Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 00 (2018) 1–22
Artificial Intelligence

Learning and Evaluation of STRIPS Action Models with Classical Planning

Diego Aineto[a], Sergio Jiménez Celorrio[a], Eva Onaindia[a]

[a]*Department of Computer Systems and Computation, Universitat Politècnica de València. Spain*

Abstract

This paper presents a novel approach for learning STRIPS action models from observations of plan executions that compiles this learning task into classical planning. The compilation approach is flexible to various amount and kind of available input knowledge; learning examples can range from plans (with their corresponding initial state) to sequences of state observations or even just a set of initial and final states (where no intermediate action/state is known). The compilation accepts also partially specified action models and 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. What is more, the compilation is extensible to assess how well a given STRIPS action model matches an observation of a plan execution. This extension is relevant since it allows us to evaluate the quality of the learned models with respect to the true models but also with respect to test sets of observations of plan executions, which is useful when the actual models are not available. The empirical performance of our compilation approach is evaluated learning action models for a wide range of classical planning domains from the International Planning Competition (IPC) and using these two evaluation approaches.

Keywords: Classical planning, Learning action models, Generalized planning

1. Introduction

Besides *plan synthesis* [1], planning action models are also useful for *plan/goal recognition* [2]. At both planning tasks, an automated planner is required to reason about action models that correctly and completely capture the possible world transitions [3]. 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 [4].

Machine Learning (ML) has shown to be able to compute a wide range of different kinds of models from examples [5]. The application of inductive ML to the learning of STRIPS action models, the vanilla action model for automated planning [6], is not straightforward though:

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

Motivated by recent advances on the synthesis of different kinds of generative models with classical planning [7, 8, 9, 10], this paper introduces an innovative approach for learning STRIPS action models that can be defined as a classical planning compilation. A solution to the classical planning task that results from our compilation is a

sequence of actions that determines the preconditions and effects of a STRIPS model such that this STRIPS model satisfies the observed plan executions that are given as input.

The compilation approach is appealing by itself because it leverages off-the-shelf planners for learning and because its practicality allow us to report model learning results over a wide range of IPC planning domains. Moreover, it opens up a way towards *bootstrapping* planning action models, enabling a planner to gradually learn/update its action model. Apart from these, our classical planning compilation presents the following contributions:

1. *Input Flexibility*. The compilation is flexible to various amount and kind of available input knowledge. Learning examples can range from a set of plans (with their corresponding initial state) or state observations, to just a set of initial and final states where no intermediate action or state is observed. This fact makes the compilation robust to partially obervability of the intermediate input data. Last but not least, the compilation also accepts input knowledge about the structure of the actions in the form of partially specified action models, where some preconditions and effects are a priori known.
2. *Model Validation*. The compilation poses a general framework to assess the validation of a generative model (provided that it is STRIPS compilable) with a given set of observations. This validation capacity goes beyond the functionality of VAL [11] since we do not require a full plan or a full action model.
3. *Model Evaluation*. The compilation can assess how well a given STRIPS action model matches a *test set* with observations of plan executions. This allows us to assess learning performance without having the actual model and again, evaluation with our compilation is flexible to various amount and kind of available data in the given *test set*.

A first description of the compilation previously appeared in the conference paper [12]. Compared to the conference paper, this work includes the following novel material:

- The formulation of *model evaluation* as the task of assessing the amount of edition required by the input action model to induce given observations of plan execution that serve as a test set.
- A unified formulation for learning and evaluating STRIPS action models from observed executions of plans.
- Leveraging *background knowledge* (given as planning constraints either in the form of *state constraints* or *trajectory constraints*) for learning/evaluating/recognizing STRIPS action models.
- A more complete empirical evaluation of the compilation approach. Our evaluation analyses how the amount of input knowledge affects to the performance of the compilation approach.

Section **??** reviews related work on learning planning action models. Section **??** introduces the classical planning model with *conditional effects* (a requirement of the proposed compilation) and the STRIPS action model (the target of our learning and evaluation tasks). Section **??** formalizes the learning of STRIPS action models with regard to different amount and kind of available input knowledge. Sections **??** and **??** describe our compilation approach for addressing the formalized learning tasks its extension to evaluate and recognize STRIPS action models. Section **??** reports the data collected in a two-fold empirical evaluation of our learning approach: First, the learned STRIPS action models are tested with observations of plan executions and second, the learned models are compared to the actual models. Finally, Section **??** discusses the strengths and weaknesses of the compilation approach and proposes several opportunities for future research.

2. Background

This section serves two purposes. In subsection 2.1 we introduce the basic planning concepts as well as the classical planning model that we will use throughout the rest of the paper. Section 2.2 summarizes existing approaches to learning action models in classical planning and highlights the novelty of our contribution via the related work.

	FO States sequences	FO Actions sequences	Plan traces (PO states)	Plan traces (PO actions)	Plan traces (PO both)
ARMS		✓	✓		
AMAN			✓		
LAMP			✓		
SLAF			✓		
LOCM		✓			
FAMA	✓	✓	✓	✓	✓

Table 1: Input comparison of main learning approaches. PO=Partial Observability, FO=Full Observability

2.1. Basic planning concepts

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 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. An action $a \in A$ is defined with:

- $\mathsf{pre}(a) \in \mathcal{L}(F)$, the *preconditions* of a, is the set of literals that must hold for the action $a \in A$ to be applicable.
- $\mathsf{eff}^+(a) \in \mathcal{L}(F)$, the *positive effects* of a, is the set of literals that true after the application of the action $a \in A$.
- $\mathsf{eff}^-(a) \in \mathcal{L}(F)$, the *negative effects* of a, is the set of literals that are false after the application of the action.

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 $\theta(s, a) = \{s \setminus \mathsf{eff}^-(a)) \cup \mathsf{eff}^+(a)\}$.

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 $\pi = \langle a_1, \ldots, a_n \rangle$ that induces the *state trajectory* $\langle s_0, s_1, \ldots, s_n \rangle$ 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 π *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 π in the initial state I.

In this work, we will use the term *observation* to refer to the input knowledge of a learning task. Hence, $\langle s_0, s_1, \ldots, s_n \rangle$ is a full observable (FO) state trajectory if every state s_i is a full assignment of values to fluents and the minimal action sequence to transit from state s_i to state s_{i+1} is composed of a single action; that is, $\theta(s_i, a) = s_{i+1}$. Otherwise, $\langle s_0, s_1, \ldots, s_n \rangle$ is a partial observable (PO) state trajectory, meaning that at least one s_i is a partial assignment of values to fluents in which one or more literals are missing. Likewise, $\langle a_1, \ldots, a_n \rangle$ is a FO action sequence if it contains all the necessary actions to achieve a goal from an initial state, and a PO action sequence if at least one necessary action is missing in the sequence.

A *plan trace* is a combination of a state and an action sequence $\tau = \langle s_0, a_1, a_1, a_2, s_2, \ldots, a_n, s_n \rangle$. Additionally, τ is a plan trace with PO states if $\langle s_0, s_1, \ldots, s_n \rangle$ is a PO state sequence and a plan trace with PO actions if $\langle a_1, \ldots, a_n \rangle$ is a PO action sequence.

2.2. Related work

One can find many different approaches to learning action models in the literature. A common but distinguishing feature to all approaches is the type of input knowledge required by the each model. Due to the large variety of types of input knowledge and the lack of a common naming convention (e.g., plan examples, plan traces, training data, plan samples), we firstly present a glossary of terms that will ease the classification of the surveyed models accordingly to their input knowledge.

==========

	Plan traces (PO states)	Plan traces (PO actions)	Plan traces (PO both)
ARMS	✓		
AMAN	✓		
CAMA	✓		
LAMP	✓		
SLAF	✓		
LOCM	✓		
FAMA	✓	✓	✓

Table 2: PO=Partial Observability, FO=Full Observability

	States	Actions
LOCM	zero	FO
ARMS	zero or PO	FO
AMAN	PO	FO
CAMA	PO	FO
LAMP	PO	FO
SLAF	PO	FO
FAMA	zero or PO	zero or PO

Table 3: PO=Partial Observability, FO=Full Observability

The Crowdsourced Action-Model Acquisition (CAMA) for planning explores knowledge from both crowdsourcing (human annotators) and plan traces to learn action models for planning. CAMA relies on the assumption that obtaining enough training samples is often difficult and costly because there is usually a limited number of plan traces available. In order to overcome this limitation, CAMA builds on a set of soft constraints based on labels `true` or `false` given by the crowd and a set of soft constraints based on the input plan traces, solves then using a MAX-SAT solver and converts the solution to action models.

