

# Learning STRIPS Action Models from State-Constraints

Diego Aineto and Sergio Jiménez and Eva Onaindia

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València.

Camino de Vera s/n. 46022 Valencia, Spain

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

## Abstract

This paper presents a classical planning compilation for learning STRIPS action models from state-constraints. A plan that solves the classical planning task resulting from the compilation induces a STRIPS action model that is compliant with the input constraints. The compilation approach does not require observations of the precise executed actions since they are guessed by an off-the-shelf classical planner leveraging on the given state-constraints. The paper shows that evaluating the learning of STRIPS action models is non-trivial. When the learning task is low-constrained actions can be *reformulated* interchanging and/or redefining their semantic and still be compliant with the learning inputs. The paper ends introducing a novel evaluation able to asses the learning of STRIPS models even when actions are *reformulated*.

## 1 Introduction

Besides *plan synthesis* [Ghallab *et al.*, 2004], planning action models are also useful for *plan/goal recognition* [Ramírez, 2012]. At these planning tasks, automated planners are required to reason about an action model that correctly and completely captures 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].

The Machine Learning of planning action models is a promising alternative to hand-coding them and nowadays, there exist sophisticated algorithms like ARMS [Yang *et al.*, 2007], SLAF [Amir and Chang, 2008] or LOCM [Cresswell *et al.*, 2013]. Motivated by recent advances on the synthesis of different kinds of generative models with classical planning [Bonet *et al.*, 2009; Segovia-Aguas *et al.*, 2016; 2017], this paper introduces an innovative approach for the automatic learning of STRIPS action models that:

1. Is defined as a classical planning compilation, which opens the door to the *bootstrapping* of planning action models.

2. Does not require observations of the particular executed actions because they are guessed by an off-the-shelf classical planner leveraging on the given state-constraints.
3. Can assess the quality of the learned STRIPS models with respect to a *reference model*, even when learning is so low constrained that actions can be reformulated and still be compliant with the learning inputs.

## 2 Background

This section defines the planning models used in this work, the input of the addressed learning task (the state constraints) and the output of the learning task, an STRIPS action model.

### 2.1 Classical planning

We use  $F$  to denote the set of *fluents* (propositional variables) describing a state. A *literal*  $l$  is a valuation of a fluent  $f \in F$ , i.e. either  $l = f$  or  $l = \neg f$ . A set of literals  $L$  represents a partial assignment of values to fluents (WLOG we assume that  $L$  does not assign conflicting values to any fluent). 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, i.e.  $|s| = |F|$ , so the size of the state space is  $2^{|F|}$ . Explicitly including negative literals  $\neg f$  in states simplifies subsequent definitions but often, we will 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  $a \in A$  comprises three sets of literals:

- $\text{pre}(a) \subseteq \mathcal{L}(F)$ , called *preconditions*, the literals that must hold for the action  $a \in A$  to be applicable.
- $\text{eff}^+(a) \subseteq \mathcal{L}(F)$ , called *positive effects*, that defines the fluents set to true by the application of the action  $a \in A$ .
- $\text{eff}^-(a) \subseteq \mathcal{L}(F)$ , called *negative effects*, that defines the fluents set to false by the action application.

We say that an action  $a \in A$  is *applicable* in a state  $s$  iff  $\text{pre}(a) \subseteq s$ . The result of applying  $a$  in  $s$  is the *successor state* denoted by  $\theta(s, a) = \{s \setminus \text{eff}^-(a)\} \cup \text{eff}^+(a)$ .

A *classical planning problem* is a tuple  $P = \langle F, A, I, G \rangle$ , where  $I$  is an initial state and  $G \subseteq \mathcal{L}(F)$  is a goal condition.

A *plan* for  $P$  is an action sequence  $\pi = \langle a_1, \dots, a_n \rangle$  that induces the *state trajectory*  $\langle s_0, s_1, \dots, s_n \rangle$  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 length* is denoted with  $|\pi| = n$ . A plan  $\pi$  *solves*  $P$  iff  $G \subseteq s_n$ , i.e. if the goal condition is satisfied at the last state reached after following the application of the plan  $\pi$  in the initial state  $I$ .

