C and the two compilations TCORE and LTL-E. As a baseline, we measured the performance of the classical planner LAMA with the benchmark instances without knowledge. Then, we ran LAMA over the problems with the knowledge handled by PAC-C, TCORE, and LTL-E. The results show that, overall, the same knowledge was more effective when expressed as PAC constraints and handled by PAC-C.

For PPLTL, we compared PLAN4PAST with LTL-E and LTL-P over a set of benchmarks that feature semantically equivalent LTL$_f$ and PPLTL goals. In particular, we took previous benchmark problems with LTL$_f$ goals and formulated such instances as planning problems with PPLTL goals. Since we observed that these instances were small and trivially solved by all compilations, we designed a new set of instances with semantically equivalent LTL_f and PPLTL goals that are more challenging and increasing in size. The instances obtained by the compilations were solved using LAMA. The empirical results show that PLAN4PAST outperforms both LTL-E and LTL-P; LTL-E blows up due to the exponential nature of such a compilation, while LAMA is not able to cope with the additional spurious actions introduced by the LTL-P schema. Instead, PLAN4PAST is neither exponential nor uses additional spurious actions. As a result, PLAN4PAST is the most effective compilation over the set of considered benchmarks.

8.2 Future Directions

The work done so far explores different ways of expressing and handling trajectory constraints and temporally extended constraints over states and action sequences. When properties are natural to express using state constraints, we can use PDDL3 or PPLTL as a specification language, while PAC offers a simple and effective way to express and handle action constraints. A research line that we are currently exploring concerns the combination of state and action properties in the trajectory constraints and temporal goals. Indeed, there is a plethora of properties that require such a combination. For instance, to require that "if the truck is at the depot and there is free space available, then it must load a package" we could specify the constraint:

$$\texttt{always}\ (At(depot) \wedge FreeSpace \Rightarrow LoadPackage)$$

This constraint combines the position and space inside a truck, encoded in the state, with the load action. Another example is "sometimes we drive from $city_1$ to $city_2$ while having 5 units of fuel", which could be expressed as:

$$\texttt{sometime}\ (FuelLevel(level_5) \wedge Drive(car, city_1, city_2))$$

These are compelling properties that cannot be naturally expressed in PDDL3, or

PAC. The next step is to formalize a new language that allows expressing constraints over trajectories of actions and states. We also plan to study a compilation to handle such a language, and we are interested in theoretically and experimentally proving the effectiveness of the proposed approach.

Another research line concerns the extension of PAC constraints to more complex planning formalisms that allow the parallel execution of actions. Many PAC constraints such as $\texttt{sometime}\ (\mathit{Drive}(\mathit{truck}_1, \mathit{city}_1, \mathit{city}_2) \wedge \mathit{Load}(\mathit{package}_2, \mathit{truck}_2))$ cannot be satisfied by sequential plans. If we allow actions to be executed in parallel, then this constraint could be interpreted as "The actions $\mathit{Drive}(\mathit{truck}_1, \mathit{city}_1, \mathit{city}_2)$ and $\mathit{Load}(\mathit{package}_2, \mathit{truck}_2)$ must sometimes be executed in parallel in the plan".

Extending the TCORE and PAC-C compilations to constraints interpreted over infinite executions would be an interesting direction, as some properties can be captured only in this setting. Suppose that we have two processes, P_1 and P_2, a resource r, and we want to enforce the following constraint: "Process P_1 eventually uses r. Moreover, every time P_1 uses r, then P_2 has to use r in the future, and vice versa. In addition, the resource cannot be used concurrently". This property can be captured in PDDL3 with the following set of constraints[1]:

$$\begin{aligned}
&\texttt{sometime}\ (P_1(r))\\
&\texttt{sometime-after}\ (P_1(r), P_2(r))\\
&\texttt{sometime-after}\ (P_2(r), P_1(r))\\
&\texttt{always}\ (\neg(P_1(r) \wedge P_2(r))).
\end{aligned}$$

A valid solution requires alternating r between P_1 and P_2 infinitely often. To handle this class of properties[2], we could compactly represent PDDL3 and PAC constraints as Büchi automatons and then exploit well-known compilations for LTL goals by Patrizi et al. [2011, 2013] for classical and nondeterministic planning domains.

