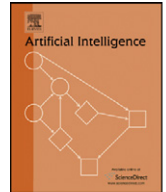
Artificial Intelligence 175 (2011) 1308–1345

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Specifying and computing preferred plans ☆

Meghyn Bienvenu [a], Christian Fritz [b], Sheila A. McIlraith [c,*]

[a] CNRS & Université Paris-Sud, France
[b] Palo Alto Research Center, USA
[c] Department of Computer Science, University of Toronto, Canada

ARTICLE INFO

Article history:
Received 7 April 2009
Received in revised form 30 July 2010
Accepted 30 July 2010
Available online 2 December 2010

Keywords:
Knowledge representation
Preferences
Planning with preferences

ABSTRACT

In this paper, we address the problem of specifying and computing preferred plans using rich, qualitative, user preferences. We propose a logical language for specifying preferences over the evolution of states and actions associated with a plan. We provide a semantics for our first-order preference language in the situation calculus, and prove that progression of our preference formulae preserves this semantics. This leads to the development of PPLAN, a bounded best-first search planner that computes preferred plans. Our preference language is amenable to integration with many existing planners, and beyond planning, can be used to support a diversity of dynamical reasoning tasks that employ preferences.
© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Research in automated planning has historically focused on classical planning – generating a sequence of actions to achieve a user-defined goal, given a specification of a domain and an initial state. However, in many real-world settings satisficing plans are plentiful, and it is the generation of *high quality plans*, meeting users' preferences and constraints, that presents the greatest challenge [50].

In this paper we examine the problem of preference-based planning – generating a plan that not only achieves a user-defined goal, but that also conforms, where possible, to a user's preferences over properties of the plan. To address the problem of preference-based planning, we require a language for specifying user preferences, as well as a means of generating plans that is capable of optimizing for the defined class of preferences. To this end, we propose $\mathcal{LPP}$, a first-order language for specifying domain-specific, qualitative user preferences. $\mathcal{LPP}$ is expressive, supporting the definition of temporally extended preferences over the evolution of actions and states associated with a plan. $\mathcal{LPP}$ harnesses much of the expressive power of first-order and linear temporal logic (LTL) [51]. We define the semantics of our first-order preference language in Reiter's version of the situation calculus [47,53]. Leveraging this semantics, we also define an extension of $\mathcal{LPP}$ that allows for the specification of preferences over the occurrence of Golog complex actions [44,53]. Golog is an Algol-inspired agent programming language that supports the construction of complex actions using programming language-like constructs over primitive and complex actions. Golog has proven of great utility in a diversity of agent programming applications.

$\mathcal{LPP}$'s situation calculus semantics enables us to reason about preferences over situations (corresponding to trajectories or partial plans) within the language, which is beneficial for a diversity of reasoning tasks where distinguishing a preferred situation or trajectory is relevant. Such tasks include but are not limited to plan understanding, diagnosis of dynamical

☆ The majority of the work presented in this paper was performed while the authors were affiliated with the University of Toronto. Revisions of the paper were carried out while the first author was at Université Paul Sabatier and Universität Bremen, and the second author at Information Sciences Institute.

* Corresponding author.
E-mail addresses: meghyn@lri.fr (M. Bienvenu), cfritz@parc.com (C. Fritz), sheila@cs.toronto.edu (S.A. McIlraith).

0004-3702/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2010.11.021

systems, and requirements modeling within software engineering. $\mathcal{LPP}$ can also be used to characterize ordered defaults and norms for default and deontic reasoning.

Despite $\mathcal{LPP}$'s roots in the situation calculus, planning with $\mathcal{LPP}$ does not require the use of deductive plan synthesis and a theorem prover. $\mathcal{LPP}$ is amenable to use by any state-of-the-art planner that can take LTL-based preferences as input. Indeed, as we will discuss later, work by Baier and McIlraith [2] provides a compilation algorithm that enables preference-based planners that do not accept LTL formulae as input to plan with $\mathcal{LPP}$-like LTL preferences. In this paper, we propose PPLAN, a bounded best-first search forward-chaining planner in the spirit of TLPlan [1] and TALPlanner [43]. PPLAN exploits *progression* to efficiently evaluate LTL preference satisfaction over partial plans. To guide search towards an optimal plan, we propose an admissible evaluation function that, in concert with A* search, results in the generation of optimal plans.

There is a significant body of research on preferences both within artificial intelligence (AI) and in related disciplines. A recent special issue of AI Magazine [36] provides a high-level overview of some of the latest AI research in this field, including research on planning with preferences [6]. In the last four years, there has been growing interest within the planning community in preference-based planning. This includes study of the specification of preferences for planning (e.g., Son and Pontelli [59,60] and Delgrande, Schaub, and Tompits [23]), and in particular an extension to the Planning Domain Definition Language (PDDL) [48] by Gerevini and Long to include preferences (PDDL3) [33]. In 2006, the biennial International Planning Competition (IPC-2006) included a track on planning with preferences specified in PDDL3. A number of preference-based planners were developed in and around this time, and subsequently (e.g., [24,25,41,61,9,10,29,3,5,35]). We discuss this related work in detail in Section 7.

This paper is organized as follows. In Section 2 we provide a brief review of the situation calculus. Then in Section 3, we introduce the syntax and semantics of our $\mathcal{LPP}$ preference language for planning, illustrating its use through a motivating example which is carried throughout the paper. With the semantics of our preference language in hand, we return to the general problem of planning with preferences. In Section 4, we define the notion of progression over $\mathcal{LPP}$ preference formulae and prove that it preserves the semantics of our preferences. We also define an admissible evaluation function, which can be used with progression and A* search to generate optimal plans. Then, in Section 5, we describe the PPLAN algorithm, a bounded best-first forward-chaining planner that plans with preferences. We prove the correctness of the PPLAN algorithm and present experimental results for a proof-of-concept implementation of the algorithm in Prolog. Finally, in Section 6 we extend $\mathcal{LPP}$ to enable definition of preferences over Golog complex actions. We correspondingly extend our notion of progression to include these new preference formulae. We conclude the paper with a discussion of related work and a summary.

2. Preliminaries

The situation calculus is a sorted, logical language with equality designed for specifying and reasoning about dynamical systems [53]. The signature of the language is specified in terms of three sorts: the set of *action* terms, $\mathcal{A}$, consists of constants or functions mapping objects and sometimes other actions into elements of type action; the set of *situation* terms consists of the constant S_0, denoting the initial state of the world, and terms of the form $do(a,s)$ where a is an action term and s is another situation term; finally *object* terms encompass everything that is neither an action nor a situation. In the situation calculus, the *state* of the world is expressed in terms of functions and relations (called *fluents*) which are relativized to a particular situation s, e.g., $F(\vec{x},s)$. In this paper, we consider only relational fluents, and we distinguish between the set $\mathcal{F}$ of fluents (e.g., $isSnowing(s)$), which are used to model dynamic properties of the world, and the set $\mathcal{R}$ of non-fluent relational formulae (e.g., $meal(spaghetti)$), which describe properties of the world that do not change over time. A situation s is a *history* of primitive actions $a \in \mathcal{A}$ performed from the initial, distinguished situation S_0. The function do maps a situation s and an action a into a new situation $do(a,s)$. The theory induces a tree of situations, rooted at S_0.

A *basic action theory* $\mathcal{D}$ in the situation calculus comprises four *domain-independent foundational axioms* and a set of *domain-dependent axioms*. The foundational axioms Σ define the situations, their branching structure and the situation predecessor relation $\sqsubset$. $s \sqsubset s'$ states that situation s precedes situation s' in the situation tree. Σ includes a second-order induction axiom. The domain-dependent axioms are strictly first-order and are of the general form described below. The reader is also directed to Appendix A for an example axiomatization of the dinner domain, which we use throughout this paper to illustrate concepts. Note that we follow the notational convention established by Reiter [53] and assume that free variables in situation calculus formulae are universally quantified from the outside, unless otherwise noted. In later sections, when discussing preferences and Golog, we also adopt the convention of referring to fluents in situation-suppressed form, e.g., $at(home)$ rather than $at(home,s)$.

- A set $\mathcal{D}_{ap}$ of *action precondition axioms* which describe the conditions under which it is possible to execute an action A in a situation s. An action precondition axiom for an action A takes the form

$$Poss\big(A(\vec{x}),s\big) \equiv \Pi_A(\vec{x},s)$$

where $\Pi_A(\vec{x},s)$ is a formula with free variables among $\vec{x}$, s which contains no situation terms other than s. For instance, here is a possible action precondition axiom for the action eat:

$$Poss\big(eat(x),s\big) \equiv meal(x) \wedge \exists y\big(at(y,s) \wedge readyToEat(x,y,s)\big)$$

- A set $\mathcal{D}_{SS}$ of *successor state axioms* which capture the effects of actions on the truth values of the fluents. A successor state axiom for a fluent F has the form

$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$$

where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among a, $\vec{x}$, s which contains no situation terms other than s. For example, a successor state axiom for the fluent *kitchenClean* might be:

$$kitchenClean(do(a, s)) \equiv a = cleanDishes \vee (kitchenClean(s) \wedge \forall y\, a \neq cook(y))$$

- A set of axioms $\mathcal{D}_{S_0}$ describing the initial situation. These will consist of formulae which only mention S_0, or mention no situation at all (e.g., for the non-fluent predicates). In this paper, we assume *complete information* about the initial situation, i.e., that for every n-ary relation F and every n-tuple of constants $\vec{c}$, we have either $\mathcal{D}_{S_0} \models F(\vec{c})$ or $\mathcal{D}_{S_0} \models \neg F(\vec{c})$.
- A set $\mathcal{D}_{una}$ of *unique names axioms* for actions. These have the forms

$$\forall\vec{x}\forall\vec{y}.A(\vec{x}) = A(\vec{y}) \rightarrow \vec{x} = \vec{y}$$

where A is an action,

$$\forall\vec{x}\forall\vec{y}\, A(\vec{x}) \neq B(\vec{y})$$

where A, B are actions such that $A \neq B$.

More details on the form of these axioms can be found in Reiter [53].

Definition 2.1 *(Planning Problem).* A *planning problem* Δ is a tuple $\langle \mathcal{D}, G \rangle$ where $\mathcal{D}$ is a basic action theory and G is a goal formula, representing properties that must hold in the final situation.

Here, a goal formula G denotes a formula that only contains one situation term, which is suppressed. We denote the instantiation of G in a situation s by $G(s)$.

In the situation calculus, planning is characterized as deductive plan synthesis [37]. Given a planning problem $\langle \mathcal{D}, G \rangle$, the task is to determine a situation s that is executable, and in which the goal holds, i.e.,

$$\mathcal{D} \models \exists s(executable(s) \wedge G(s))$$

where $executable(s) \stackrel{\text{def}}{=} \forall a \forall s'(do(a, s') \sqsubseteq s \rightarrow Poss(a, s'))$. Notice that either the goal is satisfied in the initial situation (i.e., $s = S_0$), or $s = do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$[1] in which case we have shown that executing the sequence of actions $a_1 a_2 \ldots a_n$ from S_0 enables us to reach a goal state.

We refer to this situation s as the *plan trajectory* and the (possibly empty) sequence of actions $a_1 a_2 \ldots a_n$ as the *associated plan*. The *length* of this plan is n. The set of all plans is denoted by Π, and Π^k denotes the subset of plans of length $\leqslant k$. A planning problem $\langle \mathcal{D}, G \rangle$ is *solvable* if it has at least one plan. It is *k-solvable* if it has a plan of length k or less. Note that, slightly abusing terminology, we will sometimes refer to executable sequences of actions as *partial plans*, even though not all sequences of actions can be extended into a plan.

3. Preference specification

In this section we describe the syntax and semantics of our first-order preference language. We illustrate the concepts in this paper in terms of a compelling domain we call the *Dinner Domain*. An independent contribution of this paper is the creation of this planning domain which can serve as a benchmarking domain for problems in planning with preferences. In addition to affording a number of natural and compelling temporally extended preferences, the dinner domain is easily scaled either by increasing the number of objects involved (adding more restaurants, meals, etc.) or by making the events more complex (e.g., buying groceries, cooking, etc.). A complete axiomatization of the dinner domain example used here is provided in Appendix A.

Example 3.1 *(The Dinner Domain).* It's dinner time and Claire is tired and hungry. Her goal is to be at home with her hunger sated. There are three possible ways for Claire to get food: she can cook something at home, order in take-out food, or go to a restaurant. To cook a meal, Claire needs to know how to make the meal and she must have the necessary ingredients, which might require a trip to the grocery store. She also needs a clean kitchen in which to prepare her meal. Ordering take-out is much simpler: she only has to order and eat the meal. Going to a restaurant requires getting to the restaurant, ordering, eating, and then returning home.

---

[1] We frequently abbreviate $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$ to $do([a_1, \ldots, a_n], S_0)$, or $do(\vec{a}, S_0)$.

This example is easily encoded in any number of planning systems, and given a specification of Claire's initial state, a planner could generate numerous plans that achieve Claire's goal. Nevertheless, like many of us, Claire has certain preferences concerning where and what she eats that make some plans better than others. It is the definition of these preferences and the generation of these preferred plans that is the focus of this paper.

3.1. A first-order preference language

In this section we present the syntax of a first-order language for expressing preferences about dynamical systems. Our preference language modifies and extends the preference language *PP* recently proposed by Son and Pontelli [59]. We keep their hierarchy of basic desire formulae (which we rename to trajectory property formulae), atomic preference formulae, and general preference formulae, to which we add a new class of aggregated preference formulae. Subsequent references to *preference formulae* refer to aggregated preference formulae, which encompass trajectory property formulae, atomic preference formulae, and general preference formulae. It is such a preference formula that will be given as input to a planner.

Definition 3.2 *(Trajectory Property Formula (TPF)).* A *trajectory property formula* is a sentence drawn from the smallest set $\mathcal{B}$ where:

- $\mathcal{F} \subset \mathcal{B}$,
- $\mathcal{R} \subset \mathcal{B}$,
- If $f \in \mathcal{F}$, then $\mathbf{final}(f) \in \mathcal{B}$,
- If $a \in \mathcal{A}$, then $\mathbf{occ}(a) \in \mathcal{B}$,
- If φ_1 and φ_2 are in $\mathcal{B}$, then so too are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\exists x\varphi_1$, $\forall x\varphi_1$, $\mathbf{next}(\varphi_1)$, $\mathbf{always}(\varphi_1)$, $\mathbf{eventually}(\varphi_1)$, and $\mathbf{until}(\varphi_1, \varphi_2)$.

TPFs are used to describe properties of trajectories (sequences of actions and states). The TPFs f, r, and $\mathbf{final}(f)$ are used to describe the static properties of states belonging to a trajectory, while the TPF $\mathbf{occ}(a)$ allows one to describe the types of actions that occur along a trajectory. These basic TPFs then serve as building blocks for creating more complex TPFs using the standard Boolean connectives, quantifiers, and temporal operators.

We now illustrate the kind of properties that can be expressed by TPFs by giving some sample TPFs from our motivating example. The formal semantics of TPFs will be presented in Section 3.2.

$$hasIngredients(spaghetti) \wedge knowsHowToMake(spaghetti) \tag{P1}$$

$$\exists x\big(hasIngredients(x) \wedge knowsHowToMake(x)\big) \tag{P2}$$

$$\mathbf{final}(kitchenClean) \tag{P3}$$

$$\mathbf{always}\big(at(home)\big) \tag{P4}$$

$$\exists x\ \mathbf{eventually}\big(\mathbf{occ}\big(cook(x)\big)\big) \tag{P5}$$

$$\exists x \exists y\ \mathbf{eventually}\big(\mathbf{occ}\big(orderTakeout(x, y)\big)\big) \tag{P6}$$

$$\exists x \exists y\ \mathbf{eventually}\big(\mathbf{occ}\big(orderRestaurant(x, y)\big)\big) \tag{P7}$$

$$\mathbf{always}\big(\neg\big(\exists x \exists y\ \mathbf{occ}\big(drive(x, y)\big) \wedge isSnowing\big)\big) \tag{P8}$$

$$\mathbf{always}\big(\neg\big(\exists x\big(\mathbf{occ}\big(eat(x)\big) \wedge chinese(x)\big)\big)\big) \tag{P9}$$

The first TPF (P1) states that in the initial situation Claire has the ingredients and the know-how to cook spaghetti. (P2) is more general, expressing that in the initial situation Claire has the ingredients for something she knows how to make. Observe that fluents that are not inside temporal connectives refer only to the initial situation. (P3) states that in the final situation the kitchen is clean. The TPF (P4) states that Claire remains at home throughout the trajectory. (P5)–(P7) state respectively that at some point in time Claire cooks, orders in take-out, or orders a meal in a restaurant. The TPF (P8) states that at no point does Claire drive while it is snowing. Finally (P9) tells us that Claire never eats any Chinese food.

TPFs can be used to express simple "all-or-nothing" preferences. For example, the TPF (P7) can be used to indicate a preference for going to a restaurant, and the TPF (P4) could be used to express a desire to stay at home. However, TPFs do not allow us to express more complex preferences like the fact that Claire prefers cooking to ordering takeout to going to a restaurant. Notice that preferences of the latter type can be satisfied to a certain degree: Claire is happiest when she cooks, less happy if she orders take-out, and least happy when going out to a restaurant. Moreover, Claire might find cooking only slightly better than take-out, and take-out much more appealing than going out. This suggests a need to specify preferences over alternatives in which the user can indicate the level of preference for the different alternatives. These considerations motivate the introduction of atomic preference formulae.

Definition 3.3 *(Atomic Preference Formula (APF)).* Let $\mathcal{V}$ be a totally ordered set with minimal element v_{min} and maximal element v_{max}. An *atomic preference formula* is a formula $\varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_n[v_n]$, where each φ_i is a TPF, each $v_i \in \mathcal{V}$, $v_i < v_j$ for $i < j$, and $v_0 = v_{min}$. When $n = 0$, atomic preference formulae correspond to TPFs.

Note that the requirement that $v_i < v_j$ for $i < j$, i.e., that the different values be distinct, is without loss of generality since a set of TPFs with the same value could be replaced by their disjunction.

An atomic preference formula expresses a preference over alternatives. Each of the alternatives in the APF is annotated with a value from a totally ordered set $\mathcal{V}$ which describes how far that alternative is from the ideal. The lower the value, the closer to the ideal, the more satisfied the user. In what follows, we let $\mathcal{V} = [0, 1]$ for parsimony, but we could just as easily choose a strictly qualitative set like $\{best < good < indifferent < bad < worst\}$.

Returning to our example, the following APF expresses Claire's preference over what to eat (spaghetti, followed by pizza, followed by crêpes)[2]:

$$\mathbf{occ}'\big(eat(spaghetti)\big)[0] \gg \mathbf{occ}'\big(eat(pizza)\big)[0.4] \gg \mathbf{occ}'\big(eat(crêpes)\big)[0.5] \qquad \text{(P10)}$$

From the values that Claire assigned to the various options, we can see that she has a strong preference for spaghetti but finds pizza and crêpes about equally appealing. If instead Claire is in a hurry, tired, or very hungry, she may be more concerned about how long she will have to wait for her meal, giving rise to the following preference:

$$\text{P6}[0] \gg \text{P5} \wedge \text{P4}[0.2] \gg \text{P7}[0.7] \gg \text{P5} \wedge \neg\text{P4}[0.9] \qquad \text{(P11)}$$

This preference tells us that Claire's first choice is take-out, followed by cooking if it doesn't involve going out to get groceries, followed by going to a restaurant, and lastly cooking when it requires leaving her home. We can see here that Claire really prefers options that don't involve going out.

To reiterate, an atomic preference formula represents a preference over alternatives φ_i. We wish to satisfy the TPF φ_i with the lowest index i. Consequently, if Claire eats pizza *and* crêpes, this is no better nor worse with respect to (P10) than situations in which Claire eats only pizza, and it is strictly better than situations in which she just eats crêpes. Note that there is always implicitly one last option, which is to satisfy none of the φ_i, and this option is the least preferred.

Atomic preference formulae contribute significantly to the expressivity of our preference language, but we still lack a way to combine atomic preferences together. In order to allow the user to specify more complex preferences, we introduce our third class of formulae, which extends our language with conditional, conjunctive, and disjunctive preferences.

Definition 3.4 *(General Preference Formula (GPF)).* A formula Φ is a *general preference formula* if one of the following holds:

- Φ is an atomic preference formula
- Φ is $\gamma : \Psi$, where γ is a TPF and Ψ is a general preference formula [Conditional]
- Φ is one of
  - $\Psi_1 \& \ldots \& \Psi_n$ [General And]
  - $\Psi_1 \mid \ldots \mid \Psi_n$ [General Or]

where $n \geqslant 1$ and each Ψ_i is a general preference formula.

Here are some example general preference formulae:

$$\text{P2} : \text{P5} \wedge \text{P4} \qquad \text{(P12)}$$

$$\text{P10} \,\&\, \text{P11} \qquad \text{(P13)}$$

$$\text{P10} \mid \text{P11} \qquad \text{(P14)}$$

(P12) states that if Claire initially has the ingredients for something she can make, then she prefers to stay in and cook. Preferences (P13) and (P14) show the two ways we can combine Claire's food and time preferences. (P13) maximizes the satisfaction of both Claire's food and time preferences, whereas (P14) says that she is content if either of the two were satisfied.

Our final class of formulae allows us to combine our general preferences using a number of well-known preference aggregation operators (cf., e.g., [26]).

Definition 3.5 *(Aggregated Preference Formula (AgPF)).* A formula Φ is an *aggregated preference formula* if one of the following holds:

[2] For legibility, we abbreviate **eventually**(**occ**(φ)) by **occ**$'(\varphi)$, and we refer to preference formulae by their labels.

- Φ is a general preference
- Φ is one of
  - $\mathbf{lex}(\Phi_1, \ldots, \Phi_n)$
  - $\mathbf{leximin}(\Phi_1, \ldots, \Phi_n)$
  - $\mathbf{sum}(\Phi_1, \ldots, \Phi_n)$ [if there is a sum operation associated with $\mathcal{V}$]

where $n \geqslant 1$ and each Φ_i is a general preference formula.

Note that summing elements in $\mathcal{V}$ (if such an operation exists) could possibly yield values outside of $\mathcal{V}$, e.g., if we use the standard arithmetic sum $+$ on $\mathcal{V} = [0, 1]$, we may obtain numbers greater than 1. All that we require is that the sum operation is defined for every multi-set of elements from $\mathcal{V}$, and that it outputs elements from a totally ordered set (possibly different from $\mathcal{V}$).

This concludes our description of the syntax of our preference language. Our language extends and modifies the *PP* language recently proposed by Son and Pontelli [59]. Quantifiers, variables, non-fluent relations, a conditional construct, and aggregation operators (AgPF) have been added to our language. In *PP* it is impossible to talk about arbitrary action or fluent arguments or their properties, and difficult or even impossible to express the kinds of preferences given above. *PP*'s APFs are *ordinal* rather than *qualitative* making relative differences between ordered preferences impossible to articulate. Finally, our semantics, which we present in the next section, gives a different, and we argue more natural, interpretation of *General And* and *General Or*. Relative to *quantitative* dynamical preferences, we argue that our language is more natural for a user.

3.2. The semantics of our language

We appeal to the situation calculus to define the semantics of our preference language. TPFs are interpreted as situation calculus formulae and are evaluated relative to an action theory $\mathcal{D}$. In order to define the semantics of more complex preference formulae, which can be satisfied to a certain degree, we associate a qualitative value or *weight* with a situation term, depending upon how well it satisfies a preference formula. Weights are elements of $\mathcal{V}$, with v_{min} indicating complete satisfaction and v_{max} complete dissatisfaction. The motivation for introducing values was that purely ordinal preferences (such as the atomic preference formulae in [59]) can be combined in only very limited, and not necessarily very natural, ways, in addition to leading to great incomparability between outcomes. Replacing ordinal preferences by qualitative preferences allows us to give a more nuanced representation of the user's preferences.

Since TPFs may refer to properties that hold at various situations in a situation history, we use the notation $\varphi[s, s']$ proposed by Gabaldon [32] to explicitly denote that φ holds in the sequence of situations originating in s and terminating in $s' = do([a_1, \ldots, a_n], s)$.[3] Recall that fluents are represented in situation-suppressed form and $F[s]$ denotes the re-insertion of situation term s.