## 2.2 Classical planning with conditional effects

Our approach for learning STRIPS action models is compiling the learning task into a classical planning task with conditional 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 *International Planning Competition* [Vallati *et al.*, 2015] and many classical planners cope with conditional effects without compiling them away.

An action  $a \in A$  is now defined as a set of *preconditions*  $\text{pre}(a) \in \mathcal{L}(F)$  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 a state  $s$  if and only if  $\text{pre}(a) \subseteq s$ , and the *triggered effects* resulting from the action application are the effects whose conditions hold in  $s$ :

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

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

## 2.3 State-constraints

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

A useful kind of state-constraints are state invariants, traditionally used for computing more compact state representations [Helmert, 2009] and/or for making *satisfiability planning* and *backward search* more efficient [Rintanen, 2014; Alcázar and Torralba, 2015]. A *state invariant* is a formula  $\phi$  that holds at the initial state of a given classical planning problem,  $I \models \phi$ , and at every state  $s$  that is reachable from  $I$ . Given a planning problem  $P = \langle F, A, I, G \rangle$ , the formula  $\phi_{I,A}^*$  represents the *strongest invariant* and exactly characterizes the set of all states reachable from  $I$  using the actions  $A$ . A *mutex* (mutually exclusive) is a particular state invariant that takes the form of a binary clause and indicates a pair of different properties that cannot be simultaneously true [Kautz and Selman, 1999]. For instance in a three-blocks *blocksworld*,  $\phi_1 = \neg \text{on}(\text{block}_1, \text{block}_2) \vee \neg \text{on}(\text{block}_1, \text{block}_3)$  are mutex because  $\text{block}_1$  can only be on top of a single block.

```
(: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: STRIPS operator schema coding, in PDDL, the *stack* action from the *blocksworld*.

A *lifted invariant* (also called schematic invariant) is a state invariant defined as a first order formula [Rintanen and others, 2017]. A *domain invariant* is an instance-independent state invariant, i.e. holds for any possible initial state, so often they are lifted invariants. For instance in the *blocksworld*,  $\phi_2 = \forall x : (\neg \text{handempty} \vee \neg \text{holding}(x))$ , is a *lifted domain mutex* because the robot hand is never empty and holding a block at the same time. This work exploits *domain invariants* to constrain the space of possible STRIPS action models.

## 2.4 STRIPS action schemes

This work addresses the learning of PDDL action schemes that follow the STRIPS requirement [McDermott *et al.*, 1998; Fox and Long, 2003]. Figure 1 shows the *stack* action schema, coded in PDDL, from a four-operator *blocksworld* [Slaney and Thiébaux, 2001].

To formalize the output of the learning task, we assume that fluents  $F$  are instantiated from a set of *predicates*  $\Psi$ , as in PDDL. Each predicate  $p \in \Psi$  has an argument list of arity  $\text{ar}(p)$ . Given a set of *objects*  $\Omega$ , the set of fluents  $F$  is 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  $\Omega$ .

Let  $\Omega_v = \{v_i\}_{i=1}^{\max_{a \in A} \text{ar}(a)}$  be a new set of objects  $\Omega \cap \Omega_v = \emptyset$ , denoted as *variable names*, and that is bound by the maximum arity of an action in a given planning frame. For instance, in a three-block *blocksworld*  $\Omega = \{\text{block}_1, \text{block}_2, \text{block}_3\}$  while  $\Omega_v = \{v_1, v_2\}$  because the operators with the maximum arity, *stack* and *unstack*, have two parameters each.