CAMA uses plan traces with PO states. To capture the knowledge from the crowd, CAMA enumerates all possible preconditions and effects for each action, considering all predicates whose parameters are included by the parameters of the action. Hence, if the parameters of a predicate p are included by the parameters of a, CAMA queries the crowd for three questions of the form "Is the fact p a precondition/positive effect/negative effect of action a?", and human annotators reply "yes", "no" or "cannot tell". The goal is to find an optimal estimator of the `true` labels given the observation, minimizing the average bit-wise error rate.

==========

Back in the 90's various systems aimed learning operators mostly via interaction with the environment. LIVE captured and formulated observable features of objects and used them to acquire and refine operators [13]. OBSERVER updated preconditions and effects by removing and adding facts, respectively, accordingly to observations [14]. These early works were based on lifting the observed states supported by exploratory plans or external teachers, but none provided a theoretical justification for this second source of knowledge.

More recent work on learning planning action models [15] shows that although learning STRIPS operators from pure interaction with the environment requires an exponential number of samples, access to an external teacher can provide solution traces on demand.

Whilst the aforementioned works deal with full state observability,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 [16]. In order to efficiently solve the large MAX-SAT representations, ARMS implements a hill-climbing method that models the actions approximately. SLAF also deals with partial observability [17]. 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. This update makes sure that the new formula represents all the transition relations consistent

with the actions and observations. The formula returned at the end includes all consistent models, which can then be retrieved with additional processing.

Unlike the previous approaches, the one described in [18] deals with both missing and noisy predicates in the observations. An action model is first learnt by constructing a set of kernel classifiers which tolerate noise and partial observability and then STRIPSrules are derived from the classifiers' parameters.

LOCM only requires the example plans as input without need for providing information about predicates or states [19]. This makes LOCM be most likely the learning approach that works with the least information possible. The lack of available information is addressed by LOCM 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 (FSM). The intuitive assumptions of LOCM , like the continuity of object transitions or the association of parameters between consecutive actions in the training sequence, yield a learning model heavily reliant on the kind of domain structure. The inability of LOCM to properly derive domain theories where the state of a sort is subject to different FSMs is later overcome by LOCM2 by forming separate FSMs, each containing a subset of the full transition set for the sort [20]. LOP (LOCM with Optimized Plans [21]), the last contribution of the LOCM family, addresses the problem of inducing static predicates. Because LOCM approaches induce similar models for domains with similar structures, they face problems at generating models for domains that are only distinguished by whether or not they contain static relations (e.g. *blocksworld* and *freecell*). In order to mitigate this drawback, LOP applies a post-processing step after the LOCM analysis which requires additional information about the plans, namely a set of optimal plans to be used in the learning phase.

Recenctly classical planning compilations have been defined to learn different kinds of generative models from examples. The existing compilations for computing FSCs for generalized planning follow a *top-down* approach that interleaves *programming* the FSC with validating it and hence, they tightly integrate planning and generalization. To keep the computation of FSCs tractable, they limit the space of possible solutions bounding the maximum size of the FSC. In addition, they impose that the instances to solve share, not only the domain theory (actions and predicates schemes) but the set of fluents [22] or a subset of *observable* fluents [23]. Programs increase the readability of FSCs separating the control-flow structures from the primitive actions. Like FSCs, programs can also be computed following a *top-down* approach, e.g. exploiting compilations that program and validate the program on instances with the same state and action space [22]. Since these *top-down* approaches search in the space of solutions, it is helpful to limit the set of different control-flow instructions. For instance using only *conditional gotos* that can both implement branching and loops [24].

3. Learning task

3.1. STRIPS *action schemas*

This work addresses the learning and evaluation of PDDL action schemas that follow the STRIPS requirement [25, 26]. Figure 1 shows the *stack* action schema, coded in PDDL, from a four-operator *blocksworld* [27].

```
(: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 a four-operator *blocksworld*.

To formalize the target of the learning and evaluation tasks, we assume that fluents F are instantiated from a set of *predicates* Ψ, 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 predicates in Ψ, i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ s.t. 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 ($\Omega \cap \Omega_v = \emptyset$), 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* $\Omega = \{block_1, block_2, block_3\}$ while $\Omega_v = \{v_1, v_2\}$ because the operators with the maximum arity, `stack` and `unstack`, have arity two. We define F_v, a new set of fluents s.t. $F \cap F_v = \emptyset$, that results from instantiating Ψ using only the objects

in Ω_v, i.e. the variable names, and that defines the elements that can appear in an action schema. In *blocksworld* this set contains 11 elements, 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`)`}.

For a given operator schema ξ, we define $F_v(\xi) \subseteq F_v$ as the subset of fluents that represent the elements that can appear in that action schema. For instance, for the *stack* action schema F_v(`stack`) = F_v while F_v(`pickup`)={`handempty`, `holding(`v_1`)`, `clear(`v_1`)`, `ontable(`v_1`)`, `on(`v_1, v_1`)`} excludes the fluents from F_v that involve v_2 because the action header `pickup(`v_1`)` contains the single parameter v_1.

We assume also that actions $a \in A$ are instantiated from STRIPS operator schemas $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi)\rangle$ where:

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

Given the set of predicates Ψ and the header of the operator schema ξ, $2^{2|F_v(\xi)|}$ defines the size of the space of possible STRIPS models for that operator. Note that the previous constraints require that negative effects appear as preconditions and that they cannot be positive effects and also, that a positive effect cannot appear as a precondition. For instance, $2^{2|F_v(\xi)|} = 4194304$ for the blocksworld `stack` operator while for `pickup` is only 1024.

Last but not least, we say that two STRIPS operator schemes ξ and ξ' are *comparable* if both schemas have the same parameters so they share the same space of possible STRIPS models. Formally, iff $pars(\xi) = pars(\xi')$ so it also holds that $F_v(\xi) = F_v(\xi')$. For instance we can claim that blocksworld operators `stack` and `unstack` are *comparable* while `stack` and `pickup` are not. Likewise we say that two STRIPS action models $\mathcal{M}$ and $\mathcal{M}'$ are *comparable* iff there exists a bijective function $\mathcal{M} \mapsto \mathcal{M}^*$ that maps every $\xi \in \mathcal{M}$ to a comparable action schema $\xi' \in \mathcal{M}'$ and viceversa.

3.2. Learning STRIPS *action schemas from observations of plan executions*

The learning tasks addressed in this paper correspond to observing an agent acting in the world. This learning task is formalized as a tuple $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T}\rangle$:

- $\mathcal{M}$ is the set of *empty* operator schemas.
- Ψ is the set of predicates that define the abstract state space of a given planning frame.
- $\mathcal{T} = \langle s_0, a_{,1}, s_1, \ldots, a_n, s_n\rangle$ is a plan trace that contains the sequence of *state observations* obtained watching the execution of a plan $\pi = \langle a_1, \ldots, a_n\rangle$ such that, for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. Note that $\mathcal{M}$ can be inferred from $\mathcal{T}$ provided that actions in $\mathcal{T}$ are *diverse* enough. Meaning that $\mathcal{T}$ contains at least one instantiation of every action scheme in $\mathcal{M}$. Likewise Ψ can be inferred from $\mathcal{T}$ when at least the observation of a state $s \in \mathcal{T}$ is a full state, that is $|s| = |F|$.

A solution to a $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T}\rangle$ learning task is a set of operator schema $\mathcal{M}'$ that is compliant with the input model $\mathcal{M}$, the predicates Ψ, and the observed plan trace $\mathcal{T}$.

When all the elements in $\mathcal{T}$ are fully observed, i.e. learning from plans where the *pre-* and *post-states* of every action are available, solving the $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T}\rangle$ learning task is straightforward [28]:

- *Preconditions* are derived lifting the minimal set of literals that appears in all the pre-states of the corresponding action, that is any action that belongs to the same operator scheme.
- *Effects* are derived lifting the literals that change between the pre and post-state of the corresponding action executions.

In this paper we make the following assummptions over a $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T}\rangle$ learning task:

1. Each input action scheme $\xi \in \mathcal{M}$ is only composed of $head(\xi)$,

2. The initial state $s_0 \in \mathcal{T}$ is *fully observable*. The remaining states s_i s.t. $1 \leq i \leq n$ can be *partially observable* (some fluents in s_i are missing because it is unknown whether their value is either positive or negative). In the extreme, states s_i and actions a_i $1 \leq i \leq n$, can be missing. In such scenario solving Λ implies determining, not only the STRIPS action model $\mathcal{M}'$, but also the unobserved plan π, that explains the input observations with the learned STRIPS model.
3. Any state observation $s \in \mathcal{T}$ is *noiseless*, meaning that if the value of a fluent is observed it is correct.

Figure 2 shows an example of a learning task $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$, that corresponds to observing the execution of a four-action plan $\pi = \langle$(`unstack B A`), (`putdown B`), (`pickup A`), (`stack A B`)$\rangle$ for inverting a two-block tower. In this example $\mathcal{T} = \langle s_0,$`(unstack B A), (putdown B), (pickup A), (stack A B)`$, s_4 \rangle$ which means that only the first and last states are observed and the three intermediate states s_1, s_2 and s_3 are fully unknown.

