Explanation-based learning of action models

Diego Aineto[1], Sergio Jimnez[1 [0000−0003−0561−4880]], and Eva Onaindia[1 [0000−0001−6931−8293]]

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València.
Camino de Vera s/n. 46022 Valencia, Spain
{dieaigar,serjice,onaindia}@dsic.upv.es

Abstract.

Keywords: Learning action models · Planning and Learning · Classical Planning.

1 Introduction

2 Background

This section formalizes the models for *classical planning* and for the *observation* of the execution of a classical plan.

2.1 Classical planning with conditional effects

F is the set of *fluents* or *state variables* (propositional variables). A *literal* l is a valuation of a fluent $f \in F$, i.e. either $l = f$ or $l = \neg f$. L is a set of literals that represents a partial assignment of values to fluents, and $\mathcal{L}(F)$ is the set of all literals sets on F, i.e. all partial assignments of values to fluents. A *state* s is a full assignment of values to fluents. We explicitly include negative literals $\neg f$ in states and so $|s| = |F|$ and the size of the state space is $2^{|F|}$.

A *planning frame* is a tuple $\Phi = \langle F, A\rangle$, where F is a set of fluents and A is a set of *actions*. An action $a \in A$ is defined with *preconditions*, $\mathsf{pre}(a) \in \mathcal{L}(F)$, and *effects* $\mathsf{eff}(a) \in \mathcal{L}(F)$. The semantics of actions $a \in A$ is specified with two functions: $\rho(s, a)$ denotes whether action a is *applicable* in a state s and $\theta(s, a)$ denotes the *successor state* that results of applying action a in a state s. Then, $\rho(s, a)$ holds iff $\mathsf{pre}(a) \subseteq s$. And the result of applying a in s is $\theta(s, a) = \{s \setminus \neg\mathsf{eff}(a)) \cup \mathsf{eff}(a)\}$, with $\neg\mathsf{eff}(a) = \{\neg l : l \in \mathsf{eff}(a)\}$.

A *planning problem* is defined as a tuple $P = \langle F, A, I, G\rangle$, where I is the initial state in which all the fluents of F are assigned a value true/false and G is the goal set. A *plan* π for P is an action sequence $\pi = \langle a_1, \ldots, a_n\rangle$, and $|\pi| = n$ denotes its *plan length*. The execution of π in the initial state I of P induces a *trajectory* $\tau(\pi, P) = \langle s_0, a_1, s_1, \ldots, a_n, s_n\rangle$ such that $s_0 = I$ and, for each $1 \leq i \leq n$, it holds $\rho(s_{i-1}, a_i)$ and $s_i = \theta(s_{i-1}, a_i)$. A trajectory $\tau(\pi, P)$ that solves P is one in which $G \subseteq s_n$.

An action $a_c \in A$ with conditional effects is defined as a set of preconditions $\mathsf{pre}(a_c) \in \mathcal{L}(F)$ and a set of *conditional effects* $\mathsf{cond}(a_c)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a_c)$ is composed of two sets of literals: $C \in \mathcal{L}(F)$, the *condition*, and $E \in \mathcal{L}(F)$, the *effect*. An action $a_c \in A$ is applicable in a state s if and only if $\mathsf{pre}(a_c) \subseteq s$, and the *triggered effects* resulting from the action application are the effects whose conditions hold in s:

$$triggered(s, a_c) = \bigcup_{C \rhd E \in \mathsf{cond}(a_c), C \subseteq s} E.$$

The result of applying a_c in state s follows the same definition of successor state, $\theta(s, a)$, but applied to the conditional effects in $triggered(s, a_c)$.

2.2 The observation model

Given a planning problem $P = \langle F, A, I, G \rangle$, a plan π and a trajectory $\tau(\pi, P)$, we define the *observation of the trajectory* as a sequence of partial states that represents the observation from the execution of π in P.

Formally, $\mathcal{O}(\tau) = \langle s_0^o, s_1^o \ldots, s_m^o \rangle$, $s_0^o = I$ is a sequence of possibly *partially observable states*, except for the initial state s_0^o, which is fully observable. A partially observable state s_i^o is one in which $|s_i^o| < |F|$; i.e., a state in which at least a fluent of F is not observable. Note that this definition also comprises the case $|s_i^o| = 0$, when the state is fully unobservable. Whatever the sequence of observed states of $\mathcal{O}(\tau)$ is, it must be consistent with the sequence of states of $\tau(\pi, P)$, meaning that $\forall i, s_i^o \subseteq s_i$. In practice, the number of observed states, m, range from 1 (the initial state, at least), to $|\pi|+1$, and the observed intermediate states will comprise a number of fluents between $[1, |F|]$.

We assume a bijective monotone mapping between actions/states of trajectories and observations [Ramírez and Geffner2009], thus also granting the inverse consistency relationship (the trajectory is a superset of the observation). Therefore, transiting between two consecutive observed states in $\mathcal{O}(\tau)$ may require the execution of more than a single action $(\theta(s_i^o, \langle a_1, \ldots, a_k \rangle) = s_{i+1}^o$, where $k \geq 1$ is unknown but finite. In other words, having $\mathcal{O}(\tau)$ does not imply knowing the actual length of π.

