Available online at www.sciencedirect.com

ScienceDirect

Artificial Intelligence 00 (2018) 1–29

Artificial Intelligence

Learning action models with minimal observability

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: Action model learning, AI planning, Machine Learning

1. Introduction

There is common agreement in the planning community that the unavailability of a complete domain model is a bottleneck in the applicability of planning technology to many real-world domains [1]. Motivated by the difficulty and cost of crafting action models, research in action-model learning has seen huge advances since the emergence of the pioneer learning system ARMS [2]. Ever since, we have seen models able to learn action models with quantifiers [3, 4], from noisy actions or noisy states [5, 6], from null state information [7], from incomplete domain models [8, 9] and many more.

The primary underlying motivation for acquiring planning action models is to be able to solve model-based planning tasks afterwards. A learning system receives as an input observations of the agent's plan execution and outputs an abstract version of the capability model that reflects the physics of the real-world domain. Recently, model-based representations are also becoming popular in explainable AI planning as they form a common basis for communicating with users and facilitate the generation of transparent and explainable decisions [10] as well as explanations in terms of the differences with a human mental model [11].

Two different types of data sources are generally identified in the generation of explanations [12, 13]: (1) data that embody enough information to fully trace the steps of the decision-making process of the agent and so accurate and detailed explanations can be generated; (2) data that do not accurately represent the decision-making process because the observed external behaviour of the agent responds to a black-box model. A similar distinction can be established with the type of observations that algorithms use to learn planning action models. Up to date, all existing learning approaches assume that a given input plan trace contains a fully observed sequence of the executed actions. This seriously restricts the applicability of the learning approach to contexts in which the behaviour of the agent is fully observable and a human annotator correctly labels the observed actions.

In this paper we claim that a planning model is learnable even though an accurate representation of the agent's behaviour is not available. Particularly, we present a novel learning algorithm, called FAMA, capable of inferring the preconditions and effects of STRIPS action models, the vanilla action model for automated planning [14], under minimum input knowledge. While current learning systems accept varying degrees of observability in the states traversed by the input observed actions, none of them allow partial observability in the sequence of actions. FAMA, on the other hand, allows incomplete or empty action sequences. This way, the minimal observability case acceptable by FAMA is when the algorithm is only fed with the input and output of a plan trace, namely the initial and final state of a planning task. Like many Machine Learning (ML) techniques, FAMA is able to operate with only correct input/ouput pairs and an unknown or a partially known model of the agent. Unlike ML algorithms, FAMA requires a symbolic structured representation of the input knowledge. In this sense, recent investigations tackle the problem of learning symbolic representations from low-level sensing information and unstructured data [15, 16].

FAMA is a solving schema, based on AI planning technology, that automatically compiles the task of learning STRIPS actions into a planning task which is then solved with a planner [17]. FAMA is thereby a model-based approach that automatically builds its own planning model by logical inference from the input observations. A solution to this planning task is a sequence of actions that determines the preconditions and effects of the STRIPS action models. The construct of the planning model starts out with the input observations of the agent's behaviour that, as commented above, may comprise none of the actions executed by the agent. FAMA can thus be regarded as a solving process that generates an instance of a planning model, and whose solution must be *compliant* with the input plan traces. This behaviour largely differs from ML techniques, which aim to minimize an error function on the training data. Moreover, FAMA requires far less sample data (input observations) than ML algorithms, thus alleviating the dependency on the assumption that there are enough data for learning the action models [18].

A key aspect in action-model learning is the evaluation method to assess the quality and performance of the learning approach. The most common method is to use a syntax-based evaluation that compares the learned model with a reference model. FAMA instead proposes two novel semantic evaluation metrics that build upon two well-known syntactic ML metrics, *precision* and *recall* [19], to evaluate the output action models with respect to observations of plan executions. Our semantic evaluation is generally more informative that counting the number of error between two models and alleviates the limitations of a purely syntax-based assessment: (a) that the learned model is syntactically different from the reference model but semantically correct and (b) that the learned model comprises correct though unnecessary preconditions in regards to the reference model. This latter issue is concerned with the qualification problem, the actual impossibility of listing all the preconditions required for a real world action to have its intended effects [20].

