Learning STRIPS Action Models
with Classical Planning

Diego Aineto, Sergio Jiménez, 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 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 pair of initial and final states (no intermediate action or state is given). Moreover, the compilation accepts partially specified action models and it can be used to validate whether the observation of a plan execution follows a given STRIPS action model, even if this model is not fully specified.

Introduction

Besides *plan synthesis* (Ghallab, Nau, and Traverso 2004), planning action models are also useful for *plan/goal recognition* (Ramírez 2012). In both planning tasks, an automated planner is required to reason about action models that correctly and completely capture the possible world transitions (Geffner and Bonet 2013). Unfortunately, building planning action models is complex, even for planning experts, and this knowledge acquisition task is a bottleneck that limits the potential of AI planning (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 learning STRIPS action models, the vanilla action model for planning (Fikes and Nilsson 1971), is not straightforward though:

- The *input* to ML algorithms (the learning/training data) is usually a finite set of vectors that represent the value of some fixed object features. The input for learning planning action models is, however, observations of plan executions (where each plan possibly has a different length).

- The *output* of ML algorithms is usually a scalar value (an integer, in the case of *classification* tasks, or a real value, in the case of *regression* tasks). When learning action models the output is, for each action, the preconditions, negative and positive effects that define the possible state transitions.

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

1. It is flexible to various amounts of input knowledge. Learning examples range from a set of plans (with their corresponding initial and final states) to just a pair of initial and final states where no intermediate state or action is observed.

2. It accepts previous knowledge about the structure of the actions in the form of partially specified action models. In the extreme, the compilation can validate whether an observed plan execution is valid for a given STRIPS action model, even if this model is not fully specified.

The second section of the paper formalizes the classical planning model, its extension to *conditional effects* (a requirement of the proposed compilation) and the STRIPS action model (the output of the addressed learning task). The third section formalizes the task of learning action models with different amounts of available input knowledge. The fourth and fifth sections describe our compilation approach to tackle the formalized learning tasks. Finally, the last sections show the experimental evaluation, discuss the strengths and weaknesses of the compilation approach and propose several opportunities for future research.

Background

This section defines the planning model and the output of the learning tasks addressed in the paper.

Classical planning with conditional effects

Our approach to learning STRIPS action models is compiling this learning task into a classical planning task with con-

ditional effects. Conditional effects allow us to compactly define actions whose effects depend on the current state. Supporting conditional effects is now a requirement of the IPC (Vallati et al. 2015) and many classical planners cope with conditional effects without compiling them away.

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 (without loss of generality, we will assume that L does not contain conflicting values). We use $\mathcal{L}(F)$ to denote the set of all literal sets on F; i.e. all partial assignments of values to fluents.

A *state* s is a full assignment of values to fluents; $|s| = |F|$, so the size of the state space is $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions but often we will abuse of notation by defining a state s only in terms of the fluents that are true in s, as it 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. An action $a \in A$ is defined with *preconditions*, $\mathsf{pre}(a) \subseteq \mathcal{L}(F)$, *positive effects*, $\mathsf{eff}^+(a) \subseteq \mathcal{L}(F)$, and *negative effects* $\mathsf{eff}^-(a) \subseteq \mathcal{L}(F)$. We say that an action $a \in A$ is *applicable* in a state s iff $\mathsf{pre}(a) \subseteq s$. The result of applying a in s is the *successor state* denoted by $\theta(s, a) = \{s \setminus \mathsf{eff}^-(a)) \cup \mathsf{eff}^+(a)\}$.

An action $a \in A$ with conditional effects is defined as a set of *preconditions* $\mathsf{pre}(a)$ and a set of *conditional effects* $\mathsf{cond}(a)$. Each conditional effect $C \triangleright E \in \mathsf{cond}(a)$ is composed of two sets of literals: $C \subseteq \mathcal{L}(F)$, the *condition*, and $E \subseteq \mathcal{L}(F)$, the *effect*. An action $a \in A$ is *applicable* in a state s iff $\mathsf{pre}(a) \subseteq s$, and the *triggered effects* resulting from the action application are the effects whose conditions hold in s:

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

The result of applying action a in state s is the *successor* state $\theta(s, a) = \{s \setminus \mathsf{eff}_c^-(s, a)) \cup \mathsf{eff}_c^+(s, a)\}$ where $\mathsf{eff}_c^-(s, a) \subseteq triggered(s, a)$ and $\mathsf{eff}_c^+(s, a) \subseteq triggered(s, a)$ are, respectively, the triggered *negative* and *positive* effects.

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

STRIPS action schemas and *variable name* objects

Our approach is aimed at learning PDDL action schemas that follow the STRIPS requirement (McDermott et al. 1998; Fox and Long 2003). Figure 1 shows the *stack* schema of a four-operator *blocksworld* domain (Slaney and Thiébaux 2001) encoded in PDDL.