Figure **??** illustrates a partial observation of a six-state trajectory $\{$`<(xcoord 0)(ycoord 0)>, <(xcoord 1)(ycoord 0)>, <(xcoord 2)(ycoord 0)>, <(xcoord 3)(ycoord 0)>, <(xcoord 3)(ycoord 1)>, <(xcoord 2)(ycoord 1)>`$\}$. This observation only contains fluents of the predicates `(xcoord ?v)` and `(ycoord ?v)`, and the value of the remaining fluents, corresponding to predicates `(next ?v1 ?v2)`, `(q0)` and `(q1)`, is unobservable in the six states.

We introduce a particular class of $\mathcal{O}(\tau)$ observations. This new class allows us to distinguish between *observable* state variables, whose value may be read from sensors, and *hidden* or *latent* state variables that cannot be observed.

Definition 1 (Φ-observation). *Given a subset of fluents $\Phi \subseteq F$ we say that $\mathcal{O}(\tau)$ is a Φ-observation of the execution of π on P iff, for every $1 \leq i \leq m$, each observed state s_i^o only contains fluents in Φ.*

We consider that the **observed actions** are fluents consistent with π, which means that $\langle a_1^o, \ldots, a_l^o \rangle$ is a sub-sequence of π s.t. $a_i^o \in s_i^o$ $(0 \leq i < m)$. Specifically, the number of observed actions, l, can range from 0 (fully unobservable action sequence) to $|\pi|$ (fully observable action sequence).

3 Explanation-based learning of action models

The *one-shot* learning task to learn action models from *domain-specific knowledge* is defined as a tuple $\Lambda = \langle \mathcal{M}, \mathcal{O}, \Phi \rangle$, where:

- $\mathcal{M}$ is the *initial empty model* that contains only the header of each action model to be learned.
- $\mathcal{O}$ is a single learning example or plan observation; i.e. a sequence of (partially) observable states representing the evidence of the execution of an observed agent.
- Φ is a set of logic formulae that define *domain-specific knowledge.*

A *solution* to a learning task $\Lambda = \langle \mathcal{M}, \mathcal{O}, \Phi \rangle$ is a model $\mathcal{M}'$ s.t. there exists a plan computable with $\mathcal{M}'$ that is consistent with the headers of $\mathcal{M}$, the observed states of $\mathcal{O}$ and the given domain knowledge in Φ.

3.1 The space of Strips action models

We analyze here the solution space of a learning task $\Lambda = \langle \mathcal{M}, \mathcal{O}, \Phi \rangle$; i.e., the space of Strips action models. In principle, for a given action model ξ, any element of $\mathcal{I}_{\xi,\Psi}$ can potentially appear in $pre(\xi)$, $del(\xi)$ and $add(\xi)$. In practice, the actual space of possible Strips schemata is bounded by:

1. **Syntactic constraints**. The solution $\mathcal{M}'$ must be consistent with the Strips constraints: $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ and $pre(\xi) \cap add(\xi) = \emptyset$. *Typing constraints* would also be a type of syntactic constraint [McDermott *et al.*1998].
2. **Observation constraints**. The solution $\mathcal{M}'$ must be consistent with these *semantic constraints* derived from the learning samples $\mathcal{O}$, which in our case is a single plan observation. Specifically, the states induced by the plan computable with $\mathcal{M}'$ must comprise the observed states of the sample, which further constrains the space of possible action models.

Considering only the syntactic constraints, the size of the space of possible Strips models is given by $2^{2 \times |\mathcal{I}_{\Psi,\xi}|}$ because one element in $\mathcal{I}_{\xi,\Psi}$ can appear both in the preconditions and effects of ξ. In this work, the belonging of an $e \in \mathcal{I}_{\Psi,\xi}$ to the preconditions, positive effects or negative effects of ξ is handled with a refined propositional encoding that uses fluents of two types, pre_ξ_e and eff_ξ_e, instead of the three fluents used in the BLS. The four possible combinations of these two fluents are sumarized in Figure 1. This compact encoding allows for a more effective exploitation of the syntactic constraints, and also yields the solution space of Λ to be the same as its search space.

Combination	**Meaning**
$\neg pre\_\xi\_e \wedge \neg eff\_\xi\_e$	e belongs neither to the preconditions nor effects of ξ ($e \notin pre(\xi) \wedge e \notin add(\xi) \wedge e \notin del(\xi)$)
$pre\_\xi\_e \wedge \neg eff\_\xi\_e$	e is only a precondition of ξ ($e \in pre(\xi) \wedge e \notin add(\xi) \wedge e \notin del(\xi)$)
$\neg pre\_\xi\_e \wedge eff\_\xi\_e$	e is a positive effect of ξ ($e \notin pre(\xi) \wedge e \in add(\xi) \wedge e \notin del(\xi)$)
$pre\_\xi\_e \wedge eff\_\xi\_e$	e is a negative effect of ξ ($e \in pre(\xi) \wedge e \notin add(\xi) \wedge e \in add(\xi)$)

