Available online at www.sciencedirect.com

ScienceDirect

Artificial Intelligence 00 (2018) 1–24

Artificial Intelligence

A general framework for learning planning action models

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 or 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. On the other hand, the compilation is extensible to assess how well a given STRIPS action model matches the observation of a plan execution. This extension allows us to evaluate the quality of the learned models with respect to the actual models but also, with respect to a *test set* of observations of plan executions. The 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 following 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, automated planners reason about action models that correctly and completely capture the possible world transitions [3]. Unfortunately, modeling planning actions 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 observed plan possibly has a different number of steps and involves 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 the possible state transitions.

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 model satisfies the plan executions given as input.

The compilation approach is appealing by itself because it leverages off-the-shelf planners and because its practicality allow us to report learning results over a wide range of domains from the *International Planning Competition* (IPC). 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 compilation exhibits the following contributions:

1. ***Input flexibility***. Our classical planning compilation is flexible to various amount and kind of available input knowledge. The action model to learn can be partially specified and the 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.
2. ***Model validation***. The compilation poses a novel framework to assess the validation of a STRIPS model with respect to plan executions. This validation capacity goes beyond the functionality of VAL [11] since it does require neither a full action model nor a fully observed plan to determine validation.
3. ***Model evaluation***. The compilation can assess how well a given STRIPS action model matches a plan execution, which allows us to assess learning performance without knowing the actual action model. The idea is to assess the amount of edition that is required by the input action model to induce the given observations of plan executions.

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

- A unified formulation for learning and evaluating STRIPS action models from observed executions of plans. Further, these executions can only comprise state observations.
- A redefinition of the ML metrics *precision* and *recall* to evaluate STRIPS action models with respect to observations of plan executions.
- A complete empirical evaluation of the compilation approach for learning of STRIPS action models. Our evaluation analyses how input knowledge affects to the performance of the compilation approach when learning and evaluating STRIPS action models.

Section 2 introduces classical planning and reviews related work on learning planning action models. Section 3 motivates our compilation approach for learning of STRIPS action models from observations of plan executions. Section 4 formalizes the learning of STRIPS action models with regard to different amount and kind of available input knowledge and describe how to address this learning task with a classical planning compilation. Sections 5 describes our compilation approach for evaluating the learned STRIPS action models with respect to the actual models but also, with respect to a *test set* of observations of plan executions. Section 6 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 7 discusses the strengths and weaknesses of the compilation approach and proposes several opportunities for future research.

2. Background

This section serves two purposes: first we introduce the basic planning concepts as well as the classical planning model that we will use throughout the rest of the paper; and secondly, we summarize the existing approaches to learn action models in classical planning and highlight our contributions via the related work.

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.

Like in PDDL [13], we assume that fluents F are instantiated from a set of *predicates* Ψ. 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.

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 it is common in STRIPS planning.

A *classical planning frame* is a tuple $\Phi = \langle F, A\rangle$, where F is a set of fluents and A is a set of actions. An action $a \in A$ is defined with:

- $\mathsf{pre}(a) \in \mathcal{L}(F)$, 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 are 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 assume that $\mathsf{eff}^-(a) \subseteq \mathsf{pre}(a)$, $\mathsf{eff}^-(a) \cap \mathsf{eff}^+(a) = \emptyset$ and $\mathsf{pre}(a) \cap \mathsf{eff}^+(a) = \emptyset$ and that actions $a \in A$ are instantiated from given action schemas, as in PDDL. 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* $s = \langle s_0, s_1, \ldots, s_n\rangle$ 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. A solution plan is *optimal* if it has minimum length.

In this work, the term *plan trace* refers to the *observation* of a plan execution that starts on a given initial state and interleaves observations of the executed actions and the produced states. *Plan traces* constitute the input knowledge of the learning tasks addressed in this paper.

With regard to the observed states in a plan trace, we say that $\langle s_0, s_1, \ldots, s_n\rangle$ is a fully-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, $\theta(s_i, \langle a\rangle) = s_{i+1}$. Otherwise, it is a partially-observable (PO) state trajectory, meaning that at least one state s_i is a partial assignment of values to fluents in which one or more literals are missing (formally, $|s_i| < |F|$). This implies that in a PO state trajectory all fluents of a state s_i may be missing, in which case, s_i is a *missing* or *empty* state. This general definition of PO gives rise to two particular cases:

- when **all** the $n-1$ intermediate states of a trajectory s are **missing**, s is a *non-observable* (NO) state trajectory
- when **none** of the $n-1$ intermediate states of a trajectory s is **missing**, we will refer to s as a PO* state trajectory.

Table 1 summarizes the four types of state trajectories according to the observed information, which ultimately affects the number of observed intermediate states and the number of literals comprised in each intermediate state. PO comprises both PO* and NO, and it thus encompasses trajectories with some missing state.

	# intermediate states	state type
FO	$n-1$	$\forall i, 1 \leq i < n$ s_i is a full assignment
PO*	$n-1$	$\exists i, 1 \leq i < n$ s_i is a partial assignment
PO	$\leq n-1$	$\exists i, 1 \leq i < n$ s_i is a partial assignment
NO	0	$\forall i, 1 \leq i < n$ $\lvert s_i\rvert = 0$

Table 1: Classification of state trajectories accordingly to observation

With regard to the observed actions in a plan trace, we say that an action sequence, $\langle a_1, \ldots, a_n\rangle$ is a FO action sequence if it contains all the necessary actions to transit every state s_{i-1} to the corresponding successor state s_i, for each $1 \leq i \leq n$. We say it is a PO action sequence if at least one of these necessary actions is missing in the sequence and a NO action sequence in case the sequence of observed actions is empty.

