Learning action models from *state-invariants*

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

[1]Departamento de Sistemas Informáticos y Computación. Universitat Politècnica de València. Valencia, Spain

{dieaigar,serjice,onaindia}@dsic.upv.es

Abstract

This paper addresses the learning of STRIPS action models from *state-invariants* (i.e logic formulae that specify constraints about the possible states of a given domain). The benefit of exploiting *state-invariants* is two-fold, they constrain the space of possible action models and they can complete learning examples that are only partially observed. Our approach for learning STRIPS action from *state-invariants* is a *classical planning* compilation that is flexible to different kinds of input knowledge (e.g., partially observations of plan executions including partially observed intermediate states and/or actions). The experimental results demonstrate that, even at unfavorable scenarios where input observations are minimal (just an *initial state* and the *goals*), *state-invariant* are helpful to learn good quality STRIPS action models.

1 Introduction

The specification of planning action models is a complex process that limits, too often, the application of *model-based planning* to real-world tasks [Kambhampati, 2007]. The *machine learning* of action models relieves this *knowledge acquisition bottleneck* of *model-based planning* and nowadays, there exists a wide range of effective approaches for learning action models [Arora *et al.*, 2018]. Many of the most successful approaches for learning planning action models are however purely *inductive* [Yang *et al.*, 2007; Pasula *et al.*, 2007; Mourao *et al.*, 2010; Zhuo and Kambhampati, 2013], linking learning performance exclusively to the *amount* and *quality* of the input learning examples.

This paper addresses the learning of action models exploiting a different source of knowledge, *deductive* knowledge. Our approach leverages *state-invariants*, i.e. logic formulae that specify constraints about the possible states of a given domain, to cushion the negative impact of insufficient learning examples. Given an action model, state-of-the-art planners infer *state-invariants* from that model to reduce the search space and make the planning process more efficient [Helmert, 2009]. In this paper we follow the opposite direction and leverage *state-invariants* to learn the planning action model. The benefit of learning action models from *state-invariants* is two-fold, *state-invariants* constrain the space of possible action models and can complete learning examples that are only partially observed.

Our approach for learning STRIPS action models from *state-invariants* is compile the learning task into a classical planning task. Our compilation is flexible to different kinds of input knowledge (e.g., partially/fully observations of actions of plan executions as well as partially/fully observed intermediate states) and outputs an action model that is *consistent* with the given input knowledge. The experimental results demonstrate that, even at unfavorable scenarios where input observations are minimal (just an *initial state* and the *goals*), *state-invariant* help to learn better STRIPS models.

2 Background

This section formalizes the *classical planning model* we follow in this work and introduces the classical planning compilation for the learning of STRIPS action models [Aineto *et al.*, 2018]. Finally, the section formalizes *state-invariants*.

2.1 Classical planning with conditional effects

Let F be the set of propositional state variables (*fluents*) describing a state. A *literal* l is a valuation of a fluent $f \in F$; i.e. either $l = f$ or $l = \neg f$. A set of literals L represents a partial assignment of values to fluents (without loss of generality, we will assume that L does not contain conflicting values). Given L, let $\neg L = \{\neg l : l \in L\}$ be its complement. We use $\mathcal{L}(F)$ to denote the set of all literal sets on F; i.e. all partial assignments of values to fluents. A *state* s is a full assignment of values to fluents; $|s| = |F|$.

A *classical planning action* $a \in A$ has: a precondition $\mathsf{pre}(a) \in \mathcal{L}(F)$, a set of effects $\mathsf{eff}(a) \in \mathcal{L}(F)$, and a positive action cost $cost(a)$. 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 $\mathsf{pre}(a) \subseteq s$, i.e. if its precondition holds in s. The result of executing an applicable action $a \in A$ in a state s is a new state $\theta(s, a) = (s \setminus \neg\mathsf{eff}(a)) \cup \mathsf{eff}(a)$. Subtracting the complement of $\mathsf{eff}(a)$ from s ensures that $\theta(s, a)$ remains a well-defined state. The subset of action effects that assign a positive value to a state fluent is called *positive effects* and denoted by $\mathsf{eff}^+(a) \in \mathsf{eff}(a)$ while $\mathsf{eff}^-(a) \in \mathsf{eff}(a)$ denotes the *negative effects* of an action $a \in A$.