The semantic evaluation in FAMA is carried out using the same compilation scheme for solving a learning task. In particular, FAMA also accepts as an input an initial action model of the agent's behaviour, either complete or partially specified [8, 9], alongside the observations. In this case, FAMA returns a model that follows the input model and is compliant with the observations. Following this behaviour, we design an adjustment mechanism that serves to correct the input model to the output model, which in turn defines an assessment of the accuracy with which the input model explains the observations. Interpreting the adjustment value as a distance-based concept between two models could be exploited in model reconciliation [21].

All in all, FAMA is a planning-based solving scheme that outputs a STRIPS action model using an automatically built planning model from minimal input knowledge. Unlike extensive data-ML approaches, FAMA only requires a small amount of input plan traces. Unlike most relevant action-model learning algorithms, FAMA does not require the traces to contain any observed action executed by the agent. Ultimately, FAMA adopts certain ML-like characteristics, like the ability to learn from data that do not explicitly exhibit the agents behaviour or the capacity of correcting an initially given action model.

Motivated by recent advances on the synthesis of different kinds of generative models with classical planning [22, 23, 24, 25], this paper presents FAMA, an innovative approach for learning STRIPS action models defined as a classical planning compilation. A first description of the compilation scheme previously appeared in our previous conference paper [17]. FAMA brings the following contributions over the first version of the compilation:

- A unified formulation for learning and evaluating action models from observations of plan executions. In the case of minimal observability, these executions only comprise the initial and final state of the plan traces.

- A thorough elaboration of two semantic evaluation metrics that build upon the notions of *precision* and *recall* to evaluate the output action models with respect to observations of plan executions.
- An exhaustive empirical evaluation over 14 domains from the International Planning Competitions (IPCs). We include an analysis of the impact that the size of the input knowledge has in the performance of FAMA, a comparison with ARMS and a detailed experimentation when FAMA is executed with minimal input knowledge.

The paper is organized as follows. Section 2 introduces classical planning concepts and reviews related work on learning planning action models. Section 3 motivates our compilation-to-planning approach for learning action models. Section 4 formalizes the learning task and presents the compilation scheme, the core of FAMA. Sections 5 explains the evaluation of a learned model with respect to a reference model (syntactic evaluation) and with respect to a set of plan traces (semantic evaluation). Section 6 reports the results of the experimental evaluation and, 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 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 classical planning action models and highlight our contributions comparing with 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.

A *state* s is a full assignment of values to fluents and we explicitly include negative literals $\neg f$ in states; i.e. $|s| = |F|$, so the size of the state space is $2^{|F|}$. Like in PDDL [26], 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. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ such that Ω^k is the k-th Cartesian power of Ω.

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 for P 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. A plan trace $\tau = \langle s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_n\rangle$ is generally defined as an interleaved combination of a sequence of executed actions $\langle a_1, \ldots, a_n\rangle$ and the induced state trajectory $\langle s_0, s_1, \ldots, s_n\rangle$. *Plan traces* constitute the input knowledge of the learning tasks addressed in this paper.

Our approach copes with the partial observability of the plan execution adopting the *open world assumption*. 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 might 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 the observed information.

With regard to the observed actions in a plan trace, we say that $\langle a_1, \ldots, a_n \rangle$ 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 when the sequence of observed actions is empty.

A plan trace $\tau = \langle s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_n \rangle$ for a planning frame $\Phi = \langle F, A \rangle$ holds that $a_i \in A$ for every action in τ and that $s_i \in \mathcal{L}(F)$ for each $1 \leq i \leq n$. 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 regard 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 PO considers 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 predicates and actions headers are easily extractable from the state

	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

sequence and action sequence of the input plan traces, respectively. A requirement, however, is that the input plan traces comprise at least a grounded sample of every predicate and operator schema in 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 [14] 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**) [2] 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 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 [3]. 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

	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	15 25 actions

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

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*' [3]. 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**) [4] 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 harder to learn.

The Action Model Acquisition from Noisy plan traces (**AMAN**) [5] 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 [6]. 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**) [18] 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 [27, 7]. 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 [28].

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 [28]. It is worth noting the last contribution of the LOCM family, called **LOP** (LOCM with Optimized Plans), addresses the problem of inducing static predicates [29]. 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

In this section, we will highlight the principal distinctive features of our approach FAMA with respect to the related work reviewed in Section 2.2.

