

# Bootstrapping classical planning action models

**Guillem Francés**

Information and Communication Technology  
Universitat Pompeu Fabra  
Roc Boronat 138, 08018 Barcelona, Spain  
@upf.edu

**Sergio Jiménez**

Computing and Information Systems  
University of Melbourne  
Parkville, Victoria 3010, Australia  
sjimenez@unimelb.edu.au

**Nir Lipovetzky**

Computing and Information Systems  
University of Melbourne  
Parkville, Victoria 3010, Australia  
@unimelb.edu.au

**Miguel Ramírez**

Computing and Information Systems  
University of Melbourne  
Parkville, Victoria 3010, Australia  
@unimelb.edu.au

## Abstract

This paper presents a novel approach for learning classical planning action models from minimal input knowledge and using exclusively existing classical planners. First, the paper defines a classical planning compilation to learn action models from examples. Interestingly the compilation accepts partially specified action models and is flexible to different levels of available knowledge in the input examples. Second, the paper explains how to autonomously collect informative examples using a classical planner based on pure exploratory search.

## Introduction

Off-the-shelf planners reason about action models that correctly and completely capture the possible world transitions (Geffner and Bonet 2013). Unfortunately building such models is complex, even for planning experts, limiting the potential of automated planning (Kambhampati 2007).

Despite there are many Machine Learning (ML) techniques to compute models from examples (Michalski, Carbonell, and Mitchell 2013) its application to learning planning action models is not straightforward. First, the inputs to ML algorithms usually are finite sets of values encoding objects features. On the other hand planning tasks include actions and are closer to the representation of procedures and behaviours than to object representation. 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). The output for learning planning action models is not a scalar but a declarative generative model of the possible state transitions. Last but not least, collecting *informative* examples for learning planning action models is challenging. 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).

Learning classical planning 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 the full knowledge of all the

states traversed by example plans. Motivated by recent advances on learning generative models with classical planning (Segovia-Aguas, Jiménez, and Jonsson 2017) and on the effective exploration of planning state spaces (Francés et al. 2017), this paper introduces an innovative approach for learning classical planning action models. The contribution of this work is two-fold:

1. An inductive learning algorithm that minimizes the required input knowledge, that is flexible to different levels of available input knowledge, and that can be defined as a classical planning compilation.
2. A method for collecting *informative* learning examples using an exploration-based classical planner.

## 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 a total assignment of values to fluents, i.e.  $|s| = |F|$ , so the number of states is  $2^{|F|}$ .

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 the *preconditions*, a set of positive effects  $\text{add}(a) \in \mathcal{L}(F)$  and a set of negative effects  $\text{del}(a) \in \mathcal{L}(F)$ . 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  $\theta(s, a) = (s \setminus \neg \text{del}(a)) \cup \text{add}(a)$ .

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  $i$  such that  $1 \leq i \leq n$ ,  $a_i$  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  $I$ .

We assume that fluents are instantiated from predicates, as in PDDL (Fox and Long 2003). There exists a set of predicates  $\Psi$ , each  $p \in \Psi$  with an argument list of arity  $ar(p)$ . 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.  $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$  s.t. given a set  $X$ ,  $X^n$  is the  $n$ -th Cartesian power of  $X$ .