Fig. 1. Combinations of the fluent propositional encoding and their meaning

3.2 The sampling space

The single plan observation of $\mathcal{O}$ is defined as $\mathcal{O} = \langle s_0^o, s_1^o \ldots, s_m^o \rangle$, a sequence of possibly *partially observed states* except for the initial state s_0^o which is a *fully observable* state. As commented before, the predicates Ψ and the objects that shape the fluents F are then deducible from s_0^o. A partially observed state s_i^o, $1 \leq i \leq m$, is one in which $|s_i^o| < |F|$; i.e., a state in which at least a fluent of F was not observed. Intermediate states can be *missing*, meaning that they are unobservable, so transiting between two consecutive observed states in $\mathcal{O}$ may require the execution of more than one action ($\theta(s_i^o, \langle a_1, \ldots, a_k \rangle) = s_{i+1}^o$ (with $k \geq 1$ is unknown but finite). The minimal expression of a learning sample must comprise at least two state observations, a full initial state s_0^o and a partially observed final state s_m^o so $m \geq 1$.

Figure 2 shows a learning example that contains an initial state of the blocksworld where the robot hand is empty and three blocks (namely `blockA`, `blockB` and `blockC`) are on top of the table and clear. The observation represents a partially observable final state in which `blockA` is on top of `blockB` and `blockB` on top of `blockC`.

3.3 The domain-specific knowledge

One can introduce domain-specific knowledge to constrain further the space of possible schemata. For instance, back to the *blocksworld* domain, one can argue that `on(`v_1`,`v_1`)` and `on(`v_2`,`v_2`)` will not appear in the $pre(\xi)$, $del(\xi)$ and $add(\xi)$ of any action model ξ because, in this specific domain, a block cannot be on

```
(:predicates (on ?x ?y) (ontable ?x) (clear ?x) (handempty) (holding ?x))

(:objects blockA blockB blockC)

(:init (ontable blockA) (clear blockA) (ontable blockB) (clear blockB)
       (ontable blockC) (clear blockC) (handempty))

(:observation (on blockA blockB) (on blockB blockC))
```

Fig. 2. Example of a two-state observationn for the learning STRIPS action models.

top of itself. Further we can use domain-specific knowledge to complete partially observed states.

State invariants is a useful type of state constraints for computing more compact state representations of a given planning problem [Helmert2009] and for making *satisfiability planning* or *backward search* more efficient [Rintanen2014,Alcázar and Torralba2015]. Given a planning problem $P = \langle F, A, I, G\rangle$, a state invariant is a formula ϕ that holds in I, $I \models \phi$, and in every state s built out of F that is reachable by applying actions of A in I. Recently, some works point at extracting *lifted* invariants, also called *schematic* invariants [Rintanen2017], that hold for any possible state and any possible set of objects. Invariant templates obtained by inspecting the lifted representation of the domain have also been exploited for deriving *lifted mutex* [Bernardini *et al.*2018]. *Schematic invariants* can identify mutually exclusive properties of a given type of objects [Fox and Long1998].

We define a schematic mutex as a $\langle p, q\rangle$ pair where both $p, q \in \mathcal{I}_{\xi,\Psi}$ are predicates that shape the preconditions or effects of a given action scheme ξ and they satisfy the formulae $\neg p \vee \neg q$, considering that their corresponding variables are universally quantified. For instance, $holding(v_1)$ and $clear(v_1)$ from the *blocksworld* are *schematic mutex* while $clear(v_1)$ and $ontable(v_1)$ are not because $\forall v_1, \neg clear(v_1) \vee \neg ontable(v_1)$ does not hold for every possible state. Figure 3 shows an example of four clauses that define schematic mutexes for the *blocksworld* domain.

$\forall x_1, x_2\ \neg ontable(x_1) \vee \neg on(x_1, x_2).$
$\forall x_1, x_2\ \neg clear(x_1) \vee \neg on(x_2, x_1).$
$\forall x_1, x_2, x_3\ \neg on(x_1, x_2) \vee \neg on(x_1, x_3)\ \textit{such that}\ x_2 \neq x_3.$
$\forall x_1, x_2, x_3\ \neg on(x_2, x_1) \vee \neg on(x_3, x_1)\ \textit{such that}\ x_2 \neq x_3.$

Fig. 3. *Schematic mutexes* for the *blocksworld* domain.

4 Learning Strips action models with classical planning

Our proposal to address a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ is to transform Λ into a planning task P_Λ. The intuition behind the compilation is that when P_Λ is solved with a planner, the solution plan π_Λ is a sequence of actions that build the action models of the output domain model $\mathcal{M}'$ and verify that $\mathcal{M}'$ is consistent with the actions and states of the observed plan trace $\tau = \langle s_0, \ldots, s_n \rangle$. Hence, π_Λ will comprise two differentiated blocks of actions: a first set of actions each defining the **insertion** of a fluent as a precondition, a positive effect or a negative effect of an action model $\xi \in \mathcal{M}'$; and a second set of actions that determine the **application** of the learned ξs while successively **validating** the effects of the action application in every observable point of τ, including that the final reached state comprises s_n. Roughly speaking, in the *blocksworld* domain, the format of the first set of actions of π_Λ will look like `(insert_pre_stack_holding_v1)`,`(insert_eff_stack_clear_v1)`,`(insert_eff_stack_clear_v2)`, where the first effect denotes a positive effect and the second one a negative effect to be inserted in $name(\xi)$ =`stack`; and the format of the second set of actions of π_Λ will be like `(apply_unstack blockB blockA)`,`(apply_putdown blockB)` and `(validate_1)`,`(validate_2)`, where the last two actions denote the points at which the states generated through the action application must be validated with the observed states of τ.