Let us also define  $F_v$ , a new set of fluents  $F \cap F_v = \emptyset$ , that results from instantiating  $\Psi$  using only the objects in  $\Omega_v$  and that defines the elements that can appear in an action schema. For instance, in the *blocksworld*,  $F_v = \{\text{handempty}, \text{holding}(v_1), \text{holding}(v_2), \text{clear}(v_1), \text{clear}(v_2), \text{ontable}(v_1), \text{ontable}(v_2), \text{on}(v_1, v_1), \text{on}(v_1, v_2), \text{on}(v_2, v_1), \text{on}(v_2, v_2)\}$ .

Finally, we assume that actions  $a \in A$  are instantiated from STRIPS operator schemes  $\xi = \langle \text{head}(\xi), \text{pre}(\xi), \text{add}(\xi), \text{del}(\xi) \rangle$  where:

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

$del(\xi) \subseteq F_v$ , and the positive effects  $add(\xi) \subseteq F_v$  such that,  $del(\xi) \subseteq pre(\xi)$ ,  $del(\xi) \cap add(\xi) = \emptyset$  and  $pre(\xi) \cap add(\xi) = \emptyset$ .

### 3 Learning STRIPS action models

Learning STRIPS action models from fully available input knowledge, i.e. from plans where every action in the plan is available as well as its corresponding *pre*- and *post*-states, is straightforward. In this case, STRIPS operator schemes 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 that appears in all the pre-states of the corresponding actions.

We formalize a more challenging learning task, where less input knowledge is available. This learning task, denoted by  $\Lambda = \langle \Psi, \Sigma, \Phi \rangle$ , corresponds to observing an agent acting in the world but watching only the results of its plan executions. The actual executed actions are unobserved:

- $\Psi$  is the set of predicates that define the abstract state space of a given planning domain.
- $\Sigma = \{\sigma_1, \dots, \sigma_\tau\}$  is a set of (*initial*, *final*) state pairs, that we call *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 unobserved plan  $\pi_t$  starting from the *initial* state  $s_0^t$ .
- $\Phi$  is a set of *lifted domain invariants* that do not necessarily contain the *strongest invariant*.

A solution to  $\Lambda$  is a set of operator schema  $\Xi$  compliant with the predicates in  $\Psi$ , the labels  $\Sigma$ , and the state-constraints  $\Phi$ . A planning compilation is a suitable approach for addressing  $\Lambda$  learning tasks because a solution must not only determine the STRIPS action model  $\Xi$  but also, the *unobserved* plans  $\pi_t = \langle a_1^t, \dots, a_n^t \rangle$ ,  $1 \leq t \leq \tau$  that can explain  $\Sigma$  and  $\Phi$ . Figure 2 shows a  $\Lambda$  task for learning a STRIPS action model in the blocksworld from a single label  $\sigma_1$  and a set of nine state invariants.

#### 3.1 Learning with classical planning

Our approach for addressing the learning task  $\Lambda$ , is 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:

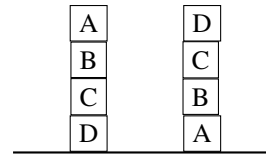
1. Programs the STRIPS action model  $\Xi$ . A solution plan starts with a *prefix* that, for each  $\xi \in \Xi$ , determines which fluents  $f \in F_v$  belong to its  $pre(\xi)$ ,  $del(\xi)$  and  $add(\xi)$  sets.
2. Validates the programmed STRIPS action model  $\Xi$  in  $\Sigma$  and  $\Phi$ . A solution plan continues with, for every  $\sigma_t \in \Sigma$ , a postfix that produces a final state  $s_n^t$  starting from the corresponding initial state  $s_0^t$  using the programmed action model  $\Xi$  and satisfying any constraint  $\phi \in \Phi$  at every new reached state. We call this process the validation of the programmed STRIPS action model  $\Xi$ , at the  $t^{th}$  learning example  $1 \leq t \leq \tau$ .

To formalize our compilation 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 (same fluents and actions but different

;;; Predicates in  $\Psi$

```
(handempty) (holding ?o - object)
(clear ?o - object) (ontable ?o - object)
(on ?o1 - object ?o2 - object)
```

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



;;; Lifted domain invariants in  $\Phi$

```
(forall (?o1 - object)
  (not (and (on ?o1 ?o1))))

(forall (?o1 - object)
  (not (and (handempty) (holding ?o1))))

