Planning with Qualitative Action-Trajectory Constraints in PDDL*

Luigi Bonassi , **Alfonso Emilio Gerevini** and **Enrico Scala**
Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Brescia, Italy
{l.bonassi005, alfonso.gerevini, enrico.scala}@unibs.it

Abstract

In automated planning the ability of expressing constraints on the structure of the desired plans is important to deal with solution quality, as well as to express control knowledge. In PDDL3 this is supported through state-trajectory constraints corresponding to a class of LTL_f formulae. In this paper, first we introduce a formalism to express trajectory constraints over actions in the plan, rather than over traversed states; the new class of constraints retains the same temporal modal operators of PDDL3, and adds two useful modalities. Then we investigate compilation-based methods to deal with action-trajectory constraints in propositional planning, and propose a new simple effective method. Finally, we experimentally study the usefulness of our action-trajectory constraints as a tool to express control knowledge. The experimental results show that the performance of a classical planner can be significantly improved by exploiting knowledge expressed by action constraints and handled by our compilation, while the same knowledge turns out to be less beneficial when specified as state constraints and handled by two state-of-the-art systems supporting state constraints.

1 Introduction

In automated planning temporal extended goals are constraints over the state trajectory of a plan that can be used to express desired properties of the solutions for a planning problem or domain control knowledge aimed at helping the planner. Linear Temporal Logic over finite traces (LTL_f) [Pnueli, 1977; Giacomo *et al.*, 2014] and PDDL3 [Gerevini *et al.*, 2009] are two of the most popular languages used to formulate such constraints in domain-independent planning.

In this paper, similarly to what done in [Bienvenu *et al.*, 2011], we study an alternative way of expressing plan constraints and control knowledge through constraints over trajectories of actions rather than states, and we propose a new simple formalism extending classical planning with such constraints. For instance, in a logistics domain, for a planning problem we could request that a certain truck should drive from `city1` to `city2` during its journey, or that it should be refueled before driving.

Action-trajectory constraints (hereinafter called action constraints) cannot be easily and naturally expressed using state-trajectory constraints (hereinafter state constraints). This requires that the domain is carefully modified by introducing additional fluents and revising the action models, for each problem in the domain that has different action constraints.[1] On the contrary, to formulate action constraints we do not have to know how states and actions are modeled: we just need to use the labels of the (instantiated) actions and relate them via a temporal modal operator. Moreover, since action constraints are independent from the state representation, they could also be used with a more complex state representation, such as in numeric planning [Fox and Long, 2003].

The proposed language to express action constraints retains the same temporal modal operators of PDDL3, and adds two useful modalities (`always-next` and `pattern`). The language was designed with the purpose of expressing useful knowledge without incurring in significant computational overheads to handle it at planning time.

After introducing classical planning enriched with action constraints (PAC), we investigate compilation-based methods to deal with action constraints in PAC planning, and propose a new effective method. Our method not only is polynomial, but it also generates a compiled problem that has solutions with exactly the same length of the solutions for the original problem. This is an advantage over other existing formalisms for expressing control knowledge in planning, such as those based on LTL_f, that need more complex and costly compilations or increase the length of the compiled plans [Torres and Baier, 2015; Bienvenu *et al.*, 2011; Baier *et al.*, 2008].

Then we experimentally study the usefulness of action constraints as a tool to express control knowledge and plan quality. The experimental results show that the performance of a classical planner can be significantly improved by exploit-

*This paper has been published in the Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22).

[1] E.g., stating `(sometime (at truck1 city2))` in PDDL3 does not work to simply request that the truck should drive from `city1` to `city2`, if in the domain the truck can reach `city2` from more cities.

ing knowledge expressed by action constraints and handled by our compilation method, while the same knowledge turns out to be less beneficial when formulated as state constraints and handled by two state-of-the-art systems supporting state constraints [Baier and McIlraith, 2006; Bonassi *et al.*, 2021].

The rest of the paper is organised as follows: Section 2 introduces PAC; Section 3 shows how action constraints can be compiled away in classical planning; Section 4 presents our experimental analysis; Section 5 briefly comments on related work; finally Section 6 gives the conclusions.