To formalize the output of the learning task, we assume that fluents F are instantiated from a set of *predicates* Ψ,

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

Figure 1: The *stack* operator schema of the *blocksworld* domain specified in PDDL.

as in PDDL. Each predicate $p \in \Psi$ has an argument list of arity $ar(p)$. Given a set of *objects* Ω, the set of fluents F is induced by assigning objects in Ω to the arguments of the predicates in Ψ; i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$, where Ω^k is the k-th Cartesian power of Ω.

Let $\Omega_v = \{v_i\}_{i=1}^{\max_{a \in A} ar(a)}$ be a new set of objects denoted as *variable names* ($\Omega \cap \Omega_v = \emptyset$). Ω_v is bound to the maximum arity of an action in a given planning frame. For instance, in a three-block *blocksworld*, $\Omega = \{block_1, block_2, block_3\}$ while $\Omega_v = \{v_1, v_2\}$ because the operators with the maximum arity, stack and unstack, have two parameters each.

Let F_v be a new set of fluents, $F \cap F_v = \emptyset$, that results from instantiating the predicates in Ψ using exclusively objects of Ω_v. F_v defines the elements of the preconditions and effects of an action schema. For instance, in the *blocksworld* domain, $F_v=$\{handempty, holding(v_1), holding(v_2), clear(v_1), clear(v_2), ontable(v_1), ontable(v_2), on(v_1,v_1), on(v_1,v_2), on(v_2,v_1), on(v_2,v_2)\}.

Finally, we assume that an action $a \in A$ is instantiated from a STRIPS operator schema $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi) \rangle$ where:

- $head(\xi) = \langle name(\xi), pars(\xi) \rangle$ is the operator *header* defined by its name and the corresponding *variable names*, $pars(\xi) = \{v_i\}_{i=1}^{ar(\xi)}$. For instance, the headers of a four-operator *blocksworld* domain are: pickup(v_1), putdown(v_1), stack(v_1, v_2) and unstack(v_1, v_2).

- $pre(\xi) \subseteq F_v$ is the set of preconditions, $del(\xi) \subseteq F_v$ the negative effects and $add(\xi) \subseteq F_v$ the positive effects such that $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ and $pre(\xi) \cap add(\xi) = \emptyset$.

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 the plans are known, is straightforward. When intermediate states are available, operator schemas are derived lifting the literals that change between the pre and post-state of each action execution. Preconditions of an action are derived lifting the minimal set of literals that appears in all the pre-states of the corresponding action (Jiménez et al. 2012).

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

Learning from (initial, final) state pairs. This learning task amounts to observing agents acting in the world but

watching only the result of their plans execution. No intermediate information about the actions in the plans is given. This learning task is formalized as $\Lambda = \langle \Psi, \Sigma \rangle$:

- Ψ is the set of predicates that define the abstract state space of a given planning domain.

- $\Sigma = \{\sigma_1, \ldots, \sigma_\tau\}$ is a set of $(initial, final)$ state pairs called *labels*. Each label $\sigma_t = (s_0^t, s_n^t)$, $1 \leq t \leq \tau$, comprises the *final* state s_n^t resulting from executing an unknown plan $\pi_t = \langle a_1^t, \ldots, a_n^t \rangle$ in the *initial* state s_0^t.

Learning from labeled plans. We augment the input knowledge with the actions executed by the observed agent and define the learning task $\Lambda' = \langle \Psi, \Sigma, \Pi \rangle$:

- $\Pi = \{\pi_1, \ldots, \pi_\tau\}$ is a given set of example plans where $\pi_t = \langle a_1^t, \ldots, a_n^t \rangle$, $1 \leq t \leq \tau$, is an action sequence that induces the corresponding state sequence $\langle s_0^t, s_1^t, \ldots, s_n^t \rangle$ such that a_i^t, $1 \leq i \leq n$, is applicable in s_{i-1}^t and generates $s_i^t = \theta(s_{i-1}^t, a_i^t)$.

Figure 2 shows an example of a learning task Λ' of the *blocksworld* domain. This task has a single learning example, $\Pi = \{\pi_1\}$ and $\Sigma = \{\sigma_1\}$, that corresponds to observing the execution of an eight-action plan ($|\pi_1| = 8$) for inverting a four-block tower.

Learning from partially specified action models. In case that partially specified operator schemas are available, we can incorporate this information within the learning task. The new leaning task is defined as $\Lambda'' = \langle \Psi, \Sigma, \Pi, \Xi_0 \rangle$:

- Ξ_0 is a partially specified action model in which some preconditions and effects are known a priori.