(forall (?o1 - object)
  (not (and (holding ?o1) (clear ?o1))))

(forall (?o1 - object)
  (not (and (holding ?o1) (ontable ?o1))))

(forall (?o1 ?o2 - object)
  (not (and (on ?o1 ?o2) (holding ?o1))))

(forall (?o1 ?o2 - object)
  (not (and (on ?o1 ?o2) (holding ?o2))))

(forall (?o1 ?o2 - object)
  (not (and (on ?o1 ?o2) (clear ?o2))))

(forall (?o1 ?o2 - object)
  (not (and (on ?o1 ?o2) (ontable ?o1))))

(forall (?o1 ?o2 - object)
  (not (and (on ?o1 ?o2) (on ?o2 ?o1))))
```

Figure 2: Example of a task for learning a STRIPS action model in the blocksworld from a single label and nine state-invariants.

initial state and/or goals). Fluents  $F$  are built instantiating the predicates in  $\Psi$  with the objects appearing in the input labels  $\Sigma$ . Formally  $\Omega = \{o|o \in \bigcup_{1 \leq t \leq \tau} obj(s_0^t)\}$ , where  $obj$  is a function that returns the set of objects that appear in a fully specified state. The set of actions,  $A = \emptyset$ , is empty because the action model is initially unknown. Finally, the initial state  $I_t$  is given by the state  $s_0^t \in \sigma_t$  while goals  $G_t$ , are defined by the state  $s_n^t \in \sigma_t$ .

Now we are ready to formalize the compilation. Given a learning task  $\Lambda = \langle \Psi, \Sigma, \Phi \rangle$  the compilation outputs a classical planning task  $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$ :

- $F_\Lambda$  extends  $F$  with:
  - Fluents representing the programmed action model  $pre_f(\xi)$ ,  $del_f(\xi)$  and  $add_f(\xi)$ , for every  $f \in F_v$  and  $\xi \in \Xi$ . If a fluent  $pre_f(\xi)/del_f(\xi)/add_f(\xi)$  holds, it means that  $f$  is a precondition/negative effect/positive effect in the STRIPS operator schema  $\xi \in \Xi$ . For instance, the preconditions of the *stack* schema (Figure 1) are represented by fluents  $pre\_holding\_stack\_v_1$  and  $pre\_clear\_stack\_v_2$ .
  - A fluent  $mode\_prog$  indicating whether the operator schemes are being programmed or validated (already programmed) and fluents  $\{test_t\}_{1 \leq t \leq \tau}$ , indicating the example where the action model is being validated.
- $I_\Lambda$  contains the fluents from  $F$  that encode  $s_0^1$  (the initial state of the first label) and every  $pre_f(\xi) \in F_\Lambda$  and  $mode\_prog$  set to true. Our compilation assumes that initially operator schemas are programmed with every possible precondition, 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.
- $A_\Lambda$  comprises three kinds of actions:
  1. Actions for *programming* operator schema  $\xi \in \Xi$ :
    - Actions for **removing** a *precondition*  $f \in F_v$  from the action schema  $\xi \in \Xi$ .
$$pre(programPre_{f,\xi}) = \{\neg del_f(\xi), \neg add_f(\xi), mode\_prog, pre_f(\xi)\},$$

$$cond(programPre_{f,\xi}) = \{\emptyset\} \triangleright \{\neg pre_f(\xi)\}.$$
    - Actions for **adding** a *negative* or *positive* effect  $f \in F_v$  to the action schema  $\xi \in \Xi$ .
$$pre(programEff_{f,\xi}) = \{\neg del_f(\xi), \neg add_f(\xi), mode\_prog\},$$

$$cond(programEff_{f,\xi}) = \{pre_f(\xi)\} \triangleright \{del_f(\xi)\},$$

$$\{\neg pre_f(\xi)\} \triangleright \{add_f(\xi)\}.$$
  2. Actions for *applying* an already programmed operator schema  $\xi \in \Xi$  bound with the objects  $\omega \subseteq \Omega^{ar(\xi)}$ . We assume operators headers are known