2 Classical Planning with Action Constraints

A classical planning problem is a tuple $\Pi = \langle F, A, I, G\rangle$ where F is a set of atoms, $I \subseteq F$ is the initial state, G is a formula over F, and A is a set of actions. An action $a \in A$ is a pair $\langle \mathit{Pre}(a), \mathit{Eff}(a)\rangle$, where $\mathit{Pre}(a)$ is a formula over F expressing the preconditions of a, and $\mathit{Eff}(a)$ is a set of conditional effects, each of the form $c \triangleright e$, where c is a formula and e is a set of literals, both over F. With e^- and e^+ we indicate the partition of e featuring only negative and positive literals, respectively. A state s is a subset of F, with the meaning that if $p \in s$, then p is true in s, and if $p \notin s$, p is false in s. An action is applicable in s if $s \models \mathit{Pre}(a)$, and the application of an action a in s yields the state $s' = (s \setminus \bigcup_{\substack{c \triangleright e \in \mathit{Eff}(a) \\ \text{with } s \models c}} e^-) \cup \bigcup_{\substack{c \triangleright e \in \mathit{Eff}(a) \\ \text{with } s \models c}} e^+$. We indicate with $s[a]$ the state resulting from applying action a in s, and, with a little abuse of notation, we write a conditional effect of the form $\top \triangleright e$ as a simple unconditional effect e. A plan π for a problem $\Pi = \langle F, A, I, G\rangle$ is a sequence of actions $\langle a_1, a_2, \ldots, a_n\rangle$ from A; π is valid for Π (a solution) iff the sequence $\langle s_1 = I, s_2 = s_1[a_1], \ldots, s_{n+1} = s_n[a_n]\rangle$ of states (state trajectory) is such that $\forall\ i \in [1, \ldots, n]\ s_i \models \mathit{Pre}(a_i)$, and $s_{n+1} \models G$. We denote with $|\pi|$ the length of plan π, and with $\pi(t)$ the t-th action of π.

The following definition formalizes the semantics of a formula ϕ in Negation Normal Form (NNF) defined over a set A of actions, intended as the set of atoms formed by the labels naming the actions in A.

Definition 1. *Let A be a set of actions of a planning problem. Given a plan π, 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. I.e., given a plan formed by a set of actions $\{a_1, a_2, \ldots, a_n\}$, the following formulae over the action atoms hold for $t = 1 \ldots |\pi|$:

$\pi(t) = a_i \ \Rightarrow\ \pi(t) \neq a_j\ , \forall j \in \{1 \ldots n\},\ j \neq i$

$\pi(t) = a_1 \vee \pi(t) = a_2 \vee \ldots \vee \pi(t) = a_n$.

Due to these properties of a sequential plan, there is only a restricted class of relevant formulae over action literals. For instance, action formula $a_1 \vee \neg a_2$ can be rewritten as the disjunction of 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 $\neg a_1 \wedge a_2$. Such formula is equivalent to a_2, since a_2 is the only action satisfying $\neg a_1 \wedge a_2$.

In general, it can be proven that for a sequential plan *every action formula can be rewritten as an equivalent formula ϕ that is either a disjunction of positive action literals, $\top$ or $\bot$.* An action atom a satisfies a disjunction of positive literals ϕ if a is a disjunct of ϕ, and we denote this by $a \in \phi$ (analogously, $a \notin \phi$ denotes that a is not a disjunct of ϕ). In the rest of the paper, we assume that all action formulae are rewritten as disjunctions of positive literals. If a formula is equivalent to $\top$ (resp. $\bot$), we can rewrite it as the disjunction of all action atoms of the planning problem (resp. the empty disjunction).