4.1 Compilation

Our compilation scheme builds upon the approach presented in [Aineto *et al.*2018] but comes up with a more general and flexible scheme able to capture any type of input plan trace.

A learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ is compiled into a planning task P_Λ with conditional effects in the context of a planning frame $\Phi = \langle F, A \rangle$. We use conditional effects because they allow us to compactly define actions whose effects depend on the current state. An action $a \in A$ with conditional effects is defined as a set of preconditions $\mathsf{pre}(a) \in \mathcal{L}(F)$ and a set of *conditional effects* $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two sets of literals $C \in \mathcal{L}(F)$, the *condition*, and $E \in \mathcal{L}(F)$, the *effect*. An action $a \in A$ is applicable in a state s if and only if $\mathsf{pre}(a) \subseteq s$, and the *triggered effects* resulting from the action application are the effects whose conditions hold in s; that is, $triggered(s,a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E$. The result of applying a in state s follows the same definition of successor state, $\theta(s,a)$, introduced in section **??** but applied to the conditional effects in $triggered(s,a)$.

A solution plan π_Λ to P_Λ induces the output domain model $\mathcal{M}'$ that solves the learning task Λ. Specifically, a solution plan π_Λ serves two purposes:

1. **To build the action models of $\mathcal{M}'$.** π_Λ comprises a first block of actions (plan *prefix*) that set the predicates $p \in \Psi_\xi$ of $pre(\xi)$, $del(\xi)$ and $add(\xi)$ for each $\xi \in \mathcal{M}$.

2. **To validate the action models of $\mathcal{M}'$.** π_Λ also comprises a second block of actions (plan *postfix*) which is aimed at validating of the observed plan trace τ with the built action models $\mathcal{M}'$.

Given a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$, with τ formed by an n-action sequence $\langle a_1, \ldots, a_n \rangle$ and a m-state trajectory $\langle s_0, s_1, \ldots, s_m \rangle$ ($\tau = \langle s_0, a_1, \ldots, a_n, s_m \rangle$), the compilation outputs a classical planning task $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$ such that:

– F_Λ extends F with the model fluents to represent the preconditions and effects of each $\xi \in \mathcal{M}$ as well as some other fluents to keep track of the validation of τ. Specifically, F_Λ contains:
  - The set of fluents obtained from s_0; i.e., F.
  - The model fluents $pre_p(\xi)$, $del_p(\xi)$ and $add_p(\xi)$, for every $\xi \in \mathcal{M}$ and $p \in \Psi_\xi$, defined as explained in section **??**
  - A set of fluents $F_\pi = \{plan(name(a_i), \Omega^{ar(a_i)}, i)\}_{1 \leq i \leq n}$ to represent the i^{th} observable action of τ. In the example of Figure **??**, the two observed actions `(putdown B)` and `(stack A B)` would be encoded as fluents `(plan-putdown B i1)` and `(plan-stack A B i2)` to indicate that `(putdown B)` is observed in the first place and `(stack A B)` is the second observed action.
  - Two fluents, at_i and $next_{i,i+1}$, $1 \leq i \leq n$, to iterate through the n observed actions of τ. The former is used to ensure that actions are executed in the same order as they are observed in τ. The latter is used to iterate to the next planning step when solving P_Λ.
  - A set of fluents $\{test_j\}_{0 \leq j \leq m}$, to point at the state observation $s_j \in \tau$ where the action model is validated. In the example of Figure **??** two tests are required to validate the programmed action model, one test at s_0 and another one at s_4.
  - A fluent, $mode_{prog}$, to indicate whether action models are being programmed or validated.

– I_Λ encodes s_0 and the following fluents set to true: $mode_{prog}$, $test_0$, F_π, at_1 and $\{next_{i,i+1}\}$, $1 \leq i \leq n$. Our compilation assumes that action models are initially programmed with no precondition, no negative effect and no positive effect.

– G_Λ includes the positive literals at_n and $test_m$. When these two goals are achieved by the solution plan π_Λ, we will be certain that the action models of $\mathcal{M}'$ are validated in all the actions and states observed in the input plan trace τ.