```
;;;;;; Action headers in M

(pickup v1) (putdown v1) (stack v1 v2} (unstack v1 v2)


;;;;;; Predicates Ψ

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


;;;;;; Plan trace T

;;; state observation #0
(clear B) (on B A) (ontable A) (handempty)

;;; state observation #4
(clear A) (on A B) (ontable B) (handempty)

;;; Plan π

0: (unstack B A)
1: (putdown B)
2: (pickup A)
3: (stack A B)
```

Figure 2: Task $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$ for learning a *blocksworld* STRIPS action model from a four-action plan and two state observations.

The previous definition formalized the learning of STRIPS action models from the observation of a single plan execution. This definition is however extensible to the more general case where the execution of multiple plans is observed. In this case, $\mathcal{T} = \{t_1, \ldots, t_\tau\}$ where each $t \in \mathcal{T}$ is a plan trace $t = \langle s_0^t, a_1^t, s_1^t, \ldots, a_n^t, s_n^t \rangle$ such that, for each $1 \leq i \leq n^t$, a_i^t is applicable in s_{i-1}^t and generates the successor state $s_i^t = \theta(s_{i-1}^t, a_i^t)$.

4. Compilation scheme

Our approach for addressing a Λ learning task is compiling it into a classical planning task P_Λ with conditional effects. A planning compilation is a suitable approach because computing a solution for Λ involves, not only determining the STRIPS action model $\mathcal{M}'$, but also the *unobserved* plan that explains the given inputs to the learning task. The intuition behind the compilation is that a solution to the resulting classical planning task is a sequence of actions that:

1. **Programs the action model** $\mathcal{M}'$. A solution plan starts with a *prefix* that, for each $\xi \in \mathcal{M}$, determines which fluents $f \in F_v(\xi)$ belong to its $pre(\xi)$, $del(\xi)$ and $add(\xi)$ sets.
2. **Validates the action model** $\mathcal{M}'$. The solution plan continues with a postfix that reproduces the given input knowledge (the available observations of the plan executions) with the programmed action model $\mathcal{M}'$.

4.1. 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 IPC [29] and many classical planners cope with conditional effects without compiling them away.

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

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

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.

4.2. Learning from from observations of plan executions

Here we formalize the compilation for learning STRIPS action models with classical planning. Given a learning task $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$ the compilation outputs a classical planning task $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$:

- F_Λ contains:
  - The set of fluents F built instantiating the predicates Ψ with the objects Ω that appear in the input observations, i.e. the blocks `A` and `B` in the example of Figure 2. Formally, $\Omega = \bigcup_{s \in O} obj(s)$, where obj is a function that returns the objects that appear in a given state.
  - Fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_v(\xi)$, that represent the programmed action model. If a fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ holds, it means that f is a precondition/negative/positive effect in the schema $\xi \in \mathcal{M}'$. For instance, the preconditions of the *stack* schema (Figure 1) are represented by the pair of fluents `pre_holding_stack_`v_1 and `pre_clear_stack_`v_2 set to True.
  - Fluents $F_\pi = \{plan(name(a_j), \Omega^{ar(a_j)}, j)\}_{1 \le j \le |\pi|}$ to code the j^{th} step of the observed plan $\pi = \langle a_1, \ldots, a_n \rangle$ that corresponds to action a_j. The static facts $next_{j,j+1}$ and the fluents at_j, $1 \le j < |\pi|$, are also added to iterate through the steps of plan π
  - The fluents $mode_{prog}$ and $mode_{val}$ to indicate whether the operator schemas are programmed or validated, and the fluents $\{test_i\}_{1 \le i \le n}$, indicating the state observation $s_i \in \mathcal{T}$ where the action model is validated.
- I_Λ encodes the first state observation, $s_0 \subseteq F$ and sets to true $mode_{prog}$ as well as the fluents F_π plus fluents at_1 and $\{next_{j,j+1}\}$, $1 \le j < |\pi|$, for tracking the plan step where the action model is validated. Our compilation assumes that initially, operator schemas are programmed with every possible precondition (the most specific learning hypothesis), no negative effect and no positive effect. Therefore fluents $pre_f(\xi)$, for every $f \in F_v(\xi)$, hold also at the initial state.
- $G_\Lambda = \bigcup_{1 \le i \le n} \{test_i\}$, requires that the programmed action model is validated in all the input state observations $s_i \in \mathcal{T}$.
- A_Λ comprises three kinds of actions:
  1. Actions for *programming* operator schema $\xi \in \mathcal{M}$:
     - Actions for **removing** a *precondition* $f \in F_v(\xi)$ from the action schema $\xi \in \mathcal{M}$.

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

– Actions for **adding** a *negative* or *positive* effect $f \in F_v(\xi)$ to the action schema $\xi \in \mathcal{M}$.

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

2. Actions for *applying* a programmed operator schema $\xi \in \mathcal{M}$ bound with objects $\omega \subseteq \Omega^{ar(\xi)}$. Since operators headers are given as input, the variables $pars(\xi)$ are bound to the objects in ω that appear at the same position. Figure 3 shows the PDDL encoding of the action for applying a programmed operator *stack* from *blocksworld*.

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

```
(: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 are coded as disjunctions).

The actions for *applying* an already programmed operator includes the following extra conditional effects $\{at_j, plan(name(a_j), \Omega^{ar(a_j)}, j)\} \rhd \{\neg at_j, at_{j+1}\}_{\forall j \in [1,n]}$ for advancing to the next plan step. This mechanism ensures that $\mathsf{apply}_{\xi,\omega}$ actions are applied, exclusively, in the same order as in $\mathcal{T}$.

3. Actions for *validating* the partially observed state $s_i \in \mathcal{T}$, $1 \leq i < n$.

$$\begin{aligned}\mathsf{pre}(\mathsf{validate_i}) =& s_i \cup \{test_j\}_{j \in 1 \leq j < i} \cup \{\neg test_j\}_{j \in i \leq j \leq |O|} \cup \{mode_{val}\},\\ \mathsf{cond}(\mathsf{validate_i}) =& \{\emptyset\} \rhd \{test_i, \neg mode_{val}\}.\end{aligned}$$

Further, known preconditions and effects (that is, a partially specified STRIPS action model) can be encoded as fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$ set to true at the initial state I_Λ. In this case, the corresponding programming actions, $\mathsf{programPre}_{f,\xi}$ and $\mathsf{programEff}_{f,\xi}$, become unnecessary and are removed from A_Λ making the classical planning task P_Λ easier to be solved. When a *fully* or *partially specified* STRIPS action model $\mathcal{M}$ is given, the compilation validates whether the observation of the plan execution follows the given model:

- $\mathcal{M}$ is proved to be a *valid* STRIPS action model for the given input data if a solution plan for P_Λ can be found.
- $\mathcal{M}$ is proved to be a *invalid* STRIPS action model for the given input data if P_Λ is unsolvable. This means that $\mathcal{M}$ cannot be compliant with the given observation of the plan execution.