A *plan trace* $\tau = \langle s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_m \rangle$ is an interleaved combination of a state sequence $\langle s_0, s_1, \ldots, s_m \rangle$ and an action sequence $\langle a_1, \ldots, a_n \rangle$. A plan trace τ for a planning frame $\Phi = \langle F, A \rangle$ satisfies that every action $a_i \in A$ for every $1 \leq i \leq n$ and that every state $s_j \in \mathcal{L}(F)$ for every $1 \leq j \leq m$. Plan traces can be classified accordingly to the type of observed state trajectory (FO, PO*, PO or NO) and action sequence (FO, PO or NO).

2.2. Related work

In this section we summarize the most recent and relevant approaches to learning action models found in the literature. Approaches will be examined according to the following parameters: the input knowledge (plan traces) accepted by the system, the expressiveness of the learned action model and the principal technique used for learning the action model (Table 2), as well as the characteristics of the evaluation method to validate the learned models (Table 3).

The first column of Table 2 shows the constraints imposed on the input plan traces with regards to observability. Since all approaches except ours deal only with FO action sequences, constraints are exclusively concerned with the type of state trajectory. This directly affects the complexity of the task, which can be sorted from the least to the most constrained following this order: 1) NO, 2) PO, 3) PO*, and 4) FO. Note that PO is less constrained than PO* because it also includes the possibility of having some missing state in the trajectory.

The task of learning from less constrained traces subsumes learning from more constrained ones. Consequently, approaches to learning from, for instance, traces with PO state trajectories will also be able to learn from traces with PO* state trajectories. All the approaches analyzed in this work accept the more constrained definition of partial observations of intermediate states PO*, and most of them also allow the sequence of intermediate states to be empty. Exceptionally, a NO state sequence in LOCM is a fully-empty trajectory, with neither initial or final state.

Most approaches assume that a set of predicates and a set of action headers are provided alongside the input traces. Others do not explicitly say so but the fact is that the predicates and the actions headers are easily extractable from the state sequence and action sequence of the plan traces, respectively. A requirement, however, for these two sets to be representative is that set of input plan traces comprises at least a grounded sample of all predicates and operator schemas of the domain model.

The expressiveness of the learned action models varies across approaches (second column of Table 2). All the presented systems are able to learn action models in a STRIPS representation [6] and some propose algorithms to learn more expressive action models that include quantifiers, logical implications or the type hierarchy of a PDDL domain.

Table 3 summarizes the main characteristics of the evaluation of the learned action models based on the type of evaluation method (first column of Table 3 – almost all approaches rely on a comparison between the learned model and a ground-truth model), the metrics used in the comparison (second column of Table 3) and the number of tested domains alongside the size of the training dataset (third column of Table 3).

In the following, we present a comprehensive insight of the particularities of the seven systems presented in Table 2 and Table 3. This exposition will help us to highlight in section 3 the value of our contribution FAMA.

The Action-Relation Modeling System (**ARMS**) [14] is one of the first learning algorithms able to learn from plan traces with partial or null observations of intermediate states. ARMS uncovers a number of constraints from the plan traces in the training data that must hold for the plans to be correct. These constraints are then used to build and solve a weighted propositional satisfiability problem with a MAX-SAT solver. Three types of constraints are considered: 1) constraints imposed by general axioms of correct STRIPS actions, 2) constraints extracted from the distribution of actions in the plan traces and 3) constraints obtained from the PO states, if available. Frequent subsets of actions in which to apply the two latter types of constraints are found by means of frequent set mining.

ARMS defines an error metric and a redundancy metric to measure the correctness and conciseness of an action model over the test set of input plan traces using a cross-validation evaluation. The model evaluation is posed as an optimization task that returns the model that best explains the input traces by minimizing the error and redundancy functions. This yields a model that is approximately correct (100% correctness is not required so as to ensure generality and avoid overfitting), approximately concise (low redundancy rates), and that can explain as many examples as

	Input plan traces	Learned action model	Technique
ARMS	NO states FO actions	STRIPS	MAX-SAT
SLAF	PO* states FO actions	universal quantifiers in eff	logical inference SAT solver
LAMP	PO states FO actions	quantifiers logical implications	Markov logic networks
AMAN	NO states noisy actions	STRIPS	graphical model estimation
NOISTA	PO* and noisy states FO actions	STRIPS	classification STRIPS rules derivation
CAMA	PO states FO actions	STRIPS	crowdsourcing annotation MAX-SAT
LOCM2	NO states FO actions	predicates and types	Finite State Machines
FAMA	NO states NO actions	STRIPS	compilation to planning

Table 2: Characteristics of action-model learning approaches

possible. Hence, there is no guarantee that the learned model of ARMS explains all observed plans, not even that it correctly explains any of the plan traces of the test set.

The ARMS system became a benchmark in action-model learning, showing empirically that is is feasible lo learn a model in a reasonably efficient way using a weighted MAX-SAT even with NO state trajectories.

A tractable and exact solution of action models in partially observable domains using a technique known as Simultaneous Learning and Filtering (**SLAF**) is presented in [15]. SLAF alongside ARMS can be considered another of the precursors of the modern algorithms for action-model learning, able to learn from partially observable states. Given a formula representing the initial belief state, a sequence of executed actions and the corresponding partially observed states, SLAF 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 such that the new transition belief formula represents all possible transition relations consistent with the actions and observations at every time step.