A solution to Λ is a set of operator schemas Ξ that is compliant with the predicates in Ψ and the set of initial and final states Σ. In a Λ learning scenario, a solution must not only determine a possible STRIPS action model but also the plans π_t, $1 \leq t \leq \tau$, that explain the given labels Σ using the learned model. A solution to Λ' is a set of STRIPS operator schemas Ξ (one schema $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi) \rangle$ for each action that has a different name in the example plans Π) that is compliant with the predicates Ψ, the example plans Π, and their corresponding labels Σ. Finally, a solution to Λ'' is a set of STRIPS operator schemas Ξ that is also compliant with the provided partially specified action model Ξ_0.

Learning STRIPS action models with planning

In our approach, a learning task Λ, Λ' or Λ'' is solved by compiling it into a classical planning task with conditional effects. The intuition behind the compilation is that a solution to the resulting classical planning task is a sequence of actions that:

1. *Programs the* STRIPS *action model* Ξ. A solution plan has a *prefix* that, for each $\xi \in \Xi$, determines the fluents from F_v that belong to $pre(\xi)$, $del(\xi)$ and $add(\xi)$.

2. *Validates the programmed* STRIPS *action model* Ξ *in the given input knowledge* (the labels Σ and Π, and/or Ξ_0 if available). For every label $\sigma_t \in \Sigma$, a solution plan

```
;;; Predicates in Ψ
(handempty) (holding ?o  - object)
(clear ?o - object) (ontable ?o - object)
(on ?o1 - object ?o2 - object)

;;; Plan π₁            ;;; Label σ₁ = (s₀¹, sₙ¹)
0: (unstack A B)
1: (putdown A)          A        D
2: (unstack B C)
3: (stack B A)          B        C
4: (unstack C D)        C        B
5: (stack C B)
6: (pickup D)           D        A
7: (stack D C)
```
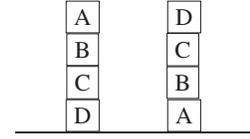
Figure 2: Learning task of the *blocksworld* domain from a single labeled plan.

has a postfix that produces a final state s_n^t using the programmed action model Ξ in the corresponding initial state s_0^t. This process is the validation of the programmed action model Ξ with the set of learning examples $1 \leq t \leq \tau$.

To formalize our compilation we first define a set of 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 but different initial states and goals). Fluents F are built instantiating the predicates in Ψ with the objects of the input labels Σ. Formally, $\Omega = \bigcup_{1 \leq t \leq \tau} obj(s_0^t)$, where obj is a function that returns the 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 $s_0^t \in \sigma_t$, and the goals G_t are defined by the state $s_n^t \in \sigma_t$.

We can now formalize the compilation approach. We start with Λ as it requires the least input knowledge. Given a learning task $\Lambda = \langle \Psi, \Sigma \rangle$, the compilation outputs a classical planning task $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$:

- F_Λ extends F with:

  - Fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_v$ and $\xi \in \Xi$ that represent the programmed action model. If a fluent of $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ holds, it means that f is a precondition/negative effect/positive effect of the operator schema $\xi \in \Xi$. For instance, the preconditions of the *stack* schema (Figure 1) are represented by fluents `pre_holding_stack_v₁` and `pre_clear_stack_v₂`.

  - A fluent $mode_{prog}$ to indicate whether the operator schemas are being programmed or validated (when already programmed)

  - Fluents $\{test_t\}_{1 \leq t \leq \tau}$ which represent the examples where the action model will be validated.

- I_Λ contains the fluents from F that encode s_0^1 (the initial state of the first label), the fluents in every $pre_f(\xi) \in F_\Lambda$ and the fluent $mode_{prog}$ set to true. Our compilation assumes that any operator schema is initially programmed with every possible precondition (the most specific learning hypothesis), no negative effect and no positive effect.

- $G_\Lambda = \bigcup_{1 \leq t \leq \tau}\{test_t\}$ indicates that the programmed action model is validated in all the learning examples.

- A_Λ contains actions of three kinds:

1. Actions for *programming* an operator schema $\xi \in \Xi$:
   - Actions for **removing** a *precondition* $f \in F_v$ from ξ.

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

   - Actions for **adding** a *negative* or *positive* effect $f \in F_v$ to ξ.

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

2. Actions for *applying* an already programmed operator schema $\xi \in \Xi$ bound to the objects $\omega \subseteq \Omega^{ar(\xi)}$. We assume operators headers are known so the binding of ξ is done implicitly by order of appearance of the action parameters, i.e. variables $pars(\xi)$ are bound to the objects in ω that appear in the same position. Figure 3 shows the PDDL encoding of the action for applying a programmed operator $stack$.

