

# Learning STRIPS action models with classical planning

Diego Aineto and Sergio Jiménez and Eva Onaindia

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

This paper presents a novel approach for learning STRIPS action models from examples that compiles this particular inductive learning task into a classical planning task. Interestingly, the compilation approach is flexible to different amounts of available input knowledge; the learning examples can range from a set of plans (with their corresponding initial and final states) to just a set of initial and final states where neither actions nor intermediate states are given. What is more, the compilation accepts partially specified action models and can also be used to validate whether certain observations of plan executions follow a given STRIPS action model (even if the action model is not fully specified).

## Introduction

Besides *plan synthesis* (Ghallab, Nau, and Traverso 2004; Geffner and Bonet 2013), planning action models are also useful for *plan/goal recognition* (Ramírez 2012). At both planning tasks, an automated planner is required to reason about action models that correctly and completely capture the possible world transitions. Unfortunately, building planning action models is complex, even for planning experts, and this knowledge acquisition task is a bottleneck that limits the potential of automated planning (Kambhampati 2007).

On the other hand, Machine Learning (ML) has shown to be able to compute a wide range of different kinds of models from examples (Michalski, Carbonell, and Mitchell 2013). The application of inductive ML to the learning of STRIPS action models, the vanilla action model for planning (Fikes and Nilsson 1971), is not straightforward though:

- The inputs to ML algorithms (the learning data) usually are finite vectors that encode the value of fixed features for a given set of objects. The input for learning planning action models are observations of plan executions (where each plan possibly has a different length).
- The traditional output of ML algorithms is a scalar value (an integer, in the case of *classification* tasks, or a real value, in the case of *regression* tasks). When learning STRIPS action models the output is, for each action, the sets of preconditions, negative and positive effects, that

define the possible state transitions of the corresponding planning domain.

The learning of STRIPS action models is a well-studied problem with sophisticated algorithms, like ARMS (Yang, Wu, and Jiang 2007), SLAF (Amir and Chang 2008) or LOCM (Cresswell, McCluskey, and West 2013) that do not require full knowledge of the intermediate states traversed by the example plans. Motivated by recent advances on learning different kinds of generative models with classical planning (Bonet, Palacios, and Geffner 2009; Segovia-Aguas, Jiménez, and Jonsson 2016; 2017), this paper introduces an innovative approach for learning STRIPS action models that can be defined as a classical planning compilation, so an off-the-shelf planner can address the learning task. The compilation approach is appealing because it opens the door to the bootstrapping of planning action models but also because:

1. Is flexible to different amounts of available input knowledge. The learning examples can range from a set of plans (with their corresponding initial and final states) to just a set of initial and final states where no intermediate state or action is observed.
2. Accepts previous knowledge about the structure of the actions in the form of partially specified action models. In the extreme, the compilation could be used to validate whether observed plan executions are valid for a given STRIPS action model.

The paper is organized as follows. The first section presents the classical planning model, its extension to conditional effects (which is a requirement of the proposed compilation) and formalizes the STRIPS action model (the output of the addressed learning task). The second section formalizes the learning of STRIPS action models with regard to different amounts of available input knowledge. The third and fourth sections describe our approach for addressing this particular learning task by compiling it into classical planning and how previous knowledge can be introduced in the compilation to constrain the space of possible action models and make learning more practicable. Finally, the last sections report the data collected during the empirical evaluation of our approach, discuss the strengths and weaknesses of our approach and propose several opportunities for future research.

## Background

This section defines the planning models used on this work as well as the STRIPS action model, the output of the learning task addressed in this paper.

### Classical planning

We use  $F$  to denote the set of *fluents* (propositional variables) 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 (WLOG we assume that  $L$  does not assign conflicting values to any fluent) and let  $\neg L = \{\neg l : l \in L\}$  be the complement of  $L$ . 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 total assignment of values to fluents, i.e.  $|s| = |F|$ , so the size of the state space is  $2^{|F|}$ . Explicitly including negative literals  $\neg f$  in states simplifies subsequent definitions but often, we abuse notation by defining a state  $s$  only in terms of the fluents that are true in  $s$ , as is common in STRIPS planning.

A *classical planning frame* is a tuple  $\Phi = \langle F, A \rangle$ , where  $F$  is a set of fluents and  $A$  is a set of actions. Each action  $a \in A$  has a set of literals  $\text{pre}(a) \in \mathcal{L}(F)$ , called *preconditions*, a set of *effects*  $\text{eff}(a) \in \mathcal{L}(F)$ . An action  $a \in A$  is *applicable* in state  $s$  iff  $\text{pre}(a) \subseteq s$ , and the result of applying  $a$  in  $s$  is the *successor state*  $\theta(s, a) = (s \setminus \neg \text{eff}(a)) \cup \text{eff}(a)$ .

A *classical planning problem* is a tuple  $P = \langle F, A, I, G \rangle$ , where  $I$  is an initial state and  $G \in \mathcal{L}(F)$  is a goal condition. A *plan* for  $P$  is an action sequence  $\pi = \langle a_1, \dots, a_n \rangle$  that induces a state sequence  $\langle s_0, s_1, \dots, s_n \rangle$  such that  $s_0 = I$  and, for each  $1 \leq i \leq n$ ,  $a_i$  is applicable in  $s_{i-1}$  and generates the successor state  $s_i = \theta(s_{i-1}, a_i)$ . A plan  $\pi$  *solves*  $P$  iff  $G \subseteq s_n$ , i.e. if the goal condition is satisfied at the last state reached after following the application of  $\pi$  in  $I$ .

### Classical planning with conditional effects

Our approach for learning STRIPS action models is compiling this learning task into a classical planning task with conditional effects. Conditional effects allow us to compactly define actions whose particular effects depend on the current state. Nowadays many classical planners cope with conditional effects without compiling them away. In fact, the support of conditional effects was a requirement for participating at the IPC-2014 (Vallati et al. 2015) and IPC-2018.