Lastly, we are interested in developing effective planning approaches that can handle complex formalisms like PPLTL directly into the search engine. Although some work addresses the problem of planning with temporally extended goals directly in

[1]This example has been adapted from De Giacomo et al. [2014].

[2]We remind the reader that PPLTL formulas can be interpreted only over finite executions.

the search engine, such as Bacchus and Kabanza [1998], current approaches have been developed in the context of using temporal logic as control knowledge to guide the search, rather than an objective to be achieved. Indeed, developing a planner that employs a PPLTL-aware heuristic is a very relevant challenge in AI planning that has yet to be addressed.

Acknowledgments

I extend my heartfelt gratitude to all of those who, directly or indirectly, played a role in my personal and professional development throughout my Ph.D.

First and foremost, I am deeply thankful to my supervisors Alfonso Gerevini and Enrico Scala, whose invaluable expertise and support have been instrumental in achieving significant milestones. This dissertation would not have been possible without your guidance. Moreover, I thank Enrico for being a mentor and a friend who has contributed to my personal and professional growth.

Next, I want to thank Giuseppe De Giacomo for his supervision during my academic visit to Oxford, his collaboration as a co-author, and his feedback on my thesis. Also, the interactions with Antonio Di Stasio and Shufang Zhu during my time at the University of Oxford were enriching experiences, for which I am sincerely grateful. I also thank Nir Lipovetzky for giving me valuable suggestions and feedback to improve this thesis.

I express my gratitude to my esteemed co-author and friend, Francesco Fuggitti, whose dedication and countless hours of insightful discussions have been invaluable in shaping both my personal and professional achievements. A special appreciation is reserved for Francesco Percassi who has been a co-author, a supervisor of my Master's Thesis, and most importantly a friend who gave me invaluable insights into research, career advancement, and life itself.

I am fortunate to have been surrounded by talented colleagues and friends throughout my academic journey. I extend my heartfelt thanks to Mattia Chiari, who has been a steadfast companion from the beginning of my Ph.D. journey till the very end. I also would like to thank all those colleagues who made the open space a peaceful and stimulating environment. Among these, a special mention goes to Luca, Matteo, and Diego, and to the students I supervised, Alberto and Valerio.

I am deeply grateful to my family members, including Girolamo, Mariagrazia, Gloria, and Luca, my grandmothers Anna and Matilde, and my uncles and cousins. I also want to remember my grandparents, Giuseppe and Bruno. Thank you for your unwavering encouragement, care, love, and support throughout my life. Your presence has helped me endure and overcome every obstacle!

To my childhood and high school friends, Giovanni, Matteo, Luca, Michele, Simone, and Christian, I extend my sincere appreciation for the enduring friendship and shared memories. Your presence has helped me a lot. I also want to mention Davide Scassola, whose unique intellect and friendship have enriched my life in countless ways.

Finally, I express deep gratitude to Pietro Bonardi, an invaluable friend who helped me make the most impactful changes in my life. His presence has made me a great, confident person with high self-esteem. Thank you, Pietro! My life would not have been the same without you.

Grazie di cuore a tutti!
Luigi

Bibliography

B. Aminof, G. De Giacomo, A. Murano, and S. Rubin. Planning under LTL environment specifications. In *ICAPS*, pages 31–39. AAAI Press, 2019.

F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Ann. Math. Artif. Intell.*, 22(1-2):5–27, 1998.

F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artif. Intell.*, 116(1-2):123–191, 2000.

