Explanation-based learning of action models

Diego Aineto[1], Sergio Jiménez[1], and Eva Onaindia[1]

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. The paper presents the first classical planning compilation for learning Strips action models from partial observations of plan executions. The compilation is flexible to different amount and kind of available input knowledge; learning examples can range from a set of plans (with their corresponding initial and final states, as well as partially observed intermediate states) to just pairs of initial and final states where no intermediate action/state is observed. The compilation accepts also partially specified action models and it can be used to validate whether the observation of a plan execution follows a given Strips action model, even if the given model or the given observation are incomplete.

Keywords: Learning action models · Classical planning.

1 Introduction

Besides *plan synthesis* [9], planning action models are also useful for *plan/goal recognition* [18]. In both tasks, off-the-shelf automated planners are required to reason about action models that correctly and completely capture the possible world transitions [8]. Unfortunately building planning action models is complex, even for planning experts, and this knowledge acquisition task is a bottleneck that limits the potential of AI planning [13].

Machine Learning (ML) has shown to be able to induce a wide range of different kinds of models from examples [16]. The application of inductive ML to learning Strips action models, the vanilla action model for planning [6], is not straightforward though:

- The *input* to ML algorithms (the learning/training data) is usually a finite vector that represents the value of some fixed object features. The input for learning planning action models is, however, observations of plan executions (where each plan possibly has a different length and plan length is not *a priori* bound).
- The *output* of ML algorithms is usually a scalar value (an integer, in the case of *classification* tasks, or a real value, in the case of *regression* tasks). When learning action models the output is, for each action, the set of preconditions and effects that define the possible state transitions of a planning task.

Learning STRIPS action models is a well-studied problem with sophisticated algorithms such as ARMS [25], SLAF [2] or LOCM [4], which do not require full knowledge of the intermediate states traversed by the example plans. Motivated by recent advances on the synthesis of different kinds of generative models with classical planning [3, 20, 21, 22], this paper describes an innovative planning compilation approach for learning STRIPS action models. The compilation approach is appealing by itself, because it opens up the door to the bootstrapping of planning action models, but also because it is flexible to different amount and kind of available input knowledge:

1. *Learning examples* can range from a set of plans (with their corresponding initial and final states, as well as partially observed intermediate states) to just a pair of initial and final states where no intermediate state/action is observed.
2. *Partially specified action models* expressing *a priori* knowledge about the structure of actions can also be provided to the compilation. In the extreme, the compilation can validate whether an observed plan execution is valid for a given STRIPS action model, even if the model is not fully specified or the observation is incomplete.

2 Background

This section formalizes the models we follow for *classical planning*, for the *observations* of executions of classical plans and for the explanation of a given observation.

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. Therefore $\rho(s, a)$ 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* $\tau(\pi, P) = \langle s_0, a_1, s_1, \dots, a_n, s_n\rangle$ 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 plan π solves P if the induced trajectory $\tau(\pi, P)$ holds that $G \subseteq s_n$. A solution plan is *optimal* iff its length is minimal.

Now we define actions with conditional effects because they allow us to compactly define our compilation. 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 $P = \langle F, A, I, G\rangle$ and a plan π, the *observation of the trajectory* $\tau(\pi, P)$ is a sequence of partial states that captures what is observed from the execution of π in P. Formally, $\mathcal{O}(\tau) = \langle s_0^o, s_1^o \dots, s_m^o\rangle$, $s_0^o = I$ is a sequence of possibly *partially observable states* (except for the initial state s_0^o which is fully observable). A partially observable state is one in which $|s_i^o| < |F|$, $1 \leq i \leq n$; i.e., a state in which at least a fluent of F is not observable.

The *observation model* comprises the case $|s_i^o| = 0$, when an intermediate state is fully unobservable. This model consider also *observed actions* as fluents that indicate the action applied in a given state. This means that a sequence of *observed actions* $\langle a_1^o, \dots, a_l^o\rangle$ is a sub-sequence of π s.t. $a_i^o \in s_i^o$ $(0 \leq i < m)$. The number of *observed actions*, l, can then range from 0 (in a fully unobservable action sequence) to $|\pi|$ (in a fully observed action sequence).