When learning action models from observations of plan executions there are two main sources of partial observability:

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 the partial state observability (PO and NO). The assumption of having FO state trajectories means that the sensors are able to capture every state change at every instant, which typically is

unrealistic. Normally, the process for obtaining state feedback from sensors (or the processing of the sensor readings) is associated with a given sampling frequency that misses intermediate data between two subsequent 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 as 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, for instance, from unstructured data involves transforming the sensor or image information into a predicate-like format before applying the action-model learning approach, and it also requires the ability of identifying action symbols [16].

FAMA represents one step ahead towards learning action models without assuming observed actions. 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. Table 4 shows that the worst case complexity of learning STRIPS action models becomes PSPACE-complete when combining PO and NO state trajectories and action sequences.

In this particular scenario, the actual number of plan traces associated to a given input observation 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 [30]. This is the reason that SAT solving is a common technique in the approaches presented in section 2.

	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

When we assume partial observability in both actions and states, a complete approach must consider the length of the input plan traces to be unknown. FAMA 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, as our proposal of compiling the learning task to a classical planning problem.

When the plan trace is fully observed, learning STRIPS action models is straightforward [31]. In this case the *pre-* and *post-states* of every action are available and 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.

Regarding the evaluation of the learned action models, we can observe in Table 3 that most of the approaches use a similar syntax-based 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. The set of preconditions and effects of the GTM is usually smaller than the set of all possible preconditions and effects and thereby it turns out that these syntax-based metrics may output error rates below 100% for totally wrong learned models. To overcome this limitation we propose to use two standard metrics from ML, *precision* and *recall*, that are frequently used in pattern recognition, information retrieval and binary classification [19].

Pure syntax-based evaluation metrics, like the ones mentioned in the above paragraph, can report low scores for learned models that are actually *sound* and *complete* but syntactically different from the GTM. Semantic evaluation metrics add a distinctive value over the syntactic ones, which is that they evaluate the learned model with a set of observations of plan executions and hence they are appropriate for scenarios where the GTM is not available. In this sense, FAMA also contributes with a novel semantic-based error measure that builds upon the *precision* and *recall* metrics. Unlike the semantic metric used in ARMS [2], our semantic version of *precision* and *recall* is not sensitive to the repetition of one same flaw in the model evaluation.

On the other hand, 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, as it will be shown in section 6, FAMA is exhaustively evaluated, syntactically and semantically, over a wide range of domains (13 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

This section details the FAMA approach for learning action models. The notion of the learning task is defined in section 4.1 and the components of the compilation scheme are described in sections 4.2 and 4.3. Subsequently, the compilation approach is fully detailed in section 4.4 and the last section presents some theoretical properties of the compilation scheme.

FAMA addresses the learning and evaluation of PDDL action models that follow the STRIPS requirement [32, 26]. An STRIPS action model is a tuple $\xi = \langle name(\xi), pars(\xi), pre(\xi), add(\xi), del(\xi)\rangle$ where:

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

As an example, Figure 1 shows the action model of the *stack* operator from the four-operator *blocksworld* domain [33] encoded in PDDL.

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

Figure 1: PDDL encoding of the action model of the *stack* operator from the four-operator *blocksworld* domain.

4.1. Learning Task

Our *learning task* consists in learning classical planning action models by observing one or more agents acting in a world definable by a *classical planning frame* $\Phi = \langle F, A\rangle$. The learning task is formalized by the pair $\Lambda = \langle \mathcal{M}, \tau\rangle$:

- $\mathcal{M}$ is the set of **initial action models**. This set is *empty*, when learning from scratch, or *partially specified*, when some fragments of the action models are known a priori.
- 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 a *fully observed* state including positive and negative fluents, i.e. $|s_0| = |F|$. Consequently, the corresponding set of predicates Ψ and objects Ω that shape the fluents in F are inferrable from s_0.
3. The header of an action model is either given by $\mathcal{M}$ or inferrable from τ. In the latter case, τ must contain at least one instantiation of the respective action model header.
4. We allow plan traces with NO state trajectories and action sequences. In the extreme, all actions and intermediate states may be missing, provided that the final state is at least partially observed. The least informative plan trace is thus $\tau = \langle s_0, s_n \rangle$.