Definition 3.6. We define the following set of macros, providing an interpretation of TPFs in the situation calculus [32][4]:

$$\begin{aligned}
f[s, s'] &\stackrel{\text{def}}{=} f[s], \quad \text{for all } f \in \mathcal{F} \\
r[s, s'] &\stackrel{\text{def}}{=} r, \quad \text{for all } r \in \mathcal{R} \\
\mathbf{final}(f)[s, s'] &\stackrel{\text{def}}{=} f[s'], \quad \text{for all } f \in \mathcal{F} \\
\mathbf{occ}(a)[s, s'] &\stackrel{\text{def}}{=} do(a, s) \sqsubseteq s' \\
(\varphi \wedge \psi)[s, s'] &\stackrel{\text{def}}{=} \varphi[s, s'] \wedge \psi[s, s'] \\
(\varphi \vee \psi)[s, s'] &\stackrel{\text{def}}{=} \varphi[s, s'] \vee \psi[s, s'] \\
(\neg\varphi)[s, s'] &\stackrel{\text{def}}{=} \neg(\varphi[s, s']) \\
(\exists x\varphi)[s, s'] &\stackrel{\text{def}}{=} \exists x(\varphi[s, s']) \\
(\forall x\varphi)[s, s'] &\stackrel{\text{def}}{=} \forall x(\varphi[s, s']) \\
\mathbf{eventually}(\varphi)[s, s'] &\stackrel{\text{def}}{=} (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s')\varphi[s_1, s']
\end{aligned}$$

[3] Actually, in [32], the notation $\varphi[s', s]$ is used, where s' is used for the start situation and s for the end situation. We chose to invert the roles of s and s' to keep with the situation calculus convention that s precedes s'.

[4] We use the following abbreviations:

$(\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s')\Phi \stackrel{\text{def}}{=} \exists s_1(s \sqsubseteq s_1 \wedge s_1 \sqsubseteq s' \wedge \Phi)$

$(\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s')\Phi \stackrel{\text{def}}{=} \forall s_1((s \sqsubseteq s_1 \wedge s_1 \sqsubseteq s') \rightarrow \Phi)$.

$$\mathbf{always}(\varphi)[s,s'] \stackrel{\text{def}}{=} (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s')\varphi[s_1,s']$$
$$\mathbf{next}(\varphi)[s,s'] \stackrel{\text{def}}{=} \exists a(do(a,s) \sqsubseteq s' \wedge \varphi[do(a,s),s'])$$
$$\mathbf{until}(\varphi,\psi)[s,s'] \stackrel{\text{def}}{=} (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s')(\psi[s_1,s'] \wedge (\forall s_2 : s \sqsubseteq s_2 \sqsubset s_1)\varphi[s_2,s'])$$

That is, a fluent which is not embedded inside some temporal connective holds in a sequence of situations just in the case that it holds in the first situation in the sequence. For $r \in \mathcal{R}$, we have nothing to do as r is already a situation calculus formula. A TPF **final**(f) just means that the fluent f holds in the final situation. The TPF **occ**(a) tells us that the first action executed is a. The Boolean connectives and quantifiers are already part of the situation calculus and so require no special translation. Finally, we interpret all temporal connectives in exactly the same way as in [32]. Since each TPF is shorthand for a situation calculus expression, a simple model-theoretic semantics follows.

Definition 3.7 *(Trajectory Property Satisfaction).* Let $\mathcal{D}$ be an action theory. A trajectory property formula φ is satisfied by a situation s just in the case that

$$\mathcal{D} \models \varphi[S_0, s]$$

We define $w_s(\varphi)$ to be the weight of TPF φ with respect to situation s. $w_s(\varphi) = v_{min}$ if φ is satisfied by s, otherwise $w_s(\varphi) = v_{max}$.

We can extend this definition to the more general case as follows:

Definition 3.8. Let $\mathcal{D}$ be an action theory, and let s and s' be two situations such that $s \sqsubseteq s'$. A trajectory property formula φ is satisfied by the sequence of situations between s and s' just in the case that

$$\mathcal{D} \models \varphi[s, s']$$

We define $w_{s,s'}(\varphi)$ to be the weight of TPF φ with respect to the situations s and s'. We define $w_{s,s'}(\varphi) = v_{min}$ if $\mathcal{D} \models \varphi[s,s']$, otherwise $w_{s,s'}(\varphi) = v_{max}$.

Clearly Definition 3.7 is just a special case of Definition 3.8 since w_s is simply short-hand for $w_{S_0,s}$. In most circumstances, the short-hand w_s notation of Definition 3.7 will suffice, with the advantage of being easier to read and understand. Consequently, we use it throughout the paper. Nevertheless, in proving properties of our semantics relative to progression, we will revert to the two-situation notation of Definition 3.8.

Example 3.9. We evaluate the example TPFs presented above with respect to the plan trajectory $s_1 = do([cook(crêpes), eat(crêpes), cleanDishes], S_0)$, and the initial situation S_0 in which Claire is at home with a clean kitchen and ingredients for crêpes, and she knows how to make both spaghetti and crêpes. Recall that for these examples we assume $\mathcal{V} = [0,1]$, i.e., $v_{min} = 0$ and $v_{max} = 1$. (See Appendix A for a more detailed description of S_0.)

$$w_{s_1}(\text{P1}) = 1, \quad w_{s_1}(\text{P2}) = 0, \quad w_{s_1}(\text{P3}) = 0$$
$$w_{s_1}(\text{P4}) = 0, \quad w_{s_1}(\text{P5}) = 0, \quad w_{s_1}(\text{P6}) = 1$$
$$w_{s_1}(\text{P7}) = 1, \quad w_{s_1}(\text{P8}) = 0, \quad w_{s_1}(\text{P9}) = 0$$

The weight of an atomic preference formula is simply defined to be the value associated with the first satisfied component TPF:

Definition 3.10 *(Atomic Preference Satisfaction).* Let s be a situation and $\Phi = \varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_n[v_n]$ be an atomic preference formula. Then

$$w_s(\Phi) = \begin{cases} v_i & \text{if } \mathcal{D} \models \varphi_i[S_0,s] \text{ and } \mathcal{D} \not\models \varphi_j[S_0,s] \text{ for all } 0 \leqslant j < i \\ v_{max} & \text{if no such } i \text{ exists} \end{cases}$$

Example 3.11. We evaluate the atomic preferences (P10) and (P11) with respect to the trajectory s_1 and initial situation S_0 from Example 3.9:

$$w_{s_1}(\text{P10}) = 0.5, \quad w_{s_1}(\text{P11}) = 0.2$$

For the trajectory $s_2 = do([drive(home, store), buyIngredients(spaghetti), drive(store, home), cook(spaghetti), eat(spaghetti)], S_0)$, we obtain:

$$w_{s_2}(\text{P10}) = 0, \qquad w_{s_2}(\text{P11}) = 0.9$$

For the trajectory $s_3 = do([drive(home, italianRest), orderRestaurant(spaghetti, italianRest), eat(spaghetti), drive(italianRest, home)], S_0)$, we have instead:

$$w_{s_3}(\text{P10}) = 0, \qquad w_{s_3}(\text{P11}) = 0.7$$

Finally, for the trajectory $s_4 = do([orderTakeout(pizza, pizzaPlace), eat(pizza)], S_0)$, we get:

$$w_{s_4}(\text{P10}) = 0.4, \qquad w_{s_4}(\text{P11}) = 0$$

Definition 3.12 *(General Preference Satisfaction).* Let s be a situation and Φ be a general preference formula. Then $w_s(\Phi)$ is defined as follows:

- $w_s(\varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_n[v_n])$ is defined above
- $w_s(\gamma : \Psi) = \begin{cases} v_{min} & \text{if } w_s(\gamma) = v_{max} \\ w_s(\Psi) & \text{otherwise} \end{cases}$
- $w_s(\Psi_1 \& \Psi_2 \& \ldots \& \Psi_n) = \max\{w_s(\Psi_i)\colon 1 \leqslant i \leqslant n\}$
- $w_s(\Psi_1 \mid \Psi_2 \mid \ldots \mid \Psi_n) = \min\{w_s(\Psi_i)\colon 1 \leqslant i \leqslant n\}$

Observe that the semantics of our generalized Boolean connectives extends the semantics of their Boolean counterparts: a conjunction $\Psi_1 \& \ldots \& \Psi_n$ is fully satisfied (i.e., has weight v_{min}) if all of the component preferences Ψ_i are fully satisfied; a disjunction $\Psi_1 \mid \ldots \mid \Psi_n$ is fully satisfied if at least one of the disjuncts Ψ_i is fully satisfied; and a conditional preference $\gamma : \Psi$ is fully satisfied if either the condition γ is false (i.e., has weight v_{max}) or the component preference formula Ψ is fully satisfied. Returning to our example GPFs from page 1312:

Example 3.13. We evaluate our general preference formulae with respect to the trajectories s_1, s_2, s_3, s_4 and the initial situation S_0 from above (also see $\mathcal{D}_{S_0}$ in Appendix A):

	s_1	s_2	s_3	s_4
P12 = P2 : P5 $\wedge$ P4	0	1	1	1
P13 = P10 & P11	0.5	0.9	0.7	0.4
P14 = P10 | P11	0.2	0	0	0

We conclude this section with the following definition which shows us how to compare two situations with respect to an aggregated preference formula:

Definition 3.14 *(Preferred Situations).* A situation s_1 is *at least as preferred* as a situation s_2 with respect to a preference formula Φ, written $s_1 \succcurlyeq_\Phi s_2$, if one of the following holds:

- Φ is a GPF, and $w_{s_1}(\Phi) \leqslant w_{s_2}(\Phi)$
- $\Phi = \mathbf{lex}(\Phi_1, \ldots, \Phi_n)$, and either $w_{s_1}(\Phi_i) = w_{s_2}(\Phi_i)$ for all i, or there is some i such that $w_{s_1}(\Phi_i) < w_{s_2}(\Phi_i)$ and for all $j < i$, $w_{s_1}(\Phi_j) = w_{s_2}(\Phi_j)$
- $\Phi = \mathbf{leximin}(\Phi_1, \ldots, \Phi_n)$, and either $|\{i\colon\ w_{s_1}(\Phi_i) = v\}| = |\{i\colon\ w_{s_2}(\Phi_i) = v\}|$ for all $v \in \mathcal{V}$ or there is some v such that $|\{i\colon\ w_{s_1}(\Phi_i) = v\}| > |\{i\colon\ w_{s_2}(\Phi_i) = v\}|$ and for all $v' < v$ we have $|\{i\colon\ w_{s_1}(\Phi_i) = v'\}| = |\{i\colon\ w_{s_2}(\Phi_i) = v'\}|$
- $\Phi = \mathbf{sum}(\Phi_1, \ldots, \Phi_n)$, and $\sum_{i=1}^{n} w_{s_1}(\Phi_i) \leqslant \sum_{i=1}^{n} w_{s_2}(\Phi_i)$

Strict preference ($\succ$) and equivalence ($\approx$) are defined in the standard way.

Thus, when comparing situations s_1 and s_2 with respect to the preference $\mathbf{lex}(\Phi_1, \ldots, \Phi_n)$, we simply apply the standard lexicographic ordering to the tuples of weights $(w_{s_1}(\Phi_1), \ldots, w_{s_1}(\Phi_n))$ and $(w_{s_2}(\Phi_1), \ldots, w_{s_2}(\Phi_n))$. For the preference formula $\mathbf{leximin}(\Phi_1, \ldots, \Phi_n)$, we check value by value (starting with the minimal value v_{min}) which of the situations has more elements of that value, privileging those situations which have more small (= good) values. In other words, we first sort the weights in the tuples $(w_{s_1}(\Phi_1), \ldots, w_{s_1}(\Phi_n))$ and $(w_{s_2}(\Phi_1), \ldots, w_{s_2}(\Phi_n))$ in ascending order, then apply the lexicographic ordering to the reordered tuples. Finally, if we have the preference $\mathbf{sum}(\Phi_1, \ldots, \Phi_n)$, then we simply sum up the weights in the tuples $(w_{s_1}(\Phi_1), \ldots, w_{s_1}(\Phi_n))$ and $(w_{s_2}(\Phi_1), \ldots, w_{s_2}(\Phi_n))$ and then compare the resulting values.

Example 3.15. We evaluate some aggregated preference formulae with respect to the trajectories s_1, s_2, s_3, s_4 and the initial situation S_0 from above. If Claire places greater importance on satisfying her food preference than her time preference, we might have the AgPF $\Phi_1 = \mathbf{lex}(\text{P10}, \text{P11})$, which gives the following order on situations:

$$s_3 \succ_{\Phi_1} s_2 \succ_{\Phi_1} s_4 \succ_{\Phi_1} s_1$$

Suppose now that Claire wants to satisfy her food and time preferences, but she accords equal importance to both preferences. This might be expressed by the AgPF $\Phi_2 = \textbf{leximin}(\text{P10}, \text{P11})$, which yields the following preference ordering:

$$s_4 \succ_{\Phi_2} s_3 \succ_{\Phi_1} s_2 \succ_{\Phi_1} s_1$$

Since we are using a numerical set of values, the **sum** operator could also be used to combine Claire's preferences, $\Phi_3 = \textbf{sum}(\text{P10}, \text{P11})$, allowing her to maximize the average level of satisfaction:

$$s_4 \succ_{\Phi_3} s_1 \approx_{\Phi_3} s_3 \succ_{\Phi_3} s_2$$

Notice that because our set $\mathcal{V}$ of weights is assumed to be totally ordered, for each general preference formula Φ and every pair of trajectories s_1 and s_2, we must have either $w_{s_1}(\Phi) < w_{s_2}(\Phi)$ or $w_{s_1}(\Phi) = w_{s_2}(\Phi)$ or $w_{s_1}(\Phi) > w_{s_2}(\Phi)$. It follows that either $s_1 \succcurlyeq_\Phi s_2$ or $s_2 \succcurlyeq_\Phi s_1$, i.e., $\succcurlyeq_\Phi$ defines a complete pre-order over situations. This continues to hold when we replace Φ by an aggregated preference formula.

Remark 3.16. Given any aggregated preference formula Φ, the relation $\succcurlyeq_\Phi$ defines a complete pre-order over situations.

What is interesting about our framework is that we are capable of representing ordinal, qualitative, and simple quantitative preferences. For example, if we want to avoid specifying any values at all, we can still apply the aggregation operators to a set of TPFs. If we combine them using **leximin**, we generate a pre-order that ranks plans based on the number of satisfied preferences, whereas by using the lexicographic operator (**lex**), we can classify plans according to criteria of varying importance. By annotating APFs with qualitative values, which is a focus of this paper, the user can specify the relative differences in levels of preference for different alternatives, which allows her to combine preferences in a number of natural ways. Finally, our approach works equally well in the case where the relative differences are expressed numerically, in which case the user can make use of the sum aggregation operator which allows for compensation between different preferences.

4. Planning with preferences

With a preference language in hand, we return to the problem of planning with preferences.

Definition 4.1 *(Preference-Based Planning Problem).* A *preference-based planning problem* $\Delta^{\mathcal{P}}$ is a tuple $\langle \mathcal{D}, G, \Phi \rangle$, where $\mathcal{D}$ is an action theory, G is a goal formula, and Φ is a preference formula.

Definition 4.2 *(Preferred Plan).* Consider a preference-based planning problem $\Delta^{\mathcal{P}} = \langle \mathcal{D}, G, \Phi \rangle$, with plan trajectories s_1 and s_2, and associated plans $\vec{a}_1$ and $\vec{a}_2$. We say that plan $\vec{a}_1$ is *preferred* to plan $\vec{a}_2$ iff $s_1 \succ_\Phi s_2$.

Following the work on planning with domain control knowledge (e.g., TALPlanner [43], TLPlan [1]), it is interesting to consider the generalized problem of planning with hard constraints, or domain knowledge, combined with preference formulae. This *constrained planning problem with preferences* can be easily accommodated in our framework by simply adding a TPF ϕ_c representing the control knowledge and requiring that all plans satisfy ϕ_c:

Definition 4.3 *(Constrained Planning Problem with Preferences).* A *constrained planning problem with preferences* $\Delta^{\mathcal{P}}_{\mathcal{C}}$ is a tuple $\langle \mathcal{D}, G, \phi_c, \Phi \rangle$, where ϕ_c is a TPF and $\mathcal{D}$, G, and Φ are as above. A plan for $\Delta^{\mathcal{P}}_{\mathcal{C}}$ is any plan $\vec{a}$ of $\langle \mathcal{D}, G \rangle$ such that $\mathcal{D} \models \phi_c[S_0, do(\vec{a}, S_0)]$.

As an example, a user of the dinner domain in conjunction with any of the previously exemplified preference formulae may want to rule out any plans that involve leaving the house. She can do so by using $\phi_c = \textbf{always}(at(home))$.

Definition 4.4 *(Ideal Plan).* Given a (constrained) preference-based planning problem with associated preference formula Φ, an *ideal plan* is any plan $\vec{a}$ such that $w_{do(\vec{a},S_0)}(\Phi) = v_{min}$ if Φ is a GPF, or such that $w_{do(\vec{a},S_0)}(\Phi_i) = v_{min}$ for all $1 \leqslant i \leqslant n$, if $\Phi = \textbf{lex}(\Phi_1, \ldots, \Phi_n)$, $\textbf{leximin}(\Phi_1, \ldots, \Phi_n)$, or $\textbf{sum}(\Phi_1, \ldots, \Phi_n)$.

Thus, an ideal plan is a plan which fully satisfies all of the user's preferences.

Definition 4.5 *(Optimal Plan).* Given a (constrained) preference-based planning problem with associated preference formula Φ, an *optimal plan* is any plan $\vec{a}$ such that there does not exist a plan $\vec{b}$ such that $do(\vec{b}, S_0) \succ_\Phi do(\vec{a}, S_0)$.

An optimal plan is thus a plan which best satisfies the user's preferences among all possible plans. Note that, since optimality is relative, every preference-based planning problem for which at least one plan exists has at least one optimal plan, but ideal plans do not always exist, as it may be impossible to achieve the goal while fully satisfying all of the agent's preferences.

Definition 4.6 *(k-Optimal Plan).* Given a (constrained) preference-based planning problem with preference formula Φ and a length bound k, a *k-optimal plan* is any plan $\vec{a} \in \Pi^k$ such that there does not exist another plan $\vec{b} \in \Pi^k$ such that $do(\vec{b}, S_0) \succ_\Phi do(\vec{a}, S_0)$.

Thus, the set of k-optimal plans consists of those plans that best satisfy the user's preferences among the plans composed of at most k actions.

For the purposes of this paper, we restrict our attention to planning problems with finite domains, i.e., problems whose planning domain definitions are representable in propositional logic. This restriction to finite domains is consistent with the state of the art in automated classical planning. Before elaborating on our approach to generating preference-based plans, we digress slightly to discuss issues related to the complexity of preference-based planning.

4.1. Complexity of preference-based planning

While a complete analysis of the computational complexity of preference-based planning is beyond the scope of this paper, in this subsection we provide insight into some of the key issues related to the complexity of preference-based planning.

The computational complexity of classical planning is generally examined with respect to two fundamental decision problems: (i) the plan existence problem – informally, does there exist a plan; and (ii) the bounded plan existence problem – does there exist a plan of length n or less. A key factor in determining the complexity of these decision problems is the expressiveness of the planning domain description. It is well established that classical planning with STRIPS using first-order terms is undecidable [28], as is numeric STRIPS [39]. However, when the STRIPS planning domain is propositional, plan existence is PSPACE-complete [19]. It takes severe syntactic restrictions on the planning domain to guarantee even NP-completeness, though such drastic restrictions can yield propositional STRIPS domains that are tractable. The most closely related complexity result to the problem we examine here is that of van den Briel et al. who show that determining whether a propositional partial satisfaction planning problem has a quality of at least k is PSPACE-complete [62]. However as noted in Section 7, while partial satisfaction planning problems are related to the notion of preference-based planning defined here, their notion of quality (or preference) is defined and evaluated in quite a different way.

In order to study the computational complexity of preference-based planning as defined here, we must examine the decision problems associated with the definitions we provided above. In particular, we must examine the decision problems associated with testing whether a given plan is ideal, optimal, or k-optimal. The decision problem relating to ideal plans necessitates testing whether the sequence of states induced by an action sequence satisfies each of the component TPFs in the preference formula. In contrast, the optimal and (k-)optimality problems not only involve evaluating the preference formula with respect to the given plan but also verifying that there does not exist any more preferred plan. Note that once we have decided which of the component TPFs is satisfied by a given plan, it is possible to determine all of the possible combinations of TPFs that may give rise to more preferred plans.

Example 4.7. Consider a GPF Φ of the form

$$\big(\varphi_1[0] \gg \varphi_2[0.2] \gg \varphi_3[0.7]\big) \& \big(\varphi_4[0] \gg \varphi_5[0.5] \gg \varphi_6[0.9]\big)$$

where $\varphi_1, \dots, \varphi_6$ are all TPFs. Suppose we have identified a plan trajectory s which satisfies φ_1, φ_5, and φ_6. The weight of s is 0.5, so every more preferred plan must have a weight less than 0.5. There are two ways of achieving this: either satisfy $\varphi_1 \wedge \varphi_4$ (to get a weight of 0) or satisfy $\varphi_2 \wedge \varphi_4$ (to obtain a weight of 0.2).

Thus, we see that the two key computational problems in our setting are to decide whether a plan satisfies a given TPF and to decide whether there exists a plan that satisfies a given TPF. The latter problem can be seen as a generalization of the plan existence problem.

Existing complexity results related to classical planning emphasize the expressiveness of the planning domain description as a determining factor in the complexity of plan existence. With respect to preference-based planning, since our preference language is intended to be used in combination with a diversity of action representations including but not limited to situation calculus, a proper analysis would require us to parameterize each of the above decision problems by the form of the action theory and perhaps also by the form of the preference formula. In the remainder of this subsection, we briefly discuss the complexity of the plan existence problem for TPFs for the case of propositional STRIPS action theories and propositional preference formulae, leaving an investigation of other settings to future work. We begin by noting that the complexity of the plan existence problem for temporally extended goals expressed in LTL has been studied previously by Baral et al. [7]. They show that the plan existence problem is NP-complete for propositional action theories when a

polynomial bound is placed on plan length. As their LTL goals are very similar to our TPFs, this result is trivially extended to our setting.

Of greater interest is the case where no length bound is provided (or where the length bound is succinctly encoded) since then the minimal-length plans might be exponentially long, preventing us from applying the "guess-and-check" method exploited by Baral et al. We know from the classical setting that it is PSPACE-complete to determine if an instance of a propositional STRIPS planning problem has any solutions [19]. The PSPACE lower bound clearly transfers to our setting. The proof of PSPACE membership is based upon a recursive computation of reachability in the transition system induced by the action theory. Initially, we want to test, for each final state s_f, whether s_f can be reached from the initial state s_0 in at most 2^n steps, where n is the number of propositional variables. This test can be performed by first guessing an intermediate state s_i and testing whether (i) s_i can be reached from s_0 in 2^{n-1} steps and (ii) s_f can be reached from s_i in 2^{n-1} steps. The recursion stops when we are only allowed a single step, in which case the two states must be identical or the second state must be reachable via a single action from the first state. Since the recursive calls can be processed independently, and there is a recursion depth of $\log 2^n = n$, we obtain membership in PSPACE. A crucial point is that we never materialize the transition system, nor even the path from s_0 to s_f, as both might be exponentially large.

At first glance, there appear to be two obstacles to extending the PSPACE upper bound for classical propositional planning to our setting:

- In classical planning, one only needs to consider plans in which each state is visited at most once, which is why we can assume paths of length at most 2^n. Such a restriction is not valid in our setting as some states might need to be visited multiple times in order to satisfy the TPF. This means the length of the shortest plan satisfying a TPF might be larger than the number of states.
- The recursive reachability algorithm does not allow us to keep track of the satisfaction of temporal properties.