The sequence of observed states $\mathcal{O}(\tau)$ must be *consistent* with the sequence of states in $\tau(\pi, P)$. In practice, the number m of observed states ranges from 1 (the initial state, at least), to $|\pi| + 1$, and the observed intermediate states will comprise a number of fluents between $[1, |F|]$. This means that we assume a bijective monotone mapping between actions/states of trajectories and observations [17], 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 $(\theta(s_i^o, \langle a_1, \dots, a_k\rangle) = s_{i+1}^o$, where $k \geq 1$ is unknown but finite. In other words, having $\mathcal{O}(\tau)$ does not imply knowing the actual length of the trajectory τ.

This observation model can also distinguish between *observable state variables*, whose value may be read from sensors, and *hidden* (or *latent*) *state variables*, that cannot be observed. Given a subset of fluents $\Gamma \subseteq F$ we say that $\mathcal{O}(\tau)$ is a Γ-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 Γ.

2.3 Explaining observations with classical planning

Given a *classical planning frame* $\Phi = \langle F, A\rangle$ and a observation of the execution of a plan within the given planning frame $\mathcal{O} = \langle s_0^o, s_1^o \ldots, s_m^o\rangle$, then $P_{\mathcal{O}}$ is a classical planning problem that is built as follows $P_{\mathcal{O}} = \langle F, A, s_0^o, s_m^o\rangle$.

Definition 1 (Explanation). *We say that a plan π* explains *$\mathcal{O}$ (denoted $\pi \mapsto \mathcal{O}$) iff π is a solution for P that is* consistent *with the state trajectory constraints imposed by the sequence of partial states $\mathcal{O}$. If π is also optimal, we say that π is the* best explanation *for the input observation $\mathcal{O}(\tau)$.*

The *observation* $\mathcal{O}$ is then considered a sequence of ordered *landmarks* for the $P_{\mathcal{O}}$ classical planning problem, because all the literals in the observation must be achieved by any plan that solves $P_{\mathcal{O}}$ and in the same order as are defined in the observation [10].

3 Explanation-based learning of Strips action models

The task of learning action models by explaining the observation of a plan execution is defined as a tuple $\Lambda = \langle \mathcal{M}, \mathcal{O}\rangle$, where:

- $\mathcal{M}$ is the *initial empty model* that contains only the *header* (i.e., the corresponding *name* and *parameters*) of each action model to be learned.
- $\mathcal{O} = \langle s_0^o, s_1^o \ldots, s_m^o\rangle$ is a sequence of partially observed states.

A *solution* to a $\Lambda = \langle \mathcal{M}, \mathcal{O}\rangle$ learning task is a model $\mathcal{M}'$ s.t. it is consistent with the headers of $\mathcal{M}$ and that it explains $\mathcal{O}$. We say that a model *explains* an observation $\mathcal{O}$ iff, when the $\langle \rho, \theta\rangle$ functions of the actions in $P_{\mathcal{O}}$ are given by that model, there exists a solution plan for $P_{\mathcal{O}}$ that *explains* $\mathcal{O}$.

3.1 The space of Strips action models

We analyze here the solution space of the addressed learning task; i.e., the space of Strips action models.

A Strips *action model* is defined as $\xi = \langle name(\xi), pars(\xi), pre(\xi), add(\xi), del(\xi)\rangle$, where $name(\xi)$ and parameters, $pars(\xi)$, define the header of ξ; and $pre(\xi)$, $del(\xi)$ and $add(\xi))$ are sets of fluents that represent the *preconditions*, *negative effects* and *positive effects*, respectively, of the actions induced from the action model ξ.

Let Ψ be the set of *predicates* that shape the fluents F (the initial state of an observation is a full assignment of values to fluents, $|s_0^o| = |F|$, and so the predicates Ψ are extractable from the observed state s_0^o). The set of propositions that can appear in $pre(\xi)$, $del(\xi)$ and $add(\xi)$ of a given ξ, denoted as $\mathcal{I}_{\xi,\Psi}$, are FOL interpretations of Ψ over the parameters $pars(\xi)$. For instance, in a four-operator *blocksworld* [23], the $\mathcal{I}_{\xi,\Psi}$ set contains five elements for the `pickup`(v_1) model, $\mathcal{I}_{pickup,\Psi}$={`handempty`, `holding`($v_1$),`clear`($v_1$),`ontable`($v_1$), `on`($v_1, v_1$)} and eleven elements for the model of `stack(`v_1`,`v_2`)`, $\mathcal{I}_{stack,\Psi}$={`handempty`, `holding(`v_1`)`, `holding(`v_2`)`, `clear(`v_1`)`,`clear(`v_2`)`,`ontable(`v_1`)`,`ontable(`v_2`)`, `on(`v_1, v_1`)`,`on(`v_1, v_2`)`, `on(`v_2, v_1`)`, `on(`v_2, v_2`)`}. Hence, solving a $\Lambda = \langle \mathcal{M}, \mathcal{O} \rangle$ learning task is determining which elements of $\mathcal{I}_{\xi,\Psi}$ will shape the preconditions, positive and negative effects of each action model $\xi \in \mathcal{M}$.