```
(: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_hanempty_stack)) (hanempty)))
: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_hanempty_stack) (not (hanempty)))
        (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_hanempty_stack) (hanempty))
        (when (modeProg) (not (modeProg)))))
```

Figure 3: Action for applying an already programmed schema *stack* as encoded in PDDL (implications coded as disjunctions).

so the binding of the operator schema is done implicitly by order of appearance of the action parameters, i.e. variables  $vars(\xi)$  are bound to the objects in  $\omega$  appearing at the same position. Figure 3 shows the PDDL encoding of the action for applying a programmed operator *stack*.

$$pre(apply_{\xi,\omega}) = \{pre_f(\xi) \implies p(\omega)\}_{\forall p \in \Psi, f=p(vars(\xi))},$$

$$cond(apply_{\xi,\omega}) = \{del_f(\xi)\} \triangleright \{\neg p(\omega)\}_{\forall p \in \Psi, f=p(vars(\xi))},$$

$$\{add_f(\xi)\} \triangleright \{p(\omega)\}_{\forall p \in \Psi, f=p(vars(\xi))},$$

$$\{mode\_prog\} \triangleright \{\neg mode\_prog\}.$$

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

$$pre(validate_t) = G_t \cup \{test_j\}_{j \in 1 \leq j < t}$$

$$\cup \{\neg test_j\}_{j \in t \leq j \leq \tau} \cup \{\neg mode\_prog\},$$

$$cond(validate_t) = \{\emptyset\} \triangleright \{test_t\}.$$

### 3.2 Constraining the learning hypothesis space

Here we show how *state constraints* are introduced into the compilation to reduce the space of the possible STRIPS action models and make learning more practicable. The intuition is using the input constraints as new preconditions and goals of the classical planning task that results from the compilation.

With regard to the *state invariants*  $\Phi$ :

- Every invariant  $\phi \in \Phi$  is added as an extra precondition of the  $\text{apply}_{\xi,\omega}$  actions for *applying* an already programmed operator schema.
- Every invariant  $\phi \in \Phi$  is added as an extra goal to the  $G_t$ ,  $1 \leq t \leq \tau$ , goal sets because  $\phi$  must hold at every reached state, including the last state.

With regard to *state trajectory constraints*. Additionally, if *state trajectories*  $\mathcal{O}_\pi = (s_0, s_1, \dots, s_n)$  obtained observing the execution of an *unobserved* plan  $\pi$  are available, they can be included in the compilation to constrain further the learning hypothesis space. In this case  $\Sigma = \{\sigma_1, \dots, \sigma_\tau\}$  is no longer a set of (*initial, final*) state pairs but a set of state pairs  $\sigma_t = (s_i, s_{i+1})$ ,  $1 \leq i < n$  s.t. only one  $\text{apply}_{\xi,\omega}$  action can be executed to produce state  $s_{i+1}$  from state  $s_i$ . Introducing *state trajectories* as new constraints to the compilation is done straightforward by:

- Extending  $F_\Lambda$  with a new fluent *applied*.
- For every  $\text{apply}_{\xi,\omega}$  action, extending its precondition set with the negation of the *applied* fluent while its set of positive effects is extended with the *applied* fluent.
- The *applied* fluent is set to false at the initial state  $I_\Lambda$ . Likewise  $\text{validate}_t$  actions set the *applied* fluent to true while this fluent is added as an extra goal to the  $G_t$ ,  $1 \leq t \leq \tau$ , goal sets.