Fortunately, both points have already been addressed by the LTL model-checking community. Indeed, the key to the PSPACE membership proof for model checking of LTL formulae, originally shown in [54], is the construction of an enhanced transition system (more precisely, a Büchi automaton) in which the states contain not only propositional atoms, but also the temporal formulae which need to be satisfied at the state. Reachability analysis can be performed on this modified transition system,[5] which is still at most exponentially large and can be constructed on-the-fly. The LTL approach cannot be directly applied to our setting, since standard LTL semantics treats infinite, rather than finite, state sequences and our TPFs contain some non-standard connectives such as **occ** and **final**. However, these differences are largely superficial, and by modifying the definition of a final state in the transition system to account for finiteness and the **final** connective and by compiling away the **occ** connective using fluents, we can obtain membership in PSPACE for TPF plan existence in the purely propositional setting. A result by Baral et al. shows that verifying that a given plan satisfies a TPF (our second key decision problem) is feasible in polynomial time. Thus, by combining these results, and applying the reduction from general preferences to TPFs suggested above, we can show PSPACE-completeness of the optimality problem (and its bounded variant).

4.2. Progression

We now return to the question of how to compute preferred plans. Again, recall that we are restricting our attention to planning problems with finite domains, in keeping with the state of the art in classical planning. In Section 5 we will present an algorithm for planning with preferences, based on forward-chaining planning. As has been done with control knowledge containing linear temporal logic formulae [1,43], we evaluate our preference formulae by progressing them as we construct our plan. Progression takes a situation and a temporal logic formula (TLF), evaluates the TLF with respect to the state of the situation and generates a new formula representing those aspects of the TLF that remain to be satisfied in subsequent situations. In this section, we define the notion of progression with respect to our preference formulae, and prove that the semantics of preference formulae is preserved through progression.

Our objective in this section is to develop a method of planning for finite domain problems and as such we define progression for preferences ranging over finite domains. This is consistent with previous definitions of progression (e.g., [1,43]). We believe that it is possible to extend the definition of progression to handle our first-order preference language, at least under some syntactic restrictions, but investigation of this issue goes beyond the scope of the present paper and is left for future work.

In order to define the progression operator, we add the propositional constants TRUE and FALSE both to the situation calculus and to our set of TPFs, where $\mathcal{D} \models$ TRUE and $\mathcal{D} \not\models$ FALSE for every action theory $\mathcal{D}$. To capture the progression of **occ**(a), we also add the TPF **occLast**(a), $a \in \mathcal{A}$, whose semantics is defined by **occLast**$(a)[s, s'] \stackrel{\text{def}}{=} \exists s''(s = do(a, s''))$.

[5] More precisely, we need to determine the emptiness of the Büchi automaton. This can be done by finding a final state s_f which is reachable from the initial state *and* is reachable from itself.

Definition 4.8 *(Progression of a Trajectory Property Formula).* Let s be a situation. The *progression* of a trajectory property formula φ through s, written $\rho_s(\varphi)$, is given by:

- For all $f \in \mathcal{F}$, $\rho_s(f) \stackrel{\text{def}}{=} \begin{cases} \text{TRUE} & \text{if } \mathcal{D} \models f[s] \\ \text{FALSE} & \text{otherwise} \end{cases}$
- For all $r \in \mathcal{R}$, $\rho_s(r) \stackrel{\text{def}}{=} \begin{cases} \text{TRUE} & \text{if } \mathcal{D} \models r \\ \text{FALSE} & \text{otherwise} \end{cases}$
- $\rho_s(\mathbf{occ}(a)) \stackrel{\text{def}}{=} \mathbf{occLast}(a)$
- $\rho_s(\mathbf{occLast}(a)) \stackrel{\text{def}}{=} \begin{cases} \text{TRUE} & \text{if } \mathcal{D} \models \exists s'(s = do(a, s')) \\ \text{FALSE} & \text{otherwise} \end{cases}$
- $\rho_s(\mathbf{final}(\psi)) \stackrel{\text{def}}{=} \mathbf{final}(\psi)$
- $\rho_s(\neg\psi) \stackrel{\text{def}}{=} \neg\rho_s(\psi)$
- $\rho_s(\psi_1 \wedge \psi_2) \stackrel{\text{def}}{=} \rho_s(\psi_1) \wedge \rho_s(\psi_2)$
- $\rho_s(\psi_1 \vee \psi_2) \stackrel{\text{def}}{=} \rho_s(\psi_1) \vee \rho_s(\psi_2)$
- $\rho_s(\exists x\psi) \stackrel{\text{def}}{=} \bigvee_{c \in \mathcal{C}} \rho_s(\psi^{c/x})$
- $\rho_s(\forall x\psi) \stackrel{\text{def}}{=} \bigwedge_{c \in \mathcal{C}} \rho_s(\psi^{c/x})$
- $\rho_s(\mathbf{next}(\psi)) \stackrel{\text{def}}{=} \psi$
- $\rho_s(\mathbf{always}(\psi)) \stackrel{\text{def}}{=} \rho_s(\psi) \wedge \mathbf{always}(\psi)$
- $\rho_s(\mathbf{eventually}(\psi)) \stackrel{\text{def}}{=} \rho_s(\psi) \vee \mathbf{eventually}(\psi)$
- $\rho_s(\mathbf{until}(\psi_1, \psi_2)) \stackrel{\text{def}}{=} (\rho_s(\psi_1) \wedge \mathbf{until}(\psi_1, \psi_2)) \vee \rho_s(\psi_2)$
- $\rho_s(\text{TRUE}) \stackrel{\text{def}}{=} \text{TRUE}$
- $\rho_s(\text{FALSE}) \stackrel{\text{def}}{=} \text{FALSE}$

where $\psi^{c/x}$ denotes the result of substituting the constant c for all instances of the variable x in φ.

Example 4.9. With S_0 defined as before, we show how to progress some example TPFs:

- $\rho_{S_0}(\mathbf{occ}(cook(crêpes))) = \mathbf{occLast}(cook(crêpes))$
- $\rho_{S_0}(\mathbf{eventually}(kitchenClean))$
  $= \rho_{S_0}(kitchenClean) \vee \mathbf{eventually}(kitchenClean)$
  $= \text{TRUE} \vee \mathbf{eventually}(kitchenClean) \equiv \text{TRUE}$
- $\rho_{S_0}(\exists x hasIngredients(x)) = \bigvee_{c \in \mathcal{C}} \rho_{S_0}(hasIngredients(c))$
  $= \rho_{S_0}(hasIngredients(crêpes)) \vee \cdots \vee \rho_{S_0}(hasIngredients(pizza))$
  $= \text{TRUE} \vee \cdots \vee \text{FALSE} \equiv \text{TRUE}$

Progression of atomic and general preference formulae is defined in a straightforward fashion by progressing the individual TPFs that comprise these more expressive formulae.

Definition 4.10 *(Progression of Atomic, General, and Aggregated Preference Formulae).* Let s be a situation, and let Φ be an atomic or general preference formula. The *progression* of Φ through s is defined by:

- $\rho_s(\varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_n[v_n]) \stackrel{\text{def}}{=} \rho_s(\varphi_0)[v_0] \gg \cdots \gg \rho_s(\varphi_n)[v_n]$
- $\rho_s(\gamma : \Psi) \stackrel{\text{def}}{=} \rho_s(\gamma) : \rho_s(\Psi)$
- $\rho_s(\Psi_1 \& \ldots \& \Psi_n) \stackrel{\text{def}}{=} \rho_s(\Psi_1) \& \ldots \& \rho_s(\Psi_n)$
- $\rho_s(\Psi_1 \mid \ldots \mid \Psi_n) \stackrel{\text{def}}{=} \rho_s(\Psi_1) \mid \ldots \mid \rho_s(\Psi_n)$
- $\rho_s(\mathbf{lex}(\Phi_1, \ldots, \Phi_n)) \stackrel{\text{def}}{=} \mathbf{lex}(\rho_s(\Phi_1), \ldots, \rho_s(\Phi_n))$
- $\rho_s(\mathbf{leximin}(\Phi_1, \ldots, \Phi_n)) \stackrel{\text{def}}{=} \mathbf{leximin}(\rho_s(\Phi_1), \ldots, \rho_s(\Phi_n))$
- $\rho_s(\mathbf{sum}(\Phi_1, \ldots, \Phi_n)) \stackrel{\text{def}}{=} \mathbf{sum}(\rho_s(\Phi_1), \ldots, \rho_s(\Phi_n))$

Note that progression can lead to a potentially exponential increase in the size of a TPF. In practice, we can (and do) greatly reduce the size of progressed formulae by the use of Boolean simplification and bounded quantification, cf. [1].

Definitions 4.8 and 4.10 show us how to progress a preference formula one step, through one situation. We extend this to the notion of iterated progression.

Definition 4.11 *(Iterated Progression).* The *iterated progression* of a preference formula Φ through situation $s = do(\vec{a}, S_0)$, written $\rho^*_s(\Phi)$, is defined by:

$$\rho^*_{S_0}(\Phi) \stackrel{\text{def}}{=} \rho_{S_0}(\Phi)$$
$$\rho^*_{do(a,s)}(\Phi) \stackrel{\text{def}}{=} \rho_{do(a,s)}\big(\rho^*_s(\Phi)\big)$$

To prove our progression theorem, we will make use of a more general form of iterated progression, which takes two situation arguments:

Definition 4.12 *(General Iterated Progression).* The *iterated progression* of a preference formula Φ starting from situation s_1 through situation s_2 (where $s_1 \sqsubseteq s_2$), written $\rho^*_{s_1,s_2}(\Phi)$, is defined as follows:

$$\rho^*_{s_1,s_1}(\Phi) \stackrel{\text{def}}{=} \rho_{s_1}(\Phi)$$
$$\rho^*_{s_1,do(a,s_3)}(\Phi) \stackrel{\text{def}}{=} \rho_{do(a,s_3)}\big(\rho^*_{s_1,s_3}(\Phi)\big)$$

Finally we prove that the progression of our preference formulae preserves their semantics, i.e., that our action theory entails a preference formula over the situation history of s if and only if it entails the progressed formula up to (but not including) s in the state associated with s. We will exploit this in proving the correctness of our algorithm in the section to follow.

Theorem 4.13 *(Correctness of Progression). Let s_1 and $s_2 = do([a_1, \ldots, a_n], s_1)$ be two situations where $n \geqslant 1$, and let φ be a TPF. Then*

$$\mathcal{D} \models \varphi[s_1, s_2] \quad \textit{iff}\ \mathcal{D} \models \rho^*_{s_1,s_3}(\varphi)[s_2, s_2]$$

where $s_2 = do(a_n, s_3)$.

Proof. Refer to Appendix B. □

In the context of planning, we will be most interested in the case where $s_1 = S_0$:

Corollary 4.14. *Let $s = do([a_1, \ldots, a_n], S_0)$ be a situation with $n \geqslant 1$, and let φ be a TPF. Then*

$$\mathcal{D} \models \varphi[S_0, s] \quad \textit{iff}\ \mathcal{D} \models \rho^*_{s'}(\varphi)[s, s]$$

where $s = do(a_n, s')$.

From Corollary 4.14, we can prove that the weight of a preference formula with respect to a situation (plan trajectory) is equal to the weight of the progressed preference formula with respect to the final situation, disregarding its history.

Corollary 4.15. *Let $s = do([a_1, \ldots, a_n], S_0)$ be a situation with $n \geqslant 1$ and let Φ be a preference formula. Then*

$$w_s(\Phi) = w_{s,s}\big(\rho^*_{s'}(\Phi)\big)$$

where $s = do(a_n, s')$.

4.3. An evaluation function for best-first search

In this section, we propose an admissible evaluation function for best-first search. To this end, we introduce the notion of *optimistic* and *pessimistic* weights of a situation relative to a GPF Φ. These weights provide a bound on the best and worst weights of any successor situation with respect to Φ. As a result, our evaluation function is non-decreasing and will never over-estimate the actual weight, thus enabling us to define an optimal search algorithm.

Optimistic (resp. pessimistic) weights are defined based on optimistic (resp. pessimistic) satisfaction of TPFs. Intuitively, optimistic satisfaction, denoted $\varphi[s, s']^{opt}$, assumes that any parts of the TPF not yet falsified will eventually be satisfied, i.e., that there is a continuation s'' of situation s' such that $\varphi[s, s'']$ is entailed by the action theory, but without doing look-ahead. Pessimistic satisfaction, denoted $\varphi[s, s']^{pess}$, assumes the opposite, namely that anything not yet satisfied will eventually be falsified. The definition of optimistic and pessimistic satisfaction largely follows the definition of (normal) satisfaction of TPFs given earlier. The key difference is in the definition of $\mathbf{next}(\varphi)$, $\mathbf{occ}(a)$, and $\mathbf{final}(\varphi)$:

$$\mathbf{final}(\varphi)\big[s,s'\big]^{opt} \stackrel{\text{def}}{=} \text{TRUE}$$
$$\mathbf{final}(\varphi)\big[s,s'\big]^{pess} \stackrel{\text{def}}{=} \text{FALSE}$$
$$\mathbf{occ}(a)\big[s,s'\big]^{opt} \stackrel{\text{def}}{=} do(a,s) \sqsubseteq s' \vee s = s'$$
$$\mathbf{occ}(a)\big[s,s'\big]^{pess} \stackrel{\text{def}}{=} do(a,s) \sqsubseteq s'$$
$$\mathbf{next}(\varphi)\big[s,s'\big]^{opt} \stackrel{\text{def}}{=} \exists a\big(do(a,s) \sqsubseteq s' \wedge \varphi\big[do(a,s),s'\big]^{opt} \vee s = s'\big)$$
$$\mathbf{next}(\varphi)\big[s,s'\big]^{pess} \stackrel{\text{def}}{=} \exists a\big(do(a,s) \sqsubseteq s' \wedge \varphi\big[do(a,s),s'\big]^{pess}\big)$$

It follows that when $s = s'$, $\mathbf{occ}(a)[s,s']^{pess} \equiv \text{FALSE}$ and $\mathbf{next}(\varphi)[s,s']^{pess} \equiv \text{FALSE}$. For later use with progression, we also define $\mathbf{occLast}(a)[s,s']^{opt/pess} \stackrel{\text{def}}{=} \mathbf{occLast}(a)[s,s']$. We define the other temporal formulae in terms of **next**.

$$\mathbf{eventually}(\varphi)\big[s,s'\big]^{opt/pess} \stackrel{\text{def}}{=} \varphi\big[s,s'\big]^{opt/pess} \vee \mathbf{next}\big(\mathbf{eventually}(\varphi)\big)\big[s,s'\big]^{opt/pess}$$
$$\mathbf{always}(\varphi)\big[s,s'\big]^{opt/pess} \stackrel{\text{def}}{=} \varphi\big[s,s'\big]^{opt/pess} \wedge \mathbf{next}\big(\mathbf{always}(\varphi)\big)\big[s,s'\big]^{opt/pess}$$
$$\mathbf{until}(\varphi,\psi)\big[s,s'\big]^{opt/pess} \stackrel{\text{def}}{=} \psi\big[s,s'\big]^{opt/pess} \vee \big(\varphi\big[s,s'\big]^{opt/pess} \wedge \mathbf{next}\big(\mathbf{until}(\varphi,\psi)\big)\big[s,s'\big]^{opt/pess}\big)$$

For the purpose of creating an admissible evaluation function for planning, we are really only interested in optimistic evaluation. The reason why we also need pessimistic evaluation is simple: the TPF $\neg\varphi$ is optimistically satisfied if and only if φ is not pessimistically satisfied. That is, it is optimistic to assume that there is a way to falsify φ which in turn will satisfy the negation. We thus define:

$$(\neg\varphi)\big[s,s'\big]^{opt} \stackrel{\text{def}}{=} \neg\big(\varphi\big[s,s'\big]^{pess}\big)$$
$$(\neg\varphi)\big[s,s'\big]^{pess} \stackrel{\text{def}}{=} \neg\big(\varphi\big[s,s'\big]^{opt}\big)$$

For all other elements of the language, the definitions are the same as for normal TPF satisfaction.

We can now define optimistic and pessimistic weights of TPFs in terms of optimistic and pessimistic TPF satisfaction:

$$w^{opt}_{s,s'}(\varphi) = \begin{cases} v_{min} & \text{if } \mathcal{D} \models \varphi[s,s']^{opt} \\ v_{max} & \text{otherwise} \end{cases}$$

and

$$w^{pess}_{s,s'}(\varphi) = \begin{cases} v_{min} & \text{if } \mathcal{D} \models \varphi[s,s']^{pess} \\ v_{max} & \text{otherwise} \end{cases}$$

For readability, we abbreviate $w^{opt}_{S_0,s}$ and $w^{pess}_{S_0,s}$ by w^{opt}_s and w^{pess}_s respectively.

For APFs and GPFs the definitions of optimistic and pessimistic weights are straightforward.

Definition 4.16 *(Optimistic/Pessimistic Atomic Preference Satisfaction).* Let s be a situation and $\Phi = \varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_n[v_n]$ be an atomic preference formula. Then

$$w^{opt/pess}_s(\Phi) = \begin{cases} v_i & \text{if } \mathcal{D} \models \varphi_i[S_0,s]^{opt/pess} \text{ and } \mathcal{D} \not\models \varphi_j[S_0,s]^{opt/pess} \text{ for all } 0 \leqslant j < i \\ v_{max} & \text{if no such } i \text{ exists.} \end{cases}$$

Definition 4.17 *(General Preference Satisfaction).* Let s be a situation and Φ be a general preference formula. Then $w^{opt}_s(\Phi)$, respectively $w^{pess}_s(\Phi)$, is defined as follows:

- $w^{opt/pess}_s(\varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_n[v_n])$ is defined above
- $w^{opt/pess}_s(\gamma : \Psi) = \begin{cases} v_{min} & \text{if } w^{pess/opt}_s(\gamma) = v_{max} \\ w^{opt/pess}_s(\Psi) & \text{otherwise} \end{cases}$
- $w^{opt/pess}_s(\Psi_1 \& \ldots \& \Psi_n) = \max\{w^{opt/pess}_s(\Psi_i){:}\ 1 \leqslant i \leqslant n\}$
- $w^{opt/pess}_s(\Psi_1 \mid \ldots \mid \Psi_n) = \min\{w^{opt/pess}_s(\Psi_i){:}\ 1 \leqslant i \leqslant n\}$

The following theorem describes some of the important properties of our optimistic and pessimistic weight functions.

Theorem 4.18. *Let s be any situation, let $[a_1,\ldots,a_n]$ be an action sequence, and for every i such that $0 \leqslant i \leqslant n$, let $s_i = do([a_1,\ldots,a_i],s)$. Let further φ be a TPF and Φ a general preference formula. Then for any $0 \leqslant i \leqslant j \leqslant k \leqslant n$:*

1. *If* $\mathcal{D} \models \varphi[s, s_i]^{pess}$, *then* $\mathcal{D} \models \varphi[s, s_j]$, *and if* $\mathcal{D} \not\models \varphi[s, s_i]^{opt}$, *then* $\mathcal{D} \not\models \varphi[s, s_j]$,
2. $w_{s_i}^{opt}(\Phi) \leqslant w_{s_j}^{opt}(\Phi)$ *and* $w_{s_j}^{opt}(\Phi) \leqslant w_{s_k}(\Phi)$,
3. $w_{s_i}^{pess}(\Phi) \geqslant w_{s_j}^{pess}(\Phi)$ *and* $w_{s_j}^{pess}(\Phi) \geqslant w_{s_k}(\Phi)$.

Proof. Refer to Appendix C. □

Intuitively, 1. states that the pessimistic satisfaction of a formula over a sequence of situations implies that any continuation of this sequence also satisfies the formula. It further states that a formula that does not hold optimistically cannot be made true by any continuation. Correspondingly, 2. states that optimistic weight is non-decreasing and bounded from above by the real weight of any future situation. Finally, 3. gives the analogue for pessimistic weights: they are non-increasing and bounded from below by the real weight of any future situation.

One immediate implication of this is the following:

Corollary 4.19. *For any two situations* s, s', *such that* $s \sqsubseteq s'$:

$$\text{If } w_s^{opt}(\Phi) = w_s^{pess}(\Phi) \text{ then } w_{s'}(\Phi) = w_s^{opt}(\Phi) = w_s^{pess}(\Phi).$$

Since these definitions are compatible with those defined for progression, we have the following corollary to Theorem 4.18:

Corollary 4.20. *Let* $s = do([a_1, \ldots, a_{n-1}], S_0)$ *and* $s' = do(a_n, s)$ *be situations where* $n \geqslant 1$, *and let* φ *be a TPF. Then* $\mathcal{D} \models \varphi[S_0, s']^{opt/pess}$ *iff* $\mathcal{D} \models \rho_s^*(\varphi)[s', s']^{opt/pess}$.

Proof. Refer to Appendix D. □

This corollary states that we can still use progression when computing optimistic and pessimistic weights. Intuitively this is because the optimistic (pessimistic) part of the evaluation is only concerned with the future whereas progression deals with the past. Since the past won't change, there is no room for optimism or pessimism.

We can now define our evaluation function f_Φ.

Definition 4.21 *(Evaluation function).* Let $s = do(\vec{a}, S_0)$ be a situation and let Φ be a general preference formula. Then $f_\Phi(s)$ is defined as follows:

$$f_\Phi(s) = \begin{cases} w_s(\Phi) & \text{if } \vec{a} \text{ is a plan} \\ w_s^{opt}(\Phi) & \text{otherwise} \end{cases}$$

From Theorem 4.18 we see that the optimistic weight is non-decreasing and never over-estimates the real weight. Thus, f_Φ is admissible and when used in best-first search, the search is optimal.

5. The PPLAN algorithm and implementation

In this section, we describe PPLAN, a bounded best-first search forward chaining planner for computing preferred plans. PPLAN is currently implemented in Prolog and has not been optimized. Rather, its Prolog implementation provides a means of experimenting with different heuristics and search techniques and a framework for reasoning with preferences in the situation calculus. As a result of this, PPLAN's subsequent integration with a Prolog interpreter for the agent programming language Golog was straightforward [58]. In what follows, we describe the PPLAN algorithm and prove properties of the system, as well as presenting experiments that illustrate the effectiveness of its heuristic to guide search. The PPLAN code and test cases are available online [12].

The PPLAN algorithm is outlined in Algorithm 1. PPLAN takes as input an initial state *init*, a goal formula *goal*, a general preference formula *pref*,[6] and a plan length bound *maxLength*. The algorithm returns two outputs: a plan and its weight with respect to *pref*. The *frontier* is a list of nodes of the form $(w_1, w_2, path, state, pref)$, where w_1, w_2 denote weights, which usually hold the optimistic and pessimistic weight, respectively, *path* is the considered partial plan, *state* is the state reached by it, and *pref* denotes the progressed preference formula. Recall that weights are values drawn from the totally ordered set $\mathcal{V}$, and may be qualitative concepts such as "excellent" or "good", just as easily as numeric values. The frontier is sorted by w_1, then by w_2, and by length (in increasing order). The frontier is initialized to the empty partial plan, its optimistic and pessimistic weights, **optW** and **pessW**, with respect to the initial situation and the preference formula *pref*. In

[6] For simplicity, we present our algorithm for general rather than aggregated preference formulae. We discuss the extension to aggregated preference formulae later in the section.

Algorithm 1: PPLAN($\mathcal{D}$, init, goal, pref, maxLength)

```
1  begin
2    frontier ← [ (optW(pref, [ ], init), pessW(pref, [ ], init), [ ], init, pref) ];
3    if D ⊨ goal[init] then
4      node1 ← (realW(pref, [ ], init), realW(pref, [ ], init), [ ], init, pref);
5      frontier ← sortNmergeByVal( [node1] , frontier);
6    while frontier ≠ ∅ do
7      node ← removeFirst(frontier);
8      if D ⊨ goal[node.state] and node.w1 = node.w2 then
9        return node.path, node.w1;
10     successors ← expand(node.path, node.state, node.pref, maxLength);
       /* expand(path, state, pref, maxLength) returns a list of new nodes to add to the frontier. Each
          node is of the form (w1, w2, path, state, pref). If path, the sequence of actions so far, has
          length equal to maxLength, expand returns the empty list ([]). Otherwise, expand determines all
          the executable actions in the given situation and returns a list which contains, for each of
          these executable actions a node
            (optW(pref′, path′, state′), pessW(pref′, path′, state′), path′, state′, pref′)
          and for each action leading to a situation that satisfies the goal, a second node
            (realW(pref′, path′, state′), realW(pref′, path′, state′), path′, state′, pref′).
          Here optW, pessW, and realWdenote functions that return the optimistic, pessimistic, and real
          weight for the given (progressed) preference formula, action sequence, and state.
                                                                                                   */
11     frontier ← sortNmergeByVal(successors, frontier);
12   return "no solution", ∞;
13 end
```

the case where the initial situation satisfies the goal, in addition, another node is added to the frontier, representing the real weight for the empty plan (Lines 3–5). On each iteration of the **while** loop, PPLAN removes the first node from the frontier and places it in *node*. If the partial plan of *node* satisfies the goal and its two weights are equal, then PPLAN returns *node*'s partial plan and weight. Otherwise, we call the function **expand** using the elements of *node* as input. If *path* has length equal to *maxLength* then no new nodes are added to the frontier. Otherwise, **expand** generates a new set of nodes of the form $(optW, pessW, path', state', pref')$, one for each action executable in *state*. For actions leading to goal states, **expand** also generates a second node of the same form but with *optW* and *pessW* replaced by the actual weight achieved by the plan. The reason that we need two nodes is that on the one hand, we need to record the actual weight associated with the plan that we have found, and on the other hand, to ensure completeness, we need to be able to reach the node's successors. The new nodes generated by **expand** are then sorted by their two weights and length and are merged with the remainder of the frontier. If we reach the empty frontier, we exit the **while** loop and return "no solution".