In principle, for a given STRIPS 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 [15].
2. **Observation constraints**. The solution $\mathcal{M}'$ must be consistent with these *semantic constraints* derived from the input observation $\mathcal{O}$. 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. The belonging of a $p \in \mathcal{I}_{\Psi,\xi}$ to the preconditions, positive effects or negative effects of ξ is handled with a refined propositional encoding that uses fluents of two types, $pre_{p,\xi}$ and $eff_{p,\xi}$, instead of the three fluents used in the original compilation [1]. 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.

Encoding	**Meaning**
$\neg pre_{p,\xi} \wedge \neg eff_{p,\xi}$	p belongs neither to the preconditions nor effects of ξ $(p \notin pre(\xi) \wedge p \notin add(\xi) \wedge p \notin del(\xi))$
$pre_{p,\xi} \wedge \neg eff_{p,\xi}$	p is only a precondition of ξ $(p \in pre(\xi) \wedge p \notin add(\xi) \wedge p \notin del(\xi))$
$\neg pre_{p,\xi} \wedge eff_{p,\xi}$	p is a positive effect of ξ $(p \notin pre(\xi) \wedge p \in add(\xi) \wedge p \notin del(\xi))$
$pre_{p,\xi} \wedge eff_{p,\xi}$	p is a negative effect of ξ $(p \in pre(\xi) \wedge p \notin add(\xi) \wedge p \in add(\xi))$

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

To illustrate better this encoding, Figure 2 shows the PDDL encoding of the `stack(?v1,?v2)` schema and our propositional representation for this same schema with `pre_e_stack` and `eff_e_stack` fluents ($e \in \mathcal{I}_{\Psi,stack}$).

```
(:action stack
   :parameters (?v1 ?v2)
   :precondition (and (holding ?v1) (clear ?v2))
   :effect (and (not (holding ?v1)) (not (clear ?v2)) (clear ?v1) (handempty) (on ?v1 ?v2)))


(pre_holding_v1_stack) (pre_clear_v2_stack)
(eff_holding_v1_stack) (eff_clear_v2_stack)
(eff_clear_v1_stack) (eff_handempty_stack) (eff_on_v1_v2_stack)
```

Fig. 2. PDDL encoding of the `stack(?v1,?v2)` schema and our propositional representation for this same schema.

3.2 The sampling space

According to our *observation model* the minimal expression of an observation must comprise at least two state observations, a full initial state s_0^o and a partially observed final state s_m^o so $m \geq 1$. On the other hand, the maximal expression of an observation corresponds to a fully observed trajectory $\mathcal{O}(\tau) = \tau$ (meaning that all traversed states and applied actions are fully observed). In between there is a grey scale of different kinds of possible observations, including the observation of the initial state and the executed plan, that is frequently used for previous system that learn planning action models such as ARMS [25] or SLAF [2].