This feature of our compilation is beyond the functionality of VAL, the plan validation tool [11], because VAL requires (1) a full plan and (2), a full action model for plan validation.

The classical plan of Figure 4 shows a solution to a learning task $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$ for getting the *blocksworld* action model where operator schemes for `pickup`, `putdown` and `unstack` are specified in $\mathcal{M}$. This plan programs and validates the operator schema `stack` from *blocksworld*, using the plan π and the two state observations s_0 and s_4 shown in Figure 2. Plan steps $[0, 8]$ program the preconditions of the `stack` operator, steps $[9, 13]$ program the operator effects and steps $[14, 18]$ validate the programmed operators following the four-action plan shown in Figure 2.

```
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 blockB blockA i1 i2)
15 :  (apply_putdown blockB i2 i3)
16 :  (apply_pickup blockA i3 i4)
17 :  (apply_stack blockA blockB i4 i5)
18 :  (validate_1)
```

Figure 4: Plan for programming and validating the *stack* schema using plan π and state observations O (shown in Figure 2) as well as previously specified operator schemes for *pickup*, *putdown* and *unstack*.

Note that the P_Λ compilation is flexible to an unbound number of *missing states* or *missing actions* in the input plan trace $\mathcal{T}$. If any reference to the $mode_{val}$ fluent is removed the compilation can learn STRIPS action models when there are missing state observations in $\mathcal{T}$. Likewise some of the actions in $\mathcal{T}$ may be unknown so they are not specified in I_Λ. In both cases the output planning becomes harder since the plan horizon is no longer bound (classical planning is PSPACE complete while SAT is NP) and the classical planner must determine how many *apply* actions are necessary between any two state observations, i.e. between the application of two *validate* actions.

Last but not least we explain how to address learning STRIPS action models from the observation of the execution of multiple plans $\Pi = \{\pi_1, \ldots, \pi_\tau\}$, $1 \leq t \leq \tau$. Let us 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). 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 and the plan π_t, and the goals G_t are defined by the state s_n^t. Addressing the learning task $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$ where $\mathcal{T} = \{t_1, \ldots, t_\tau\}$ requires introducing a small modification to our compilation. In particular, the actions in P_Λ for *validating* the plan $\pi_t \in \Pi$, $1 \leq t \leq \tau$ resets the current state, and the current plan, and are now defined as:

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

4.3. Compilation properties

Lemma 1. *Soundness. Any classical plan π that solves P_Λ induces an action model $\mathcal{M}'$ that solves $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$.*

Proof sketch. Once operator schemas $\mathcal{M}'$ are programmed, they can only be applied and validated, because of the $mode_{prog}$ fluent. In addition, P_Λ is only solvable if fluents $\{test_i\}$, $1 \leq i \leq n$ hold at the last reached state. These goals can only be achieved executing an applicable sequence of programmed operator schemas that reaches every state $s_i \in \mathcal{T}$, starting from the corresponding initial state and following the sequence of actions defined by the plans in Π. This means that the programmed action model $\mathcal{M}'$ complies with the provided input knowledge and hence, solves Λ. □

Lemma 2. *Completeness. Any* STRIPS *action model $\mathcal{M}'$ that solves a $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$ learning task, is computable solving the corresponding classical planning task P_Λ.*

Proof sketch. By definition, $F_v(\xi) \subseteq F_\Lambda$ fully captures the full set of elements that can appear in a STRIPS action schema $\xi \in \mathcal{M}$ given its header and the set of predicates Ψ. The compilation does not discard any possible STRIPS action schema definable within F_v that satisfies the state trajectory constraint given by $\mathcal{T}$. □

The size of the classical planning task P_Λ output by the compilation depends on:

- The arity of the actions headers in $\mathcal{M}$ and the predicates Ψ that are given as input to the Λ learning task. The larger these numbers, the larger the size of the $F_v(\xi)$ sets. This is the term that dominates the compilation size because it defines the $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ fluents set and the corresponding set of *programming* actions.
- The number of given state observations. The larger n, the more $test_i$ fluents and $\mathsf{validate_i}$ actions in P_Λ.

4.4. Optimizing the compilation with background knowledge

A distinctive feature of Inductive Logic Programming (ILP) is that ILP can leverage *background knowledge* to learn logic programs from data [30]. Inspired by ILP, we show that our approach for the learning of STRIPS action models can also leverage *background knowledge* in this case to optimize the performance of the P_Λ compilation.

4.4.1. Static predicates

A *static predicate* $p \in \Psi$ is a predicate that does not appear in the effects of any action [31]. 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 objects* Ω_v, can be discarded for every $\xi \in \Xi$.
- Actions $\mathsf{programEff_{f,\xi}}$ (s.t. $f \in F_v$ is an instantiation of p in Ω_v) can also be discarded for every $\xi \in \Xi$.

Static predicates can also constrain the space of possible preconditions by looking at the given set of state observations in $\mathcal{T}$. 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 observations in $\mathcal{T}$ then, fluents $pre_f(\xi)$ and actions $\mathsf{programPre_{f,\xi}}$ can be discarded for every $\xi \in \mathcal{M}$. For instance, in the *zenotravel* [32] 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 at any state in $\mathcal{T}$.

Furthermore looking as well at the given example plans, fluents $pre_f(\xi)$ and actions $\mathsf{programPre}_{t,\xi}$ are also 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 $\mathcal{T}$. 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 state observations contain the static literal `(next fl2 fl3)` but does not contain `(next fl2 fl2)`, `(next fl3 fl3)` or `(next fl3 fl2)` the only possible precondition including the static predicate is $pre\_next\_fly\_v5\_v4$.

4.4.2. 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, *state-constraints* in planning allow to specify the set of states where a given action is applicable, the set of states where a given *derived predicate* holds or the set of states that are considered goal states.

State invariants is a kind of state-constraints useful for computing more compact state representations [33] or making *satisfiability planning* and *backward search* more efficient [34, 35]. Given a classical planning problem $P = \langle F, A, I, G\rangle$, a *state invariant* is a formula ϕ that holds at the initial state of a given classical planning problem, $I \models \phi$, and at every state s, built from F, that is reachable from I.

The formula $\phi^*_{I,A}$ represents the *strongest invariant* and exactly characterizes the set of all states reachable from I with the actions in A. For instance Figure 5 shows five clauses that define the *strongest invariant* for *blocksworld*. There are infinitely many strongest invariants, but they are all logically equivalent, and computing the strongest invariant is PSPACE-hard as hard as testing plan existence.

$$\forall x_1, x_2\ ontable(x_1) \leftrightarrow \neg on(x_1, x_2).$$
$$\forall x_1, x_2\ clear(x_1) \leftrightarrow \neg on(x_2, x_1).$$
$$\forall x_1, x_2, x_3\ \neg on(x_1, x_2) \vee \neg on(x_1, x_3)\ such\ that\ x_2 \neq x_3.$$
$$\forall x_1, x_2, x_3\ \neg on(x_2, x_1) \vee \neg on(x_3, x_1)\ such\ that\ x_2 \neq x_3.$$
$$\forall x_1, \ldots, x_n\ \neg(on(x_1, x_2) \wedge on(x_2, x_3) \wedge \ldots \wedge on(x_{n-1}, x_n) \wedge on(x_n, x_1)).$$

Figure 5: An example of the strongest invariant for the *blocksworld* domain.

A *mutex* (mutually exclusive) is a state invariant that takes the form of a binary clause and indicates a pair of different properties that cannot be simultaneously true [36]. For instance in a three-block *blocksworld*, $\phi_1 = \neg on(block_A, block_B) \vee \neg on(block_A, block_C)$ is a mutex because $block_A$ can only be on top of a single block.