– A_Λ includes three types of actions that give rise to the actions of π_Λ.
  1. Actions for *inserting* a component (precondition, positive effect or negative effect) in $\xi \in \mathcal{M}$ following the syntactic constraints of STRIPS models. These actions will form the prefix of the solution plan π_Λ. Among the *inserting* actions, we find:
     - Actions which support the addition of a *precondition* $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$. A precondition p is inserted in ξ when neither pre_p, del_p nor add_p exist in ξ.

$$\begin{aligned}\mathsf{pre}(\mathsf{insertPre}_{\mathsf{p},\xi}) =& \{\neg pre_p(\xi), \neg del_p(\xi), \neg add_p(\xi), mode_{prog}\},\\ \mathsf{cond}(\mathsf{insertPre}_{\mathsf{p},\xi}) =& \{\emptyset\} \rhd \{pre_p(\xi)\}.\end{aligned}$$

- Actions which support the addition of a *negative* or *positive* effect $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$. A positive effect is inserted in ξ under the same conditions of a precondition insertion, and a negative effect is inserted in ξ when neither del_p nor add_p appear in ξ but pre_p does.

$$\begin{aligned}\mathsf{pre}(\mathsf{insertEff}_{\mathsf{p},\xi}) =& \{\neg del_p(\xi), \neg add_p(\xi), mode_{prog}\},\\ \mathsf{cond}(\mathsf{insertEff}_{\mathsf{p},\xi}) =& \{pre_p(\xi)\} \rhd \{del_p(\xi)\},\\ & \{\neg pre_p(\xi)\} \rhd \{add_p(\xi)\}.\end{aligned}$$

For instance, given $name(\xi)$ =`stack` and $C_{pre-stack} = \{$`(pre_stack_holding_v1)`,`(pre_stack_holding_v2)`, `(pre_stack_on_v1_v2)`,`(pre_stack_clear_v1)`,`(pre_stack_clear_v1)`,...$\}$, the insertion of each item $c \in C_{pre-stack}$ in ξ will generate a different alternative in the search space when solving P_Λ as long as $c \notin pre(\xi)$, $c \notin add(\xi)$ and $c \notin del(\xi)$. The same applies to effects with respect to sets $C_{add-stack}$ and $C_{del-stack}$ that would include all fluents starting with prefix `add` and `del`, respectively.

Note that executing an insert action, e.g.`(insert_pre_stack_holding_v1)`, will add the corresponding model fluent `(pre_stack_holding_v1)` to the successor state. Hence, the execution of the insert actions of π_Λ yield a state containing the valuation of the model fluents that shape every $\xi \in \mathcal{M}$. For example, executing the insert actions that shape the action model $name(\xi)$ =`putdown` leads to a state containing the positive literals `(pre_putdown_holding_v1)`,`(eff_putdown_holding_v1)`, `(eff_putdown_clear_v1)`, `(eff_putdown_ontable_v1)`,`(eff_putdown_handempty)`.

2. Actions for *applying* the action models $\xi \in \mathcal{M}$ built by the insert actions and bounded to objects $\omega \subseteq \Omega^{ar(\xi)}$. Since action headers are known, the variables $pars(\xi)$ are bounded to the objects in ω that appear in the same position.

$$\begin{aligned}\mathsf{pre}(\mathsf{apply}_{\xi,\omega}) =& \{pre_p(\xi) \implies p(\omega)\}_{\forall p \in \Psi_\xi},\\ \mathsf{cond}(\mathsf{apply}_{\xi,\omega}) =& \{del_p(\xi)\} \rhd \{\neg p(\omega)\}_{\forall p \in \Psi_\xi},\\ & \{add_p(\xi)\} \rhd \{p(\omega)\}_{\forall p \in \Psi \xi},\\ & \{mode_{prog}\} \rhd \{\neg mode_{prog}\}.\end{aligned}$$

These actions will be part of the postfix of the plan π_Λ and they determine the application of the learned action models according to the values of the model fluents in the current state configuration. Figure 4 shows the PDDL encoding of `(apply_stack)` for applying the action model of the *stack* operator.