A naive implementation of such a planner would require computing alternative plan trajectories and then evaluating their relative weights. This is computationally explosive, requiring computation of numerous plan trajectories, caching of relevant trajectory state, and redundant evaluation of preference formula weights. Instead, we make use of Theorem 4.13 to compute weights as we construct plans, progressing the preference formula as we go. Exploiting progression enables the development of a best-first search strategy that orders search by weight, and evaluates preference formulae across shared partial plans. Progression is commonly used to evaluate domain control knowledge in forward chaining planners such as TLPlan [1] and TALPlanner [43], where progression of hard constraints prunes the search space. In contrast, we are unable to prune less preferred partial plans, because they may yield the final solution, hence the need for a best-first strategy.

Note that the length bound is necessary to prevent the algorithm from exploring long or even infinite action sequences that have optimistic weight zero but do not reach a goal state. This can, for instance, happen when a **final**(φ) TPF is used with a formula φ that cannot ever be achieved using the available actions given the initial state. Since our heuristic does not perform look-ahead – approximate or otherwise – it would not be able to detect such branches and could get stuck in an infinite loop.

The following theorem asserts both the completeness and the k-optimality of PPLAN.

Theorem 5.1 *(Correctness of PPLAN algorithm). Given as input a preference-based planning problem $\mathcal{P}$ and a length bound k, PPLAN returns a k-optimal plan, if $\mathcal{P}$ is k-solvable, and returns "no solution" otherwise.*

Proof. First, we prove that the algorithm terminates. There are two ways that PPLAN halts: either the first node on the frontier is a plan and has $w_1 = w_2$ in which case PPLAN returns this plan, or we reach the empty frontier, in which case PPLAN returns "no solution". Let us then suppose that the first condition is never met. In this case, we will stay in the while loop, expanding one node on each iteration. But since the successor nodes generated by **expand** always have length one greater than their parent, and since **expand** returns an empty list whenever a node has a partial plan of length equal to k, and there are only finitely many actions to consider in each node, we will eventually run out of nodes and reach the empty frontier. Thus, the algorithm always terminates.

Next, we prove that the output satisfies the conditions of the theorem. This is obvious for the case where $\mathcal{P}$ is not k-solvable as in this case, we will never find a plan and thus will stay in the while loop until we reach the empty frontier, finally returning "no solution".

We now treat the case where $\mathcal{P}$ is k-solvable. By definition, this means that there exists at least one plan of length less than or equal to k. As PPLAN systematically explores the search space, at some point **expand** will create a node whose partial plan satisfies the goal and will set w_1 and w_2 to the actual weight. This means that the frontier will contain a node satisfying the conditions of the **if**-statement, and hence, at some point, we will enter the **if**-statement and return a non-empty plan. It remains to be shown that the plan returned is k-optimal.

Suppose for a contradiction that we return a plan p with weight w which is not k-optimal. This means that there exists a plan p' of length less than or equal to k which has a weight $w' < w$. There are two possibilities: either (1) we have generated a node corresponding to p' and placed it behind p on the frontier, which is a contradiction as the frontier is sorted in non-decreasing order by node.w_1, which for nodes corresponding to plans is equal to their real weights, i.e., w and w' in this case, or (2) there is an ancestor node of p' which is behind p in the frontier. But this is not possible either, as according to Theorem 4.18, any ancestor of p' must have an optimistic weight less than or equal to $w' < w$ (and hence would again be before p on the frontier). We have thus shown that if $\mathcal{P}$ is k-solvable, the returned plan is k-optimal, concluding the proof. □

Note that our algorithm is straightforwardly modified to handle aggregated preference formulae. It suffices to associate a *tuple* of optimistic and pessimistic weights with each node, in order to keep track of the optimistic and pessimistic weights of each of the component GPFs. We then sort the frontier by comparing the tuples according to Definition 3.14. So for instance, given an AgPF $\Psi = \mathbf{lex}(\Phi_1, \Phi_2, \Phi_3)$, our frontier would contain nodes of the form $((w_1^1, w_1^2, w_1^3), (w_2^1, w_w^2, w_2^3)$, path, state, pref), where w_1^i and w_2^i are the optimistic and pessimistic weights associated with the GPF Φ_i (for $i = 1, 2, 3$). When sorting the frontier, we would place a node whose first component is $(0, 1, 1)$ before a node with first component $(1, 0, 0)$, since $(0, 1, 1)$ precedes $(1, 0, 0)$ in the lexicographic ordering. It is easy to show that Theorem 5.1 continues to hold for the modified algorithm which takes aggregated preference formulae as input.

5.1. Experiments

As noted previously, PPLAN was implemented in Prolog as a testbed for planning with rich, temporally extended preferences and was not optimized to support large-scale experimental evaluation. As such, its performance is not competitive with recent state-of-the-art preference-based planners such as those that competed in IPC-2006. We discuss these planners and their relationship to PPLAN in Section 7.

We were interested in evaluating whether the combination of progression to evaluate LTL satisfaction and our proposed admissible heuristic provided an approach that could help guide a planner toward finding an optimal plan. From the outset, we had two concerns. The first was that while progression and blind search had proven effective in planners like TLPlan, the strength of progression had been rooted in its ability to prune states that did not comply with LTL domain-control knowledge, thus vastly reducing the search space. With LTL *preferences* no comparable pruning could be done and as such the merit of progression was in question. Further, our objective was ambitious – to generate an *optimal* plan – and as such we were using an admissible evaluation function; however it is widely accepted that admissible heuristics often don't provide sufficient guidance relative to inadmissible heuristics.

To assess the behaviour of our planner, we ran 60 instances of our dinner domain.[7] Each instance was run with PPLAN, PPLAN^C (PPLAN augmented with domain control knowledge – see below), and with depth-first search (DFS) and breadth-first search (BFS) algorithms. In order to facilitate comparison, DFS and BFS were passed the k-optimal weight as a parameter and run until they found a plan with this weight (or ran out of memory). All algorithms were implemented in Prolog using the same code base, to the extent possible. Each run was compared with respect to the length of the returned plan and the number of nodes expanded. Results are reported in Fig. 1 with instances numbered in order of increasing PPLAN running time. Fig. 2 plots test cases against their running time.

The 60 individual instances differed with respect to the initial state, the goal, the size and nature of the GPF, and the length of the optimal plan. In most experiments the agent is initially at home and has at least the goal of being sated. The initial state varies with respect to ingredients that are available at home, or things the agent knows how to cook. Preferences reflect the type of food the agent would like to eat, and how and where the agent obtains her meal. Most GPFs contained multiple TPFs and APFs, but the domain did not warrant LTL nesting. Most GPFs contained one or more **eventually**(**occc**()) formulae. GPFs further differed with respect to whether formulae were ground or quantified, contained conditionals, etc. Most optimal plans were of length 6 or less. The preferences expressible as GPFs are too diverse in form and complexity to draw conclusions regarding any correspondence between the size of a GPF formula and the scalability of the planner. Much depends on the specifics of the problem instance [12].

---

[7] We also ran an early version of PPLAN on the simple school travel example presented in [59], but we were unable to get comparative statistics in order to compare the two approaches.

	PPLAN		PPLAN^C		BFS		DFS			PPLAN		PPLAN^C		BFS		DFS	
Test #	l	NE	l	NE	l	NE	l	NE	Test #	l	NE	l	NE	l	NE	l	NE
1	2	7	2	7	2	61	6	481	31	3	15	3	15	3	426	7	395
2	3	55	3	10	3	426	7	395	32	2	8	2	3	2	51	6	406
3	2	7	2	7	2	51	6	406	33	3	68	3	23	3	408	3	385
4	2	8	2	8	2	61	6	481	34	3	15	3	15	3	426	7	395
5	2	9	2	9	2	71	7	510	35	3	408	3	119	3	408	7	389
6	3	52	3	22	3	432	7	414	36	5	13	6	15	2	60	7	432
7	3	55	3	10	3	426	7	395	37	5	29	6	11	4	1975	7	1113
8	2	61	2	22	2	61	7	395	38	5	22	5	22	2	61	6	481
9	2	51	2	21	2	51	6	406	39	4	29	4	19	4	1975	6	11767
10	4	29	4	19	4	1975	7	1113	40	4	29	4	19	4	1975	6	11767
11	2	2	2	2	2	51	6	406	41	5	49	5	29	∗	∗	7	15049
12	2	3	2	3	2	61	∗	∗	42	5	23	5	23	2	71	7	510
13	6	171	6	43	∗	∗	∗	∗	43	2	597	2	169	2	51	2	402
14	2	3	2	3	2	61	∗	∗	44	4	27	5	31	∗	∗	∗	∗
15	3	54	3	10	3	495	7	461	45	3	55	3	10	3	432	7	414
16	3	51	3	21	3	408	7	389	46	4	60	4	8	4	2537	7	1526
17	3	64	3	12	3	421	7	377	47	4	59	4	7	4	2088	7	1015
18	2	8	2	8	2	61	6	481	48	5	28	6	11	4	2088	7	1015
19	2	10	2	10	2	82	7	590	49	3	65	3	12	3	432	7	414
20	3	55	3	10	3	426	7	395	50	7	115	5	36	∗	∗	7	2540
21	6	108	6	54	4	2479	7	1688	51	2	7	2	7	2	51	6	406
22	2	7	2	7	2	51	6	406	52	7	57	7	37	4	2417	7	1617
23	2	8	2	3	2	51	6	406	53	2	702	2	163	2	61	2	477
24	3	585	3	151	3	408	3	385	54	3	55	3	10	2	61	7	395
25	2	9	2	3	2	61	6	481	55	3	597	3	169	3	505	3	493
26	3	55	3	10	3	426	7	395	56	4	37	4	22	4	2479	7	1688
27	3	15	3	15	2	71	7	377	57	7	257	6	19137	∗	∗	∗	∗
28	3	15	3	15	3	408	7	389	58	6	1254	6	85	4	2599	7	1597
29	3	16	3	16	3	495	7	461	59	6	51753	6	8157	∗	∗	∗	∗
30	3	15	3	15	2	61	7	395	60	3	16878	3	340	3	432	∗	∗

Fig. 1. Plan length (l) and nodes expanded (NE) by PPLAN, PPLAN augmented by hard constraints (PPLAN^C), breadth-first search (BFS), and depth-first search (DFS). The symbol ∗ indicates out of memory (1 GB limit).

Overall, the results were quite positive. In 55 of the 60 test cases PPLAN expanded fewer nodes than BFS and DFS, generally by a significant margin and often even an order of magnitude.

The four cases where BFS and DFS outperformed PPLAN are all cases where there was a short k-optimal but non-ideal plan. In these cases, PPLAN quickly found the plan, but had to continue the search in order to ensure that no other better plan existed. The poor performance of PPLAN relative to BFS and DFS in these cases is a result of the experimental setup which gave BFS and DFS an unfair advantage by supplying them with the k-optimal weight. If PPLAN had received the same input, it would not have resulted in a larger number of expanded nodes. In two cases, PPLAN expanded a comparatively large number of nodes ($> 10{,}000$). This speaks to the difficulty of the task. A good way to cope with this problem is to add domain-dependent control knowledge to reduce the search space, as was done in TLPlan. In order to test out this idea, we reran PPLAN on the test suite, this time pruning all nodes whose partial plans contained two consecutive *drive* actions or those containing *orderTakeout*, *orderRestaurant*, or *cook* actions not immediately followed by an *eat* action. As the results show, adding these simple pieces of control knowledge allowed PPLAN to find a plan while expanding far fewer nodes in the process. Taken all together, we feel that these results speak to the effectiveness of our evaluation function in guiding the search but also to the interest of combining this approach with domain-dependent control knowledge, or some other means of pruning the search space. Regarding the running times plotted in Fig. 2, there is a rough correspondence between the numbers of nodes expanded by PPLAN and an instance's running time. Interestingly, while DFS generally expands many more nodes than PPLAN, it is still comparatively fast. This, however, is once again a consequence of knowing the k-optimal value. If this value had to be found incrementally, long search horizons would need to be explored exhaustively first, before the optimal plan/value would be found.

It is interesting to note test cases 24, 43, 53, 55, and 60 where PPLAN demonstrates poorer performance than BFS and DFS. Recall that PPLAN's best-first search explores plans based on weight *then* length. As a consequence, PPLAN can be led astray, investigating a long plan with low weight, whereas the best plan can end up being a shorter plan with higher weight. However, this behaviour appears to occur infrequently, and the heuristic generally leads to significantly improved performance.

Also note that BFS sometimes finds shorter plans than PPLAN (see cases 31, 33, 34, 36, 37, 39, 48, 49, 52, and 54). PPLAN uses plan length only as a third sorting criterion and hence length is only considered when the first two weights are equal. However, when a goal is found, the second weight of the newly created tuple for this plan is set to the real weight, which is often lower than the pessimistic weight. Therefore, when a plan is returned there may still be other plans with the same weight which are shorter. The comparison with DFS shows that this is, however, not the reason for PPLAN's performance improvement over BFS. We conclude that the implemented heuristics provide valuable search guidance.

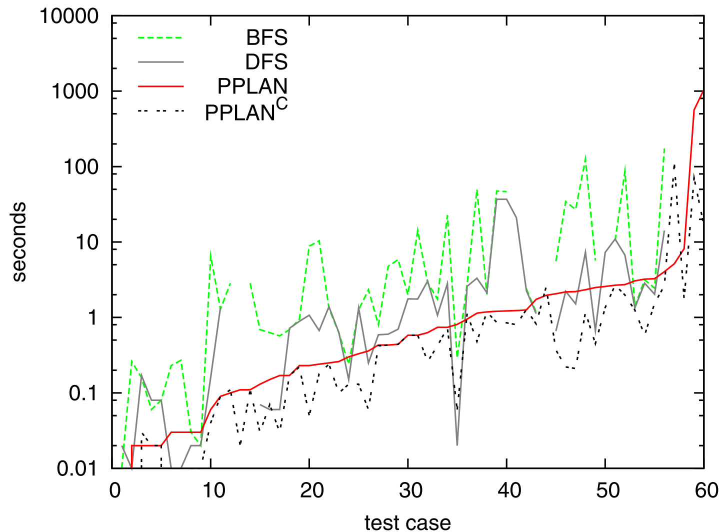


Fig. 2. Running-time of PPLAN, PPLAN augmented by hard constraints (PPLAN^C), breadth-first search (BFS), and depth-first search (DFS). Missing values indicate out of memory (1 GB limit).

5.2. Beyond PPLAN

Our original experimental results with PPLAN were encouraging, but we realized from our experience with IPC planners that much more could be done. In particular many state-of-the-art planners use some form of heuristic search, based on a relaxed plan graph (RPG) that provides a good estimate of the distance from the current state to a state in which a goal is reached (e.g., [40]). Of course, when the goal (or in this case, preference) is an LTL formula, it is more difficult to form such an estimate.

Indeed in analyzing earlier PPLAN results, it was clear that our optimistic evaluation function was lacking in two ways. First, in certain cases it could not distinguish between partial plans that made progress towards satisfying preferences and those that did not. Second, and more importantly, our evaluation function provided no estimate of the number of actions required to satisfy TPFs such as $\mathbf{eventually}(\varphi)$ nor did it have a way of determining actions to select that would make progress towards satisfaction of preferences. To illustrate the former point, consider how PPLAN evaluates the following GPF, Φ, taken from [5]:

$$\big[\mathbf{eventually}(\varphi_1) \wedge \mathbf{eventually}(\varphi_2)\big][v_1] \gg \mathbf{always}(\varphi_3)[v_2],$$

where φ_1 might be **occ**(*clean*(*kitchen*)), φ_2 might be **occ**(*eat*(*pizza*)) and φ_3 might be *at*(*home*). Our PPLAN optimistic evaluation assumes (optimistically) that each of the component predicates in the TPF can become true as new actions are added, until proven false. As such, the TPF $\mathbf{eventually}(\varphi_1) \wedge \mathbf{eventually}(\varphi_2)$ will be true whether or not either of φ_1 or φ_2 have actually been satisfied because $\mathbf{eventually}(\varphi_i)$ can never be falsified. There is always hope that φ_i will be achieved in a subsequent state of the plan. Thus, there is no distinction between a partial plan in which one or both of φ_1 or φ_2 have been achieved and one in which they have not, and as such no measure of progress towards satisfaction of the TPF. This lack of ability to distinguish progress towards satisfaction of a TPF is dependent on the form of the TPF. In contrast, the TPF $\mathbf{always}(\varphi_3)$ is falsifiable as soon as φ_3 is false in some state. To evaluate the APF/GPF, we assign a weight equal to the smallest weight TPF that is optimistically satisfied. Since the TPF $\mathbf{eventually}(\varphi_1) \wedge \mathbf{eventually}(\varphi_2)$ is always optimistically satisfied, our example Φ is always evaluated to weight v_1.

From these observations of the shortcomings of the PPLAN evaluation function, Baier and McIlraith explored whether look-ahead heuristics could be developed that would address these deficiencies. The results of this examination were reported in [5]. First $\mathcal{LPP}$ GPF preferences were decomposed into smaller constituent pieces and then translated, following [2], into parameterized non-deterministic finite-state automata whose accepting conditions corresponded to satisfaction of the component preference formulae. For each new planning instance, the planning domain description was augmented with a description of the automata representing the preference formulae – the state of each automaton, and what governed transitions between automaton states. From here, a set of inadmissible and admissible heuristics were proposed that were used together to guide search towards satisfaction of the goal and preferences. These heuristics exploited the RPG over the automata-enhanced domain and thus were able to measure progress towards satisfaction of LTL formulae. Since the use of inadmissible heuristics caused the system to lose the guarantee that the first plan returned was optimal, an incremental approach was used to search for a plan, and it was shown that if the algorithm terminated, the plan was optimal. Leveraging the insights from PPLAN of the power of pruning, this work developed a sound pruning strategy that allowed inferior partial plans to be identified and pruned from the search space, thus reducing the search space significantly.

Experimental results comparing this work with the PPLAN algorithm presented above showed that in some instances there was an order of magnitude reduction in the number of nodes expanded before an optimal plan was found. This is attributed in great part to the heuristics' ability to guide search going forward, which in turn is the result of compiling the LTL formulae into a form that can exploit a look-ahead heuristic such as an RPG heuristic. The optimistic evaluation function itself operates well in many situations, and indeed was exploited in an IPC-2006 planner to great gain, as discussed in Section 8.

6. Specifying preferences over complex actions

In previous sections we defined $\mathcal{LPP}$, a language for specifying rich, temporally extended preferences, and proposed an algorithm for computing optimal preference-based plans. In this section, we return to this language, proposing an extension with complex actions.

One of the notable features of $\mathcal{LPP}$ is the **occ**(a) statement, which allows for the specification of preferences for particular actions. We extend $\mathcal{LPP}$ with two additional constructs in order to allow for the expression of preferences concerning the occurrence of *complex actions* – actions that capture the orchestration of multiple primitive or other complex actions using well-known programming constructs. *Dine in restaurant* might be such a complex action, comprising actions that take the agent from her current location to the restaurant, where she orders and then eats a meal, and finally returns the agent back to her original location.

In many practical circumstances, people think in terms of complex actions when they describe preferred ways of achieving a given goal. It follows that allowing users to describe preferences directly in terms of these complex actions, instead of requiring them to reformulate their preferences in terms of atomic actions, may help simplify the preference elicitation process. Furthermore, the provision of such – procedural – complex actions can be used to effectively guide the search towards the goal, as complex actions often take the form of under-constrained prototypical plans.

For specifying and reasoning about complex actions, we use the Golog language and semantics [44]. Golog is a programming language defined in the situation calculus which allows a user to specify programs whose set of legal executions specifies a sub-tree of the tree of situations of a basic action theory. Golog has an Algol-inspired syntax extended with flexible non-deterministic constructs, which are later transformed into specific sequences of actions by a planner. This integration of planning and programming has proved useful in a variety of diverse applications including museum tour-guide robots [18], Web service composition [49], and soccer playing robots [30].

6.1. Golog

The set of Golog programs (without procedures) is inductively defined using the following constructs, where all appearing δ's are again Golog programs (without procedures) and the φ's are pseudo-fluent expressions. These represent situation calculus formulae with all situation terms suppressed. The expression $\varphi[s]$ denotes the instantiation of φ with all occurring fluents relativized to situation s.

a	primitive action
$\varphi?$	test condition φ
$(\delta_1; \delta_2)$	sequence
if φ **then** δ_1 **else** δ_2	conditional
while φ **do** δ'	loops
$(\delta_1 \mid \delta_2)$	non-deterministic choice
$\pi v.\delta$	non-deterministic choice of argument
δ^*	non-deterministic iteration

In addition, we introduce the term **any** to denote any action. In order to avoid ambiguity, in what follows we will call programs built from these constructs $\mathcal{LPP}$ *-programs*.

As an example, the following program may describe a preferred way of going out to a restaurant:

$$\begin{aligned} \pi r.\big(&\mathit{dineInRest}(r)?;\ \textbf{if}\ \mathit{close}(\mathit{home}, r) \\ &\textbf{then}\ \big(\mathit{walk}(\mathit{home}, r);\ \pi y.\mathit{orderRestaurant}(r, y);\ \mathit{eat}(y);\ \mathit{walk}(r, \mathit{home})\big) \\ &\textbf{else}\ \big(\mathit{drive}(\mathit{home}, r);\ \pi y.\mathit{orderRestaurant}(r, y);\ \mathit{eat}(y);\ \mathit{drive}(r, \mathit{home})\big)\big) \end{aligned} \tag{G1}$$

The program begins by picking a dine-in restaurant. Then, if the chosen place is close to home, it prescribes to walk there, order, eat, and return home. Otherwise, walking is replaced by driving.

The next example shows how non-determinism can be used to "achieve" a sub-goal, here *hasIngredients*(m). Using the **any*** construct, the program leaves it up to the planner to find a sequence of actions that will satisfy the subsequent condition. The following program describes a sensible procedure for fixing a meal at home.

$$\pi m.\big(\big(\mathit{meal}(m) \wedge \mathit{knowsHowToMake}(m)\big)?;\ \textbf{any}^*;\ \mathit{hasIngredients}(m)?;\ \mathit{cook}(m);\ \mathit{eat}(m)\big) \tag{G2}$$

According to this program, one starts by selecting a meal that one knows how to make and ensuring that one has all the necessary ingredients, which could involve a trip to the grocery store, and then the meal can be prepared and finally eaten.

Originally, the semantics of Golog programs was defined via recursive macro expansion of programs into formulae of the situation calculus. Such an "evaluation semantics" requires one to evaluate the entire program at once, which makes it difficult to use in the context of planning. This is the reason that in this paper we adopt an alternative semantics for Golog programs, the so-called "transition semantics", which was introduced in [21], where it was shown to be equivalent to the evaluation semantics. The transition semantics is defined in terms of possible transitions between program–situation pairs, called *configurations*. Roughly speaking, a configuration δ, s can lead to a configuration δ', s' (written $Trans(\delta, s, \delta', s')$) if by executing a single step of the program δ in situation s, we reach the situation s' where the remaining program is δ'. A second predicate *Final* is used to characterize the conditions under which a program has executed completely. The transition semantics is well-suited to our purposes as it permits a step-by-step evaluation of programs.