A *domain invariant* is an instance-independent invariant, i.e. holds for any possible initial state and set of objects. Therefore, if a given state s holds $s \not\models \phi$ such that ϕ is a *domain invariant*, it means that s is not a valid state. Domain invariants are often compactly defined as *lifted invariants* (also called schematic invariants) [37]. For instance, $\phi_2 = \forall x :\ (\neg handempty \vee \neg holding(x))$, is a *domain mutex* for the *blocksworld* because the robot hand is never empty and holding a block at the same time.

An interesting contribution of our compilation is that the validation of an action model can also be done with *state constraints*. Given Φ, a set of either state or trajectory constraints, our validate actions can be adapted to check that the learned model satisfy every constraint $\phi \in \Phi$:

$$\mathsf{pre}(\mathsf{validate_i}) = \phi \cup \{test_j\}_{j \in 1 \leq j < i} \cup \{\neg test_j\}_{j \in i \leq j \leq |\Phi|} \cup \{mode_{val}\},$$
$$\mathsf{cond}(\mathsf{validate_i}) = \{\emptyset\} \rhd \{test_i, \neg mode_{val}\}.$$

This redefinition of validate actions applies also to trajectory constraints because LTL formulae can be represented using classical action preconditions and goals by encoding the Non-deterministic Büchi Automaton (NBA), that is equivalent to the corresponding LTL formula, as part of the classical planning tasks [38].

Because of the combinatorial nature of the search for a solution plan, the sooner unpromising nodes are pruned from the search the more efficient the computation of a solution plan. Constraints can be used to confine earlier the set of possible STRIPS action models and reduce then the learning hypothesis space. With regard to our compilation, *domain mutex* are useful to reduce the amount of applicable actions for programming a precondition or an effect for a given action schema. For example given the *domain mutex* $\phi = (\neg f_1 \vee \neg f_2)$ such that $f_1 \in F_v(\xi)$ and $f_2 \in F_v(\xi)$,

we can redefine the corresponding programming actions for **removing** the *precondition* $f_1 \in F_v(\xi)$ from the action schema $\xi \in \mathcal{M}$ as:

$$\begin{aligned}\mathsf{pre}(\mathsf{programPre}_{\mathsf{f}_1,\xi}) &= \{\neg del_{f_1}(\xi), \neg add_{f_1}(\xi), mode_{prog}, pre_{f_1}(\xi), pre_{f_2}(\xi)\},\\ \mathsf{cond}(\mathsf{programPre}_{\mathsf{f},\xi}) &= \{\emptyset\} \rhd \{\neg pre_{f_1}(\xi)\}.\end{aligned}$$

5. Evaluation model

If the actual model is available, the quality of a learned model is quantifiable 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 [39]. Intuitively, precision gives a notion of *soundness* while recall gives a notion of the *completeness* of the learned models:

- $Precision = \frac{tp}{tp+fp}$, 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 of the learned model that should not appear).
- $Recall = \frac{tp}{tp+fn}$, where fn is the number of *false negatives* (predicates that should appear in the learned model but are missing).

The roles of two action schemes whose headers match or the roles of two action parameters that belong to the same type can be swapped by a Λ learning task. For instance, the *blocksworld* operator `stack` can be *learned* with the preconditions and effects of the `unstack` operator and vice versa, or the parameters of the `stack` operator can be swapped. Pure syntax-based evaluation metrics (like *precision* and *recall*) can 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.

Here we introduce an evaluation approach that is robust to role changes of this particular kind. The intuition of the approach is to assess how well a STRIPS action model $\mathcal{M}$ explains given observations of plan executions according to the amount of *edition* required by $\mathcal{M}$ to induce that observations. Like our learning approach, our evaluation approach is also flexible to various amount and kind of available input knowledge.

5.1. The STRIPS edit distance

We first define the two allowed *operations* to edit a given STRIPS action model $\mathcal{M}$:

- *Deletion*. A fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ is removed from the operator schema $\xi \in \mathcal{M}$, such that $f \in F_v(\xi)$.
- *Insertion*. A fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ is added to the operator schema $\xi \in \mathcal{M}$, s.t. $f \in F_v(\xi)$.

We can now formalize an *edit distance* that quantifies how similar two given STRIPS action models are. The distance is symmetric and meets the *metric axioms* provided that the two *edit operations*, deletion and insertion, have the same positive cost.

Definition 3. *Let $\mathcal{M}$ and $\mathcal{M}'$ be two* STRIPS *action models, such that they are* comparable*. The* **edit distance***, denoted as $\delta(\mathcal{M}, \mathcal{M}')$, is the minimum number of* edit operations *that is required to transform $\mathcal{M}$ into $\mathcal{M}'$.*

Since F_v is a bound set, the maximum number of edits that can be introduced to a given action model defined within F_v is bound as well. In more detail, for an operator schema $\xi \in \mathcal{M}$ the maximum number of edits that can be introduced to their precondition set is $|F_v(\xi)|$ while the max number of edits that can be introduced to the effects is twice $|F_v(\xi)|$.

Definition 4. *The* ***maximum edit distance*** *of an* STRIPS *action model $\mathcal{M}$ built from the set of possible elements F_v is* $\delta(\mathcal{M}, *) = \sum_{\xi \in \mathcal{M}} 3 \times |F_v(\xi)|$.

Normally evaluating a given learned domain with respect to the actual generative model is not possible because the actual model is not available. As the ARMS system shows, the error of a learned action model can also be estimated with respect to a set of observations of plan executions [16]. With this regard, we define now an edit distance to asses the quality of a learned action model with respect to a plan trace $\mathcal{T} = \langle s_0, a_{,1}, s_1, \ldots, a_n, s_n \rangle$ obtained watching the execution of a plan $\pi = \langle a_1, \ldots, a_n \rangle$ such that, for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$.

Definition 5. *Given $\mathcal{M}$, a* STRIPS *action model built from F_v, and a plan trace $\mathcal{T} = \langle s_0, a_{,1}, s_1, \ldots, a_n, s_n \rangle$ whose state observation are built with fluents in F. The* **observation edit distance***, denoted by $\delta(\mathcal{M}, \mathcal{T})$, is the minimal edit distance from $\mathcal{M}$ to any* comparable *model $\mathcal{M}'$, such that $\mathcal{M}'$ can produce a valid plan trace $\mathcal{T}$;*

$$\delta(\mathcal{M}, \mathcal{T}) = \min_{\forall \mathcal{M}' \to \mathcal{T}} \delta(\mathcal{M}, \mathcal{M}')$$

Unlike the error function defined by ARMS, the *observation edit distance* assess, with a single expression, the flaws in the preconditions and effects of a given learned model. The *edit distance* could be mapped into a likelihood with the following expression $1 - \frac{\delta(\mathcal{M},O)}{\delta(\mathcal{M},*)}$. Note that the error of a learned action model could also be defined quantifying the amount of edition required by the observations of the plan execution to match the given model. This would imply defining *edit operations* that modify the fluents in the state observations instead of the *edit operations* that modify the action schemes. Our definition of the edit distance is more practical since normally, F_v is smaller than F because the number of *variable objects* is smaller than the number of objects in the state observations.

5.2. Computing the observations and plans edit distance

Our compilation is extensible to compute the *observation edit distance* by simply considering that the input STRIPS model $\mathcal{M}$, given in a learning task $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$, is *non-empty*. In other words, now $\mathcal{M}$ is a set of given operator schemas, wherein each $\xi \in \mathcal{M}$ initially contains $head(\xi)$ but also the $pre(\xi)$, $del(\xi)$ and $add(\xi)$ sets. A solution to the planning task resulting from the extended compilation is a sequence of actions that:

1. **Edits the action model $\mathcal{M}$ to build $\mathcal{M}'$.** A solution plan starts with a *prefix* that modifies the preconditions and effects of the action schemes in $\mathcal{M}$ using to the two *edit operations* defined above, *deletion* and *insertion*. In theory, we could implement a third edit operation for *substituting* a fluent from a given operator schema. However, and with the aim of keeping a tractable branching factor of the planning instances that result from our compilations, we only implement *deletion* and *insertion*.
2. **Validates the edited model $\mathcal{M}'$ in the observed plan trace**. The solution plan continues with a postfix that validates the edited model $\mathcal{M}'$ on the given observations $\mathcal{T}$, as explained in Section **??** for the models that are programmed from scratch.