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

3. Actions for *validating* the learning example $1 \leq t \leq \tau$.

$$\begin{aligned}\mathsf{pre}(\mathsf{validate}_t) =&G_t \cup \{test_j\}_{1 \leq j < t} \\ &\cup \{\neg test_j\}_{t \leq j \leq \tau} \cup \{\neg mode_{prog}\}, \\ \mathsf{cond}(\mathsf{validate}_t) =&\{\emptyset\} \rhd \{test_t\} \cup \{\neg f\}_{\forall f \in G_t, f \notin I_{t+1}} \\ &\cup \{f\}_{\forall f \in I_{t+1}, f \notin G_t}.\end{aligned}$$

Lemma 1. *A classical plan π that solves P_Λ induces an action model Ξ that solves the learning task Λ.*

Proof sketch. Once operator schemas Ξ are programmed, they can only be applied and validated according to the $mode_{prog}$ fluent. To solve P_Λ, goals $\{test_t\}, 1 \leq t \leq \tau$ can only be achieved by executing an applicable sequence of programmed operator schemas that reaches the final state s_n^t, defined in σ_t, starting from s_0^t. If this is achieved for all the input examples $1 \leq t \leq \tau$, it means that the programmed action model Ξ is compliant with the provided input knowledge and hence it is a solution to Λ. $\square$

The compilation is *complete* in the sense that it does not discard any possible STRIPS action model. The size of the classical planning task P_Λ depends on:

- The arity of the actions headers in Ξ and the predicates Ψ of the learning task. The larger the arity, the larger the F_v set, which in turn defines the size of the fluent sets $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ and the corresponding set of *programming* actions.

```
(: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 3: PDDL action for applying an already programmed schema $stack$ (implications coded as disjunctions).

- The number of learning examples. The larger this number, the more $test_t$ fluents and validate$_t$ actions in P_Λ.

Constraining the learning hypothesis space with additional input knowledge

In this section, we show that further input knowledge can be used to constrain the space of possible action models and to make the learning task more practicable.

Labeled plans

We extend the compilation to consider labeled plans. Given a learning task $\Lambda' = \langle \Psi, \Sigma, \Pi \rangle$, the compilation outputs a classical planning task $P_{\Lambda'} = \langle F_{\Lambda'}, A_{\Lambda'}, I_{\Lambda'}, G_{\Lambda'} \rangle$:

- $F_{\Lambda'}$ extends F_Λ with $F_\Pi = \{plan(name(\xi), \Omega^{ar(\xi)}, j)\}$, the fluents to code the steps of the plans in Π, where $F_{\pi_t} \subseteq F_\Pi$ encodes $\pi_t \in \Pi$. Fluents at_j and $next_{j,j+1}$, $1 \leq j < n$, are also added to represent the current plan step and to iterate through the steps of a plan.

- $I_{\Lambda'}$ extends I_Λ with fluents F_{π_1} plus fluents at_1 and $\{next_{j,j+1}\}$, $1 \leq j < n$, to indicate the plan step where

the action model is validated. As in the original 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 $\{at_j\} \rhd \{\neg at_j, at_{j+1}\}_{\forall j \in [1,n]}$ for advancing to the next plan step. With this mechanism we ensure that these actions are applied in the same order as in the example plans.

  3. The actions for *validating* the current learning example have an extra precondition, $at_{|\pi_t|}$, to indicate that the current plan π_t is fully executed and extra conditional effects to remove plan π_t and initiate the next plan π_{t+1}:

$$\{\emptyset\} \rhd \{\neg at_{|\pi_t|}, at_1\} \cup \{\neg f\}_{f \in F_{\pi_t}} \cup \{f\}_{f \in F_{\pi_{t+1}}}.$$

Partially specified action models

The known preconditions and effects of a partially specified action model are encoded as fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$ set to true in the initial state $I_{\Lambda'}$. The programming actions, programPre$_{f,\xi}$ and programEff$_{f,\xi}$, become now unnecessary and they are removed from $A_{\Lambda'}$, thus making the planning task $P_{\Lambda'}$ be easier to solve.

To illustrate this, the plan of Figure 4 is a solution to a learning task $\Lambda'' = \langle \Psi, \Sigma, \Pi, \Xi_0 \rangle$ for acquiring the *blocksworld* action model where operator schemas for `pickup`, `putdown` and `unstack` are specified in Ξ_0. This plan programs and validates the operator schema `stack` from *blocksworld* using the plan π_1 and label σ_1 shown in Figure 2. Plan steps $[0, 8]$ program the preconditions of the `stack` operator, steps $[9, 13]$ program the operator effects and steps $[14, 22]$ validate the programmed operators following the plan π_1 shown in Figure 2.

In the extreme, when a fully specified STRIPS action model is given in Ξ_0, the compilation validates whether an observed plan follows the given model. In this case, if a solution plan is found for $P_{\Lambda'}$, it means that the given action model is *valid* for the provided examples. If $P_{\Lambda'}$ is unsolvable then it means that the action model is invalid because 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 (Fox and Long 1998). Therefore, one can get rid of the mechanism for programming these predicates in the effects of any action schema while keeping the compilation complete. 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 names* Ω_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 Ω_v) can also be discarded for every $\xi \in \Xi$.

