

# 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

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

## Abstract

This paper presents a novel approach for learning STRIPS action models from examples that compiles this inductive learning task into a classical planning task. Interestingly the proposed compilation 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 actions or intermediate states are observed. What is more, the compilation can reuse previous knowledge (it is able to start from partially specified action models) and can be used to validate whether an observed plan follows a given action model.

## Introduction

Besides plan synthesis, planning action models are also useful for plan/goal recognition (Ramírez and Geffner 2010). In both tasks, off-the-shelf planners require reasoning about action models that correctly and completely capture the possible world transitions (Ghallab, Nau, and Traverso 2004; Geffner and Bonet 2013). Unfortunately, building such planning action models is complex, even for planning experts, so this knowledge acquisition bottleneck limits the potential of automated planning (Kambhampati 2007).

On the other hand, Machine Learning (ML) techniques are able to compute a wide range of different kinds of models from examples (Michalski, Carbonell, and Mitchell 2013). However, the application of inductive ML techniques for learning planning action models is not straightforward. First, the inputs to ML algorithms usually are finite vectors encoding the value of a given set of features in a set of objects. The input for learning planning action models traditionally are sets of observations of plan executions (each with a possibly different length). Second, the traditional output of off-the-shelf ML techniques 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 not a scalar but a model of the preconditions and the effects of each action that defines the possible state transitions of a given planning domain.

Learning 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 all the intermediate states traversed by the example plans. Motivated by recent advances on learning 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 classical planning action models that:

- Can be defined as a classical planning compilation.
- It 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 actions or intermediate states are observed.
- Can exploit previous knowledge about the action model in the form of partially specified action models.

The paper is organized as follows. The first section presents the classical planning model and its extension to define conditional effects. The second section formalizes the learning task addressed in the paper. The third section describes our approach for addressing the learning of STRIPS action models by compiling this task into a classical planning task. The fourth section evaluates the performance of our approach. Finally the last section discusses the strengths and weaknesses of our approach and proposing some future work.

## Background

This section defines the planning models used on this work.

### 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.  $l = f$  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). 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 then a total assignment of values to fluents, i.e.  $|s| = |F|$ , so the size of the state space  $2^{|F|}$ . Explicitly including negative literals  $\neg f$  in states simplifies subsequent definitions, but we

often 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 positive effects  $\text{eff}^+(a) \in \mathcal{L}(F)$ , and a set of negative 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 a new state  $\theta(s, a) = (s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)$ .

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  $s_i = \theta(s_{i-1}, a_i)$ . The plan  $\pi$  *solves*  $P$  if and only if  $G \subseteq s_n$ , i.e. if the goal condition is satisfied following the application of  $\pi$  in  $I$ .

In this work we assume that the fluents in  $F$  are instantiated from predicates, as in PDDL (McDermott et al. 1998; Fox and Long 2003). There exists a set of predicates  $\Psi$ , each  $p \in \Psi$  with 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 each action in  $A$  is instantiated from an STRIPS operator schema. Figure 1 shows the *stack* operator schema for a STRIPS blocksworld represented in PDDL.

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

Figure 1: Example of a *stack* operator schema for a STRIPS blocksworld represented in PDDL.

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

Let us define a new set of fluents  $F_v$  that results instantiating  $\Psi$  but using only the *variable objects*  $\Omega_v$ . 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 operator schema as a tuple  $\xi = \langle \text{head}(\xi), \text{pre}(\xi), \text{add}(\xi), \text{del}(\xi) \rangle$ :

- $\text{head}(\xi) = \langle \text{name}(\xi), \text{pars}(\xi) \rangle$ , represents an operator *header* defined by its corresponding action name and a list of variables,  $\text{pars}(\xi) \in \Omega_v^{\text{ar}(\xi)}$ . The headers for the blocksworld operators 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 positive effects  $\text{add}(\xi) \subseteq F_v$ , and the negative effects  $\text{del}(\xi) \subseteq F_v$  such that,  $\text{del}(\xi) \subseteq \text{pre}(\xi)$ ,  $\text{del}(\xi) \cap \text{add}(\xi) = \emptyset$  and  $\text{pre}(\xi) \cap \text{add}(\xi) = \emptyset$ .

## 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. We use conditional effects because they allow us to compactly define actions whose effects depend on the current state. Many classical planners cope with conditional effects without compiling them away. In fact, the support of PDDL conditional effects was a requirement for participating at IPC-2014 (Vallati et al. 2015).

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  $a$  in  $s$  is a new state  $\theta(s, a) = (s \setminus \text{eff}^-(s, a)) \cup \text{eff}^+(s, a)$ , where  $\text{eff}^-(s, a)$  and  $\text{eff}^+(s, a)$  are the negative and positive effects in  $\text{eff}(s, a)$ .