We introduce *action constraints* as a class of temporal logic formulae over a sequence π of actions. Such constraints use the same modal operators of the qualitative state constraints in PDDL3, except for the additional operators `always-next` and `pattern`. Specifically, they can be of the following types (where ϕ and ψ are action formulae): (`always` ϕ), shortened as A_ϕ, requires that only actions that satisfy ϕ are in π; (`sometime` ϕ), shortened as ST_ϕ, requires that at least one action satisfying ϕ is in π; (`at-most-once` ϕ), shortened as AO_ϕ, requires that an action satisfying ϕ can appear in π only if no action satisfying ϕ is in π before; (`sometime-before` ϕ ψ), shortened as $\mathsf{SB}_{\phi,\psi}$, 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; (`sometime-after` ϕ ψ), shortened as $\mathsf{SA}_{\phi,\psi}$, 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; (`always-next` ϕ ψ), shortened as $\mathsf{AX}_{\phi,\psi}$, requires that if an action satisfying ϕ is in π, then it is immediately followed by an action satisfying ψ;[2] (`pattern` $\phi_1 \ldots \phi_k$), shortened as $\mathsf{PA}_{\phi_1 \ldots \phi_k}$, requires that, for $i = 1 \ldots k-1$, there exists an action in π satisfying ϕ_i followed at some later time by an action satisfying ϕ_{i+1}.

Definition 2. *Given a plan $\pi = \langle a_1, a_2, \ldots, a_n\rangle$, the following rules define when an action constraint is satisfied by π:*

π satisfies (`always` ϕ) *iff* $\forall t : 1 \leq t \leq |\pi| \cdot \pi(t) \in \phi$

π satisfies (`sometime` ϕ) *iff* $\exists t : 1 \leq t \leq |\pi| \cdot \pi(t) \in \phi$

π satisfies (`at-most-once` ϕ) *iff* $\forall t_1 : 1 \leq t_1 \leq |\pi| \cdot$ *if* $\pi(t_1) \in \phi$ *then* $\forall t_2 : t_1 < t_2 \leq |\pi| \cdot \pi(t_2) \notin \phi$

π satisfies (`sometime-after` ϕ ψ) *iff* $\forall t_1 : 1 \leq t_1 \leq |\pi| \cdot$ *if* $\pi(t_1) \in \phi$ *then* $\exists t_2 : t_1 \leq t_2 \leq |\pi| \cdot \pi(t_2) \in \psi$

π satisfies (`sometime-before` ϕ ψ) *iff* $\forall t_1 : 1 \leq t_1 \leq |\pi| \cdot$ *if* $\pi(t_1) \in \phi$ *then* $\exists t_2 : 1 \leq t_2 < t_1 \cdot \pi(t_2) \in \psi$

π satisfies (`always-next` ϕ ψ) *iff* $\forall t : 1 \leq t < |\pi| \cdot if\ \pi(t) \in \phi$ *then* $\pi(t+1) \in \psi$ *and* $\pi(|\pi|) \notin \phi$

π satisfies (`pattern` $\phi_1 \ldots \phi_k$) *iff* $\exists$ *a sequence of actions* $\langle a_1, \ldots, a_k\rangle$ *from π that are ordered as in π, such that* $\forall i \in \{1, \ldots, k\}\ a_i \in \phi_i$.

We call PAC the class of classical planning problems enriched with action constraints.

[2] This constraint can express some complex valid action sequences. E.g., $\mathsf{AX}_{a,a\vee b}$ accepts plans with any number of consecutive a's repetitions, if this sequence terminates with action b.

Definition 3. *A classical planning problem with action constraints (a* PAC *problem) is a tuple* $\langle \Pi, C \rangle$ *where* Π *is a classical planning problem and* C *is a set of action constraints.*

The valid plans of $\langle \Pi, C \rangle$ are the valid plans of Π that satisfy all constraints in C. PAC allows the definition of action constraints that cannot be captured by using state constraints. Indeed, the *same* sequence (trajectory) of states can be generated by different sequences (trajectories) of actions. Therefore, in general, it is not possible to distinguish the different action trajectories by constraints over the state trajectory only. For instance, consider a problem where the set of actions A is $\{a_1, a_2, a_3\}$ with $a_1 = \langle p_0, \{e_0\}\rangle$, $a_2 = \langle p_1, \{e_0\}\rangle$, $a_3 = \langle e_0, \{goal\}\rangle$, $I = \{p_0, p_1\}$, and $G = goal$. Both plans $\pi_1 = \langle a_1, a_3\rangle$ and $\pi_2 = \langle a_2, a_3\rangle$ solve the problem inducing the same state trajectory $\langle \{p_0, p_1\}, \{p_0, p_1, e_0\}, \{p_0, p_1, e_0, goal\}\rangle$. Consider now action constraint (`sometime` 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 by just looking at the (same) sequence of states induced by π_1 and π_2: there is no state-trajectory constraint that rules π_1 out.