```
00 : (program_pre_clear_stack_v1)
01 : (program_pre_handempty_stack)
02 : (program_pre_holding_stack_v2)
03 : (program_pre_on_stack_v1_v1)
04 : (program_pre_on_stack_v1_v2)
05 : (program_pre_on_stack_v2_v1)
06 : (program_pre_on_stack_v2_v2)
07 : (program_pre_ontable_stack_v1)
08 : (program_pre_ontable_stack_v2)
09 : (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 4: Plan for programming and validating the *stack* schema using plan π_1 and label σ_1 (shown in Figure 2) as well as previously specified operator schemas for *pickup*, *putdown* and *unstack*.

Static predicates can also constrain the space of possible preconditions by looking at the given set of labels Σ. One can assume that if a precondition $f \in F_v$ (s.t. $f \in F_v$ is an instantiation of a static predicate in Ω_v) is not compliant with the labels in Σ then fluents $pre_f(\xi)$ and actions programPre$_{f,\xi}$ can be discarded for every $\xi \in \Xi$. For instance, in the *zenotravel* domain, $pre\_next\_board\_v1\_v1$, $pre\_next\_debark\_v1\_v1$, $pre\_next\_fly\_v1\_v1$, $pre\_next\_zoom\_v1\_v1$, $pre\_next\_refuel\_v1\_v1$ can be discarded (and their corresponding programming actions) because a precondition (next ?v1 ?v1 - flevel) will never hold in any state of Σ.

On the other hand, fluents $pre_f(\xi)$ and actions programPre$_{f,\xi}$ are discardable 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 Ω_v) is not possible according to Π. Back to the *zenotravel* domain, if an example plan $\pi_t \in \Pi$ contains the action (fly plane1 city2 city0 fl3 fl2) and the corresponding label $\sigma_t \in \Sigma$ contains the static literal (next fl2 fl3) but does not contain (next fl2 fl2), (next fl3 fl3) or (next fl3 fl2), the only possible precondition that would include the static predicate is $pre\_next\_fly\_v5\_v4$.

Evaluation

This section evaluates the performance of our approach for learning STRIPS action models with different amounts of available input knowledge.

Setup. The domains used in the evaluation are IPC domains that satisfy the STRIPS requirement (Fox and

Long 2003), taken from the PLANNING.DOMAINS repository (Muise 2016). We only used 5 learning examples for each domain and we fixed the examples for all the experiments so that we can evaluate the impact of the input knowledge in the quality of the learned models. All experiments are run on an Intel Core i5 3.10 GHz x 4 with 8 GB of RAM.

The classical planner we used to solve the instances that result from our compilations is MADAGASCAR (Rintanen 2014). We used MADAGASCAR due to its ability to deal with planning instances populated with dead-ends. In addition, SAT-based planners can apply the actions for programming preconditions in a single planning step (in parallel) because these actions do not interact. Actions for programming action effects can also be applied in a single planning step reducing significantly the planning horizon.

Metrics. The quality of the learned models is measured with the *precision* and *recall* metrics. These two metrics are frequently used in *pattern recognition*, *information retrieval* and *binary classification* and are more informative that simply counting the number of errors in the learned model or computing the *symmetric difference* between the learned and the reference model (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). 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).

Given the syntax-based nature of these metrics, it may happen that they report low scores for learned models that are actually good but correspond to *reformulations* of the actual model; i.e. a learned model semantically equivalent but syntactically different to the reference model. This mainly occurs when the learning task is under-constrained.

Learning from labeled plans

We start evaluating our approach with tasks $\Lambda' = \langle \Psi, \Sigma, \Pi \rangle$, where *labeled plans* are available. We then repeat the evaluation but exploiting potential *static predicates* computed from Σ, which are the predicates that appear unaltered in the initial and final states in every $\sigma_t \in \Sigma$. Static predicates are used to constrain the space of possible action models as explained in the previous section.

Table 1 shows the obtained results. Precision (**P**) and recall (**R**) are computed separately for the preconditions (**Pre**), positive effects (**Add**) and negative Effects (**Del**), while the last two columns of each setting and the last row report averages values. We can observe that identifying static predicates leads to models with better precondition *recall*. This fact evidences that many of the missing preconditions corresponded to static predicates because there is no incentive to learn them as they always hold (Gregory and Cresswell 2015).