Note that executing an apply action, e.g.`(apply_stack blockB blockA)`, will add the literals `(on blockB blockA)`,`(clear blockB)`,`(not(clear blockA))`,`(handempty)` and `(not(clear blockB))` to the successor state if $name(\xi)$ =`stack` has been correctly programmed by the insert actions. Hence, while **insert actions** add the values of the **model fluents** that shape ξ, the **apply actions** add the values of the **fluents of** F that result from the execution of ξ.

```
(:action apply_stack
  :parameters (?o1 - object ?o2 - object)
  :precondition
   (and (or (not (pre_stack_on_v1_v1)) (on ?o1 ?o1))
        (or (not (pre_stack_on_v1_v2)) (on ?o1 ?o2))
        (or (not (pre_stack_on_v2_v1)) (on ?o2 ?o1))
        (or (not (pre_stack_on_v2_v2)) (on ?o2 ?o2))
        (or (not (pre_stack_ontable_v1)) (ontable ?o1))
        (or (not (pre_stack_ontable_v2)) (ontable ?o2))
        (or (not (pre_stack_clear_v1)) (clear ?o1))
        (or (not (pre_stack_clear_v2)) (clear ?o2))
        (or (not (pre_stack_holding_v1)) (holding ?o1))
        (or (not (pre_stack_holding_v2)) (holding ?o2))
        (or (not (pre_stack_handempty)) (handempty)))
  :effect
   (and (when (del_stack_on_v1_v1) (not (on ?o1 ?o1)))
        (when (del_stack_on_v1_v2) (not (on ?o1 ?o2)))
        (when (del_stack_on_v2_v1) (not (on ?o2 ?o1)))
        (when (del_stack_on_v2_v2) (not (on ?o2 ?o2)))
        (when (del_stack_ontable_v1) (not (ontable ?o1)))
        (when (del_stack_ontable_v2) (not (ontable ?o2)))
        (when (del_stack_clear_v1) (not (clear ?o1)))
        (when (del_stack_clear_v2) (not (clear ?o2)))
        (when (del_stack_holding_v1) (not (holding ?o1)))
        (when (del_stack_holding_v2) (not (holding ?o2)))
        (when (del_stack_handempty) (not (handempty)))
        (when (add_stack_on_v1_v1) (on ?o1 ?o1))
        (when (add_stack_on_v1_v2) (on ?o1 ?o2))
        (when (add_stack_on_v2_v1) (on ?o2 ?o1))
        (when (add_stack_on_v2_v2) (on ?o2 ?o2))
        (when (add_stack_ontable_v1) (ontable ?o1))
        (when (add_stack_ontable_v2) (ontable ?o2))
        (when (add_stack_clear_v1) (clear ?o1))
        (when (add_stack_clear_v2) (clear ?o2))
        (when (add_stack_holding_v1) (holding ?o1))
        (when (add_stack_holding_v2) (holding ?o2))
        (when (add_stack_handempty) (handempty))
        (when (modeProg) (not (modeProg)))))
```

Fig. 4. PDDL action for applying an already programmed model for *stack* (implications are coded as disjunctions).

When the input plan trace contains observed actions, the extra conditional effects $\{at_i, plan(name(a_i), \Omega^{ar(a_i)}, i)\} \rhd \{\neg at_i, at_{i+1}\}_{\forall i\in[1,n]}$ are included in the $\mathsf{apply}_{\xi,\omega}$ actions to ensure that actions are applied in the same order as they appear in τ.

3. Actions for *validating* the partially observed state $s_j \in \tau$, $1 \leq j < m$. These actions are also part of the postfix of the solution plan π_Λ and they are aimed at checking that the observable data of the input plan trace τ follows after the execution of the apply actions.

$$\begin{aligned}\mathsf{pre}(\mathsf{validate_j}) =& s_j \cup \{test_{j-1}\},\\ \mathsf{cond}(\mathsf{validate_j}) =& \{\emptyset\} \rhd \{\neg test_{j-1}, test_j\}.\end{aligned}$$

There will be a validate action in π_Λ for every observed state in τ. The position of the validate actions in π_Λ will be determined by the planner by checking that the state resulting after the execution of an apply action comprises the observed state $s_j \in \tau$.

In some contexts, it is reasonable to assume that some parts of the action model are known and so there is no need to learn the entire model from scratch [Zhuo *et al.*2013]. In , when an action model ξ is partially specified, the known preconditions and effects are encoded as fluents $pre_p(\xi)$, $del_p(\xi)$ and $add_p(\xi)$ set to true in the initial state I_Λ. In this case, the corresponding insert actions, $\mathsf{insertPre_{p,\xi}}$ and $\mathsf{insertEff_{p,\xi}}$, become unnecessary and are removed from A_Λ, thereby making the classical planning task P_Λ easier to be solved.

So far we have explained the compilation for learning from a single input trace. However, the compilation is extensible to the more general case $\Lambda = \langle\mathcal{M}, \mathcal{T}\rangle$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_k\}$ is a set of plan traces. Taking this into account, a small modification is required in our compilation approach. In particular, the actions in P_Λ for *validating* the last state $s_m^t \in \tau_t$, $1 \leq t \leq k$ of a plan trace τ_t reset the current state and the current plan. These actions are now redefined as:

$$\begin{aligned}\mathsf{pre}(\mathsf{validate_j}) =& s_m^t \cup \{test_{j-1}\} \cup \{\neg mode_{prog}\},\\ \mathsf{cond}(\mathsf{validate_j}) =& \{\emptyset\} \rhd \{\neg test_{j-1}, test_j\} \cup\\ & \{\neg f\}_{\forall f \in s_m^t, f \notin s_0^{t+1}} \cup \{f\}_{\forall f \in s_0^{t+1}, f \notin s_m^t},\\ & \{\neg f\}_{\forall f \in F_{\pi_t}} \cup \{f\}_{\forall f \in F_{\pi_{t+1}}}.\end{aligned}$$