We allow all constraints to be formulated in first-order representation. For instance, by writing:

$$\forall\, bus\,.\,(\texttt{sometime}\ \exists\, city\,.\ Drive(bus, city, \textit{city-a}) \vee Drive(bus, city, \textit{city-b}))$$

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

3 Solving PAC Problems through Compilation

In this section we propose a compilation schema, called PAC-C (PAC compiler), that translates a PAC problem into an equivalent classical planning problem. We distinguish our action constraints in two classes: *Prevent Constraints* (PC) and *Request Constraints* (RC). Intuitively, PCs are constraints used to express properties that must not be violated at any time in the plan, while RCs enforce that certain actions must occur in every solution plan. PCs are: A_ϕ, $\mathsf{SB}_{\phi,\psi}$, AO_ϕ and $\mathsf{AX}_{\phi,\psi}$; RCs are: ST_ϕ, $\mathsf{SA}_{\phi,\psi}$ and $\mathsf{PA}_{\phi_1 \ldots \phi_k}$. PAC-C works by preventing the execution of actions that would violate some PCs, and forcing the planner to include in the plan the actions necessary to satisfy all RCs on a state dependent basis.

Actions that cannot appear in the plan at some time step t depend on actions scheduled before t. For instance, if (`at-most-once` a) is a required constraint, then 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 RC. To record the presence in the plan under construction of the actions relevant for the constraints, PAC-C uses a set of fresh atoms, built by taking into account the constraint at hand, as described below.

PC Atoms. For every AO_ϕ 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.

Algorithm 1: PAC-C

Input : A PAC Problem $\Pi = \langle \langle F, A, I, G\rangle, C\rangle$
Output: A classical planning problem equivalent to Π

/* Phase (I) */
1 $F' = F \cup$ *PC-atoms* $\cup$ *RC-atoms*
2 $I' = I \cup \bigcup_{\mathsf{SA}_{\phi,\psi}} got_{\phi,\psi}$
/* Phase (II) */
3 $A' = \{a \mid a \in A$ and for each $\mathsf{A}_\phi \in C,\ a \in \phi\}$
4 **foreach** $a \in A'$ **do**
5 **foreach** $c \in PC(C)$ **do**
6 **if** $c = \mathsf{AO}_\phi$ **and** $a \in \phi$ **then**
7 $Pre(a) = Pre(a) \wedge \neg done_\phi$
8 $Eff(a) = Eff(a) \cup \{done_\phi\}$
9 **if** $c = \mathsf{SB}_{\phi,\psi}$ **then**
10 **if** $a \in \phi$ **then** $Pre(a) = Pre(a) \wedge done_\psi$
11 **if** $a \in \psi$ **then** $Eff(a) = Eff(a) \cup \{done_\psi\}$
12 **if** $c = \mathsf{AX}_{\phi,\psi}$ **then**
13 **if** $a \in \phi$ **then** $Eff(a) = Eff(a) \cup \{request_\psi\}$
14 **else if** $a \in \psi$ **then** $Eff(a) = Eff(a) \cup \{\neg request_\psi\}$
15 **if** $a \notin \psi$ **then** $Pre(a) = Pre(a) \wedge \neg request_\psi$
16 **foreach** $c \in RC(C)$ **do**
17 **if** $c = \mathsf{ST}_\phi$ **and** $a \in \phi$ **then** $Eff(a) = Eff(a) \cup \{got_\phi\}$
18 **if** $c = \mathsf{SA}_{\phi,\psi}$ **then**
19 **if** $a \in \psi$ **then** $Eff(a) = Eff(a) \cup \{got_{\phi,\psi}\}$
20 **if** $a \in \phi$ **and** $a \notin \psi$ **then** $Eff(a) = Eff(a) \cup \{\neg got_{\phi,\psi}\}$
21 **if** $c = \mathsf{PA}_{\phi_1 \ldots \phi_k}$ **then**
22 **foreach** $\phi_i \in \langle \phi_1 \ldots \phi_k\rangle \cdot a \in \phi_i$ **do**
23 $Eff(a) = Eff(a) \cup \begin{cases} \{stage_c^{i-1} \triangleright stage_c^i\} & \textbf{if } i > 1 \\ \{stage_c^1\} & \textbf{otherwise} \end{cases}$
/* Phase (III) */
24 $G' = G \wedge \bigwedge_{\mathsf{SA}_{\phi,\psi} \in C} got_{\phi,\psi} \wedge \bigwedge_{\mathsf{ST}_\phi \in C} got_\phi \wedge \bigwedge_{\mathsf{AX}_{\phi,\psi} \in C} \neg request_\psi \wedge \bigwedge_{c = \mathsf{PA}_{\phi_1 \ldots \phi_k} \in C} stage_c^k$
25 **return** $\langle F', A', I', G'\rangle$