Likewise we assume that actions in  $A$  are instantiated from operator schema. Let us define  $\Omega_v = \{v_1, \dots, v_v\}$ ,  $\Omega_v \cap \Omega = \emptyset$ , a new set of objects representing variable names. The number of *variable* objects,  $|\Omega_v|$ , is given by the action with the maximum arity in a planning frame. For instance, in the blockworld  $\Omega_v = \{v_1, v_2\}$  since actions *stack* (Figure 1) and *unstack* have two parameters. Let us define a new set of fluents,  $F_v \cap F = \emptyset$ , that result instantiating  $\Psi$  but using only the objects in  $\Omega_v$ , e.g. for the blockworld example  $holding(v_1) \in F_v$  or  $clear(v_2) \in F_v$ .

```
(: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 the *stack* planning action schema from the blockworld as represented in PDDL.

Now we are ready to define an operator schema  $\xi \in \Xi$  as:

- The operator *header* that is a fluent  $header(name(\xi), pars(\xi))$  built using an action name and a list of variables,  $pars(\xi) \in \Omega_v^{ar(\xi)}$ . For instance in the blockworld example these are the required operator headers  $\{stack(v_1, v_2), unstack(v_1, v_2), pickup(v_1), putdown(v_1)\}$ .
- The operator *body* comprises: the preconditions  $pre(\xi) \subseteq F_v$ , the positive effects  $add(\xi) \subseteq F_v$ , and the negative effects  $del(\xi) \subseteq F_v$  such that,  $del(\xi) \subseteq pre(\xi)$ ,  $del(\xi) \cap add(\xi) = \emptyset$  and  $pre(\xi) \cap add(\xi) = \emptyset$ .

### Classical Planning with Conditional Effects

Conditional effects make it possible to repeatedly refer to the same action while their precise effects depend on the current state. In this case each action  $a \in A$  has a set of literals  $pre(a) \in \mathcal{L}(F)$  called the *precondition* and a set of conditional effects  $cond(a)$ . Each conditional effect  $C \triangleright E \in cond(a)$  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  $pre(a) \subseteq s$ , and the resulting set of *triggered effects* is

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

i.e. effects whose conditions hold in  $s$ . The result of applying  $a$  in  $s$  is a new state  $\theta(s, a) = (s \setminus \neg eff(s, a)) \cup eff(s, a)$ .

## Learning classical planning action models

This section formalizes the different learning tasks addressed in the paper according to the available input knowledge.

On the extreme learning from fully available input knowledge, i.e. the *pre-* and *post-states* of every action execution, is straightforward. In this case operator schema are derived lifting the literals that change between the pre and post-state of the corresponding action executions. Preconditions are derived lifting the minimal set of literals appearing in all the pre-states that correspond to the same operator.

### Learning classical planning action models from plans

This 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 describing the headers of the operators.
- $\Pi = \{\pi_1, \dots, \pi_t\}$  is the given set of example plans,
- $\Sigma = \{\sigma_1, \dots, \sigma_t\}$  is a set of labels s.t. each plan  $\pi_i$ ,  $1 \leq i \leq t$ , has an associated label  $\sigma_i = (s_i, s'_i)$  where  $s'_i$  is the state resulting from executing  $\pi_i$  starting from the state  $s_i$ .

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

### Learning classical planning action models from states

As explained in this paper we aim to reduce the amount of input knowledge provided to the learning task so we redefine the previous tasks as  $\Lambda' = \langle \Psi, \Sigma \rangle$ . While the previous task corresponds to watching an agent act, this one corresponds to watching only the results of its actions executions.

Now 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.

### Learning action models from example states using a classical planner

Our innovative approach for addressing  $\Lambda' = \langle \Psi, \Sigma \rangle$  is compiling this learning task into a classical planning task  $P_\Lambda$ . The intuition behind the compilation is that a solution to  $P_\Lambda$  is a sequence of actions that first, programs the action model (i.e. the  $pre(\xi)$ ,  $del(\xi)$  and  $add(\xi)$  for each  $\xi \in \Xi$ ) and then, validates the programmed action model in the given set of labels  $\Sigma$ , one after the other.

To formalize the compilation we define  $t$  classical planning instances  $P_1 = \langle F, A, I_1, G_1 \rangle, \dots, P_t = \langle F, A, I_t, G_t \rangle$ , that belong to the same planning frame (share the 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 in  $\Omega = \{o | o \in s_i \cup s'_i, 1 \leq i \leq t\}$ , the set of objects that appear in the states in  $\Sigma$ . The set of actions is empty  $A = \emptyset$ . Finally the initial state  $I_i$ ,  $1 \leq i \leq t$ , is given by the state  $s_i \in \sigma_i$  while goals  $G_i$  are defined by the state  $s'_i \in \sigma_i$ .