A *classical planning problem* is a tuple $P = \langle F, A, I, G\rangle$, where I is the initial state and $G \in \mathcal{L}(F)$ is the set of goal conditions over the state variables. A *plan* π is an action sequence $\pi = \langle a_1, \ldots, a_n\rangle$, with $|\pi| = n$ denoting its *plan length* and $cost(\pi) = \sum_{a\in\pi} cost(a)$ its *plan cost*. The execution of π on the initial state 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 plan π solves P iff the induced *trajectory* $\tau(\pi, P)$ reaches a final state $G \subseteq s_n$, where all goal conditions are met. A solution plan is *optimal* iff its cost is minimal.

We also define *actions with conditional effects* because they are useful to compactly formulate our approach for *goal recognition with unknown domain models*. An action $a_c \in A$ with conditional effects is 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 is applicable in a state s if $\rho(s, a_c)$ is true, and the result of applying action a_c in state s is $\theta(s, a_c) = \{s \setminus \neg\mathsf{eff}_c(s, a) \cup \mathsf{eff}_c(s, a)\}$ where $\mathsf{eff}_c(s, a)$ are the *triggered effects* resulting from the action application (conditional effects whose conditions hold in s):

$$\mathsf{eff}_c(s, a) = \bigcup_{C\rhd E\in\mathsf{cond}(a_c), C\subseteq s} E,$$

2.2 Learning action models with classical planning

This work is built on top of the *classical planning compilation* for the learning STRIPS action models [Aineto *et al.*, 2018]. This compilation receives as input an empty model $\mathcal{M}$ (which only contains the action headers), and an observation of a plan execution $\mathcal{O}(\tau)$ (extensible to a set of observations). The compilation outputs a model $\mathcal{M}'$ that specifies the preconditions and effects of each action schema included in $\mathcal{M}$ such that the validation of $\mathcal{O}(\tau)$ following $\mathcal{M}'$ is successful; i.e., it holds $\rho(s^o_{i-1}, a_i)$ for every observed action of $\mathcal{O}(\tau)$ and $s^o_i = \theta(s^o_{i-1}, a_i)$ for every observed state of $\mathcal{O}(\tau)$.

Essentially, a solution plan to the classical planning problem that results from the compialtion is a sequence of: (a) *insert actions* that insert preconditions and effects on the schemata of $\mathcal{M}$ to build $\mathcal{M}'$ and (b) *apply actions* that validate the application of the $\mathcal{M}'$ model in the observation $\mathcal{O}(\tau)$.

To illustrate this, Figure 1 shows a solution to a classical planning problem resulting from the Aineto *et al.* 2018 compilation. In the initial state of that problem three blocks (`blockA`, `blockB` and `blockC`) are clear and on top of the table, the robot hand is empty. The problem goal is having the three-block tower `blockA` on top of `blockB` and `blockB` on top of `blockC`. The plan shows the *insert* actions for the `stack` scheme (steps $00 - 01$ insert the preconditions, steps $05 - 10$ insert the effects), steps $02 - 04$ insert the preconditions of the `pickup` scheme (while steps $10 - 13$ insert the effects of this scheme). Finally, steps $14 - 17$ is the plan postfix that applies the programmed action model to achieve the goals G starting from I.

2.3 State-invariants

The notion of *state-constraint* is very general and has been used in different areas of AI and for different purposes. If we restrict ourselves to planning, *state-constraints* are abstractions for compactly specifying sets of states. For example, *state-constraints* in planning allow to specify the set of states where a given action is applicable, the set of states where a given *derived predicate* holds or the set of states that are considered goal states.

```
00:(insert_pre_stack_holding_v1)     10:(insert_eff_pickup_clear_v1)
01:(insert_pre_stack_clear_v2)       11:(insert_eff_pickup_ontable_v1)
02:(insert_pre_pickup_handempty)     12:(insert_eff_pickup_handempty)
03:(insert_pre_pickup_clear_v1)      13:(insert_eff_pickup_holding_v1)
04:(insert_pre_pickup_ontable_v1)    14:(apply_pickup blockB)
05:(insert_eff_stack_clear_v1)       15:(apply_stack blockB blockC)
06:(insert_eff_stack_clear_v2)       16:(apply_pickup blockA)
07:(insert_eff_stack_handempty)      17:(apply_stack blockA blockB)
08:(insert_eff_stack_holding_v1)     18:(validate_1)
09:(insert_eff_stack_on_v1_v2)
```

Figure 1: Plan computed when solving a problem output by the classical planning compilation.