Ultimately, we can always assume that $\mathcal{M}$ will contain predicates in Ψ as well as the headers of the actions models, either explicitly provided in $\mathcal{M}$ or inferred from τ. A *solution* to a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ is a set of action models $\mathcal{M}'$ that is compliant with the input models $\mathcal{M}$ and the observed plan trace τ.

Figure 2 shows an example of a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ corresponding to the observation of the execution of the four-action plan $\pi = \langle$(unstack B A), (putdown B), (pickup A), (stack A B)$\rangle$ for inverting a two-block tower. In this example $\tau = \langle s_0,$ (putdown B), (stack A B), $s_4 \rangle$, which means that only the first and the last state 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. The set of initial action models $\mathcal{M}$ only contains two of the four headers needed, but can be completed with the headers (putdown ?v1) and (stack ?v1 ?v2) inferred from τ.

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

(pickup ?v1) (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))

;;; Action observation
(putdown B)

;;; Action observation
(stack A B)

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

Figure 2: Task $\Lambda = \langle \mathcal{M}, \tau \rangle$ associated to the observation $\tau = \langle s_0,$ (putdown B), (stack A B), $s_4 \rangle$

The definition of the learning task is extensible to the more general case where the execution of several plans from the same action models are 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, the *learned* action models $\mathcal{M}'$ have to be compliant with the input models $\mathcal{M}$ and also with every observed plan trace $\tau \in \mathcal{T}$.

4.2. *A propositional encoding for* STRIPS *action models*

In this section we formalize a propositional encoding of an STRIPS action model. This encoding is at the core of the FAMA compilation approach for addressing the learning task defined in section 4.1.

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*, which is bound to the maximum arity of an action in a given planning frame. For instance, in a three-block *blocksworld* $\Omega = \{block_1, block_2, block_3\}$ while $\Omega_v = \{v_1, v_2\}$ because the actions with the maximum arity have arity two; i.e., any instantiation of the stack or the unstack models.

We define Ψ_v as the set of predicates Ψ parameterized with the *variable names* of Ω_v as arguments. The set Ψ_v defines the elements that can appear in the preconditions and effects of the action models. In the *blocksworld* domain, this set contains eleven elements, Ψ_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 action model ξ, we define $\Psi_\xi \subseteq \Psi_v$ as the subset of elements of Ψ_v that can appear in ξ. For instance, $\Psi_{\texttt{stack}} = \Psi_v$ whereas $\Psi_{\texttt{pickup}}$={handempty, holding(v_1), clear(v_1), ontable(v_1), on(v_1,v_1)} excludes the elements from Ψ_v that involve v_2 because pickup actions have arity one. The size of the space of possible STRIPS models for a given ξ is $2^{2|\Psi_\xi|}$ (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*, $2^{2|\Psi_{\texttt{stack}}|} = 4,194,304$ while for the pickup operator this number is only 1024.

We are now ready to define the propositional encoding of *pre*(ξ), *del*(ξ) and *add*(ξ). For every ξ and $p \in \Psi_\xi$, we create:

- $pre_p(\xi)$: fluent formed by the combination of the prefixes pre and *name*(ξ) plus a fluent of arity 0 that results from appending the elements of p (e.g. pre_stack_on_v1_v2, for *name*(ξ) = stack and p = on(v_1, v_2))
- $del_p(\xi)$: fluent formed by the combination of the prefixes del and *name*(ξ) plus a fluent of arity 0 that results from appending the elements of p (e.g. del_stack_on_v1_v2)
- $add_p(\xi)$: fluent formed by the combination of the prefixes add and *name*(ξ) plus a fluent of arity 0 that results from appending the elements of p (e.g. add_stack_on_v1_v2)

For a given action model ξ, if a fluent $pre_p(\xi)/del_p(\xi)/add_p(\xi)$ holds in a state, it means that p is a precondition/negative/positive effect of ξ. For instance, Figure 3 shows the conjunction of fluents that represents the propositional encoding for the preconditions, negative effects and positive effects of the action model of the *stack* operator shown in Figure 1.

```
(pre_stack_holding_v1) (pre_stack_clear_v2)
(del_stack_holding_v1) (del_stack_clear_v2)
(add_stack_handempty) (add_stack_clear_v1) (add_stack_on_v1_v2)
```

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

4.3. 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 [34] 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.