Now an action  $a \in A$  has a set of literals  $\text{pre}(a) \in \mathcal{L}(F)$ , called the *precondition*, and a set of conditional effects  $\text{cond}(a)$ . Each conditional effect  $C \triangleright E \in \text{cond}(a)$  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 state  $s$  if and only if  $\text{pre}(a) \subseteq s$ , and the resulting set of *triggered effects* is

$$\text{eff}(s, a) = \bigcup_{C \triangleright E \in \text{cond}(a), C \subseteq s} E,$$

i.e. effects whose conditions hold in  $s$ . The result of applying an action  $a$  in a state  $s$  is the *successor state*  $\theta(s, a) = (s \setminus \neg \text{eff}(s, a)) \cup \text{eff}(s, a)$ .

```
(:action stack
:parameters (?x1 ?x2 - object)
:precondition (and (holding ?x1)
                   (clear ?x2))
:effect (and (not (holding ?x1))
             (not (clear ?x2))
             (handempty)
             (clear ?x1)
             (on ?x1 ?x2)))
```

Figure 1: Example of a STRIPS operator schema that corresponds to the *stack* action from the blocksworld represented in PDDL.

### The STRIPS action model and the *variable* objects

This work addresses the learning of action schemes defined in the STRIPS language (Fikes and Nilsson 1971). To formalized the output of this learning task, we assume that there exists a set of *predicates*  $\Psi$  and that the fluents in  $F$  are instantiated from these predicates, as in PDDL (McDermott et al. 1998; Fox and Long 2003).

Formally, each predicate  $p \in \Psi$  has an argument list of arity  $\text{ar}(p)$ . Given a set of objects  $\Omega$ , the set of fluents  $F$  is then induced by assigning objects in  $\Omega$  to the arguments of predicates in  $\Psi$ , i.e.  $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{\text{ar}(p)}\}$  s.t.  $\Omega^k$  is the  $k$ -th Cartesian power of  $\Omega$ .

Likewise, we assume that STRIPS actions are instantiated from STRIPS operator schemes. Figure 1 shows the STRIPS operator schema that corresponds to the *stack* action from the blocksworld (Slaney and Thiébaux 2001) represented in PDDL.

Let  $\Omega_v$  be a new set of objects,  $\Omega \cap \Omega_v = \emptyset$ , that represents *variable names* and whose cardinality,  $|\Omega_v|$ , is given by the maximum arity of an action in a given planning frame  $\Phi$ . For instance, in a three-block blocksworld  $\Omega = \{\text{block}_1, \text{block}_2, \text{block}_3\}$  while  $\Omega_v = \{v_1, v_2\}$  because operators *stack* and *unstack*, with the maximum arity, have two parameters each.

Let us define also a new set of fluents  $F_v$  that results instantiating  $\Psi$  but using only the *variable objects* in  $\Omega_v$ . In the blocksworld  $F_v = \{\text{handempty}, \text{holding}(v_1), \text{holding}(v_2), \text{clear}(v_1), \text{clear}(v_2), \text{ontable}(v_1), \text{ontable}(v_2), \text{on}(v_1, v_1), \text{on}(v_1, v_2), \text{on}(v_2, v_1), \text{on}(v_2, v_2)\}$ .

We are now ready to define a STRIPS action model  $\Xi$  as a set of operator schema  $\xi = \langle \text{head}(\xi), \text{pre}(\xi), \text{add}(\xi), \text{del}(\xi) \rangle$  such that:

- $\text{head}(\xi) = \langle \text{name}(\xi), \text{pars}(\xi) \rangle$ , is the operator *header* defined by its name and a enumeration of the variable objects bound by the operator arity,  $\text{pars}(\xi) \in \Omega_v^{\text{ar}(\xi)}$ . The headers for a 4-operator blocksworld are  $\text{pickup}(v_1)$ ,  $\text{putdown}(v_1)$ ,  $\text{stack}(v_1, v_2)$  and  $\text{unstack}(v_1, v_2)$ .
- The preconditions  $\text{pre}(\xi) \subseteq F_v$ , the negative effects  $\text{del}(\xi) \subseteq F_v$ , and the positive effects  $\text{add}(\xi) \subseteq F_v$  such that,  $\text{del}(\xi) \subseteq \text{pre}(\xi)$ ,  $\text{del}(\xi) \cap \text{add}(\xi) = \emptyset$  and  $\text{pre}(\xi) \cap \text{add}(\xi) = \emptyset$ .

```

;;; Predicates in  $\Psi$ 
(handempty) (holding ?o - object)
(clear ?o - object) (ontable ?o - object)
(on ?o1 - object ?o2 - object)

;;; Plan  $\pi_1$ 
0: (unstack A B)
1: (putdown A)
2: (unstack B C)
3: (stack B A)
4: (unstack C D)
5: (stack C B)
6: (pickup D)
7: (stack D C)

;;; Label  $\sigma_1 = (s_0^1, s_n^1)$ 

```

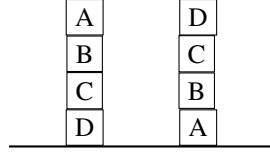


Figure 2: Example of a task for learning a STRIPS action model in the blockworld.

## Learning STRIPS action models

Learning STRIPS action models from fully available input knowledge, i.e. from plans where the *pre-* and *post-states* of every action in a plan are available, is straightforward. In this case, a STRIPS operator schema is derived by lifting the literals that change between the pre and post-state of the corresponding action executions. Likewise, the preconditions are derived lifting the minimal set of literals that appears in all the pre-states that correspond to the same operator schema.

This section formalizes more challenging learning tasks, where less input knowledge is available.