State invariants is a kind of state-constraints useful for computing more compact state representations [Helmert, 2009] or making *satisfiability planning* and *backward search* more efficient [Rintanen, 2014; Alcázar and Torralba, 2015]. Given a classical planning problem $P = \langle F, A, I, G\rangle$, a *state invariant* is a formula ϕ that holds at the initial state of a given classical planning problem, $I \models \phi$, and at every state s, built from F, that is reachable from I by applying actions in A. For instance, Figure 2 shows five clauses that define *state invariants* for the *blocksworld* planning domain [Slaney and Thiébaux, 2001].

$\forall x_1, x_2\ ontable(x_1) \leftrightarrow \neg on(x_1, x_2)$.
$\forall x_1, x_2\ clear(x_1) \leftrightarrow \neg on(x_2, x_1)$.
$\forall x_1, x_2, x_3\ \neg on(x_1, x_2) \vee \neg on(x_1, x_3)$ *such that* $x_2 \neq x_3$.
$\forall x_1, x_2, x_3\ \neg on(x_2, x_1) \vee \neg on(x_3, x_1)$ *such that* $x_2 \neq x_3$.
$\forall x_1, \ldots, x_n\ \neg(on(x_1, x_2) \wedge on(x_2, x_3) \wedge \ldots \wedge on(x_{n-1}, x_n) \wedge on(x_n, x_1))$.

Figure 2: Example of *state-invariants* for the *blocksworld* domain.

A *mutex* (mutually exclusive) is a state invariant that takes the form of a binary clause and indicates a pair of different properties that cannot be simultaneously true [Kautz and Selman, 1999]. For instance in a three-block *blocksworld*, $\neg on(block_A, block_B) \vee \neg on(block_A, block_C)$ is a *mutex* because $block_A$ can only be on top of a single block.

A *domain invariant* is an instance-independent state-invariant, i.e. holds for any possible initial state and any possible set of objects. Therefore, if a given state s holds $s \not\models \phi$ such that ϕ is a *domain invariant*, it means that s is not a valid state. Domain invariants are often compactly defined as *lifted invariants* (also called schematic invariants) [Rintanen and others, 2017].

We define a *domain mutex* as a $\langle p, q\rangle$ predicates pair where both $p \in \Psi$ and $q \in \Psi$ are predicates that shape the set of fluents F of a given planning problem and such that they satisfy the following formulae $p \rightarrow \neg q$ where the predicate variables are universally quantified. For instance, predicates $holding(x)$ and $clear(x)$ from the *blocksworld* are *domain mutex* while predicates $clear(x)$ and $ontable(x)$ are not ($\forall x\ clear(x) \leftrightarrow \neg ontable(x)$ does not hold for every possible *blocksworld* state). We pay special attention to *domain* *mutex* because they identify the *properties* of a given type of objects [Fox and Long, 1998] and because they enable (1) effectively pruning of inconsistent STRIPS action models and (2) effective completion of partially observed states.

3 Learning STRIPS action models from *state-invariants*

First, this section defines the sampling space and the space of possible action models. Then, the section formalizes the task of learning STRIPS action models from *state-invariants*.

3.1 The sampling space

We define a *learning example* $\mathcal{O} = \langle s_0^o, s_1^o \ldots, s_m^o \rangle$ as a sequence of partial states, except for the initial state s_0^o, which is fully observed. 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. Each *observed* states comprises $[1, |F|]$ fluents. The observation can still miss intermediate states that are *unobserved* so transiting between two consecutive observed states in $\mathcal{O}$ 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.