RC Atoms. For every ST_ϕ 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:

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

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

Compilation schema. Algorithm 1 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 RCs and AXs constraints.

Phase (I). The necessary PC and RC atoms are created and the initial state is setup (lines 1-2). 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 PC, 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 AO_ϕ), (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 of it triggered by the previous action (in the case of an $\mathsf{AX}_{\phi,\psi}$). For PCs, new effects are added to keep track of the execution of relevant actions. For instance, an $\mathsf{AX}_{\phi,\psi}$ constraint requires to restrict 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 13-15). If an action does not trigger the constraint and satisfies ψ instead, then $\neg request_\psi$ is added to its effects (lines 13-14), disabling the request of ψ demanded by the constraint.

For RCs, PAC-C adds a set of effects to keep track of the relevant actions that appear in the plan. A ST_ϕ constraint requires that at least one action that satisfies ϕ appears sometime in the plan. 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 19-20). For a $\mathsf{PA}_{\phi_1 \dots \phi_k}$ constraint, the algorithm checks if an action satisfies any formula in $\{\phi_1 \dots \phi_k\}$. E.g., if an action a in the plan makes formula ϕ_i true and $(\texttt{pattern}\ \phi_1 \dots \phi_{i-1})$ is already satisfied by the plan prefix up to a, then $(\texttt{pattern}\ \phi_1 \dots \phi_i)$ will become satisfied. PAC-C keeps track of this information by adding conditional effect $stage_c^{i-1} \triangleright stage_c^i$ to all actions satisfying ϕ_i (line 23).

Phase (III). The last step consists in setting up the new goals of the problem: all the ST_ϕ, $\mathsf{SA}_{\phi,\psi}$, $\mathsf{AX}_{\phi,\psi}$ and $\mathsf{PA}_{\phi_1 \dots \phi_k}$ 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 24).

The additional preconditions and effects of the compilation prevent the planner form generating any sequence of actions that violates one or more PCs, while the additional goals force the planner to satisfy all RCs. 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 the modified plan have exactly the same length.

Theorem 1. *Let $\Pi = \langle\langle F, A, I, G\rangle, C\rangle$ be a* PAC *problem and $\Pi' = \langle F', A', I', G'\rangle$ the problem obtained by compiling Π through Algorithm 1. A plan $\pi = \langle a_1, a_2, \dots, a_n\rangle$ is a solution for Π iff plan $\pi' = \langle \tau(a_1), \tau(a_2), \dots, \tau(a_n)\rangle$ is a solution for Π', where $\tau(a_i)$ is the transformation of a_i performed by Algorithm 1 (Phase II) for $i = 1 \dots n$.*

Proof Sketch. Both directions can be proven by contradiction, considering each type of constraints one by one. That is, we show that if π' (π) is not a valid plan for Π' (Π) then also π (π') cannot be a valid plan for Π (Π'). Full proof in the supplementary material.[3] □

[3]Supplementary material, benchmark domains and Python implementation of PAC-C can be found at https://bit.ly/3kerz8s.