SLAF extracts all satisfying models of the learned formula with a SAT solver. For doing so, the training data set for each domain is composed of randomly generated action-observation sequences (1,000 randomly selected actions and 10 fluents uniformly selected at random per observation). Additional processing in the form of replacement procedures or extra axioms are run into the SAT solver when finding the satisfying models. The experimentally tested SLAF version is an algorithm that learns only effects for actions that have no conditional effects and assumes that actions in the sequences are all executed successfully (without failures). This algorithm cannot effectively learn the unknown preconditions of the actions and in the resulting models '*one can see that the learned preconditions are often inaccurate*' [15]. On the other hand, it does not report any statistical evaluation of measurement error other than a manually comparison of the learned models with a ground-truth model.

The Learning Action Models from Plan Traces (**LAMP**) [16] algorithm extends the expressiveness to learning models with universal and existential quantifiers as well as logical implications. The input to LAMP is a set of plan traces with intermediate states, which are encoded by the algorithm into propositional formulas. LAMP then uses the action headers and predicates to build a set of candidate formulas that are validated against the input set using a Markov Logic Network and effectively weighting each formula. The formulas with weights larger than a certain threshold are chosen to represent preconditions and effects of the learned action models.

LAMP allows PO state trajectories up to a minimum percentage of 1/5 of non-empty states as well as PO* state trajectories with different degrees of observability in the number of propositions in each state. It uses an error metric based on counting the differences in the number of precondition and effects between the ground-truth model and the learned model. In general, the results show that the accuracy of the learned models is fairly sensitive to the threshold chosen to learn the weights of the candidate formulas, and that domains that feature more conditional effects are

	Evaluation method	**Metrics**	**#tested domains/ training data size**
ARMS	cross-validation with a test set of plan traces	error counting of #pre satisfaction and redundancy	6 1,600-4,320 actions (160 plan traces)
SLAF	manual checking wrt GTM	—	4 1,000 actions
LAMP	checking wrt GTM	error counting of extra and missing #pre and #eff	4 1,300-6,100 actions (100-200 plan traces)
AMAN	checking wrt GTM	error counting of extra and missing #pre and #eff	3 40-200 plan traces
NOISTA	checking wrt GTM	error counting of extra and missing #pre and #eff	5 5,000-20,000 actions
CAMA	checking wrt GTM	error counting of extra and missing #pre and #eff	3 15-75 plan traces
LOCM2	manual checking wrt GTM	—	—
FAMA	checking wrt GTM	precision and recall of #pre and #eff	15 25 actions

Table 3: Evaluation of action models (GTM: ground-truth model)

harder to learn.

The Action Model Acquisition from Noisy plan traces (**AMAN**) [17] introduces an algorithm able to learn action models from plan traces with NO state sequences where actions have a probability of being observed incorrectly (noisy actions). The first step of the AMAN algorithm is to build the set of candidate domain models that are compliant with the action headers and predicates. AMAN then builds a graphical model to capture the domain physics; i.e., the relations between states, correct actions, observed actions and domain models. After that, the parameters of the graphical model are learned, computing at the same time the probability distribution of each candidate domain model. AMAN finally returns the model that maximizes a reward function defined in terms of the percentage of actions successfully executed and the percentage of goal propositions achieved after the last successfully executed action.

AMAN uses the same metric as LAMP, namely counting the number of preconditions and effects that appear in the learned model and not in the ground-truth model (extra fluents) and viceversa (missing fluents). In a comparison between AMAN and ARMS on noiseless inputs, the results show that the accuracy of the learnt models are very close to each other and neither dominates the other. The convergence property of AMAN guarantees that the accuracy of the learned model with noisy input traces becomes more and more close to the case *without noise* because the distribution of noise in the plan becomes gradually closer to real distribution with the number of iterations.

Another interesting approach that deals with noisy and incomplete observations of states is presented in [18]. We will refer to this approach as **NOISTA** henceforth. In NOISTA, actions are correctly observed but they can obviously be unsuccessfully executed in the possibly noisy application state. The basis of this approach consists of two parts: a) the application of a voted Perceptron classification method to predict the effects of the actions in vectorized state descriptions and b) the derivation of explicit STRIPS action rules to predict each fluent in isolation. Experimentally, the error rates in NOISTA fall below 0.1 after 5,000 training samples for the five tested domains under a maximum of 5% noise and a minimum of 10% of observed fluents.

The Crowdsourced Action-Model Acquisition (**CAMA**) [19] 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. Then it solves the constraint-based problem using a MAX-SAT solver and converts the solution to action models.

Plan traces in CAMA are composed of 80% of empty states and each partial state was selected by 50% of propositions in the corresponding full state. An experimental comparison reveals that a manual crowdsourcing of CAMA outperforms ARMS and that as expected the difference becomes smaller as the number of plan traces becomes larger. The accuracy of CAMA for a small number of plan traces (e.g., 30) is not less than 80%, thus revealing that exploiting the knowledge of the crowd can help learning action models.

The Learning Object-Centred Models (**LOCM**) is an approach that only requires the FO action sequence as input knowledge, without need for providing any information about the predicates or the state trajectory, not even the initial or final state [20, 21]. The lack of available state information is overcome by exploiting assumptions about the structure of the actions. Particularly, LOCM assumes that objects found in the same position in the header of actions are grouped as a collection of objects (sorts) whose defined set of states is 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 dataset, yield a learning model heavily reliant on the kind of domain structure. A later work, **LOCM2**, extends the applicability of the LOCM algorithm to a wider range of domains by introducing a richer representation that allows using multiple FSMs to represent the state of a sort [22].

LOCM2 is not experimentally evaluated, only the outcome of running the LOCM2 algorithm on several benchmark domains wrt to the reference model is reported in [22]. It is worth noting the last contribution of the LOCM family, called **LOP** (LOCM with Optimized Plans), addresses the problem of inducing static predicates [23]. LOP applies a post-processing step after the LOCM analysis and it requires additional input information, particularly a set of optimal plans besides the suboptimal FO action sequences.