## Learning from labeled plans

This learning task is formalized as  $\Lambda = \langle \Psi, \Pi, \Sigma \rangle$ :

- $\Psi$ , the set of predicates that define the abstract state space of a given planning domain.
- $\Pi = \{\pi_1, \dots, \pi_\tau\}$ , the given set of example plans where each plan  $\pi_t = \langle a_1^t, \dots, a_n^t \rangle$ ,  $1 \leq t \leq \tau$ , is an action sequence that induces the corresponding state sequence  $\langle s_0^t, s_1^t, \dots, s_n^t \rangle$  such that, for each  $1 \leq i \leq n$ ,  $a_i^t$  is applicable in the state  $s_{i-1}^t$  and generates the successor state  $s_i^t = \theta(s_{i-1}^t, a_i^t)$ .
- $\Sigma = \{\sigma_1, \dots, \sigma_\tau\}$ , is a set of labels s.t. each plan  $\pi_t$ ,  $1 \leq t \leq \tau$ , has an associated label  $\sigma_t = (s_0^t, s_n^t)$  where  $s_n^t$  is the state resulting from executing the plan  $\pi_t$  starting from the state  $s_0^t$ .

Figure 2 shows an example of a learning task  $\Lambda$  in the blockworld (the figure shows the content of  $\Psi$ ,  $\Pi$  and  $\Sigma$ ). This learning task has a single learning example,  $\Pi = \{\pi_1\}$  and  $\Sigma = \{\sigma_1\}$ , that corresponds to observing the execution of a plan for inverting a four-block tower.

A solution to a learning task  $\Lambda$  is a set of STRIPS operator schema  $\Xi$  (with one schema  $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi) \rangle$  for each action with a different name in the example plans  $\Pi$ ) that is compliant with the predicates in  $\Psi$ , the example plans  $\Pi$ , and their corresponding labels  $\Sigma$ .

## Learning from initial/final states

Here we reduce the amount of provided input knowledge and define the learning task  $\Lambda' = \langle \Psi, \Sigma \rangle$  that corresponds to watching only the results of the plan executions. In this case no information about the executed plans is given. A solution to the  $\Lambda'$  learning task is a set of operator schema  $\Xi$  that is compliant only with the predicates in  $\Psi$ , and the given set of initial and final states  $\Sigma$ .

In this case, a solution must not only determine a possible STRIPS action model but also the plans that explain the given labels using the learned model (this information about the actions is no longer given in the learning examples). We assume however, that the headers of the operators schema are known.

## Learning STRIPS action models with planning

Our approach for addressing a learning task  $\Lambda$  or  $\Lambda'$ , is compiling it into a classical planning task with conditional effects. The intuition behind these compilations is that a solution to the resulting classical planning task is a sequence of actions that:

1. Programs the STRIPS action model  $\Xi$ . For each  $\xi \in \Xi$ , the solution plan has a prefix that determines the literals that belong to its *pre*( $\xi$ ), *del*( $\xi$ ) and *add*( $\xi$ ) sets.
2. Validates the programmed STRIPS action model  $\Xi$  using the given input knowledge (the set of labels  $\Sigma$ , and  $\Pi$  if available). For every label  $1 \leq t \leq \tau$ , then  $\Xi$  is used to produce a final state  $s_n^t$  starting from the corresponding initial state  $s_0^t$ . We call this process the validation of the programmed STRIPS action model  $\Xi$ , at the learning example  $1 \leq t \leq \tau$ . If information about the corresponding plan  $\pi_t \in \Pi$  is available, is used in the validation.

Figure 3 shows a classical plan for solving a learning task of the  $\Lambda$  kind. This classical plan programs and validates the operator schema *stack* from the blockworld (previously specified operator schemes for *pickup*, *putdown* and *unstack* are given), using the plan  $\pi_1$  and label  $\sigma_1$  shown in Figure 2. The plan steps [0, 8] are the actions for programming the preconditions of the *stack* operator, steps [9, 13] are the actions for programming the operator effects and steps [14, 22] are actions for validating the programmed operators. The details of the compilation are explained in the remaining of the section.

To formalize our compilations we first define  $1 \leq t \leq \tau$  classical planning instances  $P_t = \langle F, \emptyset, I_t, G_t \rangle$ , one for each learning example, that belong to the same planning frame (i.e. same fluents and actions and differ only in the initial state and goals). The set of fluents  $F$  is built instantiating the predicates in  $\Psi$  with the objects appearing in the input learning examples. Formally  $\Omega = \{o | o \in \bigcup_{1 \leq t \leq \tau} obj(s_0^t)\}$ , where *obj* is a function that returns the set of objects that appear in a fully specified state. The set of actions,  $A = \emptyset$ , is empty because the action model is initially unknown. Finally, the initial state  $I_t$  is given by the state  $s_0^t \in \sigma_t$  while goals  $G_t$ , are defined by the state  $s_n^t \in \sigma_t$ .

Now we are ready to formalize our compilations for learning STRIPS action models with classical planning. We

```

0 : (program_pre_clear_stack_v1)
1 : (program_pre_handempty_stack)
2 : (program_pre_holding_stack_v2)
3 : (program_pre_on_stack_v1_v1)
4 : (program_pre_on_stack_v1_v2)
5 : (program_pre_on_stack_v2_v1)
6 : (program_pre_on_stack_v2_v2)
7 : (program_pre_ontable_stack_v1)
8 : (program_pre_ontable_stack_v2)
9 : (program_eff_clear_stack_v1)
10 : (program_eff_clear_stack_v2)
11 : (program_eff_handempty_stack)
12 : (program_eff_holding_stack_v1)
13 : (program_eff_on_stack_v1_v2)
14 : (apply_unstack a b i1 i2)
15 : (apply_putdown a i2 i3)
16 : (apply_unstack b c i3 i4)
17 : (apply_stack b a i4 i5)
18 : (apply_unstack c d i5 i6)
19 : (apply_stack c b i6 i7)
20 : (apply_pickup d i7 i8)
21 : (apply_stack d c i8 i9)
22 : (validate_1)

```

Figure 3: Example of a plan for programming and validating the operator schema *unstack* using the plan  $\pi_1$  and label  $\sigma_1$  shown in Figure 2 as well as previously specified operator schemes for *pickup*, *putdown* and *unstack*.