4 Experimental Analysis

Our experiments are aimed at evaluating the usefulness of action constraints as knowledge that can be effectively exploited to improve problem-solution coverage and plan quality. We evaluate the behavior of a classical planner with/without this (compiled) knowledge. For comparison reasons 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 [Giacomo and Vardi, 2013] or PDDL3 [Gerevini *et al.*, 2009]. As classical planner we used LAMA [Richter and Westphal, 2010], that was run on the original benchmark problems and on the corresponding problems extended with control knowledge. Such knowledge was compiled by three different methods: PAC-C[3] (for action constraints), TCORE (for PDDL3 constraints) [Bonassi *et al.*, 2021], and LTL-C (for LTL_f constraints) [Baier and McIlraith, 2006]. To the best of our knowledge, TCORE and LTL-C are the most effective approaches to deal with the considered class of constraints.

We measured performance in terms of number of solved instances (coverage), CPU time of the planner, and plan length of the solution (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.

4.1 Benchmark Design

Since there are no available benchmarks featuring action constraints, we generated a new benchmark suite[3] 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 with the purpose of boosting problem coverage, while in the other domains the constraints were designed to improve plan quality. Our benchmark suite involves 160 instances: 30 in each of `Trucks`, `Storage`, `TPP` and `Openstack`, and 40 in `Rover`. To evaluate the use of LTL_f and PDDL3, for each instance we generated two further instances: one encoding the action constraints into an equivalent formulation in LTL_f; the other encoding an equivalent instance using PDDL3 qualitative state-trajectory constraints. Such instances were not formulated starting from the action constraints specification; that is, the constraint knowledge was directly formulated into either action constraints or state constraints (PDDL3 or LTL_f), without going through action constraints first. Note that the conversion in PDDL3 has been possible only for a subset of the considered domains. In what follows we describe the constraints introduced in each domain.

`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 plans of very bad quality. 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:

$$(\texttt{always}\ \forall\, from, to\,.\, \neg Drive(truck_2, from, to))$$
$$(\texttt{always}\ \neg Drive(truck_1, market_2, depot_1) \wedge \neg Drive(truck_1, market_1, market_2))$$

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

$$(\texttt{pattern}\ Drive(truck_1, depot_1, market_2)\ Drive(truck_1, market_2, market_1)\ Drive(truck_1, market_1, depot_1))$$

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 go back to unload the product. We enforced this by constraint

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

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

$$(\texttt{always-next}\ \exists product, \exists market.\ Buy(truck_1, product, market)\ \exists product, \exists market, \exists level.\ Load(product, truck_1, market, level))$$

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:

$$\Diamond(at(truck_1, depot_1) \wedge \bigcirc at(truck_1, market_2) \wedge \bigcirc(\Diamond(at(truck_1, market_2) \wedge \bigcirc at(truck_1, market_1) \wedge \bigcirc(\Diamond(at(truck_1, market_1) \wedge \bigcirc at(truck_1, depot_1))))))$$

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 by 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 create 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:

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

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 $\phi = \exists hoist.\ lifting(hoist, c)$:

$$\Box(\phi \Rightarrow (\phi \mathcal{U}(\Box(\neg\phi) \vee final)))$$

Each benchmark instance has from 2 to 21 constraints.

`Trucks`. This is a logistics domain concerning the delivery of 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 instances has from 3 to 20 constraints.

`Rover`. The objective is acquiring 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 there are many actions that 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 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:

$$(\texttt{sometime-before}\ \exists rover.\ Send\_soil\_data(rover, waypoint_x)\ \exists rover.\ Send\_rock\_data(rover, waypoint_y))$$

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

$$(\neg\phi \wedge \psi)\,\mathcal{R}(\neg\phi)$$
with $\phi = Communicated\_soil\_data(waypoint_x)$
$\psi = Communicated\_rock\_data(waypoint_y)$

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 to 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 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` feature these two constraints.

Domain	BASELINE	PAC-C	LTL-C	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 1: Coverage of the considered systems. In parenthesis, the number of benchmark instances for a given domain.