3. Motivation

The main novelty of FAMA with respect to other approaches lies in that our system is capable of handling PO and NO action sequences, which combined with PO and NO state trajectories, make the learning task more challenging. This essentially brings one key difference: the transition between two given observed states may now involve more than one action; i.e., $\theta(s_i, \langle a_1, \ldots, a_k\rangle) = s_{i+1}$, with $k \geq 1$, k unknown and unbound, and so the horizon of the input plan traces is no longer known now.

In this particular scenario, the actual number of plan traces that can correspond with the given input observations is also unbound and grows exponentially with the actual length of the plan trace (that is now unknown). Otherwise, the learning task is SAT compilable, which is known to be a NP-complete task [24]. This is the reason that SAT solving is a common technique in the approaches presented in section 2.

When we assume partial observability in both the sequence of actions and the state trajectory, a complete approach must consider the length of the input plan traces to be unknown. This work shows that classical planning is a complete approach for this particular scenario. Consequently, the new learning scenario features PSPACE-complete instead of NP-complete tasks, which motivates and justifies the use of planning techniques, as our proposal of compiling the learning task to a planning problem.

When the plan trace is fully observed, learning STRIPS action models is straightforward [25]. In this case the *pre-* and *post-states* of every action are available so action *effects* are derived lifting the literals that change between the pre and post-state of the corresponding action executions. Likewise *preconditions* are derived lifting the minimal set of literals that appears in all the pre-states of the corresponding action.

	state observability			
action observability	FO	PO*	PO	NO
FO	-	NP-complete	NP-complete	NP-complete
PO	NP-complete	NP-complete	**PSPACE-complete**	**PSPACE-complete**
NO	NP-complete	NP-complete	**PSPACE-complete**	**PSPACE-complete**

Table 4: Complexity of learning tasks according to the type of input trace

Table 4 displays the complexity of the learning task according to the type of input trace. The PSPACE-complexity happens when the length of the plan trace is a priori unbound, which results from the combination of the following two causes:

1. As many of the approaches in section 2 assume, there may be an unknown number of missing intermediate states in the trace because of partial state observability (PO and NO). The assumption of having FO state trajectory means that the sensors are able to capture every state change at every instant which typically is unrealistic. Normally the process of getting state feedback from sensors (or the processing of the sensor readings) is associated with a given sampling frequency that misses intermediate data between two sensor readings.
2. There may be also an unbound number of missing actions in the plan trace because of partial observability. The common assumption of having FO action sequences in a learning task is unrealistic in many domains because it implies the existence of human observers that annotate the observed action sequences. In some real-world applications, the observed and collected data are sensory data (e.g., home automation, robotics) or images (e.g. traffic) and one cannot rely on human intervention for labeling actions. Actually, learning the executed actions can also be part of the action-model learning task. Learning from unstructured data involves some prior processing to transform the sensor or image information into a predicate-like format before applying the action-model learning approach [26], and it also requires the ability of identifying action symbols. In this sense, FAMA represents a step ahead towards learning action models without assuming observed actions.

Regarding the metric to evaluate the learned action models, we can observe in Table 3 that most of the approaches use a similar metric that consists in (1) counting the missing and extra fluents that appear in the learned model wrt the GTM and (2), normalizing this error by the the total number of all the possible preconditions and effects of an action model. This is an *optimistic* metric since error rates are not normalized by the size of the actual GTM model. Hence, because the set of preconditions and effects of the GTM model is usually smaller than the set of all possible preconditions and effects, it turns out the metric may output error rates below 100% for totally wrong learned models.

In order to overcome this limitation, we propose to use two standard metrics from ML, *precision* and *recall*. 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 [27].

A striking figure of Table 3 that emphasizes a relevant feature of FAMA is the small size of the training dataset it requires in comparison to other approaches. Unlike extensive-data ML approaches, our work explores an alternative research direction to learn sound models from small amounts of plan traces. This is an important advantage, particularly in domains in which it is costly or impossible to obtain a significant number of training samples. Unlike CAMA, our approach does not require human intervention to label samples as it is able to learn from very small datasets.

Finally, we would also like to point out that, as shown in section 6, FAMA is exhaustively evaluated over a wide range of domains (15 domains compared to the scarce number of tested domains of the rest of the approaches in Table 3) and uses exclusively an *off-the shelf* classical planner so it can benefit straightforward from the last advances in classical planning.

4. Learning action models from plan executions

The *learning task* addressed in this paper corresponds to learning a classical planning action model by observing an agent(s) acting in a world that is defined by a *classical planning frame* $\Phi = \langle F, A\rangle$. This learning task is formalized by the pair $\Lambda = \langle \mathcal{M}, \tau\rangle$:

- $\mathcal{M}$ is the **initial action model**. This model is *empty*, when learning from scratch, or *partially specified*, when some fragments of the model are a priori known.
- τ is the observed **plan trace** such that:
  1. Observations in τ are *noiseless*, meaning that if the value of a fluent or an action is observed in τ, then the observation is correct.
  2. The initial state $s_0 \in \tau$ is *fully observed*, i.e. $|s_0| = |F|$. This means that the set of fluents F, and the corresponding set of predicates Ψ that shapes the fluents in F, are inferrable from s_0.
  3. The set of actions A is inferrable either from $\mathcal{M}$ or τ. In other words, the action name and parameters that shape the actions in A are given by $\mathcal{M}$ otherwise, τ contains at least one instantiation of every action model in $\mathcal{M}$.