## Learning STRIPS action models

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

This section formalizes more challenging tasks, for learning STRIPS action models, where less input knowledge is available. Next we formalize these learning tasks according to the available input knowledge.

**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. This set includes the predicates for defining the headers of the operators schema.
- $\Pi = \{\pi_1, \dots, \pi_\tau\}$ , the given set of example plans.
- $\Sigma = \{\sigma_1, \dots, \sigma_\tau\}$ , a set of labels s.t. each plan  $\pi_t$ ,  $1 \leq t \leq \tau$ , has a label  $\sigma_t = (s_0^t, s_n^t)$  where  $s_n^t$  is the state resulting from executing  $\pi_t$  starting from the state  $s_0^t$ .

To illustrate this, Figure 2, shows  $\Psi$  and  $\Pi = \{\pi_1\}$  for a learning task  $\Lambda$  with a single learning example that corresponds to invert a tower of four blocks.

```

;;; Predicates as defined in PDDL
(on ?o1 - object ?o2 - object) (ontable ?o - object)
(clear ?o - object) (holding ?o - object)
(handempty)

;;; Plan
0: (unstack A B)
1: (put-down A)
2: (unstack B C)
3: (stack B A)
4: (unstack C D)
5: (stack C B)
6: (pick-up D)

```

;;; Initial and final states

D	A
C	B
B	C
A	D

Figure 2: Example of a learning task  $\Lambda$  with a single learning example that corresponds to invert a tower of four blocks.

A solution to the learning task  $\Lambda$  is a set of operator schema  $\Xi$  (one schema for each operator header) compliant with the predicates in  $\Psi$ , the example plans  $\Pi$ , and their labels  $\Sigma$ .

**Learning from initial/final states.** Here we reduce the amount of input knowledge provided to the learning task. Now  $\Pi = \{\pi_1, \dots, \pi_\tau\}$  is replaced by  $\Pi' = \{|\pi_1|, \dots, |\pi_\tau|\}$  i.e.  $\Pi'$  that does not contain a set of plans but the number of actions of each plan, so the learning task is redefined as  $\Lambda' = \langle \Psi, \Pi', \Sigma \rangle$ . While the previous learning task,  $\Lambda$ , corresponds to watching an agent acting in the world, this new learning task  $\Lambda'$  can be understood as watching only the results of its actions executions knowing the number of different actions performed by the agent.

Finally, we can go a step further and redefine a third learning task  $\Lambda'' = \langle \Psi, \Sigma \rangle$  that corresponds to watching only the results of the plan executions. In this case 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 these two cases, a solution must not only synthesize the action models but also the actions that produced the given labels (this information is no longer given in the learning examples).

## Learning STRIPS action models with planning

Our approach for addressing a learning task  $\Lambda$ ,  $\Lambda'$  or  $\Lambda''$ , is compiling it into a classical planning task with conditional effects  $P_\Lambda$ ,  $P_{\Lambda'}$  or  $P_{\Lambda''}$ . 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. For each  $\xi \in \Xi$ , determines the literals that belong to the  $pre(\xi)$ ,  $del(\xi)$  and  $add(\xi)$  sets.
2. Validates the programmed operator schema  $\Xi$  with the given set of labels  $\Sigma = \{\sigma_1, \dots, \sigma_\tau\}$ .  $\Xi$  is used, at every  $1 \leq t \leq \tau$ , to produce a final state  $s_n^t$  starting from the corresponding initial state  $s_0^t$ .

To formalize our compilations we first define  $1 \leq t \leq \tau$  classical planning instances  $P_t = \langle F, \emptyset, I_t, G_t \rangle$ , 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  $F$

```

0 : (program_pre_clear_stack_var1)
1 : (program_pre_handempty_stack_)
2 : (program_pre_holding_stack_var2)
3 : (program_pre_on_stack_var1_var1)
4 : (program_pre_on_stack_var1_var2)
5 : (program_pre_on_stack_var2_var1)
6 : (program_pre_on_stack_var2_var2)
7 : (program_pre_ontable_stack_var1)
8 : (program_pre_ontable_stack_var2)
9 : (program_eff_clear_stack_var1)
10 : (program_eff_clear_stack_var2)
11 : (program_eff_handempty_stack_)
12 : (program_eff_holding_stack_var1)
13 : (program_eff_on_stack_var1_var2)
14 : (validate_0)
15 : (unstack a b i1 i2)
16 : (put-down a i2 i3)
17 : (unstack b c i3 i4)
18 : (stack b a i4 i5)
19 : (unstack c d i5 i6)
20 : (stack c b i6 i7)
21 : (pick-up d i7 i8)
22 : (validate_1)

```

Figure 3: Example of a plan.