Domain	Improved instances			Avg improvement		
	PAC-C	LTL-C	TCORE	PAC-C	LTL-C	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 2: Pairwise comparison of PAC-C/LTL-C/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.

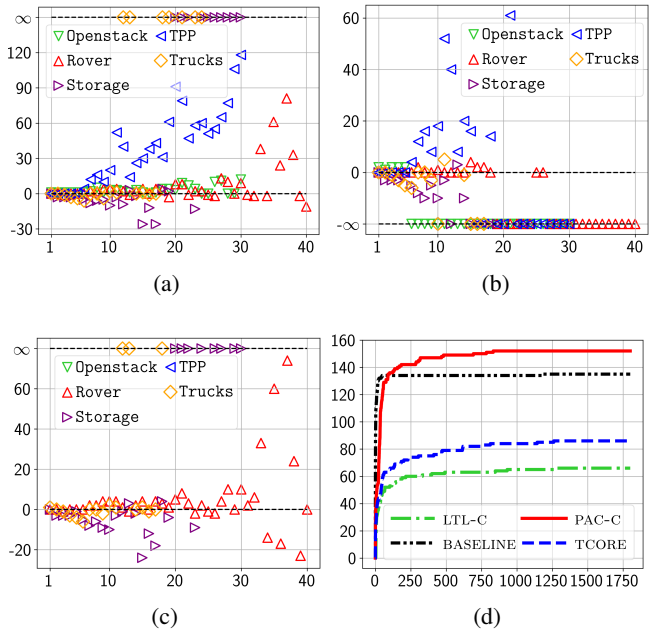

Figure 1: Quality improvement measured as difference of plan length (y-axis) between PAC-C and the BASELINE (a), LTL-C and the BASELINE (b), TCORE and the BASELINE (c), over instances solved by at least one of the two compared systems (x-axis). (d) Coverage (y-axis) versus planning time (x-axis).

4.2 Experimental Results

Coverage. Table 1 shows the overall coverage achieved using BASELINE (LAMA run on the original instances without constraints), PAC-C, LTL-C 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 will obstruct future movements inside the depot, and this is not captured by LAMA's heuristic. This cannot happen for the instances with action constrains: 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 constrains can improve the performance of a state-of-the-art classical planner. Also using TCORE coverage is incremented, 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-C, we did not obtain any improvement. Rather, the performance of LAMA was even worsened (coverage reduces for all considered domains). PAC-C turned out to be (much) more effective, coverage wise, than LTL-C and TCORE.

Plan quality. Table 2 and Figures 1a, 1b and 1c give an overall picture of the quality of the plans obtained by the compilation-based systems with respect to the baseline across all domains. From Figure 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 quality 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 lead to better quality solutions. With LTL-C 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, plan quality is worsened by all compared systems. In our benchmarks, coverage and quality is 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 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 keeps increasing coverage, outperforming the BASELINE after about 90 CPU seconds.

5 Related Work

Micheli and Scala (2019) introduced a formalism supporting action constraints specified as quantified temporal metric axioms. Such axioms define constraints over the execution tim-

ing of the actions. The constraints in PAC focus on a class of *qualitative* temporal constraints allowing to handle the constraints through a polynomial compilation into classical planning. Handling quantified temporal metric axiom is much more involved, and the computational complexity of planning with such axioms is still unknown.

The generation of *macro actions* is an approach to synthesize control knowledge where multiple consecutive actions are combined into a single macro action (e.g. [Botea *et al.*, 2005]). PAC can express sequences of (possibly disjunctive) actions through `pattern` constraints. The main differences are that (i) in this type of constraints actions are not necessarily consecutive, and (ii) macro actions do not express constraints a valid plan should satisfy (typically macros are added to the domain without removing any original action).

Hierarchical Task Network (HTN) [Erol *et al.*, 1994] is a well-know approach to planning that supports control knowledge. The main difference with PAC planning is that in HTN constraints represent specific domain knowledge, while in our approach constraints are specified at the *problem* level, and in the context of domain-independent PDDL planning rather than HTN planning.