Now we are ready to define the compilation for learning action models using an off-the-shelf classical planner. Given

a learning task  $\Lambda' = \langle \Psi, \Sigma \rangle$  the compilation outputs a classical planning task  $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$  where:

- $F_\Lambda$  extends  $F$  with:
  - Fluents  $header(name(\xi), pars(\xi))$  representing the operator header,  $pars(\xi) \in \Omega_v^{ar(\xi)}$  for each  $\xi \in \Xi$ .
  - Fluents representing the programmed action model:  $pre_f(\xi)$ ,  $del_f(\xi)$  and  $add_f(\xi)$  for every  $f \in F_v$  and  $\xi \in \Xi$ .
  - Fluents  $\{test_i\}_{1 \leq i \leq t}$ , indicating the example where the programmed model is currently being validated.
- $I_\Lambda$ , contains the fluents from  $F$  that encode the initial state  $s_1 \in P_1$  and the header of the operators. For instance,  $\{stack(v_1, v_2), unstack(v_1, v_2), pickup(v_1), putdown(v_1)\}$  for learning the blockworld action model.
- $G_\Lambda = \{test_i\}_{1 \leq i \leq t}$ , indicates that the programmed action model is validated in all the examples in  $\Sigma$ .
- $A_\Lambda$  replaces the actions in  $A$  with actions of three types:
  1. Actions for programming:
    - *Precondition*  $f \in F_v$  in the action schema  $\xi \in \Xi$ :
 
$$\begin{aligned} pre(\text{programPre}_{f,\xi}) &= \{\neg pre_f(\xi), \neg del_f(\xi), \neg add_f(\xi)\}, \\ cond(\text{programPre}_{f,\xi}) &= \{\emptyset\} \triangleright \{pre_f(\xi)\}. \end{aligned}$$
    - *Negative effect*  $f \in F_v$  in the action schema  $\xi \in \Xi$ :
 
$$\begin{aligned} pre(\text{programDel}_{f,\xi}) &= \{pre_f(\xi), \neg del_f(\xi), \neg add_f(\xi)\}, \\ cond(\text{programDel}_{f,\xi}) &= \{\emptyset\} \triangleright \{del_f(\xi)\}. \end{aligned}$$
    - *Positive effect*  $f \in F_v$  in the action schema  $\xi \in \Xi$ :
 
$$\begin{aligned} pre(\text{programAdd}_{f,\xi}) &= \{\neg pre_f(\xi), \neg del_f(\xi), \neg add_f(\xi)\}, \\ cond(\text{programAdd}_{f,\xi}) &= \{\emptyset\} \triangleright \{add_f(\xi)\}. \end{aligned}$$
  2. Actions for applying an already programmed operator schema  $\xi \in \Xi$  bounding it with objects  $\omega \subseteq \Omega^{ar(\xi)}$ 

$$\begin{aligned} pre(\text{apply}_{\xi,\omega}) &= \{header(name(\xi), pars(\xi))\} \cup \\ &\quad \{pre_f(\xi) \implies p(\omega)\}_{\forall p \in \Psi, f=p(pars(\xi))}, \\ cond(\text{apply}_{\xi,v,v'}) &= \{del_f(\xi)\} \triangleright \{\neg p(\omega)\}_{\forall p \in \Psi, f=p(pars(\xi))}, \\ &\quad \{add_f(\xi)\} \triangleright \{p(\omega)\}_{\forall p \in \Psi, f=p(pars(\xi))}. \end{aligned}$$
  3. Actions for changing the active example where the action model is currently being validated.
 
$$\begin{aligned} pre(\text{validate}_i) &= G_i \cup \{test_j\}_{j=1 \leq j < i}, \\ cond(\text{validate}_i) &= \{\emptyset\} \triangleright \{test_i\}. \end{aligned}$$

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

*Proof sketch.* Once an operator schema is programmed it cannot be modified and can only be applied. The only way of achieving a *test* fluent is by applying a sequence of programmed operator schema until achieving the goal state defined by its associated label starting from the initial state of the corresponding label. If this is done for all the labels (all the input examples) it means that the programmed action model is compliant with the learning input knowledge and hence, it is a solution to the action model learning task.  $\square$

The compilation is valid for partially specified action models since known preconditions and effects (fluents  $pre_f(\xi)$ ,  $del_f(\xi)$  and  $add_f(\xi)$ ) can be part of the initial state  $I_\Lambda$ . The approach allows also transfer learning where the action model generated for a given sub-task is encoded as *already programmed* for learning new action models in more challenging tasks.

### Learning action models from example plans using a classical planner

The compilation can be extended to the learning scenario defined by  $\Lambda$  in which a set of plans  $\Pi$  is also available. Each plan  $\pi_i \in \Pi$ ,  $1 \leq i \leq t$ , is a solution to the corresponding classical planning instance  $P_i = \langle F, A, I_i, G_i \rangle$  defined above. The compilation extensions are:

- $F_\Lambda$  includes the new set of fluents  $F_\Pi = \{plan(name(\xi), j, \Omega^{ar(\xi)})\}$  for encoding the  $j$  steps of the  $1 \leq i \leq t$  plans in  $\Pi$  with  $F_{\Pi_i} \subseteq F_\Pi$  the fluents encoding the plan corresponding to the  $i^{th}$  example. In addition fluents  $at_j$  and  $next_{j,j2}$ ,  $1 \leq j < j2 \leq n$ , represent the plan step where the programmed action model is validated ( $n$  is the max length of a plan in  $\Pi$ ).