Our sampling space follows the *open world* assumption, i.e. what is not observed is considered unknown. *State-invariants* are helpful to infer new knowledge that was unobserved in the learning examples. Given a *domain mutex* $\langle p, q \rangle$ and a state observation $s_j^o \in \mathcal{O}(\tau)$, $(1 \leq j \leq m)$, such that literal $p(\omega) \in s_j^o$ is an instantiation of predicate p over some subset of objects $\omega \subseteq \Omega^{|pars(p)|}$ then, the state observation s_j^o can be safely completed adding the new literal $\neg q(\omega)$ (despite $\neg q(\omega)$ was actually unobserved). For instance, if the literal `holding(blockA)` is observed in a particular blocksword state and we know the *domain mutex* $\forall x \; holding(x) \leftrightarrow \neg clear(x)$ we can safely extend the observation with literal $\neg$`clear(blockA)` (despite this literal was actually unobserved).

3.2 The space of STRIPS action models

A STRIPS *action schema* ξ is defined by: A list of *parameters* $pars(\xi)$, and three sets of predicates (namely $pre(\xi)$, $del(\xi)$ and $add(\xi)$) that shape the kind of fluents that can appear in the *preconditions*, *negative effects* and *positive effects* of the actions induced from that schema. Let be Ψ the set of *predicates* that shape the propositional state variables F, and a list of *parameters*, $pars(\xi)$. The set of elements that can appear in $pre(\xi)$, $del(\xi)$ and $add(\xi)$ of the STRIPS action schema ξ is the set of FOL interpretations of Ψ over the parameters $pars(\xi)$ and is denoted as $\mathcal{I}_{\Psi,\xi}$.

For instance in a four-operator *blocksworld* [Slaney and Thiébaux, 2001], the $\mathcal{I}_{\Psi,\xi}$ set contains only five elements for the `pickup(`v_1`)` schemata, $\mathcal{I}_{\Psi,pickup}$={`handempty`, `holding(`$v_1$`)`, `clear(`$v_1$`)`, `ontable(`$v_1$`)`, `on(`$v_1,v_1$`)`} while it contains eleven elements for the `stack(`v_1,v_2`)` schemata, $\mathcal{I}_{\Psi,stack}$={`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`)`}.

```
(: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)
```

Figure 3: PDDL encoding of the `stack(?v1,?v2)` schema and our propositional representation for this same schema.

Despite any element of $\mathcal{I}_{\Psi,\xi}$ can *a priori* appear in the $pre(\xi)$, $del(\xi)$ and $add(\xi)$ of schema ξ, in practice the actual space of possible STRIPS schemata is bounded by constraints of two kinds:

1. **Syntactic constraints**. STRIPS constraints require $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ and $pre(\xi) \cap add(\xi) = \emptyset$. Considering exclusively these syntactic constraints, the size of the space of possible STRIPS schemata is given by $2^{2 \times |\mathcal{I}_{\Psi,\xi}|}$. *Typing constraints* are also of this kind [McDermott *et al.*, 1998].
2. **Observation constraints**. The observation of the actions and states resulting from the execution of a plan depicts *semantic knowledge* that constraints further the space of possible action schemata.

In this work we introduce a novel propositional encoding of the *preconditions*, *negative*, and *positive* effects of a STRIPS action schema ξ that uses only fluents of two kinds `pre_e_`ξ and `eff_e_`ξ (where $e \in \mathcal{I}_{\Psi,\xi}$). This encoding exploits the syntactic constraints of STRIPS so it is more compact that the one previously proposed by Aineto *et al.* 2018 for learning STRIPS action models with classical planning. In more detail, if `pre_e_`ξ holds it means that $e \in \mathcal{I}_{\Psi,\xi}$ is a *precondition* in ξ. If `pre_e_`ξ and `eff_e_`ξ holds it means that $e \in \mathcal{I}_{\Psi,\xi}$ is a *negative effect* in ξ while if *pre_e_*ξ does not hold but `eff_e_`ξ holds, it means that $e \in \mathcal{I}_{\Psi,\xi}$ is a *positive effect* in ξ. Figure 3 shows the PDDL encoding of the `stack(?v1,?v2)` schema and our propositional representation for this same schema using the `pre_e_stack` and `eff_e_stack` fluents ($e \in \mathcal{I}_{\Psi,stack}$).

One can also introduce *domain-specific knowledge* to constrain further the space of possible schemata. For instance, in the *blocksworld* 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 $add(\xi)$ lists of an action schema ξ because, in this specific domain, a block cannot be on top of itself. *State invariants* are *domain-specific knowledge* that can be seen either as *syntactic* or *semantic* constraints. On the one hand, *state invariants* constrain the space of possible action models but on the other hand, they can complete partial observations of the states traversed by a plan.