start with  $\Lambda'$  because this learning task requires smaller input knowledge. Given a learning task  $\Lambda' = \langle \Psi, \Sigma \rangle$  the compilation outputs a classical planning task  $P_{\Lambda'} = \langle F_{\Lambda'}, A_{\Lambda'}, I_{\Lambda'}, G_{\Lambda'} \rangle$  such that:

- $F_{\Lambda'}$  extends  $F$  with:
  - Fluents representing the programmed action model  $pre_f(\xi)$ ,  $del_f(\xi)$  and  $add_f(\xi)$ , for every  $f \in F_v$  and  $\xi \in \Xi$ . If a fluent  $pre_f(\xi)/del_f(\xi)/add_f(\xi)$  holds, it means that  $f$  is a precondition/negative effect/positive effect in the STRIPS operator schema  $\xi \in \Xi$ .
  - A fluent  $mode_{prog}$  indicating whether the operator schemes are being programmed or they are being validated (already programmed).
  - Fluents  $\{test_t\}_{1 \leq t \leq \tau}$ , indicating the learning example where the programmed action model is being validated.
- $I_{\Lambda'}$ , contains the fluents from  $F$  that encode  $s_0^1$  (the initial state of the first learning example), every  $pre_f(\xi) \in F_{\Lambda'}$  (our compilation assumes that initially any operator schema is programmed with every possible precondition, no negative effect and no positive effect) and  $mode_{prog}$  set to true.
- $G_{\Lambda'} = \bigcup_{1 \leq t \leq \tau} \{test_t\}$ , indicates that the programmed action model is validated in all the learning examples.
- $A_{\Lambda'}$  contains actions of three different kinds:
  1. The actions for programming an operator schema  $\xi \in \Xi$ , which includes:
    - The actions for removing a *precondition*  $f \in F_v$  from the action schema  $\xi \in \Xi$ .

$$\begin{aligned} pre(\text{programPre}_{f,\xi}) &= \{\neg del_f(\xi), \neg add_f(\xi), \\ &\quad pre_f(\xi), mode_{prog}\}, \\ cond(\text{programPre}_{f,\xi}) &= \{\emptyset\} \triangleright \{\neg pre_f(\xi)\}. \end{aligned}$$

- The actions for adding a *negative* or a *positive* effect  $f \in F_v$  to the action schema  $\xi \in \Xi$ .

$$\begin{aligned} pre(\text{programEff}_{f,\xi}) &= \{\neg del_f(\xi), \neg add_f(\xi), \\ &\quad mode_{prog}\}, \\ cond(\text{programEff}_{f,\xi}) &= \{pre_f(\xi)\} \triangleright \{del_f(\xi)\}, \\ &\quad \{\neg pre_f(\xi)\} \triangleright \{add_f(\xi)\}. \end{aligned}$$

2. The actions for applying an already programmed operator schema  $\xi \in \Xi$  bound with the objects  $\omega \subseteq \Omega^{ar(\xi)}$ . The binding of the operator schema is done implicitly by order of appearance of the action parameters, i.e. the variables in  $vars(\xi)$  are bound to the objects in  $\omega$  appearing at the same position. Figure 4 shows the PDDL encoding of the action for applying a programmed operator *stack*.

$$\begin{aligned} pre(\text{apply}_{\xi,\omega}) &= \{pre_f(\xi) \implies p(\omega)\}_{\forall p \in \Psi, f=p(vars(\xi))}, \\ cond(\text{apply}_{\xi,\omega}) &= \{del_f(\xi)\} \triangleright \{\neg p(\omega)\}_{\forall p \in \Psi, f=p(vars(\xi))}, \\ &\quad \{add_f(\xi)\} \triangleright \{p(\omega)\}_{\forall p \in \Psi, f=p(vars(\xi))}, \\ &\quad \{mode_{prog}\} \triangleright \{\neg mode_{prog}\}. \end{aligned}$$

3. The actions for updating the learning example  $1 \leq t \leq \tau$  where the programmed action model is validated.

$$\begin{aligned} pre(\text{validate}_t) &= G_t \cup \{test_j\}_{j \in 1 \leq j < t} \cup \\ &\quad \cup \{\neg test_j\}_{j \in t \leq j \leq \tau} \cup \{\neg mode_{prog}\}, \\ cond(\text{validate}_t) &= \{\emptyset\} \triangleright \{test_t\}. \end{aligned}$$

**Lemma 1.** Any classical plan  $\pi$  that solves  $P_{\Lambda'}$  induces an action model  $\Xi$  that solves the learning task  $\Lambda'$ .

*Proof sketch.* The compilation forces that once the preconditions of an operator schema  $\xi \in \Xi$  are programmed, they cannot be altered. The same happens with the positive and negative effects that define an operator schema  $\xi \in \Xi$  (besides they can only be programmed after the operator preconditions are programmed). Furthermore, once operator schemes are programmed they can only be applied because of the  $mode_{prog}$  fluent. To solve the classical planning task  $P_{\Lambda'}$ , there is only one way of achieving goals  $\{test_t\}_{1 \leq t \leq \tau}$ : executing an applicable sequence of programmed operator schemes that reaches the final state  $s_n^t$  defined by the associated label  $\sigma_t$ , starting from  $s_0^t$ , defined in this same label. If this is achieved for all the input examples of the learning task  $\Lambda'$  (i.e. for all the labels  $\sigma_t$ ), it means that the programmed action model  $\Xi$  is compliant with the provided input knowledge and hence, it is a solution to  $\Lambda'$ .  $\square$