- $I_\Lambda$  is extended with the fluents from  $F_{\Pi_1}$  that encode the plan  $\pi_1 \in \Pi$  for solving  $P_1$ , and the fluents  $at_1$  and  $\{next_{j,j2}\}$ ,  $1 \leq j < j2 \leq n$ , for indicating the plan step where to start validating the programmed action model. Goals  $G_\Lambda$  are like in the original compilation.
- With respect to the actions in  $A_\Lambda$ ,
  1. The actions for programming the preconditions/effects of a given operator schema are the same.
  2. The actions for applying an operator schema have an extra precondition  $f \in F_{\Pi_i}$  that encodes the current plan step and extra conditional effect  $\{at_j\} \triangleright \{\neg at_j, at_{j+1}\}_{\forall j \in [1,n]}$  for advancing the plan step.
  3. The actions for changing the active test have an extra precondition,  $at_{|\Pi_i|}$ , to indicate that we simulated the full current plan  $\Pi_i$  and extra conditional effects to load the next plan  $\Pi_{i+1}$  where to validate the programmed action model:

$$\begin{aligned} \{f\} &\triangleright \{\neg f\}_{f \in F_{\Pi_i}}, \\ \{\emptyset\} &\triangleright \{f\}_{f \in F_{\Pi_{i+1}}}, \\ \{\emptyset\} &\triangleright \{\neg at_{|\Pi_i|}, at_1\}. \end{aligned}$$

### Generating informative learning examples

TBD.

Observation: If a predicate is not appearing either in the initial not in the final state it will not appear in the learned action model.

Question: Can we propose a search algorithm that generates states with the full diversity of predicates of a given domain? What about exploring the full state space for small problems (3-4 blocks blocksworld)?

## Evaluation

### Learning action models from example plans

The performance of our learning approach is evaluated for different levels of available input knowledge and using different sources for collecting this input knowledge. In all the cases we assess the performance of our learning approach using the cardinality of the *symmetric difference* sets that are computed between the set of preconditions, del and add effects (1), in the learned model and (2), in the actual models. In all the experiments the compilation is solved using the SAT-based classical planner MADAGASCAR (Rintanen 2014).

Table 1 shows the mean error and standard deviation of the learned models with respect to the actual action models when (1) using *hand-picked* examples, (2) examples collected using the classical planner FAST-DOWNWARD (Helmert 2006) and (3) examples collected *randomly*. 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

TBD.

## Related work

### Conclusions

This paper presents a novel approach for learning classical planning action models from minimal input knowledge and using exclusively existing classical planners. Learning action models from examples allows the reformulation of a domain theory. An interesting research direction is the study of domain reformulation using features that allow more compact solutions like the *reachable* or *movable* features in the Sokoban domain.

## References

- Amir, E., and Chang, A. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research* 33:349–402.
- 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.

Francés, G.; Ramrez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action representations are over-rated: 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.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)* 26:191–246.

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*.

Michalski, R. S.; Carbonell, J. G.; and Mitchell, T. M. 2013. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media.

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. 2017. Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence*.

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.

	Preconditions				Del Effects				Add Effects		
	Hand	Planner	Rand		Hand	Planner	Rand		Hand	Planner	Rand
Blocks	12.50 $\pm$ 21.65	41.67 $\pm$ 36.32			0.00	16.70 $\pm$ 3.36			5.00 $\pm$ 8.66	19.38 $\pm$ 19.40	
Grippper	19.44 $\pm$ 14.16	40.00 $\pm$ 20.00			0.00	9.17 $\pm$ 0.83			4.76 $\pm$ 6.73	74.17 $\pm$ 15.83	
Miconic	33.15 $\pm$ 16.91	61.67 $\pm$ 2.89			0.00	8.33 $\pm$ 14.43			10.00 $\pm$ 10.00	29.17 $\pm$ 18.16	
Visitall	0.00	100.00			0.00	20.00			0.00	0.00	
Llogistics	100.00	100.00			19.03 $\pm$ 6.97	19.03 $\pm$ 6.97			80.97 $\pm$ 6.97	80.97 $\pm$ 6.97	
Zenotravel	100.00	100.00			20.54 $\pm$ 9.14	20.54 $\pm$ 9.14			77.70 $\pm$ 8.55	77.70 $\pm$ 8.55	

Table 1: Mean error and standard deviation of the learned models when using hand-picked examples and examples collected using the classical planner Fast-Downward.