Sohrabi et al. (2009) extend PDDL3 to formulate action-centric preferences over HTN tasks, and propose to handle them natively through a dedicated HTN planning system (HTNPLAN-P). PAC planning does not require the specification of a hierarchical domain theory and, as we have shown, it allows efficient compilation into classical planning without constraints. This enables any classical planner supporting conditional effects to deal with PAC problems. Moreover, PAC constraints include operators that are not considered in Sohrabi et al.'s language.

$\mathcal{LPP}$ [Baier *et al.*, 2008; Bienvenu *et al.*, 2011] is a language that harnesses the expressive power of LTL_f for specifying preferred and hard constraints over trajectories of actions as well as of states. PAC focuses on a more restricted language that, as it is the case for PDDL3 constraints versus more general LTL_f formulae [Bonassi *et al.*, 2021; Percassi and Gerevini, 2019], a planner can handle more effectively (without undergoing a potentially expensive automata-based compilation). A deeper comparison in terms of the relative expressiveness and effectiveness for planning between $\mathcal{LPP}$ and PAC is left to future work.

Finally, recent work by Bonet and Geffner (2021) introduced an interesting language based on *sketch rules* that can be used to decompose a planning problem into specific subproblems, reducing the problem width [Lipovetzky and Geffner, 2012]. This language is significantly different from action constraints, and its usefulness has been theoretically studied only in the context of width-based planning algorithms.

6 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. We have presented a new language to express a class of action-trajectory constraints, and a compilation-based approach to plan with such constraints. Our compilation scheme can be used alongside any classical planner supporting conditional effects, and it 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. Future work concerns investigating trajectory constraints over both actions and states, preferences over action constraints, and further experiments with numeric domains.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. The work was partially supported by projects H2020-EU AIPlan4EU and MUR PRIN-2020 RIPER.

References

[Baier and McIlraith, 2006] Jorge A. Baier and Sheila A. McIlraith. Planning with first-order temporally extended goals using heuristic search. In *AAAI*, pages 788–795. AAAI Press, 2006.

[Baier *et al.*, 2008] Jorge A. Baier, Christian Fritz, Meghyn Bienvenu, and Sheila A. McIlraith. Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In *AAAI*, pages 1509–1512. AAAI Press, 2008.

[Bienvenu *et al.*, 2011] Meghyn Bienvenu, Christian Fritz, and Sheila A. McIlraith. Specifying and computing preferred plans. *Artif. Intell.*, 175(7-8):1308–1345, 2011.

[Bonassi *et al.*, 2021] Luigi Bonassi, Alfonso Emilio Gerevini, Francesco Percassi, and Enrico Scala. On planning with qualitative state-trajectory constraints in PDDL3 by compiling them away. In *ICAPS*, pages 46–50. AAAI Press, 2021.

[Bonet and Geffner, 2021] Blai Bonet and Hector Geffner. General policies, representations, and planning width. In *AAAI*, pages 11764–11773. AAAI Press, 2021.

[Botea *et al.*, 2005] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-ff: Improving AI planning with automatically learned macro-operators. *J. Artif. Intell. Res.*, 24:581–621, 2005.

[Erol *et al.*, 1994] Kutluhan Erol, James A. Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *AAAI*, pages 1123–1128. AAAI Press / The MIT Press, 1994.

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

[Gerevini *et al.*, 2009] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668, 2009.

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

[Giacomo *et al.*, 2014] Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *AAAI*, pages 1027–1033. AAAI Press, 2014.

[Lipovetzky and Geffner, 2012] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 540–545. IOS Press, 2012.

[Micheli and Scala, 2019] Andrea Micheli and Enrico Scala. Temporal planning with temporal metric trajectory constraints. In *AAAI*, pages 7675–7682. AAAI Press, 2019.

[Percassi and Gerevini, 2019] Francesco Percassi and Alfonso Emilio Gerevini. On compiling away PDDL3 soft trajectory constraints without using automata. In *ICAPS*, pages 320–328. AAAI Press, 2019.

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

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

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

[Sohrabi *et al.*, 2009] Shirin Sohrabi, Jorge A. Baier, and Sheila A. McIlraith. HTN planning with preferences. In *IJCAI*, pages 1790–1797, 2009.

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