The compilation is *complete* in the sense that it does not discard any possible STRIPS action model.

```

(:action apply_stack
:parameters (?o1 - object ?o2 - object)
:precondition
  (and (or (not (pre_on_stack_v1_v1)) (on ?o1 ?o1))
        (or (not (pre_on_stack_v1_v2)) (on ?o1 ?o2))
        (or (not (pre_on_stack_v2_v1)) (on ?o2 ?o1))
        (or (not (pre_on_stack_v2_v2)) (on ?o2 ?o2))
        (or (not (pre_ontable_stack_v1)) (ontable ?o1))
        (or (not (pre_ontable_stack_v2)) (ontable ?o2))
        (or (not (pre_clear_stack_v1)) (clear ?o1))
        (or (not (pre_clear_stack_v2)) (clear ?o2))
        (or (not (pre_holding_stack_v1)) (holding ?o1))
        (or (not (pre_holding_stack_v2)) (holding ?o2))
        (or (not (pre_handempty_stack)) (handempty)))
:effect
  (and (when (del_on_stack_v1_v1) (not (on ?o1 ?o1)))
        (when (del_on_stack_v1_v2) (not (on ?o1 ?o2)))
        (when (del_on_stack_v2_v1) (not (on ?o2 ?o1)))
        (when (del_on_stack_v2_v2) (not (on ?o2 ?o2)))
        (when (del_ontable_stack_v1) (not (ontable ?o1)))
        (when (del_ontable_stack_v2) (not (ontable ?o2)))
        (when (del_clear_stack_v1) (not (clear ?o1)))
        (when (del_clear_stack_v2) (not (clear ?o2)))
        (when (del_holding_stack_v1) (not (holding ?o1)))
        (when (del_holding_stack_v2) (not (holding ?o2)))
        (when (del_handempty_stack) (not (handempty)))
        (when (add_on_stack_v1_v1) (on ?o1 ?o1))
        (when (add_on_stack_v1_v2) (on ?o1 ?o2))
        (when (add_on_stack_v2_v1) (on ?o2 ?o1))
        (when (add_on_stack_v2_v2) (on ?o2 ?o2))
        (when (add_ontable_stack_v1) (ontable ?o1))
        (when (add_ontable_stack_v2) (ontable ?o2))
        (when (add_clear_stack_v1) (clear ?o1))
        (when (add_clear_stack_v2) (clear ?o2))
        (when (add_holding_stack_v1) (holding ?o1))
        (when (add_holding_stack_v2) (holding ?o2))
        (when (add_handempty_stack) (handempty))
        (when (modeProg) (not (modeProg)))))

```

Figure 4: Action for applying an already programmed operator schema *stack* as encoded in PDDL.

## Constraining the learning hypothesis space with additional input knowledge

Here we show that further input knowledge can be used to constrain the space of the possible action models and make the learning of STRIPS action models more practicable.

### Example plans

Here we extend our compilation to address the learning scenario where, a set of plans is available. Given a learning task  $\Lambda = \langle \Psi, \Pi, \Sigma \rangle$ , the compilation outputs a classical planning task  $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$  that extends  $P'_\Lambda$  as follows:

- $F_\Lambda$  extends  $F_{\Lambda'}$  with  $F_\Pi = \{plan(name(\xi), \Omega^{ar}(\xi), j)\}$ , the fluents to code the  $j$  steps of plans in  $\Pi$ , where  $F_{\pi_t} \subseteq F_\Pi$  encodes  $\pi_t \in \Pi$ . Fluents  $at_j$  and  $next_{j,j_2}$ ,  $1 \leq j < j_2 \leq n$ , are also added to represent the current and next plan steps.
- $I_\Lambda$  is extended with the fluents from  $F_\Pi$  that encode the plan  $\pi_1 \in \Pi$ , the fluents  $at_1$  and  $\{next_{j,j_2}\}$ ,  $1 \leq j < j_2 \leq$

$n$ , for indicating where to start validating the programmed action model. Goals are the same as in the previous compilation  $G_\Lambda = G'_\Lambda = \bigcup_{1 \leq t \leq \tau} \{test_t\}$ .

- With respect to  $A_\Lambda$ .

1. The actions for programming the preconditions/effects of a given operator schema  $\xi \in \Xi$  are the same.
2. The actions for applying an already programmed operator have an extra precondition  $f \in F_\Pi$ , that encodes the current plan step, and extra conditional effects  $\{at_j\} \triangleright \{\neg at_j, at_{j+1}\}_{\forall j \in [1,n]}$  for advancing to the next plan step. This mechanism forces that these actions are only applied as in the example plans.
3. The actions for updating the active test have an extra precondition,  $at_{|\pi_t|}$ , to indicate that the current plan  $\pi_t$  was fully executed and extra conditional effects to load the next plan  $\pi_{t+1}$ :

$$\{f\} \triangleright \{\neg f\}_{f \in F_{\pi_t}}, \{\emptyset\} \triangleright \{f\}_{f \in F_{\pi_t+1}}, \{\emptyset\} \triangleright \{\neg at_{|\pi_t|}, at_1\}.$$

## Previously specified action models

Our compilation does not require to start learning STRIPS action models from scratch and accepts partially specified operator schema where some preconditions and effects are a priori known.

The known preconditions and effects are encoded as fluents  $pre_f(\xi)$ ,  $del_f(\xi)$  and  $add_f(\xi)$  that are part of the initial state  $I_{\Lambda'}$ . The corresponding programming actions ( $programPre_{f,\xi}$  and  $programEff_{f,\xi}$ ) become unnecessary and are removed from  $A_{\Lambda'}$  making the classical planning task  $P_{\Lambda'}$  easier to be solved. To illustrate this, the classical plan of Figure 3 is a solution to a learning task for getting the blockworld action model where the schemes for *pickup*, *putdown* and *unstack* are known in advance.

In the extreme, when a fully specified STRIPS action model is given, the compilation validates whether an observed plan follows the given model. In this case, if a solution plan is found to  $P_{\Lambda'}$ , it means that the given STRIPS action model is *valid* for the given set of examples. If  $P_{\Lambda'}$  is unsolvable it means that the given STRIPS action model is invalid since it is not compliant with all the given examples. Tools for plan validation like VAL (Howey, Long, and Fox 2004) could also be used at this point.