3.3 The learning task

We define the task of learning a planning action model from *state-invariants* as a tuple $\Lambda = \langle \mathcal{M}, \mathcal{O}, \Phi \rangle$, where:

- $\mathcal{M}$ is the *initial empty model* that contains only the name, $name(\xi)$, and parameters, $pars(\xi)$, of each action model ξ to be learned.
- $\mathcal{O} = \langle s_0^o, s_1^o \ldots, s_m^o \rangle$ is a single learning example. This example can be reduced to its minimal expression $\mathcal{O}^* = \langle s_0^o, s_m^o \rangle$ meaning that it only contains two state observations, a full initial state s_0^o and a partially observed state s_m^o. Note that the set of predicates Ψ is deducible from $\mathcal{O}$ since s_0^o is a fully observed state.
- Φ is a set of *state-invariants* that define constraints about the set of possible states.

M is the *space of possible action models* for the $A[\cdot]$ actions (i.e., the set of possible specifications of the ρ and/or θ functions for each $a \in A[\cdot]$ action). We say that a given model $\mathcal{M} \in M$ is a *solution* to the $\Lambda = \langle \mathcal{M}, \mathcal{O}, \Phi \rangle$ learning task iff there exists a plan π that solves $P = \langle F, A[\cdot], I, G \rangle$, when the semantics of each action $a \in A[\cdot]$ is given by $\mathcal{M}$, and such that any state traversed by a trajectory $\tau(\pi, P)$ is *consistent* with the input set of *state-invariants* Φ.

Now, we show how to exploit our compact encoding of STRIPS action models to solve a $\Lambda = \langle \mathcal{M}, \mathcal{O}, \Phi \rangle$ learning task with an off-the-shelf classical planner.

4 Learning action models from *state-invariants* with classical planning

Given a $\Lambda = \langle \mathcal{M}, \mathcal{O}, \Phi \rangle$ learning task we build and solve a classical planning problem $P_\Lambda = \langle F_\Lambda, A_\Lambda, I, G_\Lambda \rangle$ such that:

- F_Λ extends F with a fluent $mode_{inval}$, to indicate whether an action model is *inconsistent* with the input *state-invariants* Φ, a fluent $mode_{insert}$, to indicate whether action models are being programmed, and the fluents for the propositional encoding of the corresponding space of STRIPS action models. As explained, this is a set of fluents of the type $\{pre\_e\_\xi, eff\_e\_\xi\}_{\forall e \in \mathcal{I}_{\Psi,\xi}}$.
- $G_\Lambda = G \cup \{\neg mode_{inval}\}$ extends the original goals G with the $\neg mode_{inval}$ literal to validate that only states *consistent* with the state constraints Φ are traversed by P_Λ solutions.
- A_Λ replaces the actions in A with two types of actions.
  1. Actions for *inserting* a *precondition*, *positive* effect or *negative* effect in ξ following the syntactic constraints of STRIPS models.
     - Actions which support the addition of a *precondition* $p \in \mathcal{I}_{\Psi,\xi}$ to the action model ξ. A precondition p is inserted in ξ when neither pre_p, eff_p exist in ξ.

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

     - Actions which support the addition of a *negative* or *positive* effect $p \in \mathcal{I}_{\Psi,\xi}$ to the action model ξ.

$$\begin{aligned}\mathsf{pre}(\mathsf{insertEff}_{\mathsf{p},\xi}) =& \{\neg eff\_p\_\xi, mode_{insert}\},\\ \mathsf{cond}(\mathsf{insertEff}_{\mathsf{p},\xi}) =& \{\emptyset\} \rhd \{eff\_p\_\xi\}.\end{aligned}$$

  2. Actions for *applying* an action model ξ built by the *insert* actions and bounded to objects $\omega \subseteq \Omega^{|pars(\xi)|}$ (where Ω is the set of *objects* used to induce the fluents F by assigning objects in Ω to the Ψ predicates, and Ω^k is the k-th Cartesian power of Ω). The action parameters, $pars(\xi)$, are bound to the objects in ω that appear in the same position. These actions validate also that any state traversed by P_Λ solutions is *consistent* with the *state-invariants* Φ. The definition $\mathsf{apply}_{\xi,\omega}$ actions is also more compact in our compilation that the one previously proposed by Aineto *et al.* 2018 since are not using disjunctions to code the possible preconditions of an action schema.