Formally, the two aforementioned predicates, *Trans* and *Final*, are defined using the following axioms [21], where $a[s]$ denotes action a with all of its arguments evaluated in s, $\varphi[s]$ denotes the truth value of formula φ in s, and δ_x^v denotes the substitution of all occurrences of v in δ by x:

$$\begin{aligned}
Trans(a, s, \delta', s') &\equiv Poss(a, s) \wedge \delta' = nil \wedge s' = do(a[s], s) \\
Trans(\mathbf{any}, s, \delta', s') &\equiv \exists a(Poss(a, s) \wedge \delta' = nil \wedge s' = do(a[s], s)) \\
Trans(\varphi?, s, \delta', s) &\equiv \varphi[s] \wedge \delta' = nil \\
Trans(\delta_1; \delta_2, s, \delta', s') &\equiv (Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')) \\
&\quad \vee \exists \gamma((\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s'))) \\
Trans(\mathbf{if}\ \varphi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2, s, \delta', s') &\equiv (\varphi[s] \wedge Trans(\delta_1, s, \delta', s')) \vee (\neg\varphi[s] \wedge Trans(\delta_2, s, \delta', s')) \\
Trans(\mathbf{while}\ \varphi\ \mathbf{do}\ \delta_1, s, \delta', s') &\equiv \varphi[s] \wedge \exists \gamma (Trans(\delta_1, s, \gamma, s') \wedge \delta' = (\gamma; \mathbf{while}\ \varphi\ \mathbf{do}\ \delta_1)) \\
Trans(\delta_1 | \delta_2, s, \delta', s') &\equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\
Trans(\pi v.\delta, s, \delta', s') &\equiv \exists x Trans(\delta_x^v, s, \delta', s') \\
Trans(\delta^*, s, \delta', s') &\equiv \exists \gamma (Trans(\delta, s, \gamma, s') \wedge \delta' = \gamma; \delta^*) \\
Final(nil, s)& \\
Final(\delta_1; \delta_2, s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \\
Final(\mathbf{if}\ \varphi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2, s) &\equiv (\varphi[s] \wedge Final(\delta_1, s)) \vee (\neg\varphi[s] \wedge Final(\delta_2, s)) \\
Final(\mathbf{while}\ \varphi\ \mathbf{do}\ \delta', s) &\equiv \neg\varphi[s] \vee Final(\delta', s) \\
Final(\delta_1 | \delta_2, s) &\equiv Final(\delta_1, s) \vee Final(\delta_2, s) \\
Final(\pi v.\delta, s) &\equiv \exists x Final(\delta_x^v, s) \\
Final(\delta^*, s)&
\end{aligned}$$

Trans and *Final* enable us to reason about the satisfaction of procedural constraints, similar to the satisfaction of the temporal constraints expressed as trajectory property formulae described earlier. By using the transitive closure of *Trans*, denoted *Trans**, we can define a new predicate *Do*, which allows one to express the fact that a program δ can terminate in situation s' when executed in situation s:

$$Do(\delta, s, s') \stackrel{\text{def}}{=} \exists \delta' (Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'))$$

We say that s, s' *satisfy* δ, or, alternatively, that s' describes a complete execution of δ in s. Since in this paper we are only concerned with finite situation terms, the above is equivalent to saying that there is a sequence of configurations $(\delta_1, s_1), \ldots, (\delta_n, s_n)$ such that: $\delta_1 = \delta$, $s_1 = s$, $s_n = s'$, and for all $1 \leqslant i < n$, $\mathcal{D} \models Trans(\delta_i, s_i, \delta_{i+1}, s_{i+1})$ and $\mathcal{D} \models Final(\delta_n, s_n)$.

Given that both our preference language and Golog define their semantics over situation trajectories in the situation calculus, they can be seamlessly integrated with one another. This enables the specification of preferences over the occurrence of complex actions, defined as the complete execution of the $\mathcal{LPP}$-programs describing these actions.

6.2. Preferred programs

We are now ready to define preferences over the occurrence of complex actions. To this end, we extend our preference language $\mathcal{LPP}$ by augmenting the set of trajectory property formulae as follows.

Definition 6.1 *(Extended Trajectory Property Formula (eTPF)).* An extended trajectory property formula is a sentence drawn from the smallest set $\mathcal{B}'$ where:

1. $\mathcal{F} \subset \mathcal{B}'$.
2. $\mathcal{R} \subset \mathcal{B}'$.
3. $f \in \mathcal{F}$, then $\mathbf{final}(f) \in \mathcal{B}'$.
4. If $a \in \mathcal{A}$, then $\mathbf{occ}(a) \in \mathcal{B}'$.
5. If φ_1 and φ_2 are in $\mathcal{B}'$, then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\exists x \varphi_1$, $\forall x \varphi_1$, $\mathbf{next}(\varphi_1)$, $\mathbf{always}(\varphi_1)$, $\mathbf{eventually}(\varphi_1)$, and $\mathbf{until}(\varphi_1, \varphi_2)$.
6. If δ is an $\mathcal{LPP}$-program and $\varphi \in \mathcal{B}'$, then $\mathbf{occC}(\delta) \in \mathcal{B}'$ and $\mathbf{afterC}(\delta, \varphi) \in \mathcal{B}'$.

Intuitively, the eTPF $\mathbf{occC}(\delta)$ states that the program δ is executed starting from the current state. To express that a complex action δ is executed at some point during a plan, we can use the **eventually** construct: $\mathbf{eventually}(\mathbf{occC}(\delta))$. The eTPF $\mathbf{afterC}(\delta, \varphi)$ stipulates that if the program δ is executed now, then φ holds in the situation where δ terminates. Note that φ may be a temporal formula. For instance, $\mathbf{afterC}(cook(x), \mathbf{eventually}(\mathbf{occ}(eat(x))))$ describes the set of trajectories where if a dish is cooked, it is eventually eaten afterwards. Using **always**, it is also possible to state that φ holds whenever δ completes executing (i.e., no matter when execution starts): $\mathbf{always}(\mathbf{afterC}(\delta, \varphi))$.

Extending the semantics of TPFs to eTPFs is straightforward given the above definition of program satisfaction:

$$\mathbf{occC}(\delta)[s, s'] \stackrel{\text{def}}{=} (\exists s_1\colon s \sqsubseteq s_1 \sqsubseteq s') Do(\delta, s, s_1)$$

$$\mathbf{afterC}(\delta, \varphi)[s, s'] \stackrel{\text{def}}{=} (\forall s_1\colon s \sqsubseteq s_1 \sqsubseteq s')\big(Do(\delta, s, s_1) \rightarrow \varphi[s_1, s']\big)$$

Note that temporally extended properties can coexist with the occurrence of complex actions, in which case these properties are also applicable to the actions forming part of the complex action. For instance, the eTPF $\mathbf{always}(\neg cold) \wedge \mathbf{eventually}(\mathbf{occC}(\text{G1}))$, where (G1) denotes the program specified above, states that eventually the program is executed, while at no time before, during, or after execution, the agent feels cold.

Using **afterC**, we can stipulate, for instance, that if cooking something at home using the above described procedure (G2), the dishes will eventually be cleaned afterwards:

$$\mathbf{afterC}\big(\text{G2}, \mathbf{eventually}\big(\mathbf{occ}(cleanDishes)\big)\big)$$

The remainder of the hierarchy of preference formulae stays the same, meaning that the change of semantics at the TPF level does not require any changes at the higher levels, APFs, GPFs, and AgPFs. To refer to such preference formulae when eTPFs are used instead of TPFs, we will use the terms eAPFs, eGPFs, and eAgPFs.

6.3. Progressing programs

The transition semantics presented in Section 6.1 is a progression of programs in disguise. To make this point clearer, we provide a formal definition of program progression. In our definition, we make reference to the set

$$\Delta'\big(\delta, do(a, s)\big) = \big\{\delta' \mid \mathcal{D} \models Trans^*\big(\delta, s, \delta', do(a, s)\big)\big\}$$

which consists of all possible remaining programs δ', after having performed a (sequence of) program transition(s) whose only primitive action is the given action a, starting in the given situation s. The reason that there may be a sequence of (program) transitions, rather than just a single transition, is that tests (ψ?) do not change the situation term. Note that in most cases, the set $\Delta'(\delta, s)$ will either be empty (if there is no possible transition using a) or only contain a single element (when there is a unique possible transition). One rather pathological example in which the set contains more than one element is: $\Delta'((a|(a; b)), do(a, S_0)) = \{nil, b\}$.

Definition 6.2 *(Progression of an $\mathcal{LPP}$-program).* Let s be a situation, and let δ be an $\mathcal{LPP}$-program. The progression of $\mathbf{occC}(\delta)$ and $\mathbf{afterC}(\delta, \psi)$ through s is given by:

- If $\varphi = \mathbf{occC}(\delta)$, then $\rho_s(\varphi) = \begin{cases} \text{TRUE}, & \text{if } \mathcal{D} \models Final(\delta, s); \\ \mathbf{occC}_{\text{trans}}(\delta), & \text{otherwise.} \end{cases}$
- If $\varphi = \mathbf{occC}_{\text{trans}}(\delta)$, then $\rho_s(\varphi) = \bigvee_{\delta' \in \Delta'(\delta, s)} \rho_s(\mathbf{occC}(\delta'))$.
- If $\varphi = \mathbf{afterC}(\delta, \psi)$, then $\rho_s(\varphi) = \begin{cases} \rho_s(\psi), & \text{if } \mathcal{D} \models Final(\delta, s); \\ \mathbf{afterC}_{\text{trans}}(\delta, \psi), & \text{otherwise.} \end{cases}$
- If $\varphi = \mathbf{afterC}_{\text{trans}}(\delta, \psi)$, then $\rho_s(\varphi) = \bigwedge_{\delta' \in \Delta'(\delta, s)} \rho_s(\mathbf{afterC}(\delta', \psi))$.

Recall that a disjunction over an empty set is false, whereas a conjunction over an empty set is true. Hence, when the situation term does not describe an execution of the program δ, then $\mathbf{occC}(\delta)$ will fail, whereas $\mathbf{afterC}(\delta, \psi)$ will trivially hold.

In these definitions we make use of auxiliary constructs, similar to **occLast**, called $\mathbf{occC}_{\text{trans}}$ and $\mathbf{afterC}_{\text{trans}}$. These, just like **occLast**, are required to do one-step bookkeeping: since the statement $\mathbf{occC}(a; b)$ states that the sequence of actions

$a; b$ is executed in the current situation, and hence regards the future, we need to keep track of this property and evaluate it (partially) in the next step. Thus, e.g., $\mathbf{occC}_{\text{trans}}(a; b)$ requires that the last action of the situation term describes a transition or sequence of transitions of the program, whose only primitive action is a.

In both $\mathbf{occC}_{\text{trans}}$ and $\mathbf{afterC}_{\text{trans}}$ we refer to a set of possible such (partial) program executions. This is mainly for technical reasons. In general, given a sequence of primitive actions, there is only one possible sequence of configurations of the program whose contained sequence of situations corresponds to this action sequence. However, in rather pathological cases, there may be several possible such configuration sequences, and hence, several possible remaining programs. For instance, this is the case for the program $(a|(a; b))$ and the single-action sequence a (cf. above). Here, either the empty program or the program b remains. Ambiguities of this kind are undesirable in programs, which should be rewritten accordingly – e.g., to $(a; (nil|b))$ in our example.

6.4. Planning with preferred programs

In order for our planning algorithm to be able to accept preference formulae involving complex actions, we need to define the optimistic/pessimistic satisfaction of extended trajectory property formulae by providing optimistic and pessimistic interpretations of the new constructs.

For the **occC** construct, we can optimistically assume that any incompletely executed program will eventually be completed.

$$\mathbf{occC}(\delta)\big[s, s'\big]^{opt} \stackrel{\text{def}}{=} \big(\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s'\big)\big(\exists \delta'\big(Trans^*\big(\delta, s, \delta', s_1\big) \wedge \big(Final\big(\delta', s_1\big) \vee s_1 = s'\big)\big)\big)$$

On the other hand, if we are pessimistic, then we would assume that incompletely executed programs will never be completed. Hence, pessimistic evaluation coincides with the original semantics of completed program execution.

$$\mathbf{occC}(\delta)\big[s, s'\big]^{pess} \stackrel{\text{def}}{=} \big(\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s'\big)Do(\delta, s, s_1)$$

Regarding post-conditions of programs (**afterC**), the optimistic assumption would be that either the program will not execute until completion, or, if it already has, to evaluate the condition optimistically.

$$\mathbf{afterC}(\delta, \psi)\big[s, s'\big]^{opt} \stackrel{\text{def}}{=} \big(\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s'\big)\big(Do(\delta, s, s_1) \rightarrow \psi\big[s_1, s'\big]^{opt}\big)$$

The pessimistic assumption would be that there will be a completed execution after which the condition will not hold. Hence, in order for the condition to be pessimistically satisfied, all possible executions of the program must already terminate within the given situation interval.

$$\begin{aligned}\mathbf{afterC}(\delta, \psi)\big[s, s'\big]^{pess} \stackrel{\text{def}}{=} & \big(\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s'\big)\big(Do(\delta, s, s_1) \rightarrow \psi\big[s_1, s'\big]^{pess}\big) \\ & \wedge \nexists \delta', s''\big(Trans^*\big(\delta, s, \delta', s''\big) \wedge s' \sqsubset s''\big)\end{aligned}$$

Theorem 6.3. *Theorem* 4.18 *continues to hold when φ is an eTPF, and Φ is an eGPF.*

Proof. Refer to Appendix E. □

Given this theorem, the algorithm described in Section 5 can be readily used to compute preferred plans also for the case where preferences refer to complex actions, i.e., when preferences are expressed as an extended general preference formula.[8]

7. Related work

There is growing interest in the issue of how to represent and reason with preferences in AI. In what follows we situate the work presented in this paper with respect to related work on the representation of preferences, with particular attention to those designed to represent preferences for planning. We also discuss the literature on the generation of preferred plans, again situating our contributions within the context of this work.

7.1. Preference languages

The literature on preference languages is extensive, much of it originating from the field of economics rather than AI. In comparing our work to other AI preference languages, many of the distinctions we raise relate to whether preference formalisms are ordinal, qualitative or quantitative; whether they model temporal preferences or solely static preferences;

[8] The algorithm can also be easily extended to treat aggregated preference formulae by associating tuples of weights with situations as we explained in Section 5.1.

whether the formalism is propositional or first order; and whether it induces a complete pre-order on the possible outcomes and if not the degree of incomparability in the ordering. In this context, our language is qualitative, models temporal preferences, is first order, and induces a complete pre-order.

As further criteria for comparison, Coste-Marquis et al. [20] evaluate some propositional logic-based preference languages with respect to expressiveness and succinctness. The issue of expressiveness is concerned with the nature of the pre-orders that can be encoded (e.g., all pre-orders, all complete pre-orders). As noted above, $\mathcal{LPP}$, by design, can encode any complete pre-order as an atomic preference formula and thus our language can represent all complete pre-orders. We cannot however express partial (or incomplete) pre-orders. We argue that for preference-based planning the restriction to complete pre-orders is reasonable and even desirable. First, we feel that this restriction is not too great since many preference frameworks adopt this same restriction. For example, in decision theory, the completeness of an agent's preferences is taken as an axiom [63]. Second, and more importantly, we feel that any disadvantages stemming from the restriction to complete pre-orders are more than compensated for by the ease of comparison of different plans. Indeed, in some other formalisms where partial orders can be encoded, the problem of deciding whether an outcome is preferred to another has been shown to be NP-complete (e.g. [13]).

Succinctness evaluates the relative space efficiency of languages, i.e., how succinctly a preference relation (an ordering) can be expressed in a language. In [20], this is done by showing that all orderings expressible in one language can be translated into another language with an at most polynomially large increase in size. With respect to $\mathcal{LPP}$, we can demonstrate using the following mapping that the preference language R_{pen} [20], which ranks outcomes by the sum of the penalties of unsatisfied preferences, can be polynomially translated into an equivalent AgPF in our language:

$$\{\langle \alpha_i, G_i \rangle, i = 1, \ldots, n\} \quad \rightsquigarrow \quad \mathbf{sum}(G_1[0] \gg \neg G_1[\alpha_i], \ldots, G_n[0] \gg \neg G_n[\alpha_i])$$

This shows that $\mathcal{LPP}$ is at least as succinct as R_{pen}. Moreover, since there exists polynomial translations of the languages R^{bo}_{prio}, R^{lexi}_{prio}, and R^{Z}_{cond} into R_{pen} [20], it follows that $\mathcal{LPP}$ is at least as succinct as these languages as well. Overall, these are rather positive results: $\mathcal{LPP}$ is expressive, being able to generate all complete pre-orders, and is at least as compact as four of the five preference languages in [20] with the same expressivity.[9] We point out however that for domain-dependent approaches, like our own, comparisons based on domain-independent criteria are less relevant, as the real test is how well the language can represent the types of preferences for which it was designed.

CP-Nets

A widely adopted language for studying user preferences in AI is the propositional CP-nets formalism [13]. CP-nets enable the description of conditional *ceteris paribus* statements about user preferences (e.g., the user prefers red wine if meat is being served and white wine if fish is being served, *all other things being equal*). User preferences are represented in a graphical notation that is compact and that reflects the conditional independence and dependence of statements. Unlike $\mathcal{LPP}$, CP-nets is restricted to static, ordinal statements about preferences. As such, CP-nets cannot express temporal preferences, nor can it express relative importance of different preferences. The CP-nets formalism is simple and elegant, however it achieves this at the expense of expressiveness. There is often a high degree of incomparability between different states because of the assumption of ceteris paribus. In [64], Wilson extends CP-Nets with stronger statements that enable the statement of preferences *irrespective of the value of other variables*. Use of such preference statements supports determining a complete pre-order on outcomes, which comes closer to the approach proposed in $\mathcal{LPP}$, but is still static and ordinal.

QCL, RKBs, and possibilistic logic

Other noteworthy work includes that of Brewka on qualitative choice logic (QCL) [17]. This preference framework is designed to represent preferences over alternatives and induces a complete pre-order over models. QCL was not developed specifically for planning and provides a subset of the expressive power of our preference language. In [16], Brewka proposes an ordinal preference language which expresses complex preferences over models in terms of ranked knowledge bases (RKBs). RKBs were originally proposed for default reasoning. In 2006, Feldmann, Brewka, and Wenzel applied this work to planning, proposing two extensions to PDDL that support the definition of preferences using RKBs [29]. In both these extensions, the preferences are on the final state of a plan. There are no temporally extended preferences. In related earlier work, Brekwa uses a variant of the QCL language to perform preference-based planning via answer set optimization in a language called $\mathcal{PLD}$ [15]. The basic elements of $\mathcal{PLD}$ are rules that code context-dependent preferences over answer sets. More complex preference formulae are formed using different aggregation operators: sum, (ranked) set inclusion, (ranked) cardinality, pareto, and lexicographic order. Finally, the possibilistic logic approach to preferences [8] is notable in that, like $\mathcal{LPP}$, it proposes a qualitative preference framework, thus allowing the relative importance of preferences to be specified.

The approaches discussed so far do not consider temporal preferences, and hence are unable to express the types of preferences that interest us. In what follows, we review some preference languages that have been designed for the task of preference-based planning or related tasks.

[9] We do not know the relationship between $\mathcal{LPP}$ and the fifth language R_H (which is based on the Hamming distance between models), but most likely there is no polynomial translation from R_H to $\mathcal{LPP}$ since R_H involves weighted sums, and $\mathcal{LPP}$ is not designed to handle any arithmetic operations beyond possibly simple sums.

Planning-oriented preference languages

In [22,23] Delgrande, Schaub, and Tompits developed a useful framework for expressing preferences for causal reasoning and planning. To this end, they proposed a general query language for histories – sequences of interleaved states and actions. This language is not unlike our TPFs. Building on this, they define a second language that supports the expression of preferences as a binary relation on histories. From these two base languages, they explored so-called *choice* and *temporal* preferences in further detail, and also extended their language with different aggregate features. One of the argued benefits of their base framework is its ability to encode other preference languages, and indeed, aside from the obvious distinction that our language is first order whereas theirs is propositional, many of the notions and constructs of our language can be nicely expressed within this framework. In particular, our TPFs, limited to finite domains, can be encoded in their query language. However our APFs cannot be encoded in their second preference language because of our capacity to denote the relative strength of preferences. Their framework has the capacity to characterize a diversity of aggregation techniques, some similar to ours.

More generally, the framework proposed in these papers has some fundamental differences to $\mathcal{LPP}$ that, in our view, underline the merits of the situation calculus. The histories employed in this framework are finite sequences of alternating states and actions. In contrast, the situation calculus foundational axioms induce a tree of situations and, when conjoined with a domain-specific action theory, characterize the space of all possible situations that follow from the domain axiomatization. As such preferences over situations can be entailed from a domain axiomatization rather than from the comparison of two specific, finite histories, as in this work.

$\mathcal{PP}$

Most noteworthy of the related work is that of Son and Pontelli [59,60] who developed a propositional language, $\mathcal{PP}$, for planning with preferences together with an implementation using answer-set programming (ASP). The original $\mathcal{PP}$ language, described in [59], served as a starting point for the development of our language and we adopted their idea of defining the language in terms of a hierarchy of formulae, and also adopted some of their nomenclature – BDF (we use TPF), APF, and GPF – and augmenting it with AgPF.

Despite the similarity in names, there are significant differences between our preference languages, both in terms of syntax and semantics. In particular, our language is first order, which affords us far more compact and simple expression of preferences. It also enables the expression of preferences over unnamed objects, which is important for online planning where groundings may not be known a priori. Planning with Web services is a good example, where the execution of the plan can provide further knowledge of objects that a planner has preferences over (e.g., specific flights or hotels in the case of Web travel planning). Furthermore, our language is *qualitative* rather than simply *ordinal*, allowing us to express, for example, that one TPF (respectively, BDF) is strongly preferred over another, as opposed to just providing an preference ordering over properties.

At the GPF level, our language includes conditional preferences, which are useful (cf. CP-nets). Like $\mathcal{PP}$, our language has the notion of *General And* (*Conjunction*) and *General Or* (*Disjunction*), but we provide a different semantics for these constructs. According to $\mathcal{PP}$'s semantics, in order for a trajectory t_1 to be preferred to a trajectory t_2 with respect to a General And preference, the trajectory t_1 must be strictly preferred to t_2 for each of the component preferences. For General Or, they require that t_1 be at least as preferred as t_2 on all component preferences and strictly preferred to t_2 for at least one component preference. We did not feel that these were natural ways of interpreting conjunction and disjunction. For example, one would expect that fully satisfying one of the component preferences should ensure satisfaction of a disjunction, but this does not follow from the $\mathcal{PP}$ semantics. In contrast our semantics is more in keeping with the Boolean connectives that give these constructs their names. Moreover, our semantics induces a complete pre-order, whereas the semantics of $\mathcal{PP}$'s general preferences leads to great incomparability between plans. Finally, at the AgPF level, we provide several further methods for aggregating preferences, which those using or reviewing our work have found to be compelling and useful, though our claim of usefulness has not been verified by a usability study. Son and Pontelli have implemented a planner using answer-set programming that can be used with a variety of black-box ASP solvers. In their later paper, they also overview how to encode their preferences to exploit answer-set optimization engines.

PDDL3

Also of interest is a comparison of $\mathcal{LPP}$ to PDDL3 [34]. Following the description in [36], PDDL3 was developed by Gerevini and Long as an extension of the Planning Domain Definition Language, PDDL [48], that provides a rich language for defining hard constraints and user preferences for planning. PDDL3 was designed for the 5th International Planning Competition (IPC-2006), which was the first international competition to include tracks for preference-based planning.