## Static predicates

A *static predicate*  $p \in \Psi$  is a predicate that does not appear in the effects of any action schema (Fox and Long 1998). Therefore, one can get rid of the mechanism for programming static predicates as action effects keeping the compilation complete. Formally, given a static predicate  $p$ :

- Fluents  $del_f(\xi)$  and  $add_f(\xi)$ , such that  $f \in F_v$  is an instantiation of the static predicate  $p$  in the set of *variable* objects  $\Omega_v$ , can be discarded for every  $\xi \in \Xi$ .
- Actions  $programEff_{f,\xi}$  (s.t.  $f \in F_v$  is an instantiation of  $p$  in  $\Omega_v$ ) can also be discarded for every  $\xi \in \Xi$ .

Static predicates can also be used to constrain the space of possible preconditions by looking at the given labels. If a precondition  $f \in F_v$  (s.t.  $f \in F_v$  is an instantiation

of a static predicate in  $\Omega_v$ ) is not possible according to  $\Sigma$ , then the fluents  $pre_f(\xi)$  and actions  $programPre_{f,\xi}$  can be discarded for every  $\xi \in \Xi$ . For instance in the *zenotravel* domain fluents  $pre\_next\_board.v1.v1$ ,  $pre\_next\_debark.v1.v1$ ,  $pre\_next\_fly.v1.v1$ ,  $pre\_next\_zoom.v1.v1$ ,  $pre\_next\_refuel.v1.v1$  can be discarded (as well as their corresponding programming actions) because a precondition  $(next \ ?v1 \ ?v1 - flevel)$  cannot be compliant with any state in  $\Sigma$ .

Likewise, static predicates can constrain the space of possible preconditions looking at the given example plans. In this case, fluents  $pre_f(\xi)$  and actions  $programPre_{f,\xi}$  are discarded for every  $\xi \in \Xi$  if a precondition  $f \in F_v$  (s.t.  $f \in F_v$  is an instantiation of a static predicate in  $\Omega_v$ ) is not possible according to  $\Pi$ . Back to the *zenotravel* domain, if a example plan  $\pi_t \in \Pi$  contains the action  $(fly \ plane1 \ city2 \ city0 \ f13 \ f12)$  and the corresponding label  $\sigma_t \in \Sigma$  contains the static fluent  $(next \ f12 \ f13)$  but it does not include  $(next \ f12 \ f12)$ ,  $(next \ f13 \ f13)$  or  $(next \ f13 \ f12)$  it means that the only possible precondition including the static predicate is  $pre\_next\_fly.v1.v2$ .

In the evaluation of our approach we do not assume that the set of static predicates is given but compute a set of candidates of static predicates from the input examples to the learning task.

## Evaluation

We evaluated our approach for learning STRIPS action models from different amounts of available input knowledge.

### Experimental setup

All the experiments run on an Intel Core i5 3.10 GHz x 4 with 4 GB of RAM. The domains used in the evaluation are IPC domains that satisfy the STRIPS requirement (Fox and Long 2003), and are taken from the PLANNING.DOMAINS repository (Muisse 2016).

**The classical planner.** The classical planner used to solve the instances that result from our compilations is the SAT-based classical planner MADAGASCAR (Rintanen 2014). We decided to use a SAT-based planner because its ability to deal with planning instances populated with dead-ends. In addition, in a SAT-based classical planner all the actions for programming the STRIPS actions preconditions can be done in parallel, in a single planning step, because they do not interact reducing significantly the planning horizon. Likewise, all the actions for programming an action effect can also be applied in a single planning step.

**The evaluation metrics.** The quality of the learned models is quantified with the *precision* and *recall* metrics. These two metrics are frequently used in pattern recognition, information retrieval and binary classification and are more informative than simply counting the number of model errors or computing the *symmetric difference* between the learned model and the baseline (Davis and Goadrich 2006).

Intuitively precision gives a notion of *soundness* while recall gives a notion of the *completeness* of the learned models. Formally  $Precision = \frac{tp}{tp+fp}$ , where  $tp$  is the number

of true positives (predicates that correctly appear in the action model) and  $fp$  is the number of false positives (predicates appear in the learned action model that should not appear). On the other hand recall is formally defined as  $Recall = \frac{tp}{tp+fn}$  where  $fn$  is the number of false negatives (predicates that should appear in the learned action model but are missing).

### The learning examples.

### Learning action models from example plans

Here we assess the performance of our learning approach when addressing  $\Lambda$  learning tasks. Here the *precision* and *recall* metrics are computed in the learned models with respect to the actual models taken as reference. Table 1 summarizes the obtained results, precision (**P**) and recall (**R**) is computed separately for the preconditions (**Pre**), positive effects (**Add**) and negative Effects (**Del**).

	Pre		Add		Del	
	P	R	P	R	P	R
Blocks	1.0	1.0	1.0	1.0	1.0	1.0
Driverlog	1.0	0.4	0.6	0.8	1.0	0.8
Ferry	0.8	0.5	1.0	1.0	1.0	1.0
Floortile	0.5	0.6	0.9	0.8	1.0	0.9
Gripper	1.0	0.6	1.0	1.0	1.0	1.0
Miconic	0.7	0.3	1.0	0.7	0.7	1.0
Satellite	0.6	0.2	1.0	1.0	1.0	0.75
Transport	1.0	0.3	0.5	0.8	1.0	0.6
Visitall	1.0	0.5	1.0	1.0	1.0	1.0
Zenotravel	1.0	0.3	0.7	0.8	1.0	0.7

Table 1: The *precision* and *recall* values achieved by the learned models in the  $\Lambda$  learning tasks.

Table 2 reports the total planning time, and the preprocessing time (in seconds) invested by the classical planner MADAGASCAR to solve the classical planning tasks that result from our compilation as well as the plan length of the found solutions.

Domain	Total time	Preprocess	Plan length
Blocks	0.04	0.00	72
Driverlog	0.10	0.06	127
Ferry	0.20	0.10	58
Floortile	2.15	1.21	170
Gripper	0.01	0.00	45
Miconic	0.10	0.09	54
Satellite	0.20	0.14	67
Transport	0.20	0.18	60
Visitall	0.43	0.39	37
Zenotravel	2.78	2.76	70

Table 2: Planning time, preprocessing time and Plan length.