4. Intermediate actions $a_i \in \tau$, and intermediate states $s_j \in \tau$, $1 \leq i, j$ are *partially observable*. In the extreme all $a_i \in \tau$, $1 \leq i \leq n$ actions could be unobserved, provided that the final state s_m is at least, partially observed. Likewise states s_j, $1 \leq j \leq m$, might be unobserved provided that at least actions a_i, $1 \leq i \leq n$, are partially observed.

A *solution* to a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ is an action model $\mathcal{M}'$ that is compliant with the input model $\mathcal{M}$ and the observed plan trace τ.

Figure 1 shows an example of a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$, that corresponds to observing the execution of the four-action plan $\pi = \langle (\texttt{unstack B A}), (\texttt{putdown B}), (\texttt{pickup A}), (\texttt{stack A B}) \rangle$ for inverting a two-block tower. In this example $\tau = \langle s_0,$ `(putdown B), (stack A B)`, $s_4 \rangle$ which means that only the first and last states are observed (the three intermediate states s_1, s_2 and s_3 are fully unknown) and that actions a_2 and a_3 are observed while a_1 and a_4 are unknown.

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

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


;;;;;; Plan trace τ

;;; Initial state observation
(clear B) (ontable A) (handempty) (on B A)
(not (clear A)) (not (ontable B)) (not (holding A)) (not (holding B))
(not (on A A)) (not (on A B)) (not (on B B))

;;; State observation
(clear A) (on A B) (ontable B) (handempty)

;;; Action observation
(putdown B)

;;; Action observation
(stack A B)
```

Figure 1: Task $\Lambda = \langle \mathcal{M}, \tau \rangle$ for learning a *blocksworld* STRIPS action model from a four-action plan and two state observations.

The definition of the learning task is extensible to the more general case where the execution of several plans generated from the same action model is observed. In this case, $\Lambda = \langle \mathcal{M}, \mathcal{T} \rangle$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_k\}$ such that each $\tau \in \mathcal{T}$ is a plan trace that satisfies the previous [1,4] assumptions. In this case a *solution* action model $\mathcal{M}'$ have to be compliant with the input model $\mathcal{M}$ but also with every observed plan trace $\tau \in \mathcal{T}$.

4.1. A propositional encoding for STRIPS action schemas

This work addresses the learning and evaluation of PDDL action schemas that follow the STRIPS requirement [28, 13]. An STRIPS action schema is a tuple $\xi = \langle name(\xi), pars(\xi), pre(\xi), add(\xi), del(\xi) \rangle$:

- The $name(\xi)$ and parameters, $pars(\xi)$, of the schema that define the *header* for that schema.
- The preconditions $pre(\xi) \subseteq \Psi$, negative effects $del(\xi) \subseteq \Psi$, and positive effects $add(\xi) \subseteq \Psi$ such that, $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ and $pre(\xi) \cap add(\xi) = \emptyset$.

As an example Figure 2 shows the *stack* action schema, coded in PDDL, from a four-operator *blocksworld* [29].

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 $pars(\xi) = pars(\xi')$. Therefore we can claim that blocksworld operators `stack` and `unstack` are *comparable* while `stack` and `pickup` are not. 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 vice versa.

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

Now we formalize a propositional encoding for the target of the learning task addressed in the paper, the STRIPS action model. This encoding is the core of our compilation approach for learning STRIPS action models with classical planning.

First, 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 actions with the maximum arity have arity two (any instantiation of the `stack` or the `unstack` schemas).

We define F_v, a new set of fluents s.t. $F \cap F_v = \emptyset$, produced instantiating Ψ using only *variable names*, and that defines the elements that can appear in the action schemes. In *blocksworld* this set contains eleven 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}. In more detail, for a given schema ξ, we define $F_\xi \subseteq F_v$ as the subset of elements that can appear in that action schema. For instance, $F_{\texttt{stack}} = F_v$ while $F_{\texttt{pickup}}$={`handempty`, `holding`_v_1, `clear`_v_1, `ontable`_v_1, `on`_v_1_v_1} excludes the elements from F_v that involve v_2 because `pickup` actions have arity one. The size of the space of possible STRIPS models for a given schema ξ is $2^{2|F_\xi|}$ (recall 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 the *blocksworld*, $2^{2|F_{\texttt{stack}}|} = 4194304$ while for the `pickup` operator this number is only 1024.

Now we are ready to define the fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_\xi$, that represent the propositional encoding for the preconditions, negative and positive effects of an action schema ξ. If a fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ holds, it means that f is a precondition/negative/positive effect in that schema. For instance, Figure 3 shows the conjunction of fluents that represents he propositional encoding for the preconditions, negative and positive effects of the *stack* schema shown in Figure 2.