Instead of enumerating the full sequence of states included in a trajectory, *state trajectory constraints* can also be implicitly defined using *Linear Temporal Logic* (LTL) [Bauer et al., 2010]. For instance the LTL *eventually* operator, denoted by  $\Diamond$ , can define constraints that, unlike *state invariants*, must be true at least at one of the reached states. LTL constraints could be included in our compilation following the ideas for compiling temporally extended goals into classical planning [Baier and McIlraith, 2006] that (1) transform the given LTL formula into an equivalent automata, (2) compute the cross product of this automata with the given classical planning task and (3) force the solution plans to always leave the LTL automata at an acceptor state by adding new goals to the classical planning task.

### 3.3 Compilation properties

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

*Proof sketch.* The compilation forces that once the preconditions of an operator schema  $\xi \in \Xi$  are programmed, they cannot be altered. The same happens with the positive and negative effects that define an operator schema  $\xi \in \Xi$  (besides they can only be programmed after preconditions are programmed). Once operator schemas are programmed they can only be applied because of the  $\text{mode}_{\text{prog}}$  fluent. To solve  $P_\Lambda$ , goals  $\{test_t\}$ ,  $1 \leq t \leq \tau$  can only be achieved: executing an applicable sequence of programmed operator schemes that reaches the final state  $s_n^t$ , defined in  $\sigma_t$ , 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  $\Xi$  is compliant with the provided input knowledge and hence, it is a solution to  $\Lambda$ .  $\square$

**Lemma 2. Completeness.** *Any STRIPS action model  $\Xi$  computable from  $\Lambda = \langle \Psi, \Sigma, \Phi \rangle$  can be obtained by solving the corresponding classical planning task  $P_\Lambda$ .*

*Proof sketch.* First,  $F_v$  completely captures the set of all the predicates that can appear in a STRIPS action schema  $\xi \in \Xi$  given  $\Psi$ ,  $\Sigma$  and  $\Phi$ . Second, any conjunction of any length built with the fluents  $F_v$ , and that hence can appear in the sets  $\text{pre}(\xi)$ ,  $\text{del}(\xi)$  and  $\text{add}(\xi)$ , is computable with a classical plan  $\pi$  that solves  $P_\Lambda$  since the classical planning compilation does not discard any possible STRIPS action model compliant with  $\Psi$ ,  $\Sigma$  and  $\Phi$ .  $\square$

## 4 Evaluation

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

### Setup and reproducibility

The domains used in the evaluation are IPC domains that satisfy the STRIPS requirement [Fox and Long, 2003], taken from the PLANNING.DOMAINS repository [Muisse, 2016]. We only use 5 learning examples for each domain and they are fixed for all the experiments so 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 4 GB of RAM.

The classical planner we use to solve the instances that result from our compilations is MADAGASCAR [Rintanen, 2014]. We use MADAGASCAR because 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.

We make fully available the compilation source code, the evaluation scripts and the used benchmarks at this anonymous repository <https://github.com/anonsub/strrips-learning> so any experimental data reported in the paper is fully reproducible.

### Metrics

The quality of the learned models is quantified with the *precision* and *recall* metrics. Intuitively, precision gives a notion of *soundness* while recall gives a notion of the *completeness* of the learned models. Formally,  $\text{Precision} = \frac{tp}{tp+fp}$ , 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  $\text{Recall} = \frac{tp}{tp+fn}$  where  $fn$  is the number of false negatives (predicates that should appear in the learned action model but are missing).