There are a number of commonalities between PDDL3 and $\mathcal{LPP}$ but ultimately some fundamental differences. In particular, PDDL3 is a quantitative (rather than qualitative) preference language. Plans are evaluated with respect to the maximization (or minimization) of a numeric objective function that is composed of a weighted linear sum of the satisfaction or violation of individual preferences. Like $\mathcal{LPP}$, individual preferences in PDDL3 are described as properties of plan trajectories that are either satisfied or violated by a plan. However, preference formulae can be quantified in such a way that a count can be taken of the number of individuals that violate a preference. This is a useful extension that could also be integrated into $\mathcal{LPP}$. For example, if you have a preference that all ingredients in the meal you are making be fresh, then a plan that uses 5 fresh ingredients and 2 frozen is more desirable than a plan that uses 1 fresh and 6 frozen, even though

neither fully satisfies the preference. This example of course also uncovers a problem that can occur with such counting. If two meals were being compared with different numbers of ingredients, then to achieve the intended interpretation of the counting, the count would need to be normalized by the total number of ingredients in each of the two different meals – something PDDL3 cannot do. Preferences over plan trajectories in PDDL3 can be temporally extended, though unlike $\mathcal{LPP}$, PDDL3 does not allow the arbitrary nesting of LTL formulae. It also does not allow for the expression of preferences over action occurrences, an important feature of $\mathcal{LPP}$. However, PDDL3 has a number of features that $\mathcal{LPP}$ does not. In particular, preferences can be related to specific times (e.g., I would like to eat dinner between 8 PM and 9 PM). There are also precondition preferences, which are state preferences that are desirable to hold in the state in which an action is being executed. One could use these to specify a *soft* precondition, and record the number of times an action is executed without this preference being satisfied.

Recently PDDL3 was extended to include preferences over decompositions of hierarchical task network (HTN) tasks [56]. This work and its application to web service composition further motivates the conceptually related extension of $\mathcal{LPP}$ with complex actions.

Other

Finally, there has been a variety of work that uses quantitative preferences for planning or temporal reasoning. This includes Eiter et al.'s work on answer set planning with respect to plan length and numeric action costs [27], work by Rossi and colleagues on reasoning with temporal soft constraints [65], Haddawy and Hanks' early work using a decision-theoretic utility function to guide planning [38], and of course the extensive research on decision-theoretic planning and MDPs [52]. The quantitative nature of these frameworks makes preference elicitation difficult. This is why in our own work we decided to focus on qualitative preferences, which are more expressive than ordinal preferences yet much easier to elicit than quantitative preferences. As a useful middle ground, Fritz and McIlraith integrate qualitative and quantitative preferences within an agent programming framework. The authors express their qualitative preferences in a restricted version of $\mathcal{LPP}$ [31].

7.2. Preference-based planners

The previous subsection noted some efforts to generate preferred plans, related to the specific languages described above. Here we provide a broad overview of some other noteworthy work in the development of preference-based planners. Detailed descriptions of many of these planners is provided in a survey article on preference-based planning by Baier and McIlraith [6]. The interested reader is directed there for further detail.

Work on decision-theoretic planning notwithstanding, one of the first pieces of work on generating preferred plans was that of Myers and colleagues at SRI on advisable planners. Myers and Lee [50] proposed a means of generating preferred plans via biases that guided a planner towards plans with certain attributes. This was followed, in and around 2004, by work on the related problem of partial satisfaction planning (PSP), also called over-subscription planning (e.g., [62,55]). Kambhampati and colleagues have developed a number of PSP planner including *Sapa*$^{\mathcal{PS}}$ [62], *AltAlt*$^{\mathcal{PS}}$ [62], *Yochan*$^{\mathcal{PS}}$ [9], and BBOP-LP [10]. In these PSP planners, the planning problem is cast as the task of finding a plan of maximal benefit, given an association of utility to facts, and costs to actions. The planners differ in how they search for such solutions alternatively using forward-chaining, backward-chaining, and incremental branch-and-bound with linear programming.

As observed in Section 1, in 2006 the biennial International Planning Competition included a track on planning with preferences specified in PDDL3. This resulted in the development of several highly optimized preference-based planners. The planners were differentiated with respect to the complexity of the preferences they could handle, starting with final-state preferences, adding temporally extended preferences, and finally extended to include more complex metric preferences. Most planners used some form of heuristic search in order to compute preferred plans. The best comparators to PPLAN are the planners that could plan with temporally extended preferences. HPLAN-P by Baier et al. [4] is one such planner. In HPLAN-P, temporally extended preferences are compiled to final-state preferences by representing them as parameterized nondeterministic finite state automata. Planning is performed via branch and bound search, incrementally generating plans of increasing quality. HPLAN-P uses a portfolio of admissible and inadmissible heuristics to guide search, together with an admissible heuristic to soundly prune partial plans that were of poorer quality than the plan previously computed. By pruning the search space, HPLAN-P is able, in some cases, to search the space exhaustively and thus guarantee optimality.

Also of note are the two planners by Edelkamp and colleagues: MIPS-BDD [24] and MIPS-XXL [25]. The former is an optimal planner that applies bidirectional breadth-first search, encoding states as binary decision diagrams. The latter is a heuristic planner based on enforced hill climbing. Both compile temporally extended preferences to grounded Büchi automata so they can be treated as final-state preferences.

Finally, SGPlan_5 [41] is also a search-based planner that can plan with temporally extended preferences. Unlike the planners described above, SGPlan_5 searches for a plan by using constraint partitioning, decomposing the original planning problem into several sub-problems. This technique stems from treating the preference-based planning problem as a standard optimization problem, where the objective function is to minimize the makespan of the plan.

We would be remiss not to mention two other related efforts to build preference-based planners – one using a constraint satisfaction problem (CSP) solver, and one using a satisfiability (SAT) solver. Like PPLAN, both of these planners are k-optimal. Similarly, neither of these planners can compete with the above state-of-the-art competition-optimized planners.

However, in contrast to PPLAN, neither of these planners can plan with temporally extended preferences. In 2005, Brafman and Chernyavsky developed PrefPlan, a preference-based planner using a CSP solver [14]. Preferences were specified over possible goal states using TCP-nets. A TCP-net is a tradeoff-enhanced CP-net, which allows the user to express priorities between variables. The most notable limitation of TCP-nets relative to $\mathcal{LPP}$ is that they cannot express temporal preferences, and suffer incomparability of states, just as CP-nets do. Their approach to planning is to compile the problem into an equivalent CSP problem, imposing variable instantiation constraints on the CSP solver, according to the TCP-net. This is a promising method for planning, though it is not clear how it will extend to temporal preferences.

satplan(P) [35] by Giunchiglia and Maratea is an extension of the award-winning Satplan planner [42] that is able to plan with final-state preferences by calling an external SAT solver. The approach is similar to PrefPlan in the sense that a variable ordering is imposed on propositional variables corresponding to final-state preferences in such a way that most-preferred plans will be explored first by the SAT solver. Preferences in satplan(P) can be defined either in a qualitative or a quantitative language. In the qualitative language, the preference ordering of plans is induced from a partial order between properties of the final state. In the quantitative language, on the other hand, each preference on the final state has an associated weight.

8. Closing remarks

In this paper we addressed the problem of preference-based planning. To this end, we proposed $\mathcal{LPP}$, an expressive first-order language for specifying domain-specific, qualitative user preferences. $\mathcal{LPP}$ supports the expression of temporally extended preferences over states as well as over actions; the actions can be either primitive or complex, in the form of Golog complex actions. In contrast to many ordinal or qualitative preference formalisms that yield significant incomparability, $\mathcal{LPP}$ provides a complete ordering over plans, which is computationally advantageous for preference-based planning. $\mathcal{LPP}$ also supports specification of the relative strength of a user's preferences. This is acknowledged by a number of practitioners to be a desirable property of preference languages for real-world applications. The semantics of $\mathcal{LPP}$ is described in the situation calculus. In the situation calculus, each executable situation corresponds to a (possible, partial) plan, and since all executable situations are described within one model of the domain theory, it means that preference for one situation over another can be expressed *within* the language. The situation calculus semantics facilitated the extension of $\mathcal{LPP}$ to Golog complex actions.

The $\mathcal{LPP}$ language, as proposed in [11], has already begun to garner interest from researchers. For example, Fritz and McIlraith combined $\mathcal{LPP}$ with quantitative preferences represented through utility functions within an agent programming language [31]. The resulting program then searched for the quantitatively optimal plan within the space of qualitatively best plans. $\mathcal{LPP}$ has also been exploited in diverse applications including the specification of user preferences for the customization of web service composition where it was integrated with a Golog interpreter by Sohrabi et al. [58], and the specification of goals for (software) requirements engineering by Liaskos et al. (e.g., [45,46]). Giunchiglia and Maratea discuss the use of $\mathcal{LPP}$ in order to extend their work on preference-based planning in satplan(P) with temporally extended preferences [35], and Sohrabi and McIlraith integrated $\mathcal{LPP}$ into an HTN planner [57]. The extension of $\mathcal{LPP}$ to include preferences over complex actions was not included in any of the above works, but is relevant and would enhance each of these applications.

In addition to $\mathcal{LPP}$, we proposed an approach to computing preferred plans via bounded best-first search in a forward chaining planner. Key components of our approach were the exploitation of progression to efficiently evaluate levels of preference satisfaction with respect to partial plans, and development of an admissible evaluation function that guarantees the optimality of best-first search. We have implemented our planner, PPLAN, and evaluated it experimentally. PPLAN was written in Prolog and was not intended to be a state-of-the-art preference-based planner, nor does it perform as one in terms of planning time. Nevertheless, experimental evaluation demonstrated that our admissible evaluation function was informative, generally expanding far fewer nodes than breadth first search. Also, in contrast to state-of-the-art IPC planners, PPLAN always returns an optimal plan.

While PPLAN itself is not a highly optimized planner, aspects of the PPLAN approach are already starting to have some impact. In particular, the heuristic of our evaluation function (as reported in [11]) has been exploited by HPlan-P, the state-of-the-art preference-based planner that received distinguished mention at IPC-2006. HPlan-P used our heuristic as one of a portfolio of different heuristics applied to different IPC domains. No one heuristic strategy worked best for all domains, but our heuristic was the best in one of six domains, and was used successfully in combination with other inadmissible heuristics in other domains [4]. Perhaps more importantly, in cases where HPlan-P used a more informative inadmissible heuristic to guide search, our (admissible) heuristic was used to soundly prune inferior partial plans as part of a branch and bound search strategy. This enabled HPlan-P to significantly reduce the search space and thus, in some cases, to search exhaustively for a plan that was provably optimal.

The work presented in this paper provides a formal foundation for specifying and generating preference-based plans. Although the work has its basis in the situation calculus, the language and the approach to planning are amenable to integration with several existing planners, and beyond planning can be used to support a diversity of reasoning tasks that employ preferences.

Acknowledgements

We gratefully acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Discovery grant program and its Undergraduate Student Research Award (USRA) program. We would like to thank the anonymous referees for their detailed and thorough comments, and Jérôme Lang for helpful comments on an earlier paper describing some of this work. Finally, we gratefully acknowledge Shirin Sohrabi Araghi for her work on the implementation of PPlan.

Appendix A. Axiomatization of the dinner example

Here we provide a formal axiomatization of the dinner domain as a basic action theory $\mathcal{D}$ of the situation calculus. A STRIPS-style specification of the domain can be found at [12]. Axioms marked with $^{(\dagger)}$ are presented for exposition purposes only, but were not used in the experiments. Recall that a basic action theory comprises four sets of domain-dependent axioms: action precondition axioms, $\mathcal{D}_{ap}$, successor state axioms, $\mathcal{D}_{SS}$, axioms describing the initial situation S_0, $\mathcal{D}_{S_0}$, and a set of unique name axioms for actions, $\mathcal{D}_{una}$. The latter is straightforward to define, and is not shown here.

Action precondition axioms ($\mathcal{D}_{ap}$). These take the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ where $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$. Following the notational convention established by Reiter [53], all free variables in situation calculus axioms are assumed to be universally quantified from the outside, unless otherwise noted.

$$
\begin{aligned}
Poss\big(drive(x,y),s\big) &\equiv location(x) \wedge location(y) \wedge x \neq y \wedge at(x,s) \\
{}^{(\dagger)}Poss\big(walk(x,y),s\big) &\equiv location(x) \wedge location(y) \wedge x \neq y \wedge at(x,s) \\
Poss\big(cook(x),s\big) &\equiv meal(x) \wedge knowsHowToMake(x) \wedge at(home,s) \\
&\quad \wedge hasIngredients(x,s) \wedge kitchenClean(s) \\
Poss\big(eat(x),s\big) &\equiv meal(x) \wedge \big(\exists y\big(at(y,s) \wedge readyToEat(x,y,s)\big)\big) \\
Poss\big(buyIngredients(x),s\big) &\equiv meal(x) \wedge \neg hasIngredients(x) \wedge at(store,s) \\
Poss\big(orderTakeout(x,y),s\big) &\equiv meal(x) \wedge takeOutRest(y) \wedge onMenu(x,y) \wedge at(home,s) \\
Poss\big(orderRestaurant(x,y),s\big) &\equiv meal(x) \wedge dineInRest(y) \wedge onMenu(x,y) \wedge at(y,s) \\
Poss(cleanDishes,s) &\equiv at(home,s)
\end{aligned}
$$

Effect axioms, which can be translated into successor state axioms ($\mathcal{D}_{SS}$) using Reiter's solution to the frame problem [53, pp. 30–32]. We provide effect axioms rather than successor state axioms, as they tend to be easier to understand for human readers. These take either the positive form $\gamma_F^+(a, \vec{x}, s) \rightarrow F(\vec{x}, do(a, s))$, or the negative form $\gamma_F^-(a, \vec{x}, s) \rightarrow \neg F(\vec{x}, do(a, s))$, where γ^+ and γ^- state the conditions under which action a makes fluent F true, respectively false, when executed from the situation s.

$$
\begin{aligned}
& a = drive(x,y) \rightarrow at\big(y, do(a,s)\big) \\
& a = drive(x,y) \rightarrow \neg at\big(x, do(a,s)\big) \\
& {}^{(\dagger)}a = walk(x,y) \rightarrow at\big(y, do(a,s)\big) \\
& {}^{(\dagger)}a = walk(x,y) \rightarrow \neg at\big(x, do(a,s)\big) \\
& {}^{(\dagger)}isSnowing(s) \wedge a = walk(x,y) \rightarrow cold\big(do(a,s)\big) \\
& a = cook(x) \rightarrow readyToEat\big(x, home, do(a,s)\big) \\
& a = cook(x) \rightarrow \neg hasIngredients\big(x, do(a,s)\big) \\
& a = cook(x) \rightarrow \neg kitchenClean\big(do(a,s)\big) \\
& a = eat(x) \rightarrow sated\big(do(a,s)\big) \\
& at(y,s) \wedge a = eat(x) \rightarrow \neg readyToEat\big(x, y, do(a,s)\big) \\
& a = buyIngredients(x) \rightarrow hasIngredients\big(x, do(a,s)\big) \\
& a = orderTakeout(x,y) \rightarrow readyToEat\big(x, home, do(a,s)\big) \\
& a = orderRestaurant(x,y) \rightarrow readyToEat\big(x, y, do(a,s)\big) \\
& a = cleanDishes \rightarrow kitchenClean\big(do(a,s)\big)
\end{aligned}
$$

Initial theory $\mathcal{D}_{S_0}$. Unless otherwise stated in the text, we use the following values of fluents in the initial state (making a closed world assumption):

$$at(home, S_0) \quad kitchenClean(S_0) \quad hasIngredients(crêpes, S_0)$$

In addition, we include in $\mathcal{D}_{S_0}$ the following axioms about situation-independent relations:

- Meals

$$meal(x) \equiv x = pizza \vee x = tacos \vee x = fajitas \vee x = spaghetti$$
$$\vee\, x = sweetsourpork \vee x = crêpes \vee x = duck \vee x = salad$$

- Types of meals

$$vegetarian(x) \equiv x = salad$$
$$italian(x) \equiv x = spaghetti \vee x = pizza$$
$$mexican(x) \equiv x = tacos \vee x = fajitas$$
$$french(x) \equiv x = crêpes \vee x = duck$$
$$chinese(x) \equiv x = sweetsourpork$$

- Locations

$$location(x) \equiv x = home \vee x = store \vee x = italianRest$$
$$\vee\, x = frenchRest \vee x = chineseRest \vee x = pizzaPlace$$
$$^{(\dagger)}close(x, y) \equiv x = home \wedge y = italianRest$$

- Types of restaurants

$$takeOutRest(x) \equiv x = chineseRest \vee x = pizzaPlace$$
$$dineInRest(x) \equiv x = italianRest \vee x = frenchRest$$

- Restaurant offerings

$$onMenu(x, y) \equiv y = italianRest \wedge (x = spaghetti \vee x = pizza)$$
$$\vee\, y = frenchRest \wedge (x = crêpes \vee x = duck)$$
$$\vee\, y = pizzaPlace \wedge x = pizza$$
$$\vee\, y = chineseRest \wedge x = sweetsourpork$$

- Knowledge of recipes

$$knowsHowToMake(x) \equiv x = crêpes \vee x = spaghetti \vee x = tacos$$
$$\vee\, x = fajitas \vee x = salad$$

Appendix B. Proof of Theorem 4.13

Proof. The proof proceeds by induction on the structural complexity of φ. We assume throughout, unless stated otherwise, that we are given two situations s_1 and $s_2 = do([a_1, \ldots, a_n], s_1)$, where $n \geqslant 1$ and $s_2 = do(a_n, s_3)$.

Case 1. $\varphi = f \in \mathcal{F}$

$$\mathcal{D} \models f[s_1, s_2] \text{ iff } \mathcal{D} \models f[s_1]$$
$$\text{iff } \rho_{s_1}(f) = \text{TRUE}$$
$$\text{iff } \rho^*_{s_1,s_3}(f) = \text{TRUE}$$
$$\text{iff } \mathcal{D} \models \rho^*_{s_1,s_3}(f)[s_2, s_2]$$

The equivalence in line 1 follows from the semantics of TPFs. For the forward direction of the equivalence between lines 1 to 2, we use Definition 4.8, and for the backwards direction, we use the fact that a progressed fluent equals TRUE only if the fluent is satisfied in the situation. The forwards directions of the equivalences between lines 2 and 3 and lines 3 and 4 are obvious, whereas for the backwards directions, we use the fact that progressed fluents are either TRUE or FALSE, plus the fact that TRUE and FALSE are unaffected by progression.

Case 2. $\varphi = r \in \mathcal{R}$

$$\begin{aligned}\mathcal{D} \models r[s_1, s_2] &\text{ iff } \mathcal{D} \models r\\ &\text{ iff } \rho_{s_1}(r) = \text{TRUE}\\ &\text{ iff } \rho^*_{s_1,s_3}(r) = \text{TRUE}\\ &\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_3}(r)[s_2, s_2]\end{aligned}$$

For backwards direction, we make use of the fact that non-fluent relations are progressed to either TRUE or FALSE, which are unaffected by further progression.

Case 3. $\varphi = \mathbf{occ}(a)$
We prove this case in two steps, first for the case where $n = 1$ and then the case where $n \geqslant 2$.

(a) $n = 1$

$$\begin{aligned}\mathcal{D} \models \mathbf{occ}(a)[s_1, s_2] &\text{ iff } s_2 = do(a, s_1)\\ &\text{ iff } \mathcal{D} \models \mathbf{occLast}(a)[s_2, s_2]\\ &\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_1}\big(\mathbf{occ}(a)\big)[s_2, s_2]\end{aligned}$$

(b) $n \geqslant 2$

$$\begin{aligned}\mathcal{D} \models \mathbf{occ}(a)[s_1, s_2] &\text{ iff } \mathcal{D} \models do(a, s_1) \sqsubseteq s_2\\ &\text{ iff } do(a_1, s_1) = do(a, s_1)\\ &\text{ iff } \exists s' do(a_1, s_1) = do\big(a, s'\big)\\ &\text{ iff } \mathcal{D} \models \mathbf{occLast}(a)\big[do(a_1, s_1), s_2\big]\\ &\text{ iff } \rho_{do(a_1,s_1)}\big(\mathbf{occLast}(a)\big) = \text{TRUE}\\ &\text{ iff } \rho_{do(a_1,s_1)}\big(\rho_{s_1}\big(\mathbf{occ}(a)\big)\big) = \text{TRUE}\\ &\text{ iff } \rho^*_{s_1,s_3}\big(\mathbf{occ}(a)\big) = \text{TRUE}\\ &\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_3}\big(\mathbf{occ}(a)\big)[s_2, s_2]\end{aligned}$$

For the backwards direction, we make use of the fact that the progression of $\mathbf{occ}(a)$ through two situations yields either TRUE or FALSE, plus the fact that TRUE and FALSE are unchanged by progression.

Case 4. $\varphi = \mathbf{occLast}(a)$

$$\begin{aligned}\mathcal{D} \models \mathbf{occLast}(a)[s_1, s_2] &\text{ iff } \mathcal{D} \models \exists s' s_1 = do\big(a, s'\big)\\ &\text{ iff } \rho_{s_1}\big(\mathbf{occLast}(a)\big) = \text{TRUE}\\ &\text{ iff } \rho^*_{s_1,s_3}\big(\mathbf{occLast}(a)\big) = \text{TRUE}\\ &\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_3}\big(\mathbf{occLast}(a)\big)[s_2, s_2]\end{aligned}$$

For the backwards direction, we utilize the fact that $\mathbf{occLast}(a)$ progresses to a Boolean constant TRUE or FALSE, which is then stable under further progression.

Case 5. $\varphi = \mathbf{final}(f)$ for some TPF $f \in \mathcal{F}$

$$\begin{aligned}\mathcal{D} \models \mathbf{final}(f)[s_1, s_2] &\text{ iff } \mathcal{D} \models f[s_2]\\ &\text{ iff } \mathcal{D} \models \mathbf{final}(f)[s_2, s_2]\\ &\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_3}\big(\mathbf{final}(f)\big)[s_2, s_2]\end{aligned}$$

Case 6. $\varphi = \neg\psi$ for some TPF ψ
We assume the result for ψ and show that the result holds for $\neg\psi$.

$$\begin{aligned}\mathcal{D} \models \neg\psi[s_1, s_2] \text{ iff } & \mathcal{D} \not\models \psi[s_1, s_2] \\ \text{iff } & \mathcal{D} \not\models \rho^*_{s_1,s_3}(\psi)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \neg\rho^*_{s_1,s_3}(\psi)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \rho^*_{s_1,s_3}(\neg\psi)[s_2, s_2]\end{aligned}$$

For the equivalence in line 1, we make use of the fact that the action theory $\mathcal{D}$ provides complete information about the initial situation, which means in particular that for every situation calculus formula γ, either $\mathcal{D} \models \gamma$ or $\mathcal{D} \models \neg\gamma$.

Case 7. $\varphi = \psi_1 \wedge \psi_2$ for TPFs ψ_1 and ψ_2
We assume the result for ψ_1 and ψ_2 and show that the result holds for $\psi_1 \wedge \psi_2$.

$$\begin{aligned}\mathcal{D} \models \psi_1 \wedge \psi_2[s_1, s_2] \text{ iff } & \mathcal{D} \models \psi_1[s_1, s_2] \text{ and } \mathcal{D} \models \psi_2[s_1, s_2] \\ \text{iff } & \mathcal{D} \models \rho^*_{s_1,s_3}(\psi_1)[s_2, s_2] \text{ and } \mathcal{D} \models \rho^*_{s_1,s_3}(\psi_2)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \big(\rho^*_{s_1,s_3}(\psi_1) \wedge \rho^*_{s_1,s_3}(\psi_2)\big)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \rho^*_{s_1,s_3}(\psi_1 \wedge \psi_2)[s_2, s_2]\end{aligned}$$

Case 8. $\varphi = \psi_1 \vee \psi_2$. As $\psi_1 \vee \psi_2 \equiv \neg(\neg\psi_1 \wedge \neg\psi_2)$, this follows immediately from cases 6 and 7.

Case 9. $\varphi = \exists x\psi$
We assume that the result holds for all TPFs of lower structural complexity than φ. In particular this means that we can assume the result for the TPFs $\psi^{c/x}$.

$$\begin{aligned}\mathcal{D} \models \exists x\psi[s_1, s_2] \text{ iff } & \text{there exists } c \in \mathcal{C} \text{ such that } \mathcal{D} \models \psi^{c/x}[s_1, s_2] \\ \text{iff } & \text{there exists } c \in \mathcal{C} \text{ such that } \mathcal{D} \models \rho^*_{s_1,s_3}\big(\psi^{c/x}\big)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \bigvee_{c\in\mathcal{C}} \rho^*_{s_1,s_3}\big(\psi^{c/x}\big)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \rho_{s_3}\left(\ldots \rho_{do(a_1,s_1)}\left(\bigvee_{c\in\mathcal{C}} \rho_{s_1}\big(\psi^{c/x}\big)\right)\ldots\right)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \rho_{s_3}\big(\ldots \rho_{do(a_1,s_1)}\big(\rho_{s_1}(\exists x\psi)\big)\ldots\big)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \rho^*_{s_1,s_3}(\exists x\psi)[s_2, s_2]\end{aligned}$$

Note that the backwards direction of the equivalence between lines 2 and 3 uses the fact that $\mathcal{D}$ completely defines the initial situation.