$$\begin{aligned}\mathsf{pre}(\mathsf{apply}_{\xi,\omega}) =& \{\neg mode_{inval}\},\\ \mathsf{cond}(\mathsf{apply}_{\xi,\omega}) =& \{pre\_p\_\xi \wedge eff\_p\_\xi\} \rhd \{\neg p(\omega)\}_{\forall p \in \mathcal{I}_{\Psi,\xi}},\\ & \{\neg pre\_p\_\xi \wedge eff\_p\_\xi\} \rhd \{p(\omega)\}_{\forall p \in \mathcal{I}_{\Psi,\xi}},\\ & \{pre\_p\_\xi \wedge \neg p(\omega)\} \rhd \{mode_{inval}\}_{\forall p \in \mathcal{I}_{\Psi,\xi}},\\ & \{\neg \phi\} \rhd \{mode_{inval}\}_{\forall \phi \in \Phi},\\ & \{\emptyset\} \rhd \{\neg mode_{insert}\},\end{aligned}$$

4.1 Pruning inconsistent action models with *domain mutex*

Our approach to implement this pruning is extending the conditional effects of the $\mathsf{insertPre}_{\mathsf{p},\xi}$ and $\mathsf{insertPre}_{\mathsf{p},\xi}$ actions (i.e., the actions that determine a solution model $\mathcal{M}$) with extra conditional effects indicating that the programmed model is *invalid* (i.e., inconsistent with a *domain mutex* in Φ). Note that this *consistency* checking is more effective than the one implemented at the $\mathsf{apply}_{\xi,\omega}$ actions since $\mathsf{insertPre}_{\mathsf{p},\xi}$ and $\mathsf{insertPre}_{\mathsf{p},\xi}$ actions appear at an earlier stage of the planning process.

Formally, given a *domain mutex* (p, q), s.t. both p and q belong to $\in \mathcal{I}_{\Psi,\xi}$, we extend the actions for setting a precondition p in a given action schema ξ as follows:

$$\begin{aligned}\mathsf{pre}(\mathsf{insertPre}_{\mathsf{p},\xi}) =& \{\neg pre_p(\xi), \neg eff_p(\xi),\\ & mode_{insert}, \neg mode_{inval}\},\\ \mathsf{cond}(\mathsf{insertPre}_{\mathsf{p},\xi}) =& \{\emptyset\} \rhd \{pre_p(\xi)\},\\ & \{pre_q(\xi)\} \rhd \{mode_{inval}\}.\end{aligned}$$

The same procedure is applied for action $insertPre_{q,\xi}$ to ban programming precondition q iff $pre_p(\xi)$ precondition is already set. A similar procedure is also applied to $\mathsf{insertEff}_{\mathsf{p},\xi}$ and $\mathsf{insertEff}_{\mathsf{q},\xi}$ actions for banning in this case, two *negative effects* (or two *positive effects*) that are *domain mutex*. Now we show the actions that ban programming a positive (or negative) p effect if its corresponding q effect is already programmed:

$$\begin{aligned}\mathsf{pre}(\mathsf{insertEff}_{\mathsf{p},\xi}) =& \{\neg eff_p(\xi), mode_{insert}, \neg mode_{inval}\},\\ \mathsf{cond}(\mathsf{insertEff}_{\mathsf{p},\xi}) =& \{\emptyset\} \rhd \{eff_p(\xi),\\ & \{pre_q(\xi), eff_q(\xi), pre_p(\xi)\} \rhd \{mode_{inval}\},\\ & \{\neg pre_q(\xi), eff_q(\xi), \neg pre_p(\xi)\} \rhd \{mode_{inval}\}.\end{aligned}$$

4.2 The bias of the initially *empty* action model

Classical planners tend to preffer shorter solution plans, so our compilation may introduce a bias to $P = \langle F, A[\cdot], I, G\rangle$ problems preferring solutions that are referred to action models with a shorter number of *preconditions/effects*. In more detail, all $\{pre\_e\_\xi, eff\_e\_\xi\}_{\forall e \in \mathcal{I}_{\Psi,\xi}}$ fluents are false at the initial state of our $P' = \langle F', A', I, G\rangle$ compilation so classical planners tend to solve P' with plans that require a shorter number of *insert* actions.