F. Bacchus, C. Boutilier, and A. J. Grove. Rewarding behaviors. In *AAAI/IAAI, Vol. 2*, pages 1160–1167. AAAI Press / The MIT Press, 1996.

F. Bacchus, C. Boutilier, and A. J. Grove. Structured solution methods for non-markovian decision processes. In *AAAI/IAAI*, pages 112–117. AAAI Press / The MIT Press, 1997.

C. Baier and J. Katoen. *Principles of model checking.* MIT Press, 2008.

J. A. Baier and S. A. McIlraith. Planning with first-order temporally extended goals using heuristic search. In *AAAI*, pages 788–795. AAAI Press, 2006a.

J. A. Baier and S. A. McIlraith. Planning with temporally extended goals using heuristic search. In *ICAPS*, pages 342–345. AAAI, 2006b.

J. A. Baier and S. A. McIlraith. Planning with preferences. *AI Mag.*, 29(4):25–36, 2008.

J. Benton, A. J. Coles, and A. Coles. Temporal planning with preferences and time-dependent continuous costs. In *ICAPS*. AAAI, 2012.

M. Bienvenu, C. Fritz, and S. A. McIlraith. Specifying and computing preferred plans. *Artif. Intell.*, 175(7-8):1308–1345, 2011.

L. Bonassi, A. E. Gerevini, F. Percassi, and E. Scala. On planning with qualitative state-trajectory constraints in PDDL3 by compiling them away. In *ICAPS*, pages 46–50. AAAI Press, 2021.

L. Bonassi, A. E. Gerevini, and E. Scala. Planning with qualitative action-trajectory constraints in PDDL. In *IJCAI*, pages 4606–4613. ijcai.org, 2022a.

L. Bonassi, E. Scala, and A. E. Gerevini. Planning with PDDL3 qualitative constraints for cost-optimal solutions through compilation (short paper). In *IPS/RiCeRcA/SPIRIT@AI*IA*, volume 3345 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022b.

L. Bonassi, G. De Giacomo, M. Favorito, F. Fuggitti, A. E. Gerevini, and E. Scala. Planning for temporally extended goals in pure-past linear temporal logic. In *ICAPS*, pages 61–69. AAAI Press, 2023a.

L. Bonassi, G. De Giacomo, M. Favorito, F. Fuggitti, A. E. Gerevini, and E. Scala. FOND planning for pure-past linear temporal logic goals. In *ECAI*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 279–286. IOS Press, 2023b.

L. Bonassi, A. E. Gerevini, and E. Scala. Dealing with numeric and metric time constraints in PDDL3 via compilation to numeric planning. In *AAAI*, pages 20036–20043. AAAI Press, 2024.

A. Camacho, M. Bienvenu, and S. A. McIlraith. Towards a unified view of AI planning and reactive synthesis. In *ICAPS*, pages 58–67. AAAI Press, 2019.

A. J. Coles and A. Coles. LPRPG-P: relaxed plan heuristics for planning with preferences. In *ICAPS*. AAAI, 2011.

G. De Giacomo and S. Rubin. Automata-theoretic foundations of FOND planning for ltlf and ldlf goals. In *IJCAI*, pages 4729–4735, 2018.

G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pages 854–860. IJCAI/AAAI, 2013.

G. De Giacomo and M. Y. Vardi. Synthesis for LTL and LDL on finite traces. In *IJCAI*, pages 1558–1564. AAAI Press, 2015.

G. De Giacomo, R. D. Masellis, and M. Montali. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *AAAI*, pages 1027–1033. AAAI Press, 2014.

G. De Giacomo, A. D. Stasio, F. Fuggitti, and S. Rubin. Pure-past linear temporal and dynamic logic on finite traces. In *IJCAI*, pages 4959–4965. ijcai.org, 2020.

C. Domshlak, J. Hoffmann, and M. Katz. Red-black planning: A new systematic approach to partial delete relaxation. *Artif. Intell.*, 221:73–114, 2015.