Finally, we will detail the composition of a solution plan π_Λ to a planning task P_Λ and the mechanism to extract the action models of $\mathcal{M}'$ from π_Λ. The plan of Figure 5 shows a solution to the task P_Λ that encodes a learning task $\Lambda = \langle\mathcal{M}, \tau\rangle$ for obtaining the action models of the *blocksworld* domain, where the models for `pickup`, `putdown` and `unstack` are already specified in $\mathcal{M}$. Therefore, the plan shows the insert actions and validate action for the action model `stack` using the input plan trace of Figure **??**. Plan steps $00-01$ insert the preconditions of the `stack` model, steps $02-06$ insert the action model effects, and steps $07-11$ form the plan postfix that applies the action models (only the `stack` model is learned) and validates the result in the plan trace of Figure **??**.

```
00 : (insert_pre_stack_holding_v1)
01 : (insert_pre_stack_clear_v2)
02 : (insert_eff_stack_clear_v1)
03 : (insert_eff_stack_clear_v2)
04 : (insert_eff_stack_handempty)
05 : (insert_eff_stack_holding_v1)
06 : (insert_eff_stack_on_v1_v2)
07 : (apply_unstack blockB blockA i1 i2)
08 : (apply_putdown blockB i2 i3)
09 : (apply_pickup blockA i3 i4)
10 : (apply_stack blockA blockB i4 i5)
11 : (validate_1)
```

Fig. 5. Plan for programming and validating the *stack* action model (using the plan trace τ of Figure **??**) as well as previously specified action models for *pickup*, *putdown* and *unstack*.

Given a solution plan π_Λ that solves P_Λ, the set of action models $\mathcal{M}'$ that solves $\Lambda = \langle \mathcal{M}, \tau \rangle$ are computed in linear time and space. In order to do so, π_Λ is executed in the initial state I_Λ and the action model $\mathcal{M}'$ will be given by the fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$ that are set to true in the last state reached by π_Λ, $s_g = \theta(I_\Lambda, \pi_\Lambda)$. For each $\xi \in \mathcal{M}'$, we build the sets of preconditions, positive effects and negative effects as follows:

$$\begin{aligned}
pre(\xi) =& \{p \mid pre_p(\xi) \in s_g\}_{\forall p \in \Psi_\xi},\\
add(\xi) =& \{p \mid add_p(\xi) \in s_g\}_{\forall p \in \Psi_\xi},\\
del(\xi) =& \{p \mid del_p(\xi) \in s_g\}_{\forall p \in \Psi_\xi}.
\end{aligned}$$

The logical inference process our approach is based on has trouble learning preconditions that do not appear as negative effects since in this case no change is observed between the pre-state and post-state of an action. This is specially relevant for static predicates that never change and, hence, only appear as preconditions in the actions. In order to address this shortcoming and complete the list of learned preconditions, we apply a post-process based on the one proposed in [Kucera and Barták2018]. The idea lies in going through every action and counting the number of cases where a literal is present before the action is executed and the number of cases where it is not present. If a literal is present in all the cases before the action, the literal is considered to be a precondition.

In order to obtain a complete trace, the proposal in [Kucera and Barták2018] applies the sequence of actions of the input trace and infers the preconditions from this action sequence. In our case, since the sequence of actions of the input trace might not be fully observable, we produce the traces by applying the actions found in the validation part of the solution plan. For instance, in the example of the figure 5, the sequence of actions used to produce the complete trace would be `(unstack blockB blockA)`, `(put-down blockB)`, `(pick-up blockA)`, and `(stack blockA blockB)`.

4.2 Properties of the compilation

Lemma 1. *Soundness. Any classical plan π that solves P' (planning task that results from the compilation) produces a model $\mathcal{M}'$ that solves the $\Lambda = \langle\mathcal{M}, \mathcal{O}, \Phi\rangle$ learning task.*

Proof (Proof). According to the P' compilation, once a given precondition or effect is inserted into the domain model $\mathcal{M}$ it cannot be undone. In addition, once an action model is applied it cannot be modified. In the compiled planning task P', only $(\texttt{apply})_{\xi,\omega}$ actions can update the value of the state fluents F. This means that a state consistent with an observation s^o_m can only be achieved executing an applicable sequence of $(\texttt{apply})_{\xi,\omega}$ actions that, starting in the corresponding initial state s^o_0, validates that every generated intermediate state s_j $(0 < j \leq m)$, is consistent with the input state observations and *state-invariants*. This is exactly the definition of the solution condition for model $\mathcal{M}'$ to solve the $\Lambda = \langle\mathcal{M}, \mathcal{O}, \Phi\rangle$ learning task.

Lemma 2. *Completeness. Any model $\mathcal{M}'$ that solves the $\Lambda = \langle\mathcal{M}, \mathcal{O}, \Phi\rangle$ learning task can be computed with a classical plan π that solves P'.*

Proof (Proof). By definition $\mathcal{I}_{\xi,\Psi}$ fully captures the set of elements that can appear in an action model ξ using predicates Ψ. In addition the P' compilation does not discard any model $\mathcal{M}'$ definable within $\mathcal{I}_{\xi,\Psi}$ that satisfies the mutexes in Φ. This means that, for every model $\mathcal{M}'$ that solves the $\Lambda = \langle\mathcal{M}, \mathcal{O}, \Phi\rangle$, we can build a plan π that solves P' by selecting the appropriate $(\texttt{insert\_pre})_{\xi,\texttt{e}}$ and $(\texttt{insert\_eff})_{\xi,\texttt{e}}$ actions for programming the precondition and effects of the corresponding action models in $\mathcal{M}'$ and then, selecting the corresponding $(\texttt{apply})_{\xi,\omega}$ actions that transform the initial state observation s^o_0 into the final state observation s^o_m.