```
(pre_holding_stack_v1) (pre_clear_stack_v2)
(del_holding_stack_v1) (del_clear_stack_v2)
(add_handempty_stack) (add_clear_stack_v1) (add_on_stack_v1_v2)
```

Figure 3: Propositional encoding for the *stack* schema from a four-operator *blocksworld*.

4.2. Classical planning 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 IPC [30] 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.

4.3. Compilation

Our approach for addressing a Λ learning task 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 action model** $\mathcal{M}'$. A solution plan starts with a *prefix* that, for each $\xi \in \mathcal{M}$, determines which fluents $f \in F_\xi$ belong to its *pre*(ξ), *del*(ξ) and *add*(ξ) sets.
2. **Validates the action model** $\mathcal{M}'$. The solution plan continues with a *postfix* that reproduces the given input knowledge (the available plan traces) with the programmed action model $\mathcal{M}'$.

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

- F_Λ contains:
  - The set of fluents F built instantiating the predicates Ψ with the objects Ω that appear in the plan trace given as input, i.e. the blocks A and B in the example of Figure 1.
  - Fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_\xi$, 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 $\xi \in \mathcal{M}'$.
  - Fluents $F_\pi = \{plan(name(a_i), \Omega^{ar(a_i)}, i)\}_{1 \leq i \leq n}$ to code the i^{th} action in a plan trace τ. The static facts $next_{i,i+1}$ and the fluents at_i, $1 \leq i < n$, are also added to iterate through the n actions of τ that are observed. In the example of Figure 1 two actions were observed $\langle$(`putdown B`), (`stack A B`)$\rangle$.
  - Fluents $\{test_j\}_{0 \leq j \leq m}$, indicating the state observation $s_j \in \tau$ where the action model is validated. In the example of Figure 1 two tests are required to validate the programmed action model at $\langle s_0, s_4 \rangle$.
  - The fluents $mode_{prog}$ and $mode_{val}$ to indicate whether the operator schemas are programmed or validated.
- I_Λ encodes the first state observation, $s_0 \subseteq F$ and sets to true $mode_{prog}$ as well as fluent $test_0$ and the fluents F_π plus at_1 and $\{next_{i,i+1}\}$, $1 \leq i < n$, for tracking the action where the programmed model is validated. Our compilation assumes that initially, schemas are programmed with no precondition, no negative effect and no positive effect.
- $G_\Lambda = \{at_n, test_m\}$, requires that the programmed action model is validated in all the actions and states observed from the input plan trace τ.
- A_Λ comprises three kinds of actions:
  1. Actions for *programming* operator schema $\xi \in \mathcal{M}$:
     - Actions for adding a *precondition* $f \in F_\xi$ from the action schema $\xi \in \mathcal{M}$.

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

     - Actions for adding a *negative* or *positive* effect $f \in F_\xi$ 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 4 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 4: PDDL action for applying an already programmed schema *stack* (implications are coded as disjunctions).

When the input plan trace contain observed actions, then the extra conditional effects $\{at_i, plan(name(a_i), \Omega^{ar(a_i)}, i)\} \rhd \{\neg at_i, at_{i+1}\}_{\forall i\in[1,n]}$ are included in the $\mathsf{apply}_{\xi,\omega}$ actions to validate that actions are applied, exclusively, in the same order as in $\mathcal{T}$.

3. Actions for *validating* the partially observed state $s_j \in \tau$, $1 \leq j < m$.

$$\begin{aligned}
\mathsf{pre}(\mathsf{validate_j}) =& s_j \cup \{test_{j-1}\} \cup \{mode_{val}\},\\
\mathsf{cond}(\mathsf{validate_j}) =& \{\emptyset\} \rhd \{\neg test_{j-1}, test_j, \neg mode_{val}\}.
\end{aligned}$$

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 P_Λ 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 in τ iff 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 τ iff P_Λ is unsolvable. This means that $\mathcal{M}$ cannot be compliant with the given observation of the plan execution.

This validation capacity of our compilation is beyond the functionality of VAL (the plan validation tool [11]) because, our P_Λ compilation can test *model validation* with a partial (or even an empty) action model and a partially observed plan trace. On the other hand, VAL requires (1) a full plan and (2), a full action model for plan validation.

The classical plan of Figure 5 shows a solution to the classical planning task P_Λ that encodes a $\Lambda = \langle \mathcal{M}, \tau \rangle$ learning task for getting the *blocksworld* action model where the headers for the schemes `pickup`, `putdown` and `unstack` are specified in $\mathcal{M}$. This plan programs and validates the operator schema `stack` from *blocksworld*, using the plan trace shown in Figure 1. Plan steps [0, 1] program the preconditions of the `stack` operator, steps [2, 6] program the operator effects and steps [7, 11] validate the programmed operators following the four-action plan shown in Figure 1.