is built instantiating the predicates in  $\Psi$  with the set of objects appearing in the example plans or their corresponding labels. Formally  $\Omega = \{o | o \in \bigcup_{1 \leq t \leq \tau} obj(\pi_t) \cup obj(s_0^t) \cup obj(s_n^t)\}$ . The set of actions is empty,  $A = \emptyset$ , because the action model is 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 start with  $\Lambda''$  because it is the learning task that requires the smallest amount of input knowledge and incrementally extend the formalized compilation until addressing  $\Lambda$ , the learning task with the largest amount of 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 of the operator  $\xi$ .
  - Fluents  $\{test_t\}_{1 \leq t \leq \tau}$ , indicating the learning example where the programmed model is being validated.
  - Fluents  $mode_{prog}$  and  $mode_{val}$  indicating whether the operator schema are being programmed or the programmed operators are being validated.
- $I_{\Lambda''}$ , contains the fluents from  $F$  that encode  $s_0^1$  (i.e. the initial state of the first learning example), every fluent  $pre_f(\xi) \in F_\Lambda$  (initially all operators have all the possible preconditions) and  $mode_{prog}$ .
- $G_{\Lambda''} = \{test_t\}_{1 \leq t \leq \tau}$ , indicates that the programmed action model is validated in all the learning examples.

- $A_{\Lambda''}$  contains actions of three different types:
  1. The actions for programming the operator schema. This includes the actions for removing a *precondition*  $f \in F_v$  from the action schema  $\xi \in \Xi$ .

$$\begin{aligned} \text{pre}(\text{programPre}_{f,\xi}) &= \{\neg \text{del}_f(\xi), \neg \text{add}_f(\xi), \\ &\quad \text{mode}_{\text{prog}}, \text{pre}_f(\xi)\}, \\ \text{cond}(\text{programPre}_{f,\xi}) &= \{\emptyset\} \triangleright \{\neg \text{pre}_f(\xi)\}. \end{aligned}$$

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

$$\begin{aligned} \text{pre}(\text{programEff}_{f,\xi}) &= \{\neg \text{del}_f(\xi), \neg \text{add}_f(\xi), \\ &\quad \text{mode}_{\text{prog}}\}, \\ \text{cond}(\text{programEff}_{f,\xi}) &= \{\text{pre}_f(\xi)\} \triangleright \{\text{del}_f(\xi)\}, \\ &\quad \{\neg \text{pre}_f(\xi)\} \triangleright \{\text{add}_f(\xi)\}. \end{aligned}$$

2. The actions for applying an already programmed operator schema  $\xi \in \Xi$  bounding it with objects  $\omega \subseteq \Omega^{ar}(\xi)$

$$\begin{aligned} \text{pre}(\text{apply}_{\xi,\omega}) &= \{\text{pre}_f(\xi) \implies p(\omega)\}_{\forall p \in \Psi, f = p(\text{pars}(\xi))}, \\ \text{cond}(\text{apply}_{\xi,\omega}) &= \{\text{del}_f(\xi)\} \triangleright \{\neg p(\omega)\}_{\forall p \in \Psi, f = p(\text{pars}(\xi))}, \\ &\quad \{\text{add}_f(\xi)\} \triangleright \{p(\omega)\}_{\forall p \in \Psi, f = p(\text{pars}(\xi))}, \\ &\quad \{\text{mode}_{\text{prog}}\} \triangleright \{\neg \text{mode}_{\text{prog}}, \text{mode}_{\text{val}}\}. \end{aligned}$$

For instance, these actions define that if an operator is programmed with the precondition  $\text{holding}(v_1) \in F_v$  it *implies* ( $\implies$ ) that  $\text{holding}(\text{block}_1) \in F$  has to be true in the current state if the operator binds the variable object  $v_1 \in \Omega_v$  with the regular object  $\text{block}_1 \in \Omega$ . The operator binding is done implicitly, i.e. variables in  $\text{pars}(\xi)$  are bound to the objects in  $\omega$  appearing in the same position.

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

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

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

*Proof sketch.* Once the preconditions of an operator schema  $\Xi$  are programmed, they cannot be altered. The same happens with the positive and negative effects that define an operator schema (besides they can only be programmed after the operator preconditions are programmed). Furthermore, once an operator schema is programmed it can only be applied. The only way of achieving a goal  $\{\text{test}_t\}, 1 \leq t \leq \tau$  is by executing an applicable sequence of programmed operator schema that achieves the final state defined by the associated label  $\sigma_t$ , starting from the initial defined in that label. If this is done for all the input examples of the learning task (for all the labels), it means that the programmed action model  $\Xi$  is compliant with the learning input knowledge and hence, it is a solution to the action model learning task.  $\square$

Interestingly, the compilation is valid for partially specified action models since known preconditions and effects (fluents  $\text{pre}_f(\xi)$ ,  $\text{del}_f(\xi)$  and  $\text{add}_f(\xi)$ ) can be part of the initial state  $I_{\Lambda''}$  and the corresponding programming actions ( $\text{programPre}_{f,\xi}$  and  $\text{programEff}_{f,\xi}$ ) be removed from  $A_{\Lambda''}$  making the classical planning task  $P_{\Lambda''}$  easier.