Case 10. $\varphi = \forall x\psi$. As $\forall x\psi \equiv \neg\exists x\neg\psi$, this follows directly from Cases 6 and 9.

Case 11. $\varphi = \mathbf{next}(\psi)$
We proceed by induction on n, the difference in length between s_1 and s_2. Our base case is $n = 1$, i.e., $s_2 = do(a_1, s_1)$:

$$\begin{aligned}\mathcal{D} \models \mathbf{next}(\psi)\big[s_1, do(a_1, s_1)\big] \text{ iff } & \mathcal{D} \models \psi\big[do(a_1, s_1), do(a_1, s_1)\big] \\ \text{iff } & \mathcal{D} \models \rho^*_{s_1,s_1}\big(\mathbf{next}(\psi)\big)\big[do(a_1, s_1), do(a_1, s_1)\big]\end{aligned}$$

Next we assume the result for all pairs of situations s_1 and $s_2 = do([a_1, \ldots, a_n], s_1)$ with $n < k$, and we demonstrate the result for the case where $n = k$:

$$\begin{aligned}\mathcal{D} \models \mathbf{next}(\psi)[s_1, s_2] \text{ iff } & \mathcal{D} \models \psi\big[do(a_1, s_1), s_2\big] \\ \text{iff } & \mathcal{D} \models \rho^*_{do(a_1,s_1),s_3}(\psi)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \rho^*_{do(a_1,s_1),s_3}\big(\rho_{s_1}\big(\mathbf{next}(\psi)\big)\big)[s_2, s_2] \\ \text{iff } & \mathcal{D} \models \rho^*_{s_1,s_3}\big(\mathbf{next}(\psi)\big)[s_2, s_2]\end{aligned}$$

Case 12. $\varphi = \mathbf{always}(\psi)$
We assume the result for ψ and prove that the result also holds for $\mathbf{always}(\psi)$. The proof proceeds by induction on n, the difference in length between s_1 and s_2, with base case $n = 1$:

$$\begin{aligned}
\mathcal{D} \models \mathbf{always}(\psi)\big[s_1, do(a_1,s_1)\big] &\text{ iff } \mathcal{D} \models \psi\big[s_1, do(a_1,s_1)\big] \wedge \psi\big[do(a_1,s_1), do(a_1,s_1)\big]\\
&\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_1}(\psi)\big[do(a_1,s_1), do(a_1,s_1)\big] \wedge \psi\big[do(a_1,s_1), do(a_1,s_1)\big]\\
&\text{ iff } \mathcal{D} \models \big(\rho_{s_1}(\psi) \wedge \mathbf{always}(\psi)\big)\big[do(a_1,s_1), do(a_1,s_1)\big]\\
&\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_1}\big(\mathbf{always}(\psi)\big)\big[do(a_1,s_1), do(a_1,s_1)\big]
\end{aligned}$$

We now suppose that the theorem holds for $n < k$, and we show that it is also true when $s_2 = do([a_1,\ldots,a_k], s_1)$:

$$\begin{aligned}
\mathcal{D} \models \mathbf{always}(\psi)[s_1,s_2] &\text{ iff } \mathcal{D} \models \psi[s_1,s_2] \wedge \mathbf{always}(\psi)\big[do(a_1,s_1), s_2\big]\\
&\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_3}(\psi)[s_2,s_2] \wedge \rho^*_{do(a_1,s_1),s_3}\big(\mathbf{always}(\psi)\big)[s_2,s_2]\\
&\text{ iff } \mathcal{D} \models \rho_{s_3}\big(\ldots\rho_{do(a_1,s_1)}\big(\rho_{s_1}(\psi) \wedge \mathbf{always}(\psi)\big)\ldots\big)[s_2,s_2]\\
&\text{ iff } \mathcal{D} \models \rho_{s_3}\big(\ldots\rho_{do(a_1,s_1)}\big(\rho_{s_1}\big(\mathbf{always}(\psi)\big)\big)\ldots\big)[s_2,s_2]\\
&\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_3}\big(\mathbf{always}(\psi)\big)[s_2,s_2]
\end{aligned}$$

Case 13. $\varphi = \mathbf{eventually}(\psi)$. Given that $\mathbf{eventually}(\psi)$ can be rewritten as $\neg\mathbf{always}(\neg\psi)$, this case follows immediately from cases 6 and 12.

Case 14. $\varphi = \mathbf{until}(\psi_1,\psi_2)$
We assume the result for ψ_1 and ψ_2 and prove that the result also holds for $\mathbf{until}(\psi_1,\psi_2)$. We prove this by induction on n, the difference in length between s_1 and s_2. Our base case is $s_2 = do(a_1,s_1)$:

$$\begin{aligned}
\mathcal{D} \models \mathbf{until}(\psi_1,\psi_2)\big[s_1, do(a_1,s_1)\big] &\text{ iff } \mathcal{D} \models \exists s\big(s_1 \sqsubseteq s \wedge s \sqsubseteq do(a,s_1) \wedge \psi_2\big[s, do(a,s_1)\big]\\
&\qquad \wedge \forall s'\big(\big(s_1 \sqsubseteq s' \wedge s' \sqsubset s\big) \rightarrow \psi_1\big[s', do(a,s_1)\big]\big)\big)\\
&\text{ iff } \mathcal{D} \models \psi_2\big[s_1, do(a_1,s_1)\big] \vee \big(\psi_1\big[s_1, do(a_1,s_1)\big] \wedge \psi_2\big[do(a_1,s_1), do(a_1,s_1)\big]\big)\\
&\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_1}(\psi_2)\big[do(a_1,s_1), do(a_1,s_1)\big] \vee \big(\rho^*_{s_1,s_1}(\psi_1)\big[do(a_1,s_1), do(a_1,s_1)\big]\\
&\qquad \wedge \psi_2\big[do(a_1,s_1), do(a_1,s_1)\big]\big)\\
&\text{ iff } \mathcal{D} \models \big(\rho_{s_1}(\psi_2) \vee \big(\rho_{s_1}(\psi_1) \wedge \mathbf{until}(\psi_1,\psi_2)\big)\big)\big[do(a_1,s_1), do(a_1,s_1)\big]\\
&\text{ iff } \mathcal{D} \models \rho_{s_1}\big(\mathbf{until}(\psi_1,\psi_2)\big)\big[do(a_1,s_1), do(a_1,s_1)\big]\\
&\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_1}\big(\mathbf{until}(\psi_1,\psi_2)\big)\big[do(a_1,s_1), do(a_1,s_1)\big]
\end{aligned}$$

For the equivalence above between lines 1 and 2, we use the fact that either $s = s_1$ or $s = do(a_1,s_1)$. In the former case, the first line simplifies to $\psi_2[s_1, do(a_1,s_1)]$, while in the latter case, we obtain $\psi_2[do(a_1,s_1), do(a_1,s_1)] \wedge \psi_1[s_1, do(a_1,s_1)]$.

We now prove the result for the case where $n = k$, under the assumption that the result holds in the case where $n < k$:

$$\begin{aligned}
\mathcal{D} \models \mathbf{until}(\psi_1,\psi_2)[s_1,s_2] &\text{ iff } \mathcal{D} \models \exists s\big(s_1 \sqsubseteq s \wedge s \sqsubseteq s_2 \wedge \psi_2[s,s_2]\\
&\qquad \wedge \forall s'\big(\big(s_1 \sqsubseteq s' \wedge s' \sqsubset s\big) \rightarrow \psi_1\big[s', s_2\big]\big)\big)\\
&\text{ iff } \mathcal{D} \models \psi_2[s_1,s_2] \vee \big(\psi_1[s_1,s_2] \wedge \mathbf{until}(\psi_1,\psi_2)\big[do(a_1,s_1), s_2\big]\big)\\
&\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_3}(\psi_2)[s_2,s_2]\\
&\qquad \vee \big(\rho^*_{s_1,s_3}(\psi_1)[s_2,s_2] \wedge \rho^*_{do(a_1,s_1),s_3}\big(\mathbf{until}(\psi_1,\psi_2)\big)[s_2,s_2]\big)\\
&\text{ iff } \mathcal{D} \models \rho_{s_3}\big(\ldots\rho_{do(a_1,s_1)}\big(\big(\rho_{s_1}(\psi_2) \wedge \mathbf{until}(\psi_1,\psi_2)\big) \vee \rho_{s_1}(\psi_2)\big)\ldots\big)[s_2,s_2]\\
&\text{ iff } \mathcal{D} \models \rho_{s_3}\big(\ldots\rho_{s_1}\big(\mathbf{until}(\psi_1,\psi_2)\big)\ldots\big)[s_2,s_2]\\
&\text{ iff } \mathcal{D} \models \rho^*_{s_1,s_3}\big(\mathbf{until}(\psi_1,\psi_2)\big)[s_2,s_2]
\end{aligned}$$

Note that the equivalence between lines 1 to 2 follows from the fact that either $s = s_1$, in which case line 1 simplifies to $\psi_2[s_1,s_2]$, or $s \neq s_1$ in which case line 1 gives us

$$\exists s\big(do(a_1,s_1) \sqsubseteq s \wedge s \sqsubseteq s_2 \wedge \psi_2[s,s_2] \wedge \forall s'\big(\big(s_1 \sqsubseteq s' \wedge s' \sqsubset s\big) \rightarrow \psi_1\big[s', s_2\big]\big)\big)$$

which is another way to write $\psi_1[s_1,s_2] \wedge \mathbf{until}(\psi_1,\psi_2)[do(a_1,s_1), s_2]$.

Case 15. $\varphi = \text{TRUE}$ or $\varphi = \text{FALSE}$
Obvious as $\mathcal{D} \models \text{TRUE}$ and $\mathcal{D} \not\models \text{FALSE}$, and TRUE and FALSE are unaffected by progression. □

Appendix C. Proof of Theorem 4.18

Lemma C.1. *Let s be any situation, $s_n = do([a_1, \ldots, a_n], s)$, $n \geqslant 0$, a collection of situations, and φ a TPF. Then for any $0 \leqslant i \leqslant j \leqslant n$: $\mathcal{D} \models \varphi[s, s_j]^{opt}$ implies $\mathcal{D} \models \varphi[s, s_i]^{opt}$, and $\mathcal{D} \not\models \varphi[s, s_j]^{pess}$ implies $\mathcal{D} \not\models \varphi[s, s_i]^{pess}$.*

Proof. The proof proceeds by induction over the structure of trajectory property formulae. Clearly, for $\varphi \in \mathcal{F}$, $\varphi \in \mathcal{R}$, and $\varphi = \mathbf{final}(\psi)$ the assumption holds. Also trivial is the case for $\varphi = \mathbf{occ}(a)$: if $\mathcal{D} \models \mathbf{occ}(a)[s, s_j]^{opt}$ then either $j = 0$ which entails $i = 0$, or $a = a_1$. In both cases we have $\mathcal{D} \models \mathbf{occ}(a)[s, s_i]^{opt}$. Similarly, for the pessimistic case, if $\mathcal{D} \not\models \mathbf{occ}(a)[s, s_j]^{pess}$ then either $j = 0$ which entails $i = 0$, or $a \neq a_1$. In both cases we have $\mathcal{D} \not\models \mathbf{occ}(a)[s, s_i]^{pess}$.

Now suppose the assumption holds for the TPFs φ_1, φ_2. Then

- For conjunction:

$$\mathcal{D} \models (\psi_1 \wedge \psi_2)[s, s_j]^{opt} \Rightarrow \mathcal{D} \models \psi_1[s, s_j]^{opt} \text{ and } \mathcal{D} \models \psi_2[s, s_j]^{opt}$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \models \psi_1[s, s_i]^{opt} \text{ and } \mathcal{D} \models \psi_2[s, s_i]^{opt} \Rightarrow \mathcal{D} \models (\psi_1 \wedge \psi_2)[s, s_i]^{opt}$$
$$\mathcal{D} \not\models (\psi_1 \wedge \psi_2)[s, s_j]^{pess} \Rightarrow \mathcal{D} \not\models \psi_1[s, s_j]^{pess} \text{ or } \mathcal{D} \not\models \psi_2[s, s_j]^{pess}$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \not\models \psi_1[s, s_i]^{pess} \text{ or } \mathcal{D} \not\models \psi_2[s, s_i]^{pess} \Rightarrow \mathcal{D} \not\models (\psi_1 \wedge \psi_2)[s, s_i]^{pess}$$

- For disjunctions:

$$\mathcal{D} \models (\psi_1 \vee \psi_2)[s, s_j]^{opt} \Rightarrow \mathcal{D} \models \psi_1[s, s_j]^{opt} \text{ or } \mathcal{D} \models \psi_2[s, s_j]^{opt}$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \models \psi_1[s, s_i]^{opt} \text{ or } \mathcal{D} \models \psi_2[s, s_i]^{opt} \Rightarrow \mathcal{D} \models (\psi_1 \vee \psi_2)[s, s_i]^{opt}$$
$$\mathcal{D} \not\models (\psi_1 \vee \psi_2)[s, s_j]^{pess} \Rightarrow \mathcal{D} \not\models \psi_1[s, s_j]^{pess} \text{ and } \mathcal{D} \not\models \psi_2[s, s_j]^{pess}$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \not\models \psi_1[s, s_i]^{pess} \text{ and } \mathcal{D} \not\models \psi_2[s, s_i]^{pess} \Rightarrow \mathcal{D} \not\models (\psi_1 \vee \psi_2)[s, s_i]^{pess}$$

  We remark that the implication from $\mathcal{D} \models (\psi_1 \vee \psi_2)[s, s_j]^{opt}$ to $\mathcal{D} \models \psi_1[s, s_j]^{opt}$ or $\mathcal{D} \models \psi_2[s, s_j]^{opt}$ in line 1 follows from the fact that $\mathcal{D}$ completely defines the initial situation.
- For negation:

$$\mathcal{D} \models \neg\varphi_1[s, s_j]^{opt} \Rightarrow \mathcal{D} \not\models \varphi_1[s, s_j]^{pess} \overset{i.h.}{\Rightarrow} \mathcal{D} \not\models \varphi_1[s, s_i]^{pess} \Rightarrow \mathcal{D} \models \neg\varphi_1[s, s_i]^{opt}$$
$$\mathcal{D} \not\models \neg\varphi_1[s, s_j]^{pess} \Rightarrow \mathcal{D} \models \varphi_1[s, s_j]^{opt} \overset{i.h.}{\Rightarrow} \mathcal{D} \models \varphi_1[s, s_i]^{opt} \Rightarrow \mathcal{D} \not\models \neg\varphi_1[s, s_i]^{pess}$$

  Note that to go from $\mathcal{D} \not\models \neg\varphi_1[s, s_i]^{pess}$ to $\mathcal{D} \models \neg\varphi_1[s, s_i]^{opt}$ in line 1, and from $\mathcal{D} \not\models \neg\varphi_1[s, s_j]^{pess}$ to $\mathcal{D} \models \varphi_1[s, s_j]^{opt}$ in line 2, we leverage the fact that $\mathcal{D}$ contains complete information about the initial situation.
- For **next**:

$$\mathcal{D} \models \mathbf{next}(\varphi_1)[s, s_j]^{opt} \Rightarrow \mathcal{D} \models \varphi_1[s_1, s_j]^{opt} \text{ or } j = 0$$
$$(\text{by i.h. and since } i \leqslant j) \Rightarrow \mathcal{D} \models \varphi_1[s_1, s_i]^{opt} \text{ or } i = 0 \Rightarrow \mathcal{D} \models \mathbf{next}(\varphi_1)[s, s_i]^{opt}$$
$$\mathcal{D} \not\models \mathbf{next}(\varphi_1)[s, s_j]^{pess} \Rightarrow \mathcal{D} \not\models \varphi_1[s_1, s_j]^{pess} \text{ or } j = 0$$
$$(\text{by i.h. and since } i \leqslant j) \Rightarrow \mathcal{D} \not\models \varphi_1[s_1, s_i]^{pess} \text{ or } i = 0 \Rightarrow \mathcal{D} \not\models \mathbf{next}(\varphi_1)[s, s_i]^{pess}$$

- The cases for **always**(φ), **eventually**(φ), and **until**(φ_1, φ_2) follow by induction hypothesis and the cases for "$\wedge$", "$\vee$", and **next**(φ). □

Lemma C.2. *Let s be any situation, $s_n = do([a_1, \ldots, a_n], s)$, $n \geqslant 0$, a collection of situations, and φ a TPF. Then for any $0 \leqslant i \leqslant j \leqslant n$: $\mathcal{D} \models \varphi[s, s_j]$ implies $\mathcal{D} \models \varphi[s, s_i]^{opt}$ and $\mathcal{D} \not\models \varphi[s, s_j]$ implies $\mathcal{D} \not\models \varphi[s, s_i]^{pess}$.*

Proof. The proof of this lemma proceeds analogously to the previous one. Again it is clear that for $\varphi \in \mathcal{F}$, $\varphi \in \mathcal{R}$, and $\varphi = \mathbf{final}(\psi)$ the assumption holds. Also trivial is the case in which $\varphi = \mathbf{occ}(a)$: if $\mathcal{D} \models \mathbf{occ}(a)[s, s_j]$ then $a = a_1$ and thus $\mathcal{D} \models \mathbf{occ}(a)[s, s_i]^{opt}$. And, for the pessimistic case, if $\mathcal{D} \not\models \mathbf{occ}(a)[s, s_j]$ then either $j = 0$ which entails $i = 0$, or $a \neq a_1$. In both cases we have $\mathcal{D} \not\models \mathbf{occ}(a)[s, s_i]^{pess}$.

Now suppose the assumption holds for TPFs φ_1, φ_2. Then

- For conjunction:

$$\mathcal{D} \models (\psi_1 \wedge \psi_2)[s, s_j] \Rightarrow \mathcal{D} \models \psi_1[s, s_j] \text{ and } \mathcal{D} \models \psi_2[s, s_j]$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \models \psi_1[s, s_i]^{opt} \text{ and } \mathcal{D} \models \psi_2[s, s_i]^{opt} \Rightarrow \mathcal{D} \models (\psi_1 \wedge \psi_2)[s, s_i]^{opt}$$
$$\mathcal{D} \not\models (\psi_1 \wedge \psi_2)[s, s_j] \Rightarrow \mathcal{D} \not\models \psi_1[s, s_j] \text{ or } \mathcal{D} \not\models \psi_2[s, s_j]$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \not\models \psi_1[s, s_i]^{pess} \text{ or } \mathcal{D} \not\models \psi_2[s, s_i]^{pess} \Rightarrow \mathcal{D} \not\models (\psi_1 \wedge \psi_2)[s, s_i]^{pess}$$

- For disjunction:

$$\mathcal{D} \models (\psi_1 \vee \psi_2)[s, s_j] \Rightarrow \mathcal{D} \models \psi_1[s, s_j] \text{ or } \mathcal{D} \models \psi_2[s, s_j]$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \models \psi_1[s, s_i]^{opt} \text{ or } \mathcal{D} \models \psi_2[s, s_i]^{opt} \Rightarrow \mathcal{D} \models (\psi_1 \vee \psi_2)[s, s_i]^{opt}$$
$$\mathcal{D} \not\models (\psi_1 \vee \psi_2)[s, s_j] \Rightarrow \mathcal{D} \not\models \psi_1[s, s_j] \text{ and } \mathcal{D} \not\models \psi_2[s, s_j]$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \not\models \psi_1[s, s_i]^{pess} \text{ and } \mathcal{D} \not\models \psi_2[s, s_i]^{pess} \Rightarrow \mathcal{D} \not\models (\psi_1 \vee \psi_2)[s, s_i]^{pess}$$

- For negation:

$$\mathcal{D} \models \neg\varphi_1[s, s_j] \Rightarrow \mathcal{D} \not\models \varphi_1[s, s_j] \overset{i.h.}{\Rightarrow} \mathcal{D} \not\models \varphi_1[s, s_i]^{pess} \Rightarrow \mathcal{D} \models \neg\varphi_1[s, s_i]^{opt}$$
$$\mathcal{D} \not\models \neg\varphi_1[s, s_j] \Rightarrow \mathcal{D} \models \varphi_1[s, s_j] \overset{i.h.}{\Rightarrow} \mathcal{D} \models \varphi_1[s, s_i]^{opt} \Rightarrow \mathcal{D} \not\models \neg\varphi_1[s, s_i]^{pess}$$

- For **next**:

$$\mathcal{D} \models \mathbf{next}(\varphi_1)[s, s_j] \Rightarrow \mathcal{D} \models \varphi_1[s_1, s_j] \text{ and } j \geqslant 1$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \models \varphi_1[s_1, s_i]^{opt} \Rightarrow \mathcal{D} \models \mathbf{next}(\varphi_1)[s, s_i]^{opt}$$
$$\mathcal{D} \not\models \mathbf{next}(\varphi_1)[s, s_j] \Rightarrow \mathcal{D} \not\models \varphi_1[s_1, s_j] \text{ or } j = 0$$
$$\overset{i.h.}{\Rightarrow} \mathcal{D} \not\models \varphi_1[s_1, s_i]^{pess} \text{ or } i = 0 \Rightarrow \mathcal{D} \not\models \mathbf{next}(\varphi_1)[s, s_i]^{pess}$$

- As before, the cases for **always**(φ), **eventually**(φ), and **until**(φ_1, φ_2) follow from the above cases for "$\wedge$", "$\vee$", and **next**(φ). □

With these lemmas in hand it is straightforward to show the theorem itself. Again we proceed by induction, over the structure of general preference formulae.

Proof of Theorem 4.18. The first item of the theorem follows directly from Lemma C.2. For the second item of the theorem, we consider the component inequalities separately.

Inequalities $w_{s_i}^{opt}(\Phi) \leqslant w_{s_j}^{opt}(\Phi)$ and $w_{s_i}^{pess}(\Phi) \geqslant w_{s_j}^{pess}(\Phi)$: Assume $w_{s_j}^{opt}(\varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_m[v_m]) = v_r$ with all φ_l trajectory property formulae. Then if $r \leqslant m$ we have that $\mathcal{D} \models \varphi_r[S_0, s_j]^{opt}$ and $\mathcal{D} \not\models \varphi_l[S_0, s_j]^{opt}, \forall l < r$. Thus with Lemma C.1 we have $\mathcal{D} \models \varphi_r[S_0, s_i]^{opt}$ and thus $w_{s_i}^{opt}(\varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_m[v_m]) \leqslant v_r$. This trivially also holds for the case where none of the φ_l holds optimistically and hence $v_r = v_{max}$. For the pessimistic case in turn, let $w_{s_j}^{pess}(\varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_m[v_m]) = v_r$ with φ_l trajectory property formulae. Then $\mathcal{D} \not\models \varphi_l[S_0, s_j]^{pess}$, $\forall l < r$. Thus by Lemma C.1 we have $\mathcal{D} \not\models \varphi_l[S_0, s_i]^{pess}, \forall l < r$ and thus $w_{s_i}^{pess}(\varphi_0[v_0] \gg \varphi_1[v_1] \gg \cdots \gg \varphi_m[v_m]) \geqslant v_r$.

Now, for the induction step, suppose the assumption holds for Ψ, Ψ_i. Then,

- For $\Phi = \gamma : \Psi$ and the optimistic case, there are two possibilities:
  - $\mathcal{D} \not\models \gamma[s, s_j]^{pess}$ and thus $w_{s_j}^{opt}(\Phi) = 0$. Then from Lemma C.1 we have that also $\mathcal{D} \not\models \gamma[s, s_i]^{pess}$ and thus $w_{s_i}^{opt}(\Phi) = 0$.
  - $\mathcal{D} \models \gamma[s, s_j]^{pess}$ and $w_{s_j}^{opt}(\Phi) = w_{s_j}^{opt}(\Psi)$. From induction hypothesis we know that $w_{s_i}^{opt}(\Psi) \leqslant w_{s_j}^{opt}(\Psi)$ and thus again $w_{s_i}^{opt}(\Phi) \leqslant w_{s_j}^{opt}(\Phi)$.