```
00 :  (program_pre_holding_stack_v1)
01 :  (program_pre_clear_stack_v2)
02 :  (program_eff_clear_stack_v1)
03 :  (program_eff_clear_stack_v2)
04 :  (program_eff_handempty_stack)
05 :  (program_eff_holding_stack_v1)
06 :  (program_eff_on_stack_v1_v2)
07 :  (apply_unstack blockB blockA i1 i2)
08 :  (apply_putdown blockB i2 i3)
09 :  (apply_pickup blockA i3 i4)
10 :  (apply_stack blockA blockB i4 i5)
11 :  (validate_1)
```

Figure 5: Plan for programming and validating the *stack* schema (using plan π and state observations shown in Figure 1) as well as previously specified operator schemes for *pickup*, *putdown* and *unstack*.

The compilation is flexible to the particular scenario where the number of *missing states* or *missing actions* in the input plan trace τ is bound. In both cases the output planning problem P_Λ becomes simpler since the plan horizon is bound so the classical planner does not require to 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 $\mathcal{T} = \{\tau_1, \ldots, \tau_k\}$ observations of the execution of plans $\Pi = \{\pi_1, \ldots, \pi_k\}$ in a *classical planning frame* $\Phi = \langle F, A \rangle$. Let us first define a set of classical planning instances $P_t = \langle F, A, I_t, G_t \rangle$, $1 \leq t \leq k$, that belong to Φ (i.e. same fluents and actions but different initial states and goals). The initial state I_t is given by the state $s_0^t \in \tau_t$ and the observed actions from the plan π_t while the goals G_t are defined by the number of observed actions and states from the corresponding plan trace τ_t. Addressing the learning task $\Lambda = \langle \mathcal{M}, \mathcal{T} \rangle$ where $\mathcal{T} = \{\tau_1, \ldots, \tau_k\}$ requires introducing a small modification to our compilation. In particular, the actions in P_Λ for *validating* last state $s_m^t \in \tau_t$ of a plan trace τ_t reset the current state and the current plan. These actions are now redefined as:

$$\begin{aligned}
\mathsf{pre}(\mathsf{validate_t}) = & G_t \cup \{test_{j-1}\} \cup \{\neg mode_{prog}\}, \\
\mathsf{cond}(\mathsf{validate_t}) = & \{\emptyset\} \rhd \{\neg test_{j-1}, test_t\} \cup \\
& \{\neg f\}_{\forall f \in G_t, f \notin I_{t+1}} \cup \{f\}_{\forall f \in I_{t+1}, f \notin G_t}, \\
& \{\neg f\}_{\forall f \in F_{\pi_t}} \cup \{f\}_{\forall f \in F_{\pi_{t+1}}}.
\end{aligned}$$

4.4. 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 at_n and $test_m$ 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 $\mathcal{T}$. 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 $\mathcal{M}'$ definable within F_v that satisfies the state trajectory constraint given by $\mathcal{T}$. This means that a solution plan can be built selecting the corresponding $\mathsf{programPre}_{\mathsf{f},\xi}$ and $\mathsf{programEff}_{\mathsf{f},\xi}$ actions according to $\mathcal{M}'$ and later, selecting the corresponding $\mathsf{apply}_{\xi,\omega}$ and $\mathsf{validate}_{\mathsf{i}}$ actions according to $\mathcal{T}$. □

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 and the corresponding set of *programming* actions.
- The length of $\mathcal{T}$, the observed plan execution. The larger $1 \leq j \leq n$, the more $\{test_j\}$ fluents and $\{\mathsf{validate}_{\mathsf{j}}\}$ actions in P_Λ.

4.5. 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 [31]. 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.

A *static predicate* $p \in \Psi$ is a predicate that does not appear in the effects of any action [32]. 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 $\mathsf{programEff}_{\mathsf{f},\xi}$ (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 $\mathsf{programPre}_{\mathsf{f},\xi}$ can be discarded for every $\xi \in \mathcal{M}$. For instance, in the *zenotravel* [33] 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 $\mathsf{programPre}_{\mathsf{f},\xi}$ 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 $pre\_next\_fly\_v5\_v4$.

5. Evaluating STRIPS Action Models with Classical Planning

If the actual GTM model is available then the quality of a learned action model is quantifiable using the ML metrics, *precision* and *recall*. *Precision* and *recall* are two syntactic metrics frequently used in *pattern recognition*, *information retrieval* and *binary classification* and are more informative than counting the number of errors between the learned and the reference model [27]. 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* (in our case, 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).

In a Λ learning task, the roles of two *comparable* action schemes (or the roles of two action parameters with the same type) can be swapped. These role swaps typically happen when the observed input data, given in $\mathcal{T}$, is scarce. For instance when $\mathcal{T}$ does not contain any intermediate action, the *blocksworld* operator `stack` can be *learned* with the preconditions and effects of the `unstack` operator and vice versa. Further, the roles of parameters of the `stack` (or the `unstack`) operator could be swapped and these learned models would still be semantically correct with respect to the given input observations.

Pure syntax-based evaluation metrics (like *precision* and *recall*) can report low scores for learned models that are actually *sound* and *complete* 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 a novel evaluation approach that is robust to role changes of this particular kind. The intuition of the approach is to *semantically* 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. This semantic evaluation approach is again 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 $\delta(\mathcal{M}, *) = \sum_{\xi \in \mathcal{M}} 3 \times |F_v(\xi)|$.*

Note that the number of deletions indicate the number of *false positives* while the number of insertions corresponds to the number of *false negatives*. As explained, these numbers can be normalized with the amount of *true positives* to compute the *precision* and *recall* of a given learned model with respect to a reference model.

Normally evaluating a given learned domain with respect to the actual GTM model is not possible because the actual GTM model is unknown. 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 that are assumed to be generated with that model [14]. With this regard, we define now an edit distance to asses the quality of a learned action model with respect to a partially observed plan trace $\mathcal{T}$.

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 $\mathcal{T}$;*

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

Following the previous correspondences (*false positives* ≡ *deletions* and *false negatives* ≡ *insertions*) we can define a *semantic* version of the *precision* and *recall* metrics of a learned model. In this case *precision* and *recall* is not computed with respect to a GTM but with respect to a partially observed plan trace $\mathcal{T} = \langle s_0, a_{,1}, s_1, \ldots, a_n, s_n\rangle$. To define the *semantic* version of *precision* and *recall* the only term to re-formalize is the number of *true positives* that, for each action schema $\xi \in \mathcal{M}$, is given by $tp(\xi) = |pre(\xi)| + |del(\xi)| + |add(\xi)| - fp(\xi)$.

5.2. Computing the edit distance with classical planning

Our compilation is extensible to compute the *observation edit distance* and hence, the semantic versions of the *precision* and *recall* metrics. This extension considers that the input STRIPS model $\mathcal{M}$, is *non-empty* so instead of learning an action model from scratch we simply edit $\mathcal{M}$ until it satisfies the given input observations. In other words, now $\mathcal{M}$ is a set of given operator schemas, wherein each $\xi \in \mathcal{M}$ initially contains *head*(ξ) but also the *pre*(ξ), *del*(ξ) and *add*(ξ) sets. A solution to the classical 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*.
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.

In more detail given $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T}\rangle$, the output of the extended compilation is a classical planning task $P'_\Lambda = \langle F_\Lambda, A'_\Lambda, I_\Lambda, G_\Lambda\rangle$ such that:

- F_Λ, I_Λ and G_Λ are defined as in the previous compilation. Note that, the input action model $\mathcal{M}$ is 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 I_Λ 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 each action for *programming* an 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 1.