Table 2 reports the total planning time, the preprocessing time (in seconds) invested by MADAGASCAR to solve the

planning instances that result from our compilation as well as the number of actions of the solution plans. All the learning tasks are solved in a few seconds. Interestingly, one can identify the domains with static predicates by just looking at the reported plan length. In these domains some of the preconditions that correspond to static predicates are directly derived from the learning examples and therefore fewer programming actions are required. When static predicates are identified, the resulting compilation is also much more compact and produces smaller planning/instantiation times.

Learning from partially specified action models

We evaluate now the ability of our approach to support partially specified action models; that is, addressing learning tasks of the kind $\Lambda'' = \langle \Psi, \Sigma, \Pi, \Xi_0 \rangle$. In this experiment, the model of half of the actions is given in Ξ_0 as an extra input of the learning task.

Tables 3 and 4 summarize the obtained results, which include the identification of static predicates. We only report the *precision* and *recall* of the *unknown* actions since the values of the metrics of the *known* action models is 1.0. In this experiment, a low value of *precision* or *recall* has a greater impact than in the corresponding Λ' tasks because the evaluation is done only over half of the actions. This occurs, for instance, in the precondition *recall* of domains such as *Floortile*, *Gripper* or *Satellite*.

Remarkably, the overall *precision* is now 0.98, which means that the contents of the learned models is highly reliable. The value of *recall*, 0.87, is an indication that the learned models still miss some information (preconditions are again the component more difficult to be fully learned). Overall, the results confirm the previous trend: the more input knowledge of the task, the better the models and the less planning time. Additionally, the solution plans required for this task are smaller because it is only necessary to program half of the actions (the other half are included in the input knowledge Ξ_0). *Visitall* and *Hanoi* are excluded from this evaluation because they only contain one action schema.

Learning from (initial,final) state pairs

Finally, we evaluate our approach when input plans are not available and thereby the planner must not only compute the action models but also the plans that satisfy the input labels. Table 5 and 6 summarize the results obtained for the task $\Lambda = \langle \Psi, \Sigma, \Xi_0 \rangle$ using static predicates. Values for the *Zenotravel* and *Grid* domains are not reported because MADAGASCAR was not able to solve the corresponding planning tasks within a 1000 sec. time bound. The values of *precision* and *recall* are significantly lower than in Table 1. Given that the learning hypothesis space is now fairly under-constrained, actions can be reformulated and still be compliant with the inputs (e.g. the *blocksworld* operator stack can be *learned* with the preconditions and effects of the unstack operator and vice versa). We tried to minimize this effect with the additional input knowledge (static predicates and partially specified action models) and yet the results are below the scores obtained when learning from labeled plans.

	No Static								Static							
	Pre		Add		Del				Pre		Add		Del			
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Blocks	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Driverlog	1.0	0.36	0.75	0.86	1.0	0.71	0.92	0.64	0.9	0.64	0.56	0.71	0.86	0.86	0.78	0.73
Ferry	1.0	0.57	1.0	1.0	1.0	1.0	1.0	0.86	1.0	0.57	1.0	1.0	1.0	1.0	1.0	0.86
Floortile	0.52	0.68	0.64	0.82	0.83	0.91	0.66	0.80	0.68	0.68	0.89	0.73	1.0	0.82	0.86	0.74
Grid	0.62	0.47	0.75	0.86	0.78	1.0	0.71	0.78	0.79	0.65	1.0	0.86	0.88	1.0	0.89	0.83
Gripper	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.89	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.89
Hanoi	1.0	0.50	1.0	1.0	1.0	1.0	1.0	0.83	0.75	0.75	1.0	1.0	1.0	1.0	0.92	0.92
Miconic	0.75	0.33	0.50	0.50	0.75	1.0	0.67	0.61	0.89	0.89	1.0	0.75	0.75	1.0	0.88	0.88
Satellite	0.60	0.21	1.0	1.0	1.0	0.75	0.87	0.65	0.82	0.64	1.0	1.0	1.0	0.75	0.94	0.80
Transport	1.0	0.40	1.0	1.0	1.0	0.80	1.0	0.73	1.0	0.70	0.83	1.0	1.0	0.80	0.94	0.83
Visitall	1.0	0.50	1.0	1.0	1.0	1.0	1.0	0.83	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Zenotravel	1.0	0.36	1.0	1.0	1.0	0.71	1.0	0.69	1.0	0.64	0.88	1.0	1.0	0.71	0.96	0.79
	0.88	0.50	0.88	0.92	0.95	0.91	0.90	0.78	0.90	0.74	0.93	0.92	0.96	0.91	0.93	0.86

Table 1: *Precision* and *recall* scores for learning tasks from labeled plans without (left) and with (right) static predicates.