### Exploiting static predicates

Here we repeat the previous evaluation but compute now a candidates of static predicates from the inputs to the learning

task and exploit this knowledge to reduce the space of possible action models. The set of considered candidate *static predicates* is the set of predicates whose instantiation appear the same in the initial and final states for all the learning examples  $1 \leq t \leq \tau$ . Tables 3 and 4 show the obtained results.

Pre		Add		Del	
P	R	P	R	P	R

Table 3: The *precision* and *recall* values achieved by the learned models in the  $\Lambda$  learning tasks exploiting the computed static predicates.

Domain	Total time	Preprocess	Plan length
--------	------------	------------	-------------

Table 4: Planning time, preprocessing time and Plan length exploiting the computed static predicates.

### Learning from partially specified action models

Here we evaluate the ability of the compilation to support partially specified action models. In this case instead of learning the action models from scratch the model of half of the actions is given as input of the learning task. Tables 5 and 6 summarizes the obtained results.

	Pre		Add		Del	
	P	R	P	R	P	R
Blocks	1.0	1.0	1.0	1.0	1.0	1.0
Driverlog	1.0	0.4	1.0	1.0	1.0	0.7
Ferry	1.0	0.6	0.5	1.0	1.0	1.0
Floortile	0.5	0.6	1.0	0.8	1.0	0.8
Gripper	1.0	0.5	1.0	1.0	1.0	1.0
Miconic	1.0	0.2	1.0	1.0	1.0	1.0
Satellite	1.0	0.5	0.5	1.0	1.0	1.0
Transport	1.0	0.2	1.0	1.0	1.0	0.5
Zenotravel	1.0	0.5	0.6	0.6	1.0	1.0

Table 5: The *precision* and *recall* values achieved by the learned models in the different domains.

Domain	Total time	Preprocess	Plan length
Blocks	0.07	0.01	54
Driverlog	0.06	0.03	71
Ferry	0.18	0.14	50
Floortile	0.81	0.68	92
Gripper	0.01	0.00	37
Miconic	0.06	0.04	40
Satellite	0.18	0.12	43
Transport	0.15	0.14	33
Zenotravel	2.54	2.54	42

Table 6: Planning time, preprocessing time and Plan length.

### Validating action models with classical planning

Here we evaluate the performance of the compilation to validate whether a given action mode is compliant with a given

set of observations. Table 7 reports the planning time, and preprocessing time used to solve the classical planning task that results from our compilation as well as the plan length of the found solutions.

Domain	Total time	Preprocess	Plan length
Blocks	0.00	0.00	
Driverlog	0.04	0.04	
Ferry	0.01	0.01	
Floortile	0.03	0.03	
Gripper	0.00	0.00	
Miconic	0.01	0.01	
Satellite	0.30	0.29	
Transport	0.02	0.02	
Visitall	0.07	0.07	
Zenotravel	11.65	11.39	

Table 7: Planning time, preprocessing time and Plan length for validating a given action model.

### Learning action models from example states

When the set of input planning task is not available, the planner determines also the actions that must satisfying the input labels so actions can reformulated and still be compliant with the learning examples. For instance a blocksworld can be learned where the operator `stack` is defined with the preconditions and effects of the `unstack` operator and vice versa.

### Learning exploiting static predicates

### Learning from partially specified action models

### Related work

The LIVE system (Shen and Simon, 1989) was an extension of the General Problem Solver (GPS) framework (Ernst and Newell, 1969) with a learning component. LIVE alternated problem solving with model learning to automatically define operators. The decision about when to alternate depended on surprises, that is situations where an action effects violated its predicted model. EXPO (Gil, 1992) generated plans with the PRODIGY system (Minton, 1988), monitored the plans execution, detected differences in the predicted and the observed states and constructed a set of specific hypotheses to fix those differences. Then the EXPO filtered the hypotheses heuristically. OBSERVER (Wang, 1994) learned operators by monitoring expert agents and applying the version spaces algorithm (Mitchell, 1997) to the observations. When the system already had an operator representation, the preconditions were updated by removing facts that were not present in the new observations pre-state; the effects were augmented by adding facts that were in the observations delta-state. All of these early works were based on direct liftings of the observed states. They also benefit from experience beyond simple interaction with the environment such as exploratory plans or external teachers, but none provided a theoretical justification for this second source of knowledge. The work recently reported in (Walsh and Littman, 2008) succeeds in bounding the number of interactions the

learner must complete to learn the preconditions and effects of a STRIPS action model. This work shows that learning STRIPS operators from pure interaction with the environment, can require an exponential number of samples, but that limiting the size of the precondition lists enable sample-efficient learning (polynomial in the number of actions and predicates of the domain). The work also proves that efficient learning is also possible without this limit if an agent has access to an external teacher that can provide solution traces on demand.

Others systems have tried to learn more expressive action models for deterministic planning in fully observable environments. Examples would include the learning of conditional costs for AP actions (Jess Lanchas and Borrajo, 2007) or the learning of conditional effects with quantifiers (Zhuo et al., 2008).

In addition action model learning has been studied in domains where there is partial state observability. ARMS uses the same kind of learning examples but assumes the examples are given and proceeds in two phases. In the first phase, ARMS extracts frequent action sets from plans that share a common set of parameters. ARMS also finds some frequent literal-action pairs with the help of the initial state and the goal state that provide an initial guess on the actions preconditions, and effects. In the second phase, ARMS uses the frequent action sets and literal-action pairs to define a set of weighted constraints that must hold in order to make the plans correct. Then, ARMS solves the resulting weighted MAX-SAT problem and produces action models from the solution of the SAT problem. This process iterates until all actions are modelled. For a complex planning domain that involves hundreds of literals and actions, the corresponding weighted MAX-SAT representation is likely to be too large to be solved efficiently as the number of clauses can reach up to tens of thousands. For that reason ARMS implements a hill-climbing method that models the actions approximately. Consequently, the ARMS output is a model which may be inconsistent with the examples.