```
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 1.

Our interest when solving the classical planning task that is output by our extended compilation is not in the resulting action model $\mathcal{M}'$ but in the number of required *edit operations* (insertions and deletions). For instance, in the example of Figure 6, $\mathcal{M}'$ is validated in the given observations after inserting two new positive effects to the initial action model, e.g. $\delta(\mathcal{M},\mathcal{T}) = 2$. 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 4). 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 [34].

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 $\mathsf{apply}_{\xi,\omega}$ or $\mathsf{validate}_i$ 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 [13], taken from the PLANNING.DOMAINS repository [35]. 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 [36]. 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^t_o and s^t_n, 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 5 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 [37].

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

Table 6 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 7 and 8 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

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

done only over half of the actions. This occurs, for instance, in the precondition *recall* of domains such as *Floortile*, *Gripper* or *Satellite*.

Remarkably, the overall *precision* is now 0.98, which means that the contents of the learned models is highly reliable. The value of *recall*, 0.87, is an indication that the learned models still miss some information (preconditions are again the component more difficult to be fully learned). Overall, the results confirm the previous trend: the more input knowledge of the task, the better the models and the less planning time. Additionally, the solution plans required for this task are smaller because it is only necessary to program half of the actions (the other half are included in the input knowledge $\mathcal{M}$). *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 7: *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 9 and 10 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 5. 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 11 shows the precision (**P**) and recall (**R**) computed separately for the preconditions (**Pre**), positive

	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 8: 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 9: *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 10: Time and plan length when learning from (initial,final) state pairs.

effects (**Add**) and negative Effects (**Del**) while the last two columns report averages values. The reason why the scores in Table 11 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 11: 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

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

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 12 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 13 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 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.

	$\delta(\mathcal{M}, \mathcal{O})$	$\delta(\mathcal{M}, *)$	$1 - \frac{\delta(\mathcal{M},\mathcal{O})}{\delta(\mathcal{M},*)}$
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 13: Evaluation of the quality of the learned models with respect to an observations test set.

The learning scores of several domains in Table 12 are above the ones reported in Table 11. 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 13. 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 [38].

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 [39] 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 [40], as well as learning from unstructured data, like state images [26]. 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 [38]. The success of recent algorithms for exploring planning tasks [41] motivates the development of novel techniques that enable to autonomously collect informative learning examples. The combination of such exploration techniques with our learning approach is an appealing research direction that opens up the door to the bootstrapping of planning action models.

In 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*

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, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 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] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains., Journal of Artificial Intelligence Research 20 (2003) 61–124.
[14] Q. Yang, K. Wu, Y. Jiang, Learning action models from plan examples using weighted MAX-SAT, Artificial Intelligence 171 (2-3) (2007) 107–143.
[15] E. Amir, A. Chang, Learning partially observable deterministic action models, Journal of Artificial Intelligence Research 33 (2008) 349–402.
[16] H. H. Zhuo, Q. Yang, D. H. Hu, L. Li, Learning complex action models with quantifiers and logical implications, Artificial Intelligence 174 (18) 1540–1569.
[17] H. H. Zhuo, S. Kambhampati, Action-model acquisition from noisy plan traces, in: International Joint Conference on Artificial Intelligence, IJCAI-13, 2013, pp. 2444–2450.
[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] H. H. Zhuo, Crowdsourced action-model acquisition for planning, in: National Conference on Artificial Intelligence, AAAI-15, 2015, pp. 3439–3446.
[20] S. Cresswell, T. L. McCluskey, M. M. West, Acquisition of object-centred domain models from planning examples, in: International Conference on Automated Planning and Scheduling, ICAPS-09, 2009.
[21] S. N. Cresswell, T. L. McCluskey, M. M. West, Acquiring planning domain models using LOCM, The Knowledge Engineering Review 28 (02) (2013) 195–213.
[22] S. Cresswell, P. Gregory, Generalised domain model acquisition from action traces, in: International Conference on Automated Planning and Scheduling, ICAPS-11, 2011.
[23] P. Gregory, S. Cresswell, Domain model acquisition in the presence of static relations in the LOP system, in: International Joint Conference on Artificial Intelligence, IJCAI-16, 2016, pp. 4160–4164.
[24] S. J. Russell, P. Norvig, Artificial intelligence: a modern approach, Pearson Education Limited,, 2016.
[25] 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.
[26] M. Asai, A. Fukunaga, Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary, in: National Conference on Artificial Intelligence, AAAI-18, 2018.
[27] J. Davis, M. Goadrich, The relationship between precision-recall and ROC curves, in: International Conference on Machine learning, ACM, 2006, pp. 233–240.
[28] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL – The Planning Domain Definition Language (1998).
[29] J. Slaney, S. Thiébaux, Blocks world revisited, Artificial Intelligence 125 (1-2) (2001) 119–153.
[30] 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.
[31] S. Muggleton, L. De Raedt, Inductive logic programming: Theory and methods, The Journal of Logic Programming 19 (1994) 629–679.
[32] M. Fox, D. Long, The automatic inference of state invariants in TIM, Journal of Artificial Intelligence Research 9 (1998) 367–421.
[33] D. Long, M. Fox, The 3rd international planning competition: Results and analysis, Journal of Artificial Intelligence Research 20 (2003) 1–59.
[34] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (1-2) (2001) 5–33.
[35] C. Muise, Planning. domains, ICAPS system demonstration.
[36] J. Rintanen, Madagascar: Scalable planning with sat, Proceedings of the 8th International Planning Competition (IPC-2014).
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] 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.