The size of P' depends on the arity of the predicates in Ψ, that shape variables F, and the number of parameters of the action models, $|pars(\xi)|$. The larger these arities, the larger $|\mathcal{I}_{\xi,\Psi}|$. The size of $\mathcal{I}_{\xi,\Psi}$ is the most dominant factor of the compilation because it defines the pre_ξ_e/eff_ξ_e fluents, the corresponding set of insert actions, and the number of conditional effects in the $(\text{apply})_{\xi,\omega}$ actions. Note that *typing* can be used straightforward to constrain the FOL interpretations of Ψ over the parameters $pars(\xi)$, which will significantly reduce $|\mathcal{I}_{\xi,\Psi}|$ and hence the size of P' output by the compilation.

Classical planners tend to prefer shorter solution plans, so our compilation (as well as the BLS) may introduce a bias to $\Lambda = \langle\mathcal{M}, \mathcal{O}, \Phi\rangle$ learning tasks preferring solutions that are referred to action models with a shorter number of preconditions/effects. In more detail, all $\{pre\_\xi\_e, eff\_\xi\_e\}_{\forall e \in \mathcal{I}_{\xi,\Psi}}$ fluents are false at the initial state of our P' compilation so classical planners tend to solve P' with plans that require a smaller number of insert actions.

This bias can be eliminated defining a cost function for the actions in P' (e.g. insert actions have *zero cost* while $(\mathtt{apply})_{\xi,\omega}$ actions have a *positive constant cost*). In practice we use a different approach to disregard the cost of insert actions since classical planners are not proficient at optimizing plan cost with zero-cost actions. Instead, our approach is to use a SAT-based planner [Rintanen2014] that can apply all actions for inserting preconditions in a single planning step (these actions do not interact). Further, the actions for inserting action effects are also applied in another single planning step. The plan horizon for programming any action model is then always bounded to 2. The SAT-based planning approach is also convenient for its ability to deal with planning problems populated with dead-ends and because symmetries in the insertion of preconditions/effects into an action model do not affect the planning performance.

An interesting aspect of our approach is that when a *fully* or *partially specified* STRIPS action model $\mathcal{M}$ is given in Λ, the P_Λ compilation also serves to validate whether the observed τ follows the given model $\mathcal{M}$:

- $\mathcal{M}$ is proved to be a *valid* action model for the given input data in τ iff a solution plan for P_Λ can be found.
- $\mathcal{M}$ is proved to be a *invalid* action model for the given input data τ iff P_Λ is unsolvable. This means that $\mathcal{M}$ cannot be consistent with the given observation of the plan execution.

The validation capacity of our compilation is beyond the functionality of VAL (the plan validation tool [Howey *et al.*2004]) because our P_Λ compilation is able to address *model validation* of a partial (or even an empty) action model with a partially observed plan trace. VAL, however, requires a full plan and a full action model for plan validation.

5 Experimental results

6 Conclussions

References

[Aineto *et al.*2018] Diego Aineto, Sergio Jiménez, and Eva Onaindia. Learning STRIPS action models with classical planning. In *International Conference on Automated Planning and Scheduling, (ICAPS-18)*, pages 399--407. AAAI Press, 2018.

[Alcázar and Torralba2015] Vidal Alcázar and Alvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *ICAPS*, pages 2--6. AAAI Press, 2015.

[Bernardini *et al.*2018] Sara Bernardini, Fabio Fagnani, and David E. Smith. Extracting mutual exclusion invariants from lifted temporal planning domains. *Artificial Intelligence*, 258:1--65, 2018.

[Fox and Long1998] Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367--421, 1998.

[Helmert2009] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503--535, 2009.

[Howey *et al.*2004] Richard Howey, Derek Long, and Maria Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 294--301. IEEE, 2004.

[Kucera and Barták2018] Jirí Kucera and Roman Barták. LOUGA: learning planning operators using genetic algorithms. In *Pacific Rim Knowledge Acquisition Workshop, PKAW-18*, pages 124--138, 2018.

[McDermott *et al.*1998] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL -- The Planning Domain Definition Language, 1998.

[Ramírez and Geffner2009] Miquel Ramírez and Hector Geffner. Plan recognition as planning. In *International Joint conference on Artifical Intelligence, (IJCAI-09)*, pages 1778--1783. AAAI Press, 2009.

[Rintanen2014] Jussi Rintanen. Madagascar: Scalable planning with SAT. In *International Planning Competition, (IPC-2014)*, 2014.

[Rintanen2017] Jussi Rintanen. Schematic invariants by reduction to ground invariants. In *National Conference on Artificial Intelligence, AAAI-17*, pages 3644--3650, 2017.

[Zhuo *et al.*2013] Hankz Hankui Zhuo, Tuan Anh Nguyen, and Subbarao Kambhampati. Refining incomplete planning domain models through plan traces. In *International Joint Conference on Artificial Intelligence, IJCAI-13*, pages 2451--2458, 2013.