S. Edelkamp and J. Hoffmann. Pddl2. 2: The language for the classical part of the 4th international planning competition. Technical report, Technical Report 195, University of Freiburg, 2004.

S. Edelkamp, S. Jabbar, and M. Nazih. Large-scale optimal pddl3 planning with mips-xxl. *5th International Planning Competition Booklet (IPC-2006)*, pages 28–30, 2006a.

S. Edelkamp, S. Jabbar, and M. Nazih. Large-scale optimal pddl3 planning with mips-xxl. *5th International Planning Competition Booklet (IPC-2006)*, pages 28–30, 2006b.

E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. Elsevier and MIT Press, 1990.

M. Fox and D. Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124, 2003.

F. Fuggitti. *Efficient Techniques for Automated Planning for Goals in Linear Temporal Logics on Finite Traces*. Phd dissertation, Sapienza University & York University, September 2023.

D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL*, pages 163–173. ACM Press, 1980.

A. Gerevini and D. Long. Bnf description of pddl3. 0. *Unpublished manuscript from the IPC-5 website*, 2005.

A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668, 2009.

P. Haslum. Optimal delete-relaxed (and semi-relaxed) planning with conditional effects. In *IJCAI*, pages 2291–2297. IJCAI, 2013.

M. Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006.

M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535, 2009.

J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, 14:253–302, 2001.

C. Hsu, B. W. Wah, R. Huang, and Y. Chen. Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In *IJCAI*, pages 1924–1929, 2007.

L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

H. J. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.

J. Li, K. Y. Rozier, G. Pu, Y. Zhang, and M. Y. Vardi. Sat-based explicit ltlf satisfiability checking. In *AAAI*, pages 2946–2953. AAAI Press, 2019.

D. Long and M. Fox. The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res.*, 20:1–59, 2003.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl – the planning domain definition language. Technical report, ICAPS, 1998.

B. Nebel. On the compilability and expressive power of propositional planning formalisms. *JAIR*, 12:271–315, 2000.

F. Patrizi, N. Lipovetzky, G. De Giacomo, and H. Geffner. Computing infinite plans for LTL goals using a classical planner. In *IJCAI*, pages 2003–2008. IJCAI/AAAI, 2011.

F. Patrizi, N. Lipovetzky, and H. Geffner. Fair LTL synthesis for non-deterministic systems using strong cyclic planners. In *IJCAI*, pages 2343–2349. IJCAI/AAAI, 2013.

J. S. Penberthy and D. S. Weld. Temporal planning with continuous change. In *AAAI*, pages 1010–1015. AAAI Press / The MIT Press, 1994.

F. Percassi and A. E. Gerevini. On compiling away PDDL3 soft trajectory constraints without using automata. In *ICAPS*, pages 320–328. AAAI Press, 2019.

A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

T. Ramesh. Traveling purchaser problem. *Opsearch*, 18(1-3):78–91, 1981.

S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.*, 39:127–177, 2010.

J. Rintanen. Regression for classical and nondeterministic planning. In *ECAI*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 568–572. IOS Press, 2008.

G. Röger, F. Pommerening, and M. Helmert. Optimal planning in the presence of conditional effects: Extending lm-cut with context splitting. In *ECAI*, volume 263 of *FAIA*, pages 765–770. IOS, 2014.

S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010.

S. Thiébaux, J. Hoffmann, and B. Nebel. In defense of pddl axioms. *AIJ*, 168(1-2): 38–69, 2005.

J. Torres and J. A. Baier. Polynomial-time reformulations of LTL temporally extended goals into final-state goals. In *IJCAI*, pages 1696–1703. AAAI Press, 2015.

W. M. P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Comput. Sci. Res. Dev.*, 23(2):99–113, 2009.

B. Wright, R. Mattmüller, and B. Nebel. Compiling away soft trajectory constraints in planning. In *KR*, pages 474–483. AAAI Press, 2018.