Figure 3 shows an example of an observation that contains only two states. 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 also a partially observable final state in which `blockA` is on top of `blockB` and `blockB` on top of `blockC`.

```
(: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. 3. Example of a two-state observation for the learning of Strips action models in the *blocksworld* domain.

4 Learning Strips action models with classical planning

Our approach to address a learning task $\Lambda = \langle \mathcal{M}, \mathcal{O} \rangle$ is to compile it into a planning task P_Λ. The intuition behind the compilation is that when P_Λ is solved, the solution plan π_Λ is a sequence of actions that build the output model $\mathcal{M}'$ and verify that $\mathcal{M}'$ is consistent with the observation $\mathcal{O}$.

A solution plan π_Λ comprise then two differentiated blocks of actions: a plan prefix with a set of actions each defining the **insertion** of a fluent as a precondition or a effect of an action model $\xi \in \mathcal{M}'$; and a plan postfix with a set of actions that determine the **application** of the learned modes while successively **validating** the effects of the action application in every partial state in $\mathcal{O}$. Roughly speaking, in the *blocksworld*, the format of the first set of actions of π_Λ looks like `(insert_pre_stack_holding_v1), (insert_eff_stack_clear_v1),(insert_eff_stack_holding_v1)`, 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 is 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 in $\mathcal{O}$.

4.1 Compilation

A learning task $\Lambda = \langle \mathcal{M}, \mathcal{O} \rangle$ is compiled into a classical planning problem with conditional effects P_Λ. 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 *explaining* the observation $\mathcal{O}$ with the built action models $\mathcal{M}'$.

Given a learning task $\Lambda = \langle \mathcal{M}, \mathcal{O} \rangle$ 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 $\mathcal{O}$. Specifically, F_Λ contains:
  - The set of fluents obtained from s_0^o; i.e., F.
  - The model fluents $pre_{p,\xi}$ and $eff_{p,\xi}$, for every $p \in \Psi_\xi$ and $\xi \in \mathcal{M}$, defined as explained in section 3.1.
  - A set of fluents $\{test_j\}_{0 \leq j \leq m}$, to point at the state observation $s_j^o \in \mathcal{O}$ where the action model is validated. In the example of Figure 3 two tests are required to validate the programmed action model, one corresponding to the initial state and the second one corresponding to the final state.

- A fluent, $mode_{prog}$, to indicate whether action models are being programmed or validated and a fluent `invalid` to indicate that the programmed action model is inconsistent with the input observation.

– I_Λ encodes s_0^o and the following fluents set to true: $mode_{prog}$, $test_0$. Our compilation assumes that action models are initially programmed with no precondition, no negative effect and no positive effect.

– G_Λ includes the positive literal $test_m$ and the negative literal $\neg$`invalid`. When these goals are achieved by the solution plan π_Λ, we will be certain that the action models of $\mathcal{M}'$ are validated in the input observation.

– A_Λ includes three types of actions that give rise to the actions of π_Λ.

1. Actions for *inserting* a component (precondition or 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 nor eff_p exist in ξ.

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

   - Actions which support the addition of a effect $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$.

$$\begin{aligned}\mathsf{pre}(\mathsf{insertEff}_{\mathsf{p},\xi}) =& \{\neg eff_{p,\xi}, mode_{prog}\},\\ \mathsf{cond}(\mathsf{insertEff}_{\mathsf{p},\xi}) =& \{\emptyset\} \rhd \{eff_{p,\xi}\}\end{aligned}$$

   For instance, given $name(\xi) =$`stack` and $F_{pre,stack} = \{$`(pre_stack_holding_v1)`, `(pre_stack_holding_v2)`, `(pre_stack_on_v1_v2)`,`(pre_stack_clear_v1)`, `(pre_stack_clear_v1)`, $\ldots\}$, the insertion of each item $p \in F_{pre,stack} \subseteq F_\Lambda$ in ξ will generate a different alternative in the search space when solving P_Λ. The same applies to effects $F_{eff,stack} = \{$`(eff_stack_holding_v1)`,`(eff_stack_holding_v2)`, `(eff_stack_on_v1_v2)`,`(eff_stack_clear_v1)`,`(eff_stack_clear_v1)`,$\ldots\}$.

   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 $\omega \subseteq \Omega^{ar(\xi)}$. 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. Since action headers are known, the variables $pars(\xi)$ are bounded to the objects in ω that appear in the same position.

$$\begin{aligned}
\mathsf{pre}(\mathsf{apply}_{\xi,\omega}) =& \{\},\\
\mathsf{cond}(\mathsf{apply}_{\xi,\omega}) =& \{pre_{p,\xi} \wedge eff_{p,\xi}\} \rhd \{\neg p(\omega)\}_{\forall p \in \Psi_\xi},\\
& \{\neg pre_{p,\xi} \wedge eff_{p,\xi}\} \rhd \{p(\omega)\}_{\forall p \in \Psi_\xi},\\
& \{pre_{p,\xi} \wedge \neg p(\omega)\}_{\forall p \in \Psi_\xi} \rhd \{invalid\},\\
& \{mode_{prog}\} \rhd \{\neg mode_{prog}\}.
\end{aligned}$$

Figure 4 shows the PDDL encoding of `(apply_stack)` for applying the action model of the *stack* operator. Let's assume the action `(apply_stack blockB blockA)` is in π_Λ. Executing this action in a state s implies activating the preconditions and effects of `(apply_stack)` according to the values of the model fluents in s. For example, if $\{$`(pre_stack_holding_v1)`,`(pre_stack_clear_v2)`$\} \subset s$ then it must be checked that positive literals `(holding blockB)` and `(clear blockA)` hold in s. Otherwise, a different set of precondition literals will be checked. The same applies to the conditional effects, generating the corresponding literals according to the values of the model fluents of s.
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.
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 $s_j \in \mathcal{O}$, $1 \leq j < m$. These actions are also part of the postfix of the solution plan π_Λ and they are aimed at checking that the observation $\mathcal{O}$ 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 $\mathcal{O}$. 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 $s_j \in \mathcal{O}$.