4.4. Compilation

In section 3, we exposed our reasons to solve the learning task via compiling it into a classical planning task. Our compilation scheme builds upon the approach presented in [17] but FAMA comes up with a more general and flexible scheme able to capture any type of input plan trace.

Our approach to solving a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ is to compile Λ into a classical planning task P_Λ with conditional effects. The intuition behind the FAMA compilation is that a solution plan π_Λ to P_Λ induces the output set of action models $\mathcal{M}'$ that solves Λ. Specifically, a solution plan π_Λ serves two purposes:

1. **To program the set of action models $\mathcal{M}'$.** π_Λ comprises a plan *prefix* whose actions determine the predicates $p \in \Psi_\xi$ that belong to $pre(\xi)$, $del(\xi)$ and $add(\xi)$ for each $\xi \in \mathcal{M}$.
2. **To validate the set of action models $\mathcal{M}'$.** π_Λ also comprises a plan *postfix* whose actions target the validation of the observed plan trace τ with the programmed action models $\mathcal{M}'$.

Here we formalize the compilation for learning STRIPS action models with classical planning. Given a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$, with τ formed by an n-action sequence $\langle a_1, \ldots, a_n \rangle$ and a m-state trajectory $\langle s_0, s_1, \ldots, s_m \rangle$ ($\tau = \langle s_0, a_1, \ldots, a_n, s_m \rangle$), 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 obtained from s_0; i.e., F.
  - The fluents $pre_p(\xi)$, $del_p(\xi)$ and $add_p(\xi)$, for every $\xi \in \mathcal{M}$ and $p \in \Psi_\xi$, defined as explained in section 4.2
  - A set of fluents $F_\pi = \{plan(name(a_i), \Omega^{ar(a_i)}, i)\}_{1 \leq i \leq n}$ to represent the i^{th} observable action of τ. In the example of Figure 2, the two observed actions `(putdown B)` and `(stack A B)` would be encoded as fluents `(plan-putdown B i1)` and `(plan-stack A B i2)` to indicate that `(putdown B)` is observed in the first place and `(stack A B)` is the second observed action.
  - Two fluents, at_i and $next_{i,i+1}$, $1 \leq i \leq n$, to iterate through the n observed actions of τ. The former is used to ensure that actions are executed in the same order as they are observed in τ. The latter is used to iterate to the next planning step when solving P_Λ.
  - A set of fluents $\{test_j\}_{0 \leq j \leq m}$, to point at the state observation $s_j \in \tau$ where the action model is validated. In the example of Figure 2 two tests are required to validate the programmed action model, one test at s_0 and another one at s_4.
  - A fluent, $mode_{prog}$, to indicate whether action models are being programmed or validated.

- I_Λ encodes s_0 and the following fluents set to true: $mode_{prog}$, $test_0$, F_π, at_1 and $\{next_{i,i+1}\}$, $1 \leq i \leq n$. Our compilation assumes that action models are initially programmed with no precondition, no negative effect and no positive effect.

- G_Λ includes the positive fluents at_n and $test_m$. When these two goals are achieved by the solution plan π_Λ, we will be certain that the programmed action models are validated in all the actions and states observed in the input plan trace τ.

- A_Λ comprises three kinds of actions:
  1. Actions for *programming* an action model $\xi \in \mathcal{M}$. These actions will form the prefix of the solution plan π_Λ and they are aimed at generating the appropriate state configuration to shape ξ. Among the programming actions, we find:
     - Actions which support the addition of a *precondition* $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$.

$$\begin{aligned}
\mathsf{pre}(\mathsf{programPre}_{\mathsf{p},\xi}) =& \{\neg pre_p(\xi), \neg del_p(\xi), \neg add_p(\xi), mode_{prog}\},\\
\mathsf{cond}(\mathsf{programPre}_{\mathsf{p},\xi}) =& \{\emptyset\} \rhd \{pre_p(\xi)\}.
\end{aligned}$$

     - Actions which support the addition of a *negative* or *positive* effect $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$.

$$\begin{aligned}
\mathsf{pre}(\mathsf{programEff}_{\mathsf{p},\xi}) =& \{\neg del_p(\xi), \neg add_p(\xi), mode_{prog}\},\\
\mathsf{cond}(\mathsf{programEff}_{\mathsf{p},\xi}) =& \{pre_p(\xi)\} \rhd \{del_p(\xi)\},\\
& \{\neg pre_p(\xi)\} \rhd \{add_p(\xi)\}.
\end{aligned}$$