When the learning hypothesis space is low constrained, the learned actions can be reformulated and still be compliant with the inputs. For instance in the *blocksworld*, given a low amount of input knowledge, operator `stack` could be *learned* with the preconditions and effects of the `unstack` operator (and vice versa) making non trivial to compute *precision* and *recall* with respect to a reference model. To address this issue

	Pre		Add		Del			
	P	R	P	R	P	R	P	R
Blocks	0.86	0.67	0.71	0.56	0.86	0.67	0.81	0.63
Driverlog	0.40	0.14	0.36	0.57	0.50	0.29	0.42	0.33
Ferry	0.50	0.29	0.33	0.50	0.50	0.50	0.44	0.43
Floortile	0.67	0.36	0.67	0.36	1.0	0.36	0.78	0.36
Grid	-	-	-	-	-	-	-	-
Gripper	1.0	0.67	0.75	0.75	0.75	0.75	0.83	0.72
Hanoi	1.0	0.5	1.0	1.0	1.0	1.0	1.0	0.83
Hiking	-	-	-	-	-	-	-	-
Parking	-	-	-	-	-	-	-	-
Satellite	0.40	0.14	0.43	0.60	0.67	0.50	0.50	0.41
Sokoban	-	-	-	-	-	-	-	-
Transport	0.75	0.30	0.67	0.80	1.0	0.60	0.81	0.57
Zenotravel	1.0	0.29	0.50	0.43	1.0	0.43	0.83	0.38
	0.73	0.36	0.60	0.62	0.81	0.57	0.71	0.52

Table 1: Precision and recall values obtained when learning from labels.

	Total time	Preprocess	Plan length
Blocks	1.40	0.00	70
Driverlog	1.50	0.00	89
Ferry	1.49	0.00	64
Floortile	351.38	0.11	156
Grid	-	-	-
Gripper	0.04	0.00	59
Hanoi	2.33	0.01	49
Hiking	-	-	-
Parking	-	-	-
Satellite	78.36	0.03	98
Sokoban	-	-	-
Transport	285.61	0.10	106
Zenotravel	6.20	0.46	71

Table 2: Planning results obtained when learning from labels.

we define the following evaluation methodology that deals with action reformulation.

Given a reference STRIPS action model  $\Xi^*$  and the learned STRIPS action model  $\Xi$  we define these two bijective functions  $f_p : \Xi \mapsto \Xi^*$  and  $f_r : \Xi \mapsto \Xi^*$  such that  $f_p$  and  $f_r$  respectively maximize the the accumulated *precision* and *recall*. With this defined we compute the *precision* of an STRIPS action  $\xi$  with respect to the action  $f_p(\xi)$ . Likewise, the *recall* of an STRIPS action  $\xi$  is computed with respect to the action  $f_r(\xi)$ .

#### 4.1 Learning from labels

Figure 1 shows the precision and recall values obtained when learning from state-invariants. Figure 2 shows the planning results obtained when learning from state-invariants.

#### 4.2 Learning from state-invariants

For each domain we provide a set of *lifted domain invariants* that are computed using the TIM algorithm [Fox and Long, 1998]. Figure 3 shows the precision and recall values obtained when learning from state-invariants. Figure 4 shows the planning results obtained when learning from state-invariants.

	Pre		Add		Del			
	P	R	P	R	P	R	P	R
Blocks	0.75	0.67	0.71	0.56	0.71	0.56	0.73	0.59
Driverlog	0.67	0.29	0.38	0.71	0.60	0.43	0.55	0.48
Ferry	0.75	0.43	0.75	0.75	0.75	0.75	0.75	0.64
Floortile	-	-	-	-	-	-	-	-
Grid	-	-	-	-	-	-	-	-
Gripper	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.89
Hanoi	0.50	0.25	0.0	0.0	0.50	0.50	0.33	0.25
Hiking	-	-	-	-	-	-	-	-
Parking	-	-	-	-	-	-	-	-
Satellite	0.40	0.14	0.43	0.60	0.67	0.50	0.50	0.41
Sokoban	-	-	-	-	-	-	-	-
Transport	-	-	-	-	-	-	-	-
Zenotravel	1.0	0.29	0.50	0.43	1.0	0.43	0.83	0.38
	0.72	0.39	0.54	0.58	0.72	0.60	0.67	0.52

Table 3: Precision and recall values obtained when learning from labels + invariants.

	Total time	Preprocess	Plan length
Blocks	652.70	0.04	76
Driverlog	14.98	0.10	65
Ferry	1.70	0.03	58
Floortile	-	-	-
Grid	-	-	-
Gripper	0.14	0.00	47
Hanoi	55.30	0.14	43
Hiking	-	-	-
Parking	-	-	-
Satellite	84.57	0.22	98
Sokoban	-	-	-
Transport	-	-	-
Zenotravel	-	-	-

Table 4: Planning results obtained when learning from labels + invariants.

	Pre		Add		Del			
	P	R	P	R	P	R	P	R
Blocks	0.71	0.56	0.60	0.67	0.71	0.56	0.68	0.59
Ferry	0.75	0.43	0.50	0.75	0.75	0.75	0.67	0.64
Gripper	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.89
Hanoi	0.50	0.25	0.0	0.0	0.50	0.50	0.33	0.25
	0.74	0.48	0.53	0.61	0.74	0.70	0.67	0.57