```
(:action apply_stack
  :parameters (?o1 - object ?o2 - object)
  :precondition (and )
  :effect (and (when (and (pre_stack_on_v1_v1) (eff_stack_on_v1_v1)) (not (on ?o1 ?o1)))
               (when (and (pre_stack_on_v1_v2) (eff_stack_on_v1_v2)) (not (on ?o1 ?o2)))
               (when (and (pre_stack_on_v2_v1) (eff_stack_on_v2_v1)) (not (on ?o2 ?o1)))
               (when (and (pre_stack_on_v2_v2) (eff_stack_on_v2_v2)) (not (on ?o2 ?o2)))
               (when (and (pre_stack_ontable_v1) (eff_stack_ontable_v1)) (not (ontable ?o1)))
               (when (and (pre_stack_ontable_v2) (eff_stack_ontable_v2)) (not (ontable ?o2)))
               (when (and (pre_stack_clear_v1) (eff_stack_clear_v1)) (not (clear ?o1)))
               (when (and (pre_stack_clear_v2) (eff_stack_clear_v2)) (not (clear ?o2)))
               (when (and (pre_stack_holding_v1) (eff_stack_holding_v1)) (not (holding ?o1)))
               (when (and (pre_stack_holding_v2) (eff_stack_holding_v2)) (not (holding ?o2)))
               (when (and (pre_stack_handempty) (eff_stack_handempty)) (not (handempty)))
               (when (and (not (pre_stack_on_v1_v1)) (eff_stack_on_v1_v1)) (on ?o1 ?o1))
               (when (and (not (pre_stack_on_v1_v2)) (eff_stack_on_v1_v2)) (on ?o1 ?o2))
               (when (and (not (pre_stack_on_v2_v1)) (eff_stack_on_v2_v1)) (on ?o2 ?o1))
               (when (and (not (pre_stack_on_v2_v2)) (eff_stack_on_v2_v2)) (on ?o2 ?o2))
               (when (and (not (pre_stack_ontable_v1)) (eff_stack_ontable_v1)) (ontable ?o1))
               (when (and (not (pre_stack_ontable_v2)) (eff_stack_ontable_v2)) (ontable ?o2))
               (when (and (not (pre_stack_clear_v1)) (eff_stack_clear_v1)) (clear ?o1))
               (when (and (not (pre_stack_clear_v2)) (eff_stack_clear_v2)) (clear ?o2))
               (when (and (not (pre_stack_holding_v1)) (eff_stack_holding_v1)) (holding ?o1))
               (when (and (not (pre_stack_holding_v2)) (eff_stack_holding_v2)) (holding ?o2))
               (when (and (not (pre_stack_handempty)) (eff_stack_handempty)) (handempty))
               (when (and (pre_stack_on_v1_v1) (not (on ?o1 ?o1))) (invalid))
               (when (and (pre_stack_on_v1_v2) (not (on ?o1 ?o2))) (invalid))
               (when (and (pre_stack_on_v2_v1) (not (on ?o2 ?o1))) (invalid))
               (when (and (pre_stack_on_v2_v2) (not (on ?o2 ?o2))) (invalid))
               (when (and (pre_stack_ontable_v1) (not (ontable ?o1))) (invalid))
               (when (and (pre_stack_ontable_v2) (not (ontable ?o2))) (invalid))
               (when (and (pre_stack_clear_v1) (not (clear ?o1))) (invalid))
               (when (and (pre_stack_clear_v2) (not (clear ?o2))) (invalid))
               (when (and (pre_stack_holding_v1) (not (holding ?o1))) (invalid))
               (when (and (pre_stack_holding_v2) (not (holding ?o2))) (invalid))
               (when (and (pre_stack_handempty) (not (handempty))) (invalid))
               (when (modeProg) (not (modeProg)))))
```

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

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 [26]. In our compilation approach, when an action model ξ is partially specified, the known preconditions and effects are encoded as fluents $pre_{p,\xi}$ and $eff_{p,\xi}$ set to true in the initial state I_Λ. In this case, the corresponding insert actions, $\mathsf{insertPre}_{\mathsf{p},\xi}$ and $\mathsf{insertEff}_{\mathsf{p},\xi}$, become unnecessary and are removed from A_Λ, thereby making the classical planning task P_Λ easier to be solved.

So far we 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 $\mathcal{T} = \{\mathcal{O}_1, \ldots, \mathcal{O}_k\}$ is a set of k observations. 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 $s^o_{m,t} \in \mathcal{O}_t$, $1 \leq t \leq k$ of an observation $\mathcal{O}_t$ reset the current state. These actions are now redefined as:

$$\begin{aligned}\mathsf{pre}(\mathsf{validate_j}) =& s^o_{m,t} \cup \{test_{j-1}\} \cup \{\neg mode_{prog}\},\\ \mathsf{cond}(\mathsf{validate_j}) =& \{\emptyset\} \rhd \{\neg test_{j-1}, test_j\} \cup \\ & \{\neg f\}_{\forall f \in s^o_{m,t}, f \notin s^o_{0,t+1}} \cup \{f\}_{\forall f \in s^o_{0,t+1}, f \notin s^o_{m,t}}.\end{aligned}$$

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 $\Lambda = \langle \mathcal{M}, \mathcal{O} \rangle$ 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`. 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 input observation.