## Constraining the hypothesis space with example plans

Here we extend our compilation to address the learning scenario defined by  $\Lambda$  where a set of plans  $\Pi$  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:

- $F_{\Lambda}$  includes the fluents  $F_{\Pi} = \{\text{plan}(\text{name}(\xi), j, \Omega^{ar}(\xi))\}$  to code the  $j$  steps of the plans in  $\Pi$  with  $F_{\Pi_t} \subseteq F_{\Pi}$  encoding  $\pi_t \in \Pi$ . We denote  $\pi_t$  as a solution plan to the classical planning instances  $P_t$  introduced above. In addition fluents  $\text{at}_j$  and  $\text{next}_{j,j_2}, 1 \leq j < j_2 \leq n$ , represent the plan step where the programmed action model is validated.
- $I_{\Lambda}$  is extended with the fluents from  $F_{\Pi_1}$  that encode the plan  $\pi_1 \in \Pi$  for solving  $P_1$ , and the fluents  $\text{at}_1$  and  $\{\text{next}_{j,j_2}\}, 1 \leq j < j_2 \leq n$ , for indicating where to start validating the programmed action model.  $G_{\Lambda} = G'_{\Lambda}$ .
- With respect to  $A_{\Lambda}$ , the actions for programming the preconditions/effects of a given operator are the same.
  1. The actions for applying an operator have an extra precondition  $f \in F_{\Pi_t}$  that encodes the current plan step and extra conditional effects  $\{\text{at}_j\} \triangleright \{\neg \text{at}_j, \text{at}_{j+1}\}_{\forall j \in [1,n]}$  for advancing the plan step.
  2. The actions for changing the active test have an extra precondition,  $\text{at}_{|\Pi_t|}$ , to indicate that the current plan  $\Pi_t$  was fully executed and extra conditional effects to load the next plan  $\Pi_{t+1}$  where the programmed operators are validated:

$$\{f\} \triangleright \{\neg f\}_{f \in F_{\Pi_t}}, \{\emptyset\} \triangleright \{f\}_{f \in F_{\Pi_{t+1}}}, \{\emptyset\} \triangleright \{\neg \text{at}_{|\Pi_t|}, \text{at}_1\}.$$

## Evaluation

We evaluated our learning approach for different amounts of available input knowledge. In all the experiments the compilation is solved using the SAT-based classical planner MADAGASCAR (Rintanen 2014) given its ability to deal with planning instance highly populated with dead-ends.

**Learning action models from example plans.** Here we assess the performance of our learning approach when addressing the  $\Lambda$  learning task. The quality of the learned models is quantified using the cardinality of the *symmetric difference* between the set of preconditions, negative and positive effects (1), in the learned model and (2), in the actual models taken as reference.

Last but not least, collecting *informative* examples for learning planning action models is 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 Giovan 2004). In addition, motivated by the success of recent algorithms for exploring planning tasks (?), we do not assume that a learning set of plans is given apriori but instead, we autonomously collect the learning examples.

Table 1: Mean error and standard deviation of the learned models.

Table 1 shows the mean error and standard deviation of the learned models with respect to the actual action models. The standard deviation provides a measure of how this error is distributed among the different operators in the domain. If this deviation is 0 it means that is equally distributed in all the domain operators.

**Learning action models from example states.** When the set of input planning task is not available the *symmetric difference* is not an acceptable measure for evaluating the quality of the learned models. The reason is that, in this situation, 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.

With this regard, the Table 2 shows the planning time and number of steps required to build a set of action models compliant with the input examples.

Table 2: Planning time and number of steps required to build a set of action models compliant with the input examples.

## Conclusions

The paper presented a novel approach for learning STRIPS action models from examples exclusively using classical planning. Interestingly 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. 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 exploitin features that allow more compact solutions like the *reachable* or *movable* features in the Sokoban domain.

With regard to efficiency, the size of the compiled classical planning instances depend 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.

## 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. N.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using locm. *The Knowledge Engineering Review* 28(02):195–213.
- Fern, A.; Yoon, S. W.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *ICAPS*, 191–199.
- 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.
- 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.
- 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.
- Ramirez, M., and Geffner, H. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the Conference of the Association for the Advancement of Artificial Intelligence (AAAI 2010)*, 1121–1126.
- 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*.
- 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.