Table 5: Precision and recall values obtained when learning from observations.

	Total time	Preprocess	Plan length
Blocks	13.36	0.00	73
Ferry	89.04	0.03	63
Gripper	0.66	0.00	43
Hanoi	98.66	0.11	45

Table 6: Planning results obtained when learning from observations.

	Pre		Add		Del		P	R
	P	R	P	R	P	R		
Blocks	-	-	-	-	-	-	-	-
Ferry	1.0	0.57	1.0	1.0	1.0	1.0	1.0	0.86
Gripper	0.75	0.50	0.75	0.75	0.75	0.75	0.75	0.67
Hanoi	0.50	0.25	0.0	0.0	0.50	0.50	0.33	0.25
	-	-	-	-	-	-	-	-

Table 7: Precision and recall values obtained when learning from observations + invariants.

	Total time	Preprocess	Plan length
Blocks	-	-	-
Ferry	253.04	0.21	61
Gripper	1.60	0.04	43
Hanoi	81.76	0.62	45

Table 8: Planning results obtained when learning from observations + invariants.

### 4.3 Learning from state-trajectory constraints

### 4.4 Empty domains with invariants

## 5 Conclusions

### References

- [Alcázar and Torralba, 2015] Vidal Alcázar and Alvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *ICAPS*, pages 2–6, 2015.
- [Amir and Chang, 2008] Eyal Amir and Allen Chang. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 33:349–402, 2008.
- [Baier and McIlraith, 2006] Jorge A Baier and Sheila A McIlraith. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 788. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.
- [Bauer et al., 2010] Andreas Bauer, Patrik Haslum, et al. Ltl goal specifications revisited. In *ECAI*, volume 10, pages 881–886, 2010.
- [Bonet et al., 2009] Blai Bonet, Héctor Palacios, and Héctor Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*, 2009.
- [Cresswell et al., 2013] Stephen N Cresswell, Thomas L McCluskey, and Margaret M West. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28(02):195–213, 2013.
- [Fox and Long, 1998] Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.
- [Fox and Long, 2003] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)*, 20:61–124, 2003.
- [Geffner and Bonet, 2013] Hector Geffner and Blai Bonet. A concise introduction to models and methods for automated planning, 2013.
- [Ghallab et al., 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.
- [Helmert, 2009] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.
- [Kambhampati, 2007] Subbarao Kambhampati. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *Proceedings of the National Conference on Artificial Intelligence*, 2007.
- [Kautz and Selman, 1999] Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325, 1999.
- [McDermott et al., 1998] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language, 1998.
- [Muise, 2016] Christian Muise. Planning. domains. *ICAPS system demonstration*, 2016.
- [Ramírez, 2012] Miquel Ramírez. *Plan recognition as planning*. PhD thesis, Universitat Pompeu Fabra, 2012.
- [Rintanen and others, 2017] Jussi Rintanen et al. Schematic invariants by reduction to ground invariants. In *AAAI*, pages 3644–3650, 2017.
- [Rintanen, 2014] Jussi Rintanen. Madagascar: Scalable planning with sat. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 2014.
- [Segovia-Aguas et al., 2016] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Hierarchical finite state controllers for generalized planning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 3235–3241. AAAI Press, 2016.
- [Segovia-Aguas et al., 2017] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence*, 2017.
- [Slaney and Thiébaux, 2001] John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.
- [Vallati et al., 2015] Mauro Vallati, Lukáš Chrpá, Marek Grzes, Thomas L McCluskey, Mark Roberts, and Scott Sanner. The 2014 international planning competition: Progress and trends. *AI Magazine*, 36(3):90–98, 2015.
- [Yang et al., 2007] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2-3):107–143, 2007.