```
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 the *stack* action model and for validating the programmed *stack* action model with 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 $\Lambda = \langle \mathcal{M}, \mathcal{O} \rangle$ learning task 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 $pre_{p,\xi}$, and $eff_{p,\xi}$ 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:

$$\begin{aligned}pre(\xi) =& \{p \mid pre_{p,\xi} \in s_g\}_{\forall p \in \Psi_\xi},\\ del(\xi) =& \{p \mid pre_{p,\xi} \wedge eff_{p,\xi} \in s_g\}_{\forall p \in \Psi_\xi},\\ add(\xi) =& \{p \mid \neg pre_{p,\xi} \wedge eff_{p,\xi} \in s_g\}_{\forall p \in \Psi_\xi}.\end{aligned}$$

The plain compilation 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. To address this shortcoming and complete the list of learned preconditions, we apply a post-process based on the one proposed by the LOUGA system [14]. The intuition is going through every action counting the number of cases where a literal is present before the action is executed. If a literal is present in all the cases before the action, the literal is considered to be a precondition. Since intermmediate states/actions may not be fully observed in a observation $\mathcal{O}$, we consider the actions/states found in the validation part of the solution plan π_Λ. For instance, in the example of Figure 5, the used sequence of actions is `(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_Λ produces a model $\mathcal{M}'$ that solves the $\Lambda = \langle\mathcal{M}, \mathcal{O}\rangle$ learning task.*

Proof (Proof). According to the P_Λ 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 problem P_Λ, 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_m^o can only be achieved executing an applicable sequence of $(\texttt{apply})_{\xi,\omega}$ actions that, starting in the corresponding initial state s_0^o, validates that every generated intermediate state s_j $(0 < j \leq m)$, is consistent with the input state observations. This is exactly the definition of the solution condition for model $\mathcal{M}'$ to solve the $\Lambda = \langle\mathcal{M}, \mathcal{O}\rangle$ learning task.

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

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_Λ compilation does not discard any model $\mathcal{M}'$ definable within $\mathcal{I}_{\xi,\Psi}$. This means that, for every model $\mathcal{M}'$ that solves the $\Lambda = \langle\mathcal{M}, \mathcal{O}\rangle$, we can build a plan π that solves P_Λ by selecting the appropriate $(\texttt{insert\_pre})_{\texttt{p},,\xi}$ and $(\texttt{insert\_eff})_{\texttt{p},\xi}$ 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_0^o into the final state observation s_m^o.