	No Static			Static		
	Total	Preprocess	Length	Total	Preprocess	Length
Blocks	0.04	0.00	72	0.03	0.00	72
Driverlog	0.14	0.09	83	0.06	0.03	59
Ferry	0.06	0.03	55	0.06	0.03	55
Floortile	2.42	1.64	168	0.67	0.57	77
Grid	4.82	4.75	88	3.39	3.35	72
Gripper	0.03	0.01	43	0.01	0.00	43
Hanoi	0.12	0.06	48	0.09	0.06	39
Miconic	0.06	0.03	57	0.04	0.00	41
Satellite	0.20	0.14	67	0.18	0.12	60
Transport	0.59	0.53	61	0.39	0.35	48
Visitall	0.21	0.15	40	0.17	0.15	36
Zenotravel	2.07	2.04	71	1.01	1.00	55

Table 2: Total planning time, preprocessing time and plan length for learning tasks from labeled plans without/with static predicates.

Related work

Action model learning has also been studied in domains where there is partial or missing state observability. ARMS works when no partial intermediate state is given. It defines a set of weighted constraints that must hold for the plans to be correct, and solves the weighted propositional satisfiability problem with a MAX-SAT solver (Yang, Wu, and Jiang 2007). In order to efficiently solve the large MAX-SAT representations, ARMS implements a hill-climbing method that models the actions approximately. In contrast to our model comparison validation which aims at covering all the training examples, ARMS maximizes the number of covered examples from a testing set.

SLAF also deals with partial observability (Amir and Chang 2008). Given a formula representing the initial belief state, a sequence of executed actions and the corresponding partially observed states, it builds a complete explanation of observations by models of actions through a CNF formula. The learning algorithm updates the formula of the belief state with every action and observation in the sequence and thus the final returned formula includes all consistent models. SLAF assesses the quality of the learned models with respect to the actual generative model.

LOCM only requires the example plans as input without need for providing information about predicates or states (Cresswell, McCluskey, and West 2013; Cresswell and Gregory 2011). The lack of available information is overcome by exploiting assumptions about the kind of domain model it has to generate. Particularly, it assumes a domain consists of a collection of objects (sorts) whose defined set of states can be captured by a parameterized Finite State Machine. LOP (LOCM with Optimized Plans (Gregory and Cresswell 2015)) incorporates static predicates and applies a post-processing step after the LOCM analysis that requires a set of optimal plans to be used in the learning phase. This is done to mitigate the limitation of LOCM of inducing similar models for domains with similar structures. LOP compares the learned models with the corresponding reference model.

Compiling an action model learning task into classical planning is a general and flexible approach that allows to accommodate various amounts and kinds of input knowledge and opens up a path for addressing further learning and validation tasks. For instance, the example plans in Π can be replaced or complemented by a set $\mathcal{O}$ of sequences of observations (i.e., fully or partial state observations with noisy or missing fluents (Sohrabi, Riabov, and Udrea 2016)), and learning tasks of the kind $\Lambda = \langle \Psi, \Sigma, \mathcal{O}, \Xi_0 \rangle$ would also be attainable. Furthermore, our approach seems extensible to learning other types of generative models (e.g. hierarchical models like HTN or behaviour trees) that can be more appealing than STRIPS models since they require less search effort to compute a a planning solution.

Conclusions

We presented a novel approach for learning STRIPS action models from examples using classical planning. The approach is flexible to various amounts of input knowledge and accepts partially specified action models. We also introduced the *precision* and *recall* metrics, widely used in ML, for evaluating the learned action models. These two metrics measure the soundness and completeness of the learned models and facilitate the identification of model flaws.

To the best of our knowledge, this is the first work on learning action models that is exhaustively evaluated over a wide range of domains and uses exclusively an *off-the-*

	Pre		Add		Del			
	P	R	P	R	P	R	P	R
Blocks	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Driverlog	1.0	0.71	1.0	1.0	1.0	1.0	1.0	0.90
Ferry	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.89
Floortile	0.75	0.60	1.0	0.80	1.0	0.80	0.92	0.73
Grid	1.0	0.67	1.0	1.0	1.0	1.0	0.84	0.78
Gripper	1.0	0.50	1.0	1.0	1.0	1.0	1.0	0.83
Miconic	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Satellite	1.0	0.57	1.0	1.0	1.0	1.0	1.0	0.86
Transport	1.0	0.75	1.0	1.0	1.0	1.0	1.0	0.92
Zenotravel	1.0	0.67	1.0	1.0	1.0	0.67	1.0	0.78
	0.98	0.71	1.0	0.98	1.0	0.95	0.98	0.87

Table 3: *Precision* and *recall* scores for learning tasks with partially specified action models.

	Total time	Preprocess	Plan length
Blocks	0.07	0.01	54
Driverlog	0.03	0.01	40
Ferry	0.06	0.03	45
Floortile	0.43	0.42	55
Grid	3.12	3.07	53
Gripper	0.03	0.01	35
Miconic	0.03	0.01	34
Satellite	0.14	0.14	47
Transport	0.23	0.21	37
Zenotravel	0.90	0.89	40

Table 4: Time and plan length learning for learning tasks with partially specified action models.