(Amir and Chang, 2008) introduced an algorithm that tractably generates all the STRIPS-like models that could have lead to a set of observations. Given a formula representing the initial belief state, a sequence of executed actions and the corresponding observed states (where partial observations of states are given), it builds a complete explanation of observations by models of actions through a Conjunctive Normal Form (CNF) formula. By linking the possible states of fluents to the effect propositions in the action models, the complexity of the CNF encoding can be controlled to find exact solutions efficiently in some circumstances. The learning algorithm updates the formula of the belief state with every action and observation in the sequence. This update makes sure that the new formula represents all the transition relations consistent with the actions and observations. The formula returned at the end includes all consistent models, which can then be retrieved with additional processing. Unlike the previous approaches, the one described in (Mourao et al., 2008) deals with both missing and noisy predicates in the observations. For each action in a given domain, they use kernel perceptrons to learn predictions of the domain prop-

erties that change because of the action execution.

In the LOCM approach for learning planning models (Cresswell, McCluskey, and West 2013), the only required input are the example training plans, without the need for providing the initial and goal state information. LOCM exploits assumptions about the kind of domain model it has to generate. Particularly, it assumes a domain consists of a collection of objects (called *sorts*), where each object has a defined set of states that can be captured by a parameterized FSM. The induction of FSM draws upon assumptions like the continuity of object transitions or the association of parameters between consecutive actions in the training sequence. The intuitive assumptions of LOCM yield a learning model heavily reliant on the kind of domain structure and not capable of properly deriving domain theories in which the state of a sort is subject to different state machines. This limitation is later overcome by LOCM2 by forming separate state machines for a sort, each containing a subset of the full transition set for the sort (Cresswell and Gregory 2011).

(Stern and Juba 2017).

## Conclusions

The paper presented a novel approach for learning STRIPS action models from examples exclusively using classical planning. The approach is flexible to different amounts of available input knowledge and accepts partially specified action models.

The empirical evaluation shows that, when example plans are available, our approach can compute accurate action models from small sets of learning examples and investing very small learning times (less than a second time in most of the domains). When action plans are not available our approach is still able to produce action models compliant with the input information. In this case, since learning is not constrained by actions it can change the semantics of the operators. 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 (Ivankovic and Haslum 2015).

With regard to scalability of the compilation approach, the size of the compiled classical planning instances depends on the number of input examples. On the other hand the empirical results show that our approach, since is based on inferences instead of statistics, is able to generate non-trivial models from very small data sets. In addition the SAT-based planner MADAGASCAR is particularly suitable for the approach because its ability to deal with planning instances populated with dead-ends and because all the actions for programming the STRIPS actions preconditions can be done in parallel, in a single planning step, because they do not interact reducing significantly the planning horizon. The same happens with all the actions for programming an action effect can also be applied in a single planning step.

As far as we know, this is the first work on learning action models using a classical planner. This novel approach opens the door to the bootstrapping of action model that is,



a planner collecting by its own the necessary data for learning an action model for a given planning task. Generating *informative* examples for learning planning action models is still a challenging open issue. Planning actions include preconditions that are only satisfied by specific sequences of actions, and often, with a low probability of being chosen by chance (Fern, Yoon, and Givan 2004). The success of recent algorithms for exploring planning tasks (Francés et al. 2017) motivates the development of novel techniques that autonomously collect the learning examples.

## Acknowledgment

Diego Aineto is partially supported by the *FPU* program funded by the Spanish government. Sergio Jiménez is partially supported by the *Ramon y Cajal* program funded by the Spanish government.

## References

- Amir, E., and Chang, A. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research* 33:349–402.
- Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*.
- Cresswell, S., and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *ICAPS*.
- Cresswell, S. N.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using locm. *The Knowledge Engineering Review* 28(02):195–213.
- Davis, J., and Goadrich, M. 2006. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, 233–240. ACM.
- Fern, A.; Yoon, S. W.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *ICAPS*, 191–199.
- Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3-4):189–208.
- Fox, M., and Long, D. 1998. The automatic inference of state invariants in tim. *Journal of Artificial Intelligence Research* 9:367–421.
- Fox, M., and Long, D. 2003. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)* 20:61–124.
- Francés, G.; Ramirez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action representations are overrated: Classical planning with simulators. In *International Joint Conference on Artificial Intelligence*.
- Geffner, H., and Bonet, B. 2013. A concise introduction to models and methods for automated planning.
- Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Elsevier.
- Howey, R.; Long, D.; and Fox, M. 2004. 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*, 294–301. IEEE.
- Ivankovic, F., and Haslum, P. 2015. Optimal planning with axioms. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.
- Kambhampati, S. 2007. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *Proceedings of the National Conference on Artificial Intelligence*.
- McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.
- Michalski, R. S.; Carbonell, J. G.; and Mitchell, T. M. 2013. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media.
- Muise, C. 2016. Planning. domains. *ICAPS system demonstration*.
- Ramírez, M. 2012. *Plan recognition as planning*. Ph.D. Dissertation, Universitat Pompeu Fabra.
- Rintanen, J. 2014. Madagascar: Scalable planning with sat. *Proceedings of the 8th International Planning Competition (IPC-2014)*.
- Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016. Hierarchical finite state controllers for generalized planning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 3235–3241. AAAI Press.
- Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2017. Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence*.
- Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125(1-2):119–153.
- Stern, R., and Juba, B. 2017. Efficient, safe, and probably approximately complete learning of action models. *IJCAI*.
- Vallati, M.; Chrapa, L.; Grzes, M.; McCluskey, T. L.; Roberts, M.; and Sanner, S. 2015. The 2014 international planning competition: Progress and trends. *AI Magazine* 36(3):90–98.
- Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence* 171(2-3):107–143.