The size of the classical planning proble P_Λ 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 $pre_{p,\xi}/eff_{p,\xi}$ fluents, the corresponding set of `insert` actions, and the number of conditional effects in the $(\texttt{apply})_{\xi,\omega}$ 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_Λ output by the compilation.

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

This bias can be eliminated defining a cost function for the actions in P_Λ (e.g. `insert` actions have *zero cost* while $(\texttt{apply})_{\xi,\omega}$ 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 [19] 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 observatoin $\mathcal{O}$ follows the given model $\mathcal{M}$:

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

This validation capacity of our compilation is beyond the functionality of VAL (the plan validation tool [11]) 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

7 Conclusions

We presented a novel approach for learning STRIPS action models from examples using classical planning. To the best of our knowledge, this is the first approach on learning action models that is exhaustively evaluated over a wide range of domains and uses exclusively an *off-the-shelf* classical planner. The work in [24] proposes a planning compilation for learning action models from plan traces following the *finite domain* representation for the state variables. This is a theoretical study on the boundaries of the learned models and no experimental results are reported.

When example plans are available, we can compute accurate action models from small sets of learning examples (five examples per domain) in little computation time (less than a second). When action plans are not available, our approach still produces action models that are compliant with the input information. In this case, since learning is not constrained by actions, operators can be reformulated changing their semantics, in which case the comparison with a reference model turns out to be tricky.

An interesting research direction related to this issue is *domain reformulation* to use actions in a more efficient way, reduce the set of actions identifying dispensable information or exploiting features that allow more compact solutions like the *reachable* or *movable* features in the *Sokoban* domain [12].

Generating *informative* examples for learning planning action models is still an open issue. Planning actions include preconditions that are only satisfied by specific sequences of actions which have low probability of being chosen by chance [5]. The success of recent algorithms for exploring planning tasks [7] motivates the development of novel techniques that enable to autonomously collect informative learning examples. The combination of such exploration techniques with our learning approach is an appealing research direction that opens up the door to the bootstrapping of planning action models.

Acknowledgments

This work is supported by the Spanish MINECO project TIN2017-88476-C2-1-R. Diego Aineto is partially supported by the *FPU16/03184* and Sergio Jiménez by the *RYC15/18009*, both programs funded by the Spanish government.

Bibliography

[1] 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.

[2] Eyal Amir and Allen Chang. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 33:349–402, 2008.

[3] Blai Bonet, Héctor Palacios, and Héctor Geffner. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In *International Conference on Automated Planning and Scheduling, (ICAPS-09)*. AAAI Press, 2009.

[4] Stephen N Cresswell, Thomas L McCluskey, and Margaret M West. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28(02):195–213, 2013.

[5] Alan Fern, Sung Wook Yoon, and Robert Givan. Learning Domain-Specific Control Knowledge from Random Walks. In *International Conference on Automated Planning and Scheduling, ICAPS-04*, pages 191–199. AAAI Press, 2004.

[6] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[7] Guillem Francès, Miquel Ramírez, Nir Lipovetzky, and Hector Geffner. Purely declarative action descriptions are overrated: Classical planning with simulators. In *International Joint Conference on Artificial Intelligence, (IJCAI-17)*, pages 4294–4301, 2017.

[8] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013.

[9] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[10] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[11] 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.

[12] Franc Ivankovic and Patrik Haslum. Optimal planning with axioms. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[13] Subbarao Kambhampati. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *National Conference on Artificial Intelligence, (AAAI-07)*, 2007.

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

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

[16] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

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

[18] Miquel Ramírez. *Plan recognition as planning*. PhD thesis, Universitat Pompeu Fabra, 2012.

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

[20] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generating Context-Free Grammars using Classical Planning. In *International Joint Conference on Artificial Intelligence, (IJCAI-17)*, pages 4391–4397. AAAI Press, 2017.

[21] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Computing hierarchical finite state controllers with classical planning. *Journal of Artificial Intelligence Research*, 62:755–797, 2018.

[22] Javier Segovia-Aguas, Sergio Jiménez Celorrio, and Anders Jonsson. Computing programs for generalized planning using a classical planner. *Artificial Intelligence*, 2019.

[23] John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

[24] Roni Stern and Brendan Juba. Efficient, safe, and probably approximately complete learning of action models. In *International Joint Conference on Artificial Intelligence, (IJCAI-17)*, pages 4405–4411, 2017.

[25] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3):107–143, 2007.

[26] 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.