This bias could be eliminated defining a cost function for the actions in A' (e.g. *insert* actions has *zero cost* while $\mathsf{apply}_{\xi,\omega}$ actions has a *positive constant cost*). In practice we use a different approach to disregard the cost of *insert* actions because classical planners are not proficiency optimizing *plan cost* with zero-cost actions. Instead, our approach is to use a SAT-based planner [Rintanen, 2014] because it 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 a single planning step so the plan horizon for programming any action model is always bound to 2, which significantly reduces the planning horizon.

Our compilation for *planning with unknown domain models* can then be understood as an extension of the SATPLAN approach for classical planning [Kautz *et al.*, 1992] with two additional initial layers: a first layer for inserting the action preconditions and a second one for inserting the action effects. These two extra layers are followed by the typical N layers of the SATPLAN encoding (extended however to apply the action models that are determined by the previous two initial layers, the $\mathsf{apply}_{\xi,\omega}$ actions). Regarding again the example of Figure 1, this means that steps [00-04] are applied in paralel in the first SATPLAN layer, steps [05-13] are applied in paralel in the second layer and each step [14-17] is applied sequentially and correponds to a differerent SATPLAN layer (so just six layers are necesary to compute the example plan of Figure 1).

The SAT-based planning approach is also convenient because its ability to deal with classical planning problems populated with dead-ends and because symmetries in the insertion of preconditions/effects into an action model do not affect to the planning performance.

4.3 Compilation properties

Lemma 1. *Soundness. Any classical plan π_Λ that solves P_Λ produces a* STRIPS *model $\mathcal{M}$ that solves the $\Lambda = \langle P, \Phi\rangle$ learning task.*

Proof. According to the P_Λ compilation, once a given precondition or effect is inserted into the action model $\mathcal{M}$ it cannot be removed back. In addition, once the action model $\mathcal{M}$ is applied it cannot be *reprogrammed*. In the compiled planning problem P_Λ, the value of the original fluents F can exclusively be modified via $\mathsf{apply}_{\xi,\omega}$ actions. Therefore, the goals of the original P classical planning task can only be achieved executing an applicable sequence of $\mathsf{apply}_{\xi,\omega}$ actions that, starting in the corresponding initial state $I = s_0$ reach a state $G \subseteq s_n$ validating that every generated intermediate state s_i, s.t. $0 \leq i \leq n$, is consistent with the input *state-invariants*. This is exactly the definition of the solution condition for an action model $\mathcal{M}$ to solve the $\Lambda = \langle P, \Phi, M\rangle$ learning task. □

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

Proof. By definition $\mathcal{I}_{\Psi,\xi}$ fully captures the set of elements that can appear in a STRIPS action schema ξ using predicates Ψ. In addition the P_Λ compilation does not discard any possible action model $\mathcal{M}$ definable within $\mathcal{I}_{\Psi,\xi}$ while it can satisfy the domain mutex in Φ. This means that for every STRIPS model $\mathcal{M}$ that solves the $\Lambda = \langle P, \Phi, M\rangle$, we can build a plan π_Λ that solves P_Λ by selecting the appropriate $\mathsf{insertPre}_{\mathsf{p},\xi}$ and $\mathsf{insertEff}_{\mathsf{p},\xi}$ actions for *programming* the precondition and effects of the corresponding action model $\mathcal{M}$ and then, selecting the corresponding $\mathsf{apply}_{\xi,\omega}$ actions that transform the initial state I into a state that satisfies the goals G. □

The size of the classical planning task P_Λ output by our compilation depends on the arity of the given *predicates* Ψ, that shape the propositional state variables F, and the number of parameters of the action models, $|pars(\xi)|$. The larger these arities, the larger $|\mathcal{I}_{\Psi,\xi}|$. The size of the $\mathcal{I}_{\Psi,\xi}$ set is the term that dominates the compilation size because it defines the pre_e_ξ/eff_e_ξ fluents, the corresponding set of *insert* actions, and the number of conditional effects in the $\mathsf{apply}_{\xi,\omega}$ actions. Note that *typing* can be used straightforward to constrain the FOL interpretations of Ψ over the parameters $pars(\xi)$ which significantly reduces $|\mathcal{I}_{\Psi,\xi}|$ and hence, the size of the classical planning task output by the compilation.