Now Λ does not formalize a learning task but the task of editing $\mathcal{M}$ to produce the plan trace $\mathcal{T}$, which results in the edited model $\mathcal{M}'$. The output of the extended compilation is a classical planning task $P'_\Lambda = \langle F_\Lambda, A'_\Lambda, I'_\Lambda, G_\Lambda \rangle$:

- F_Λ and G_Λ are defined as in the previous compilation.
- I'_Λ contains the fluents from F that encode s_0 and $mode_{prog}$ set to true. In addition, the input action model $\mathcal{M}$ is now encoded in the initial state. This means that the fluents $pre_f(\xi)/del_f(\xi)/add_f(\xi)$, $f \in F_v(\xi)$, hold in the initial state iff they appear in $\mathcal{M}$.
- A'_Λ, comprises the same three kinds of actions of A_Λ. The actions for *applying* an already programmed operator schema and the actions for *validating* an observation are defined exactly as in the previous compilation. The only difference here is that the actions for *programming* the operator schema now implement the two *edit operations* (i.e. include actions for *inserting* a precondition and for *deleting* a negative/positive effect).

To illustrate this, the plan of Figure 6 shows the plan for editing a given *blockswold* action model where again the positive effects `(handempty)` and `(clear ?v1)` of the `stack` schema are missing. In this case the edited action model is however validated at the plan shown in Figure 2.

```
00 :  (insert_add_handempty_stack)
01 :  (insert_add_clear_stack_var1)
02 :  (apply_unstack blockB blockA i1 i2)
03 :  (apply_putdown blockB i2 i3)
04 :  (apply_pickup blockA i3 i4)
05 :  (apply_stack blockA blockB i4 i5)
06 :  (validate_1)
```

Figure 6: Plan for editing a given *blockswold* schema and validating it at the plan shown in Figure 2.

Our interest when computing the *observation edit distance* is not in the resulting action model $\mathcal{M}'$ but in the number of required *edit operations* for that $\mathcal{M}'$ is validated in the given observations, e.g. $\delta(\mathcal{M}, \mathcal{T}) = 2$ for the example in Figure 6. In this case $\delta(\mathcal{M}, *) = 3 \times 2 \times (11 + 5)$ since there are 4 action schemes (`pickup`, `putdown`, `stack` and `unstack`) and $|F_v| = |F_v(stack)| = |F_v(unstack)| = 11$ while $|F_v(pickup)| = |F_v(putdown)| = 5$ (as shown in Section **??**). The *observation edit distance* is exactly computed if the classical planning task resulting from our compilation is optimally solved (according to the number of edit actions); is approximated if it is solved with a satisfying planner; and is a less accurate estimate (but faster to be computed) if the solved task is a relaxation of the classical planning task that results from our compilation [40].

Last but not least, this compilation is flexible to compute the *edit distance* between two *comparable* STRIPS action models, $\mathcal{M}$ and $\mathcal{M}'$. A solution to the planning task resulting from this compilation is a sequence of actions that edits the action model $\mathcal{M}$ to produce $\mathcal{M}'$ using to the two *edit operations*, deletion and insertion. In this case the edited model is not validated on a sequence of observations or plans but on the given action model $\mathcal{M}'$ that acts as a reference. The sets of fluents F_Λ and I_Λ are defined like in the previous compilation. With respect to the actions, A_Λ again implement the two *edit operations* (i.e. include actions for *inserting* a precondition and for *deleting* a negative/positive effect) but does not contain apply actions because the STRIPS action model are not validated in any observation of plan executions. Finally, the goals are also different and are now defined by the set of fluents, $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ that represent all the operator schema $\xi \in \mathcal{M}'$, such that $f \in F_v(\xi)$. To illustrate this, the plan of Figure 7 solves the classical planning task that corresponds to computing the distance between a *blocksworld* action model, where the positive effects `(handempty)` and `(clear ?v1)` of the `stack` schema are missing, and the actual four-operator *blocksworld* model. The plan edits first the `stack` schema, *inserting* these two positive effects. Again our interest is in the number of required *edit operations*, e.g. $\delta(\mathcal{M}, \mathcal{M}') = 2$.

```
00 :  (insert_add_handempty_stack)
01 :  (insert_add_clear_stack_var1)
```

Figure 7: Plan for computing the distance between a *blocksworld* action model, where the positive effects `(handempty)` and `(clear ?v1)` of the `stack` schema are missing, and the actual four-operator *blocksworld* model.

6. Experimental results

6.1. Setup

The domains used in the evaluation are IPC domains that satisfy the STRIPS requirement [26], taken from the PLANNING.DOMAINS repository [41]. 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 8 GB of RAM.

- **Planner**. The classical planner we use to solve the instances that result from our compilations is MADAGASCAR [34]. 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.

- **Reproducibility**. We make fully available the compilation source code, the evaluation scripts and the used benchmarks at this repository *https://github.com/sjimenezgithub/strips-learning* so any experimental data reported in the paper is fully reproducible.

6.2. Evaluating with a reference model

Here we evaluate the learned models with respect to the actual generative model.

6.2.1. Learning from plans

We start evaluating our approach with $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$ learning tasks, where the action of the executed plans are available and the state observation sequence contains only the corresponding initial and goal states, s_o^t and s_n^t, for every trace plan $t \in \mathcal{T}$. We then repeat the evaluation but exploiting potential *static predicates* computed from the observed states in $\mathcal{T}$, which are the predicates that appear unaltered in the states that belong to the same plan. Static predicates are used to constrain the space of possible action models as explained in Section **??**.

Table 4 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 [21].

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

Table 5 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.

We evaluate now the ability of our approach to support partially specified action models; that is, when the input model $\mathcal{M}$ is not empty because some preconditions and effects of the actions are initially known. In this particular experiment, the model of half of the actions is given in $\mathcal{M}$ as an extra input of the learning task. Tables 6 and 7 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 previous learning 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*.

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

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 $\mathcal{M}$). *Visitall* and *Hanoi* are excluded from this evaluation because they only contain one action schema.

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

6.2.2. Learning from state observations

Here we evaluate our approach with learning tasks of the kind $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$, where the action of the executed plans are not available but the initial and goal states are known. When input plans are not available, the planner must not only compute the action models but also the plans that satisfy the input observations. Table 8 and 9 summarize the results obtained for this using static predicates and partially specified models. 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 4. 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.

Now we evaluate our approach with learning tasks of the kind $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$, where the action of the executed plans are not available but where the plan trace $\mathcal{T}$ contains all the intermediate states, not just the initial and final states. Table 10 shows the precision (**P**) and recall (**R**) computed separately for the preconditions (**Pre**), positive effects (**Add**) and negative Effects (**Del**) while the last two columns report averages values. The reason why the

	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 7: 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 8: *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 9: Time and plan length when learning from (initial,final) state pairs.

scores in Table 10 are still low, despite more state observations are available, is because the syntax-based nature of *precision* and *recall* make these two metrics report low scores for learned models that are semantically correct but correspond to *reformulations* of the actual model (changes in the roles of actions with matching headers or parameters with matching types).

	Pre		**Add**		**Del**			
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
blocks	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.44
driverlog	0.0	0.0	0.25	0.43	0.0	0.0	0.08	0.14
ferry	1.0	0.71	1.0	1.0	1.0	1.0	1.0	0.9
floortile	0.38	0.55	0.4	0.18	0.56	0.45	0.44	0.39
grid	0.5	0.47	0.33	0.29	0.25	0.29	0.36	0.35
gripper	0.83	0.83	0.75	0.75	0.75	0.75	0.78	0.78
hanoi	0.5	0.25	0.5	0.5	0.0	0.0	0.33	0.25
hiking	0.43	0.43	0.5	0.35	0.44	0.47	0.46	0.42
miconic	0.6	0.33	0.33	0.25	0.33	0.33	0.42	0.31
npuzzle	0.33	0.33	0.0	0.0	0.0	0.0	0.11	0.11
parking	0.25	0.21	0.0	0.0	0.0	0.0	0.08	0.07
satellite	0.6	0.21	0.8	0.8	1.0	0.5	0.8	0.5
transport	1.0	0.3	0.8	0.8	1.0	0.6	0.93	0.57
visitall	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
zenotravel	0.67	0.29	0.33	0.29	0.33	0.14	0.44	0.24

Table 10: Precision and recall values obtained without computing the $f_{P\&R}$ mapping with the reference model.