  And for the pessimistic case, there are also two possibilities:
  - $\mathcal{D} \not\models \gamma[s, s_j]^{opt}$ and thus $w_{s_j}^{pess}(\Phi) = 0$ so that immediately $w_{s_i}^{pess}(\Phi) \geqslant w_{s_j}^{pess}(\Phi)$.
  - $\mathcal{D} \models \gamma[s, s_j]^{opt}$ and $w_{s_j}^{pess}(\Phi) = w_{s_j}^{pess}(\Psi)$. Using Lemma C.1 it follows that also $\mathcal{D} \models \gamma[s, s_i]^{opt}$ and thus $w_{s_i}^{pess}(\Phi) = w_{s_i}^{pess}(\Psi)$ which by induction hypothesis is greater or equal than $w_{s_j}^{pess}(\Psi)$.

- For $\Phi = \Psi_1 \& \ldots \& \Psi_m$

$$w_{s_j}^{opt}(\Psi_1 \& \ldots \& \Psi_m) = \max_{1 \leqslant l \leqslant m} w_{s_j}^{opt}(\Psi_l) \overset{i.h.}{\geqslant} \max_{1 \leqslant l \leqslant m} w_{s_i}^{opt}(\Psi_l) = w_{s_i}^{opt}(\Psi_1 \& \ldots \& \Psi_m)$$

and

$$w_{s_j}^{pess}(\Psi_1 \& \ldots \& \Psi_m) = \max_{1 \leqslant l \leqslant m} w_{s_j}^{pess}(\Psi_l) \overset{i.h.}{\leqslant} \max_{1 \leqslant l \leqslant m} w_{s_i}^{pess}(\Psi_l) = w_{s_i}^{pess}(\Psi_1 \& \ldots \& \Psi_m)$$

- For $\Phi = \Psi_1 | \ldots | \Psi_m$

$$w_{s_j}^{opt}(\Psi_1 | \ldots | \Psi_m) = \min_{1 \leqslant l \leqslant m} w_{s_j}^{opt}(\Psi_l) \overset{i.h.}{\geqslant} \min_{1 \leqslant l \leqslant m} w_{s_i}^{opt}(\Psi_l) = w_{s_i}^{opt}(\Psi_1 | \ldots | \Psi_m)$$

and

$$w_{s_j}^{pess}(\Psi_1 | \ldots | \Psi_m) = \min_{1 \leqslant l \leqslant m} w_{s_j}^{pess}(\Psi_l) \overset{i.h.}{\leqslant} \min_{1 \leqslant l \leqslant m} w_{s_i}^{pess}(\Psi_l) = w_{s_i}^{pess}(\Psi_1 | \ldots | \Psi_m)$$

Inequalities $w_{s_j}^{opt}(\Phi) \leqslant w_{s_k}(\Phi)$ and $w_{s_j}^{pess}(\Phi) \geqslant w_{s_k}(\Phi)$: The proof of these two inequalities proceeds in direct analogy to the previous ones but uses Lemma C.2 instead of Lemma C.1. □

Appendix D. Proof of Corollary 4.20

Proof. The proof proceeds by induction over the structure of φ and using Theorem 4.13.

- $\varphi = \mathbf{final}(\psi)$: By definition $\rho_s^*(\mathbf{final}(\psi))[s', s'] = \mathbf{final}(\psi)[s', s']$ and $\mathbf{final}(\psi)[S_0, s']^{opt/pess} = \mathbf{final}(\psi)[s', s']^{opt/pess}$, hence the thesis.
- $\varphi = \mathbf{occ}(a)$: We have:

$$\mathcal{D} \models \mathbf{occ}(a)\big[S_0, s'\big]^{opt} \text{ iff (by definition)}$$
$$\mathcal{D} \models do(a, S_0) \sqsubseteq s' \vee S_0 = s' \text{ iff (by assumption } n \geqslant 1)$$
$$\mathcal{D} \models do(a, S_0) \sqsubseteq s' \text{ iff } \big(\text{by definition of } \mathbf{occ}(a) \text{ and Theorem 4.13}\big)$$
$$\mathcal{D} \models \rho_s^*\big(\mathbf{occ}(a)\big)\big[s', s'\big].$$

  Also, by definition: $\mathbf{occ}(a)[S_0, s']^{pess} = \mathbf{occ}(a)[S_0, s']$ and hence, $\mathcal{D} \models \mathbf{occ}(a)[S_0, s']^{pess}$ iff $\mathcal{D} \models \rho_s^*(\mathbf{occ}(a))[s', s']$.
  Now there are two cases to consider, either $\rho_s^*(\mathbf{occ}(a))[s', s'] = \mathbf{occLast}(a)$, in which case the thesis follows by definition of $\mathbf{occLast}(a)[s, s']^{opt/pess}$, or $\rho_s^*(\mathbf{occ}(a))[s', s']$ is equal to a Boolean constant TRUE/FALSE, in which case the thesis follows trivially.
- $\varphi = \mathbf{next}(\psi)$: Again, since we are assuming $n \geqslant 1$ we get that $\mathbf{next}(\psi)[S_0, s']^{opt/pess} = \mathbf{next}(\psi)[S_0, s']$. Hence the thesis follows from Theorem 4.13.
- For the remaining temporal formulae the thesis follows from the induction hypothesis due to their definition in terms of **next**. □

Appendix E. Proof of Theorem 6.3

Proof. The theorem is easily proven by extending Lemmas C.1 and C.2 to handle the two new constructs in eTPFs as follows. With respect to Lemma C.1 we observe:

- If $\mathcal{D} \models \mathbf{occC}(\delta)[s, s_j]^{opt}$, then, by definition,

$$\mathcal{D} \models (\exists s_1 \colon s \sqsubseteq s_1 \sqsubseteq s_j)\exists\delta'\big(Trans^*\big(\delta, s, \delta', s_1\big) \wedge \big(s_1 \neq s_j \rightarrow Final\big(\delta', s_1\big)\big)\big)$$

The chosen s_1 is either such that $s_1 \sqsubseteq s_i$ or the other way round, $s_i \sqsubseteq s_1$. In the former case it follows that:

$$\mathcal{D} \models \big(\exists s_1' \colon s \sqsubseteq s_1' \sqsubseteq s_i\big)\exists\delta'\big(Trans^*\big(\delta, s, \delta', s_1'\big) \wedge Final\big(\delta', s_1'\big)\big)$$

In the latter case, it follows that:

$$\mathcal{D} \models \exists\delta' Trans^*\big(\delta, s, \delta', s_i\big)$$

Hence, together we obtain:

$$\mathcal{D} \models (\exists s_1 \colon s \sqsubseteq s_1 \sqsubseteq s_i)\exists\delta'\big(Trans^*\big(\delta, s, \delta', s_1\big) \wedge \big(s_1 \neq s_i \rightarrow Final\big(\delta', s_1\big)\big)\big)$$

and thus $\mathcal{D} \models \mathbf{occC}(\delta)[s, s_i]^{opt}$.
Furthermore: $\mathcal{D} \not\models \mathbf{occC}(\delta)[s, s_j]^{pess}$, i.e., by definition, $\mathcal{D} \not\models (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s_j)Do(\delta, s, s_1)$. Since s_i precedes s_j it is obvious that then also $\mathcal{D} \not\models (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s_i)Do(\delta, s, s_1)$ and hence by definition: $\mathcal{D} \not\models \mathbf{occC}(\delta)[s, s_i]^{pess}$.
- If $\mathcal{D} \models \mathbf{afterC}(\delta, \varphi)[s, s_j]^{opt}$, then, by definition, $\mathcal{D} \models (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s_j)Do(\delta, s, s_1) \rightarrow \varphi[s_1, s_j]^{opt}$. Since, s_i precedes s_j, it follows that: $\mathcal{D} \models \forall s_1(s_1 \sqsubseteq s_i \rightarrow s_1 \sqsubseteq s_j)$, and hence with the above, we derive: $\mathcal{D} \models (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s_i)(Do(\delta, s, s_1) \rightarrow \varphi[s_1, s_i]^{opt})$, i.e., $\mathcal{D} \models \mathbf{afterC}(\delta, \varphi)[s, s_i]^{opt}$.
Furthermore: If $\mathcal{D} \not\models \mathbf{afterC}(\delta, \varphi)[s, s_j]^{pess}$, then, by definition,

$$\mathcal{D} \not\models (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s_j)Do(\delta, s, s_1) \rightarrow \varphi[s_1, s_j]^{pess} \wedge \big(\nexists\delta', s''\big)\big(Trans^*\big(\delta, s, \delta', s''\big) \wedge s_j \sqsubset s''\big)$$

Hence, either:
1. $\mathcal{D} \models (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s_j)(Do(\delta, s, s_1) \wedge \neg\varphi[s_1, s_j]^{pess})$, or
2. $\mathcal{D} \models (\exists\delta', s'')(Trans^*(\delta, s, \delta', s'') \wedge s_j \sqsubset s'')$.

In the former case, there are again two cases to distinguish: (a) the chosen s_1 is such that $s_i \sqsubset s_1 \sqsubseteq s_j$, or (b) $s_1 \sqsubseteq s_i$. In case (a) it follows that:

$$\mathcal{D} \models \exists\delta'\exists s''\big(Trans^*\big(\delta, s, \delta', s''\big) \wedge s_i \sqsubset s''\big)$$

and in case (b) it follows that:

$$\mathcal{D} \models (\exists s_1 \colon s \sqsubseteq s_1 \sqsubseteq s_i)\big(Do(\delta, s, s_1) \wedge \neg\varphi[s_1, s_i]^{pess}\big)$$

using induction hypothesis w.r.t. φ.
In the latter case of the enumeration (2.), it follows immediately from the definition of $Trans^*$ and the fact that $s_i \sqsubset s_j$ that

$$\mathcal{D} \models \exists\delta'\exists s''\big(Trans^*\big(\delta, s, \delta', s''\big) \wedge s_i \sqsubset s''\big)$$

Hence, taking all cases together, we get that

$$\mathcal{D} \not\models (\forall s_1 \colon s \sqsubseteq s_1 \sqsubseteq s_i).Do(\delta, s, s_1) \rightarrow \varphi[s_1, s_i]^{pess} \wedge \big(\nexists\delta', s''\big)\big(Trans^*\big(\delta, s, \delta', s''\big) \wedge s_i \sqsubset s''\big)$$

i.e, by definition, $\mathcal{D} \not\models \mathbf{afterC}(\delta, \varphi)[s, s_i]^{pess}$.

With respect to Lemma C.2 we observe:

- If $\mathcal{D} \models \mathbf{occC}(\delta)[s, s_j]$, then, by definition,

$$\mathcal{D} \models (\exists s_1 \colon s \sqsubseteq s_1 \sqsubseteq s_j)\exists\delta'\big(Trans^*\big(\delta, s, \delta', s_1\big) \wedge Final\big(\delta', s_1\big)\big)$$

It then follows from definition of *Trans* that

$$\mathcal{D} \models (\exists s_1 \colon s \sqsubseteq s_1 \sqsubseteq s_i)\exists\delta'\big(Trans^*\big(\delta, s, \delta', s_1\big) \wedge \big(s_1 \neq s_i \rightarrow Final\big(\delta', s_1\big)\big)\big)$$

i.e., by definition, $\mathcal{D} \models \mathbf{occC}(\delta)[s, s_i]^{opt}$.

Furthermore: $\mathcal{D} \not\models \mathbf{occC}(\delta)[s, s_j]$, i.e., by definition, $\mathcal{D} \not\models (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s_j)Do(\delta, s, s_1)$. Since s_i precedes s_j it is obvious that then also again $\mathcal{D} \not\models (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s_i)Do(\delta, s, s_1)$ and hence by definition: $\mathcal{D} \not\models \mathbf{occC}(\delta)[s, s_i]^{pess}$.

- If $\mathcal{D} \models \mathbf{afterC}(\delta, \varphi)[s, s_j]$, then, by definition, $\mathcal{D} \models (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s_j)(Do(\delta, s, s_1) \to \varphi[s_1, s_j])$. Since, s_i precedes s_j, it follows that: $\mathcal{D} \models \forall s_1 (s_1 \sqsubseteq s_i \to s_1 \sqsubseteq s_j)$, and hence with the above and by induction hypothesis w.r.t. φ: $\mathcal{D} \models (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s_i)(Do(\delta, s, s_1') \to \varphi[s_1, s_i]^{opt})$, i.e., $\mathcal{D} \models \mathbf{afterC}(\delta, \varphi)[s, s_i]^{opt}$.
  Furthermore: If $\mathcal{D} \not\models \mathbf{afterC}(\delta, \varphi)[s, s_j]$, then, $\mathcal{D} \not\models (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s_j).Do(\delta, s, s_1) \to \varphi[s_1, s_j]$. Hence: $\mathcal{D} \models (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s_j)(Do(\delta, s, s_1) \wedge \neg\varphi[s_1, s_j])$. There are two cases to distinguish: (a) the chosen s_1 is such that $s_i \sqsubset s_1 \sqsubseteq s_j$, or (b) $s_1 \sqsubseteq s_i$. In case (a) it follows that:

$$\mathcal{D} \models \exists \delta' \exists s'' \big(Trans^*\big(\delta, s, \delta', s''\big) \wedge s_1 \sqsubset s''\big)$$

  and in case (b) it follows that:

$$\mathcal{D} \models (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s_i)\big(Do(\delta, s, s_1) \wedge \neg\varphi[s_1, s_i]^{pess}\big)$$

  using induction hypothesis w.r.t. φ. Hence, taken together we get that

$$\mathcal{D} \not\models (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s_i)\big(Do(\delta, s, s_1) \to \varphi[s_1, s_i]^{pess}\big) \wedge \big(\nexists \delta', s''\big)\big(Trans^*\big(\delta, s, \delta', s''\big) \wedge s_i \sqsubset s''\big)$$

  i.e, by definition, $\mathcal{D} \not\models \mathbf{afterC}(\delta, \varphi)[s, s_i]^{pess}$.

With these extensions in place, the proof proceeds analogously to the proof of Theorem 4.18. □

References

[1] F. Bacchus, F. Kabanza, Using temporal logics to express search control knowledge for planning, Artificial Intelligence 16 (2000) 123–191.
[2] J. Baier, S. McIlraith, Planning with first-order temporally extended goals using heuristic search, in: Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06), 2006, pp. 788–795.
[3] J.A. Baier, F. Bacchus, S.A. McIlraith, A heuristic search approach to planning with temporally extended preferences, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), 2007, pp. 1808–1815.
[4] J.A. Baier, F. Bacchus, S.A. McIlraith, A heuristic search approach to planning with temporally extended preferences, Artificial Intelligence 173 (5–6) (2009) 593–618.
[5] J.A. Baier, S.A. McIlraith, On domain-independent heuristics for planning with qualitative preferences, in: Proceedings of the 7th Workshop on Nonmonotonic Reasoning, Action and Change (NRAC-07), 2007.
[6] J.A. Baier, S.A. McIlraith, Planning with preferences, AI Magazine 29 (4) (2008) 25–36.
[7] C. Baral, V. Kreinovich, R. Trejo, Computational complexity of planning with temporal goals, in: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01), 2001, pp. 509–514.
[8] S. Benferhat, D. Dubois, H. Prade, Towards a possibilistic logic handling of preferences, in: Applied Intelligence, vol. 14, Kluwer, 2001, pp. 303–317, .
[9] J. Benton, S. Kambhampati, M.B. Do, YochanPS: PDDL3 simple preferences and partial satisfaction planning, in: Proceedings of the 5th International Planning Competition Booklet (IPC-06), 2006, pp. 54–57.
[10] J. Benton, M. van den Briel, S. Kambhampati, A hybrid linear programming and relaxed plan heuristic for partial satisfaction problems, in: Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-07), 2007, pp. 34–41.
[11] M. Bienvenu, C. Fritz, S. McIlraith, Planning with qualitative temporal preferences, in: Proceedings of the 10th International Conference on Knowledge Representation and Reasoning (KR-06), 2006, pp. 134–144.
[12] M. Bienvenu, C. Fritz, S. Sohrabi, S. McIlraith, PPLAN: Code, experiments, http://www.cs.toronto.edu/~sheila/PPLAN, 2006.
[13] C. Boutilier, R. Brafman, C. Domshlak, H. Hoos, D. Poole, CP-nets: A tool for representing and reasoning about conditional ceteris paribus preference statements, Journal of Artificial Intelligence Research 21 (2004) 135–191.
[14] R.I. Brafman, Y. Chernyavsky, Planning with goal preferences and constraints, in: Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05), 2005, pp. 182–191.
[15] G. Brewka, Complex preferences for answer set optimization, in: Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR-04), 2004, pp. 213–223.
[16] G. Brewka, A rank based description language for qualitative preferences, in: Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04), 2004, pp. 303–307.
[17] G. Brewka, S. Benferhat, D.L. Berre, Qualitative choice logic, Artificial Intelligence 157 (1–2) (2004), Special Issue on Nonmonotonic Reasoning.
[18] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, S. Thrun, Experiences with an interactive museum tour-guide robot, Artificial Intelligence 114 (1–2) (1999) 3–55.
[19] T. Bylander, The computational complexity of propositional STRIPS planning, Artificial Intelligence 69 (1–2) (1994) 165–204.
[20] S. Coste-Marquis, J. Lang, P. Liberatore, P. Marquis, Expressive power and succinctness of propositional languages for preference representation, in: Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR-04), 2004, pp. 203–212.
[21] G. De Giacomo, Y. Lespérance, H. Levesque, ConGolog, a concurrent programming language based on the situation calculus, Artificial Intelligence 121 (1–2) (2000) 109–169.
[22] J. Delgrande, T. Schaub, H. Tompits, Domain-specific preferences for causual reasoning and planning, in: Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR-04), 2004, pp. 673–682.
[23] J.P. Delgrande, T. Schaub, H. Tompits, A general framework for expressing preferences in causal reasoning and planning, Journal of Logic and Computation 17 (2007) 871–907.
[24] S. Edelkamp, Optimal symbolic PDDL3 planning with MIPS-BDD, in: Proceedings of the 5th International Planning Competition Booklet (IPC-06), 2006, pp. 31–33.
[25] S. Edelkamp, S. Jabbar, M. Naizih, Large-scale optimal PDDL3 planning with MIPS-XXL, in: Proceedings of the 5th International Planning Competition Booklet (IPC-06), 2006, pp. 28–30.
[26] M. Ehrgott, Multicriteria Optimization, Springer, Berlin, 2000.
[27] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres, Answer set planning under action costs, Journal of Artificial Intelligence Research 19 (2003) 25–71.
[28] K. Erol, D.S. Nau, V.S. Subrahmanian, Complexity, decidability and undecidability results for domain-independent planning, Artificial Intelligence 76 (1–2) (1995) 75–88.

[29] R. Feldmann, G. Brewka, S. Wenzel, Planning with prioritized goals, in: Proceedings of the 10th International Conference on Knowledge Representation and Reasoning (KR-06), 2006, pp. 503–514.
[30] A. Ferrein, C. Fritz, G. Lakemeyer, On-line decision-theoretic Golog for unpredictable domains, in: Proceedings of 27th German Conference on AI (KI-04), 2004, pp. 322–336.
[31] C. Fritz, S. McIlraith, Decision-theoretic GOLOG with qualitative preferences, in: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR-06), 2006, pp. 153–163.
[32] A. Gabaldon, Precondition control and the progression algorithm, in: Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR-04), 2004, pp. 634–643.
[33] A. Gerevini, P. Haslum, D. Long, A. Saetti, Y. Dimopoulos, Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners, Artificial Intelligence 173 (5–6) (2009) 619–668.
[34] A. Gerevini, D. Long, Plan constraints and preferences in PDDL3: The language of the fifth international planning competition. Tech. Rep., University of Brescia, 2005.
[35] E. Giunchiglia, M. Maratea, Planning as satisfiability with preferences, in: Proceedings of the 22nd Conference on Artificial Intelligence (AAAI-07), 2007, pp. 987–992.
[36] J. Goldsmith, J. Ulrich (Eds.), AI Magazine 29 (4) (2008), Winter 2008, Special Issue on Preferences.
[37] C.C. Green, Application of theorem proving to problem solving, in: Proceedings of the 1st International Joint Conference on Artificial Intelligence, 1969, pp. 219–240.
[38] P. Haddawy, S. Hanks, Representations for decision-theoretic planning: Utility functions for deadline goals, in: Proceedings of the 3rd International Conference on Knowledge Representation and Reasoning (KR-96), 1992, pp. 71–82.
[39] M. Helmert, Decidability and undecidability results for planning with numerical state variables, in: Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS-02), 2002, pp. 44–53.
[40] J. Hoffmann, B. Nebel, The FF planning system: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.
[41] C.-W. Hsu, B. Wah, R. Huang, Y. Chen, Constraint partitioning for solving planning problems with trajectory constraints and goal preferences, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), 2007, pp. 1924–1929.
[42] H.A. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), 1999, pp. 318–325.
[43] J. Kvarnström, P. Doherty, TALplanner: A temporal logic based forward chaining planner, Annals of Mathematics and Artificial Intelligence 30 (2000) 119–169.
[44] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, R.B. Scherl, GOLOG: A logic programming language for dynamic domains, Journal of Logic Programming 31 (1–3) (1997) 59–83.
[45] S. Liaskos, Acquiring and reasoning about variability in goal models, PhD in Computer Science, Department of Computer Science, University of Toronto, Toronto, Canada, 2008.
[46] S. Liaskos, S.A. McIlraith, J. Mylopoulos, Towards augmenting requirements models with preferences, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE-09), 2009, pp. 565–569.
[47] J. McCarthy, Situations, actions and causal laws. Tech. Rep., Stanford University, 1963.
[48] D.V. McDermott, PDDL—The Planning Domain Definition Language, Tech. Rep. TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
[49] S. McIlraith, T. Son, Adapting Golog for composition of semantic web services, in: Proceedings of the 8th International Conference on Knowledge Representation and Reasoning, 2002, pp. 482–493.
[50] K. Myers, T. Lee, Generating qualitatively different plans through metatheoretic biases, in: Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99), 1999, pp. 570–576.
[51] A. Pnueli, The temporal logic of programs, in: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS-77), 1977, pp. 46–57.
[52] M. Puterman, Markov Decision Processes: Discrete Dynamic Programming, Wiley, New York, 1994.
[53] R. Reiter, Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press, Cambridge, MA, 2001.
[54] A.P. Sistla, E.M. Clarke, The complexity of propositional linear temporal logics, Journal of the ACM 32 (3) (1985) 733–749.
[55] D.E. Smith, Choosing objectives in over-subscription planning, in: Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04), 2004, pp. 393–401.
[56] S. Sohrabi, J.A. Baier, S.A. McIlraith, HTN planning with preferences, in: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09), 2009, pp. 1790–1797.
[57] S. Sohrabi, S.A. McIlraith, On planning with preferences in HTN, in: Proceedings of the 4th Multidisciplinary Workshop on Advances in Preference Handling (M-Pref-08) at AAAI-08, 2008, pp. 103–109.
[58] S. Sohrabi, N. Prokoshyna, S. McIlraith, Web service composition via generic procedures and customizing user preferences, in: Proceedings of the 5th International Semantic Web Conference (ISWC-06), 2006, pp. 597–611.
[59] T.C. Son, E. Pontelli, Planning with preferences using logic programming, in: Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-04), 2004, pp. 247–260.
[60] T.C. Son, E. Pontelli, Planning with preferences using logic programming, Theory and Practice of Logic Programming 6 (5) (2006) 559–607.
[61] P.H. Tu, T.C. Son, E. Pontelli, CPP: A constraint logic programming based planner with preferences, in: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-07), 2007, pp. 290–296.
[62] M. van den Briel, R.S. Nigenda, M.B. Do, S. Kambhambati, Effective approaches for partial satisfaction (oversubscription) planning, in: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), 2004, pp. 562–569.
[63] J. von Neumann, O. Morgenstern, Theory of Games and Economic Behavior, Princeton University Press, 1994.
[64] N. Wilson, Extending CP-Nets with stronger conditional preference statements, in: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), 2004, pp. 735–741.
[65] N. Yorke-Smith, K.B. Venable, F. Rossi, Temporal reasoning with preferences and uncertainty, in: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03), 2003, pp. 1385–1390.