5 Learning from observations of plan executions

Inductive approaches for the learning of planning action models compute an action model starting from an input set of observations of plan executions. This section provides a formal model for such input observations and shows how to leverage *state-invariants* to automatically *complete* those input observations. The section ends with the extension of our compilation to exploit the *completed* observations for the learning of STRIPS action models.

6 Evaluation

7 Related work

State-invariants have been previously used to infer a HTN lanning model [Lotinac and Jonsson, 2016].

In *Inductive Logic Programming* it is very common to make the hypothesis be consistent with some form deductive knowledge apart from the examples, what is usually called *background knowledge* [Muggleton and De Raedt, 1994].

8 Conclusions

In some contexts it is however reasonable to assume that the action model is not learned from scratch, e.g. because some parts of the action model are known [Zhuo *et al.*, 2013; Sreedharan *et al.*, 2018; Pereira and Meneguzzi, 2018]. Our compilation approach is also flexible to this particular learning scenario. The known preconditions and effects are encoded setting the corresponding fluents

$\{pre\_e\_\xi, eff\_e\_\xi\}_{\forall e \in \mathcal{I}_{\Psi,\xi}}$ to true in the initial state. Further, the corresponding insert actions, $\mathsf{insertPre}_{\mathsf{p},\xi}$ and $\mathsf{insertEff}_{\mathsf{p},\xi}$, become unnecessary and are removed from A_Λ, making the classical planning task P_Λ easier to be solved. For example, suppose that the preconditions of the *blocksworld* action schema `stack` are known, then the initial state I is extended with literals, `(pre_holding_v1_stack)` and `(pre_clear_v2_stack)` and the associated actions $\mathsf{insertPre}_{\mathsf{holding_v1,stack}}$ and $\mathsf{insertPre}_{\mathsf{clear_v2,stack}}$ can be safely removed from the A_Λ action set without altering the *soundness* and *completeness* of the P_Λ compilation.

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 Torralba, 2015] 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.

[Arora *et al.*, 2018] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty. A review of learning planning action models. *The Knowledge Engineering Review*, 2018.

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

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

[Kambhampati, 2007] 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.

[Kautz and Selman, 1999] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325, 1999.

[Kautz *et al.*, 1992] Henry A Kautz, Bart Selman, et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer, 1992.

[Lotinac and Jonsson, 2016] Damir Lotinac and Anders Jonsson. Constructing hierarchical task models using invariance analysis. In *ECAI*, pages 1274–1282, 2016.

[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.

[Mourao *et al.*, 2010] Kira Mourao, Ronald PA Petrick, and Mark Steedman. Learning action effects in partially observable domains. In *ECAI*, pages 973–974. Citeseer, 2010.

[Muggleton and De Raedt, 1994] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

[Pasula *et al.*, 2007] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[Pereira and Meneguzzi, 2018] Ramon Fraga Pereira and Felipe Meneguzzi. Heuristic approaches for goal recognition in incomplete domain models. *arXiv preprint arXiv:1804.05917*, 2018.

[Rintanen and others, 2017] Jussi Rintanen et al. Schematic invariants by reduction to ground invariants. In *AAAI*, pages 3644–3650, 2017.

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

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

[Sreedharan *et al.*, 2018] Sarath Sreedharan, Tathagata Chakraborti, and Subbarao Kambhampati. Handling model uncertainty and multiplicity in explanations via model reconciliation. In *International Conference on Automated Planning and Scheduling, (ICAPS-18)*, pages 518–526, 2018.

[Yang *et al.*, 2007] 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.

[Zhuo and Kambhampati, 2013] Hankz Hankui Zhuo and Subbarao Kambhampati. Action-model acquisition from noisy plan traces. In *International Joint Conference on Artificial Intelligence, IJCAI-13*, pages 2444–2450, 2013.

[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.