To give an insight of the actual quality of the learned models, we defined a method for computing *Precision* and *Recall* that is robust to the mentioned model *reformulations*. Precision and recall are often combined using the *harmonic mean*. This expression, called the *F-measure* or the balanced *F-score*, is defined as $F = 2 \times \frac{Precision \times Recall}{Precision + Recall}$. Given the learned action model $\mathcal{M}$ and the reference action model $\mathcal{M}^*$, the bijective function $f_{P\&R} : \mathcal{M} \mapsto \mathcal{M}^*$ is the mapping between the learned and the reference model that maximizes the accumulated *F-measure* (considering swaps in the actions with matching headers or parameters with matching types).

Table 11 shows that significantly higher values of *precision* and *recall* are reported when a learned action schema, $\xi \in \mathcal{M}$, is compared to its corresponding reference schema given by the $f_{P\&R}$ mapping ($f_{P\&R}(\xi) \in \mathcal{M}^*$). The *blocksworld* and *gripper* domains are perfectly learned from only 25 state observations. These results evidence that in all of the evaluated domains, except for *ferry* and *satellite*, the learning task swaps the roles of some actions (or parameters) with respect to their role in the reference model.

6.3. Evaluating with a test set

When a reference model is not available, the learned models are tested with an observation set. Table 12 summarizes the results obtained when evaluating the quality of the learned models with respect to a test set of state observations. Each test set comprises between 20 and 50 observations per domain and is generated executing 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	0.67	0.14	0.33	0.57	0.67	0.29	0.56	0.33
ferry	1.0	0.71	1.0	1.0	1.0	1.0	1.0	0.9
floortile	0.44	0.64	1.0	0.45	0.89	0.73	0.78	0.61
grid	0.63	0.59	0.67	0.57	0.63	0.71	0.64	0.62
gripper	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
hanoi	1.0	0.5	1.0	1.0	1.0	1.0	1.0	0.83
hiking	0.78	0.6	0.93	0.82	0.88	0.88	0.87	0.77
miconic	0.8	0.44	1.0	0.75	1.0	1.0	0.93	0.73
npuzzle	0.67	0.67	1.0	1.0	1.0	1.0	0.89	0.89
parking	0.56	0.36	0.5	0.33	0.5	0.33	0.52	0.34
satellite	0.6	0.21	0.8	0.8	1.0	0.5	0.8	0.5
transport	1.0	0.3	1.0	1.0	1.0	0.6	1.0	0.63
visitall	0.67	1.0	1.0	1.0	1.0	1.0	0.89	1.0
zenotravel	1.0	0.43	0.67	0.57	1.0	0.43	0.89	0.48

Table 11: Precision and recall values obtained when computing the $f_{P\&R}$ mapping with the reference model.

plans for various instances of the IPC domains and collecting the intermediate states. The table shows, for each domain, the *observation edit distance* (computed with our extended compilation), the *maximum edit distance*, and their ratio. The reported results show that, despite learning only from 25 state observations, 12 out of 15 learned domains yield ratios of 90% or above. This fact evidences that the learned models require very small amounts of edition to match the observations of the given test set.

	$\delta(\mathcal{M}, O)$	$\delta(\mathcal{M}, *)$	$1 - \frac{\delta(\mathcal{M},O)}{\delta(\mathcal{M},*)}$
blocks	0	90	1.0
driverlog	5	144	0.97
ferry	2	69	0.97
floortile	34	342	0.90
grid	42	153	0.73
gripper	2	30	0.93
hanoi	1	63	0.98
hiking	69	174	0.60
miconic	3	72	0.96
npuzzle	2	24	0.92
parking	4	111	0.96
satellite	24	75	0.68
transport	4	78	0.95
visitall	2	24	0.92
zenotravel	3	63	0.95

Table 12: Evaluation of the quality of the learned models with respect to an observations test set.

The learning scores of several domains in Table 11 are above the ones reported in Table 10. The reason lies in the particular observations comprised by the test sets. As an example, in the *driverlog* domain, the action schema `disembark-truck` is missing from the learned model because this action is never induced from the observations in the training set; that is, such action never appears in the corresponding *unobserved* plan. The same happens with the `paint-down` action of the *floortile* domain or `move-curb-to-curb` in the *parking* domain. Interestingly, these actions do not appear either in the test sets and so the learned action models are not penalized in Table 12. Generating *informative* and *representative* observations for learning planning action models is an open issue. Planning actions include preconditions that are only satisfied by specific sequences of actions, often, with a low probability of being chosen by chance [42].

7. Conclusions

We presented a novel approach for learning STRIPS action models from examples using classical planning. The approach is flexible to various amount and kind of input knowledge and accepts partially specified action models. Unlike extensive-data ML approaches, our work explores an alternative research direction to learn sound models from very small data sets. The action models of the *blocksworld* or *gripper* domains were perfectly learned from only 25 state observations. Moreover, in 12 out of the 15 domains, the learned models yield *Precision* values over 0.75.

To the best of our knowledge, this is the first work on learning STRIPS action models from state observations, using exclusively an *off-the-shelf* classical planner, and evaluated over a wide range of different domains. Recently, the work in [43] 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.

We also introduced the *precision* and *recall* metrics, widely used in ML, for evaluating the learned action models with respect to a given reference model. These two metrics measure the soundness and completeness of the learned models and facilitate the identification of model flaws.

When example plans are available, we can compute accurate action models from small sets of learning examples in little computation time, less than a second. In many applications, the actual actions executed by the observed agent are not available but, instead, the resulting states can be observed. With this regard, we extended our approach for learning also from state observations as it broadens the range of application to external observers and facilitates the

representation of imperfect observability, as shown in plan recognition [44], as well as learning from unstructured data, like state images [45]. 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.

We also introduced a semantic method for evaluating the learned STRIPS action models with respect to observations of plan executions. 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 [42]. The success of recent algorithms for exploring planning tasks [46] 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.

Acknowledgment

This work is supported by the Spanish MINECO project TIN2017-88476-C2-1-R. Diego Aineto is partially supported by the *FPU16/03184* and Sergio Jiménez by the *RYC15/18009*, both programs funded by the Spanish government.