	Pre		Add		Del			
	P	R	P	R	P	R	P	R
Blocks	0.33	0.33	0.75	0.50	0.33	0.33	0.47	0.39
Driverlog	1.0	0.29	0.33	0.67	1.0	0.50	0.78	0.48
Ferry	1.0	0.67	0.50	1.0	1.0	1.0	0.83	0.89
Floortile	0.67	0.40	0.50	0.40	1.0	0.40	0.72	0.40
Grid	-	-	-	-	-	-	-	-
Gripper	1.0	0.50	1.0	1.0	1.0	1.0	1.0	0.83
Miconic	0.0	0.0	0.33	0.50	0.0	0.0	0.11	0.17
Satellite	1.0	0.14	0.67	1.0	1.0	1.0	0.89	0.71
Transport	0.0	0.0	0.25	0.5	0.0	0.0	0.08	0.17
Zenotravel	-	-	-	-	-	-	-	-
	0.63	0.29	0.54	0.70	0.67	0.53	0.61	0.51

Table 5: *Precision* and *recall* scores for learning from (initial,final) state pairs.

	Total time	Preprocess	Plan length
Blocks	2.14	0.00	58
Driverlog	0.09	0.00	88
Ferry	0.17	0.01	65
Floortile	6.42	0.15	126
Grid	-	-	-
Gripper	0.03	0.00	47
Miconic	0.04	0.00	68
Satellite	4.34	0.10	126
Transport	2.57	0.21	47
Zenotravel	-	-	-

Table 6: Time and plan length when learning from (initial,final) state pairs.

shelf classical planner. The work in (Stern and Juba 2017) proposes a planning compilation for learning action models from plan traces following the *finite domain* representation for the state variables. This is a theoretical study on the boundaries of the learned models and no experimental results are reported.

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

Generating *informative* examples for learning planning action models is still an open issue. Planning actions include preconditions that are only satisfied by specific sequences of actions which have low probability of being chosen by chance (Fern, Yoon, and Givan 2004). The success of recent algorithms for exploring planning tasks (Francès et al. 2017) motivates the development of novel techniques that enable to autonomously collect informative learning examples. The combination of such exploration techniques with our learning approach is an appealing research direction that opens up the door to the bootstrapping of planning action models.

In many applications, the actual actions executed by the observed agent are not available but, instead, the resulting states can be observed. We plan to extend our approach for learning from state observations as it broadens the range of application to external observers and facilitates the representation of imperfect observability, as shown in plan recogni-

tion (Sohrabi, Riabov, and Udrea 2016), as well as learning from unstructured data, like state images (Asai and Fukunaga 2018).

Acknowledgments

References

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

Asai, M., and Fukunaga, A. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *National Conference on Artificial Intelligence, AAAI-18*.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In *International Conference on Automated Planning and Scheduling, ICAPS-09*.

Cresswell, S., and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *International Conference on Automated Planning and Scheduling, ICAPS-11*.

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 *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 *International Conference on Automated Planning and Scheduling, ICAPS-04*, 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. *Journal of Artificial Intelligence Research* 20:61–124.

Francès, G.; Ramírez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely Declarative Action Descriptions are Overrated: Classical Planning with Simulators. In *International Joint Conference on Artificial Intelligence, IJCAI-17*, 4294–4301. AAAI Press.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Elsevier.

Gregory, P., and Cresswell, S. 2015. Domain Model Acquisition in the Presence of Static Relations in the LOP System. In *International Conference on Automated Planning and Scheduling, ICAPS-15*, 97–105.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2004*, 294–301. IEEE.

Jiménez, S.; De La Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *The Knowledge Engineering Review* 27(04):433–467.

Kambhampati, S. 2007. Model-lite Planning for the Web Age Masses: The Challenges of Planning with Incomplete and Evolving Domain Models. In *National Conference on Artificial Intelligence, AAAI-07*, 1601–1605.

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. http://bibbase.org/network/publication/muise-planningdomains/.

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 *International Joint Conference on Artificial Intelligence, IJCAI-16*, 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, ICAPS-17*, 4391–4397.

Slaney, J., and Thiébaux, S. 2001. Blocks World Revisited. *Artificial Intelligence* 125(1-2):119–153.

Sohrabi, S.; Riabov, A. V.; and Udrea, O. 2016. Plan Recognition as Planning Revisited. In *International Joint Conference on Artificial Intelligence, IJCAI-16*, 3258–3264. AAAI Press.

Stern, R., and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *International Joint Conference on Artificial Intelligence, IJCAI-17*, 4405–4411. AAAI Press.

Vallati, M.; Chrpa, 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.