References

[1] M. Ghallab, D. Nau, P. Traverso, Automated Planning: theory and practice, Elsevier, 2004.
[2] M. Ramírez, Plan recognition as planning, Ph.D. thesis, Universitat Pompeu Fabra (2012).
[3] H. Geffner, B. Bonet, A concise introduction to models and methods for automated planning (2013).
[4] S. Kambhampati, 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, 2007.
[5] R. S. Michalski, J. G. Carbonell, T. M. Mitchell, Machine learning: An artificial intelligence approach, Springer Science & Business Media, 2013.
[6] R. E. Fikes, N. J. Nilsson, Strips: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (3-4) (1971) 189–208.
[7] B. Bonet, H. Palacios, H. Geffner, Automatic derivation of memoryless policies and finite-state controllers using classical planners., in: International Conference on Automated Planning and Scheduling (ICAPS), 2009.
[8] J. Segovia, S. Jiménez, A. Jonsson, Generalized planning with procedural domain control knowledge, 2016.
[9] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Hierarchical finite state controllers for generalized planning, in: International Joint Conference on Artificial Intelligence, IJCAI-16, AAAI Press, 2016, pp. 3235–3241.
[10] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Generating context-free grammars using classical planning, in: International Joint Conference on Artificial Intelligence, ICAPS-17, 2017.
[11] R. Howey, D. Long, M. Fox, VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL, in: Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on, IEEE, 2004, pp. 294–301.
[12] D. Aineto, S. Jiménez, E. Onaindia, Learning strips action models with classical planning.
[13] W. Shen, H. A. Simon, Rule creation and rule learning through environmental exploration, in: International Joint Conference on Artificial Intelligence, IJCAI-89, 1989, pp. 675–680.
[14] X. Wang, Learning by observation and practice: An incremental approach for planning operator acquisition, in: International Conference on Machine Learning, 1995, pp. 549–557.
[15] T. J. Walsh, M. L. Littman, Efficient learning of action schemas and web-service descriptions, in: National Conference on Artificial Intelligence, 2008, pp. 714–719.
[16] Q. Yang, K. Wu, Y. Jiang, Learning action models from plan examples using weighted max-sat, Artificial Intelligence 171 (2-3) (2007) 107–143.
[17] E. Amir, A. Chang, Learning partially observable deterministic action models, Journal of Artificial Intelligence Research 33 (2008) 349–402.
[18] K. Mourão, L. S. Zettlemoyer, R. P. A. Petrick, M. Steedman, Learning STRIPS operators from noisy and incomplete observations, in: Conference on Uncertainty in Artificial Intelligence (UAI), 2012, pp. 614–623.
[19] S. N. Cresswell, T. L. McCluskey, M. M. West, Acquiring planning domain models using LOCM, The Knowledge Engineering Review 28 (02) (2013) 195–213.
[20] S. Cresswell, P. Gregory, Generalised domain model acquisition from action traces, in: International Conference on Automated Planning and Scheduling, ICAPS-11, 2011.
[21] P. Gregory, S. Cresswell, Domain model acquisition in the presence of static relations in the LOP system., in: International Conference on Automated Planning and Scheduling, ICAPS-15, 2015, pp. 97–105.
[22] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Generalized planning with procedural domain control knowledge, in: ICAPS, 2016.
[23] B. Bonet, H. Palacios, H. Geffner, Automatic derivation of finite-state machines for behavior control, in: AAAI, 2010.
[24] S. Jiménez, A. Jonsson, Computing Plans with Control Flow and Procedures Using a Classical Planner, in: SOCS, 2015.
[25] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL – The Planning Domain Definition Language (1998).

[26] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains., Journal of Artificial Intelligence Research 20 (2003) 61–124.
[27] J. Slaney, S. Thiébaux, Blocks world revisited, Artificial Intelligence 125 (1-2) (2001) 119–153.
[28] S. Jiménez, T. De La Rosa, S. Fernández, F. Fernández, D. Borrajo, A review of machine learning for automated planning, The Knowledge Engineering Review 27 (04) (2012) 433–467.
[29] M. Vallati, L. Chrpa, M. Grzes, T. L. McCluskey, M. Roberts, S. Sanner, The 2014 international planning competition: Progress and trends, AI Magazine 36 (3) (2015) 90–98.
[30] S. Muggleton, L. De Raedt, Inductive logic programming: Theory and methods, The Journal of Logic Programming 19 (1994) 629–679.
[31] M. Fox, D. Long, The automatic inference of state invariants in TIM, Journal of Artificial Intelligence Research 9 (1998) 367–421.
[32] D. Long, M. Fox, The 3rd international planning competition: Results and analysis, Journal of Artificial Intelligence Research 20 (2003) 1–59.
[33] M. Helmert, Concise finite-domain representations for pddl planning tasks, Artificial Intelligence 173 (5-6) (2009) 503–535.
[34] J. Rintanen, Madagascar: Scalable planning with sat, Proceedings of the 8th International Planning Competition (IPC-2014).
[35] V. Alcázar, A. Torralba, A reminder about the importance of computing and exploiting invariants in planning., in: ICAPS, 2015, pp. 2–6.
[36] H. Kautz, B. Selman, Unifying sat-based and graph-based planning, in: IJCAI, Vol. 99, 1999, pp. 318–325.
[37] J. Rintanen, et al., Schematic invariants by reduction to ground invariants., in: AAAI, 2017, pp. 3644–3650.
[38] J. A. Baier, S. A. McIlraith, Planning with first-order temporally extended goals using heuristic search, in: Proceedings of the National Conference on Artificial Intelligence, Vol. 21, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006, p. 788.
[39] J. Davis, M. Goadrich, The relationship between precision-recall and ROC curves, in: International Conference on Machine learning, ACM, 2006, pp. 233–240.
[40] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (1-2) (2001) 5–33.
[41] C. Muise, Planning. domains, ICAPS system demonstration.
[42] A. Fern, S. W. Yoon, R. Givan, Learning domain-specific control knowledge from random walks., in: International Conference on Automated Planning and Scheduling, ICAPS-04, 2004, pp. 191–199.
[43] R. Stern, B. Juba, Efficient, safe, and probably approximately complete learning of action models, in: International Joint Conference on Artificial Intelligence, IJCAI-17, 2017, pp. 4405–4411.
[44] S. Sohrabi, A. V. Riabov, O. Udrea, Plan recognition as planning revisited, in: International Joint Conference on Artificial Intelligence, IJCAI-16, 2016, pp. 3258–3264.
[45] M. Asai, A. Fugunaga, Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary, in: National Conference on Artificial Intelligence, AAAI-18, 2018.
[46] G. Francès, M. Ramírez, N. Lipovetzky, H. Geffner, Purely declarative action descriptions are overrated: Classical planning with simulators, in: International Joint Conference on Artificial Intelligence, IJCAI-17, 2017, pp. 4294–4301.

Appendix

```
(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
       (ontable ?x)
       (clear ?x)
       (handempty)
       (holding ?x)
       )

  (:action pick-up
     :parameters (?x)
     :precondition (and (clear ?x) (ontable ?x) (handempty))
     :effect
     (and (not (ontable ?x))
   (not (clear ?x))
   (not (handempty))
   (holding ?x)))

  (:action put-down
     :parameters (?x)
     :precondition (holding ?x)
     :effect
     (and (not (holding ?x))
   (clear ?x)
   (handempty)
   (ontable ?x)))
  (:action stack
     :parameters (?x ?y)
     :precondition (and (holding ?x) (clear ?y))
     :effect
     (and (not (holding ?x))
   (not (clear ?y))
   (clear ?x)
   (handempty)
   (on ?x ?y)))
  (:action unstack
     :parameters (?x ?y)
     :precondition (and (on ?x ?y) (clear ?x) (handempty))
     :effect
     (and (holding ?x)
   (clear ?y)
   (not (clear ?x))
   (not (handempty))
   (not (on ?x ?y)))))
```

Figure 8: PDDL domain file for the blocksworld domain.

```
(define (problem BLOCKS-4-1)
(:domain BLOCKS)
(:objects A C D B )
(:INIT (CLEAR B) (ONTABLE D) (ON B C) (ON C A) (ON A D) (HANDEMPTY))
(:goal (AND (ON D C) (ON C A) (ON A B)))
)
```

Figure 9: PDDL problem file for the blocksworld domain.

```
(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
       (ontable ?x)
       (clear ?x)
       (handempty)
       (holding ?x)
       )

  (:action pick-up
     :parameters (?x)
     :precondition (and (clear ?x) (ontable ?x) (handempty))
     :effect
     (and (not (ontable ?x))
   (not (clear ?x))
   (not (handempty))
   (holding ?x)))

  (:action put-down
     :parameters (?x)
     :precondition (holding ?x)
     :effect
     (and (not (holding ?x))
   (clear ?x)
   (handempty)
   (ontable ?x)))
  (:action stack
     :parameters (?x ?y)
     :precondition (and (holding ?x) (clear ?y))
     :effect
     (and (not (holding ?x))
   (not (clear ?y))
   (clear ?x)
   (handempty)
   (on ?x ?y)))
  (:action unstack
     :parameters (?x ?y)
     :precondition (and (on ?x ?y) (clear ?x) (handempty))
     :effect
     (and (holding ?x)
   (clear ?y)
   (not (clear ?x))
   (not (handempty))
   (not (on ?x ?y)))))
```

Figure 10: Compiled PDDL domain file for the blocksworld domain.

```
(define (problem BLOCKS-4-1)
(:domain BLOCKS)
(:objects A C D B )
(:INIT (CLEAR B) (ONTABLE D) (ON B C) (ON C A) (ON A D) (HANDEMPTY))
(:goal (AND (ON D C) (ON C A) (ON A B)))
)
```

Figure 11: Compiled PDDL problem file for the blocksworld domain.