2. Actions for *applying* a programmed action model $\xi \in \mathcal{M}$ bound to objects $\omega \subseteq \Omega^{ar(\xi)}$. These actions will be part of the postfix of the solution plan π_Λ and they execute ξ according to the current state configuration, i.e., the values of $pre_p(\xi)$, $del_p(\xi)$ and $add_p(\xi)$. Since action headers are known, the variables $pars(\xi)$ are bound to the objects in ω that appear in the same position. Figure 4 shows the PDDL encoding for applying a programmed action model of the *stack* operator from the *blocksworld* domain.

$$\begin{aligned}
\mathsf{pre}(\mathsf{apply}_{\xi,\omega}) =& \{pre_p(\xi) \implies p(\omega)\}_{\forall p \in \Psi_\xi},\\
\mathsf{cond}(\mathsf{apply}_{\xi,\omega}) =& \{del_p(\xi)\} \rhd \{\neg p(\omega)\}_{\forall p \in \Psi_\xi},\\
& \{add_p(\xi)\} \rhd \{p(\omega)\}_{\forall p \in \Psi_\xi},\\
& \{mode_{prog}\} \rhd \{\neg mode_{prog}\}.
\end{aligned}$$

```
(:action apply_stack
  :parameters (?o1 - object ?o2 - object)
  :precondition
   (and (or (not (pre_stack_on_v1_v1)) (on ?o1 ?o1))
        (or (not (pre_stack_on_v1_v2)) (on ?o1 ?o2))
        (or (not (pre_stack_on_v2_v1)) (on ?o2 ?o1))
        (or (not (pre_stack_on_v2_v2)) (on ?o2 ?o2))
        (or (not (pre_stack_ontable_v1)) (ontable ?o1))
        (or (not (pre_stack_ontable_v2)) (ontable ?o2))
        (or (not (pre_stack_clear_v1)) (clear ?o1))
        (or (not (pre_stack_clear_v2)) (clear ?o2))
        (or (not (pre_stack_holding_v1)) (holding ?o1))
        (or (not (pre_stack_holding_v2)) (holding ?o2))
        (or (not (pre_stack_handempty)) (handempty)))
  :effect
   (and (when (del_stack_on_v1_v1) (not (on ?o1 ?o1)))
        (when (del_stack_on_v1_v2) (not (on ?o1 ?o2)))
        (when (del_stack_on_v2_v1) (not (on ?o2 ?o1)))
        (when (del_stack_on_v2_v2) (not (on ?o2 ?o2)))
        (when (del_stack_ontable_v1) (not (ontable ?o1)))
        (when (del_stack_ontable_v2) (not (ontable ?o2)))
        (when (del_stack_clear_v1) (not (clear ?o1)))
        (when (del_stack_clear_v2) (not (clear ?o2)))
        (when (del_stack_holding_v1) (not (holding ?o1)))
        (when (del_stack_holding_v2) (not (holding ?o2)))
        (when (del_stack_handempty) (not (handempty)))
        (when (add_stack_on_v1_v1) (on ?o1 ?o1))
        (when (add_stack_on_v1_v2) (on ?o1 ?o2))
        (when (add_stack_on_v2_v1) (on ?o2 ?o1))
        (when (add_stack_on_v2_v2) (on ?o2 ?o2))
        (when (add_stack_ontable_v1) (ontable ?o1))
        (when (add_stack_ontable_v2) (ontable ?o2))
        (when (add_stack_clear_v1) (clear ?o1))
        (when (add_stack_clear_v2) (clear ?o2))
        (when (add_stack_holding_v1) (holding ?o1))
        (when (add_stack_holding_v2) (holding ?o2))
        (when (add_stack_handempty) (handempty))
        (when (modeProg) (not (modeProg)))))
```

Figure 4: PDDL action for applying an already programmed model for *stack* (implications are coded as disjunctions).

When the input plan trace contains observed actions, 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 ensure that actions are applied in the same order as in τ.

3. Actions for *validating* the partially observed state $s_j \in \tau$, $1 \leq j < m$. These actions are also part of the postfix of the solution plan π_Λ and they are aimed at checking that the input plan trace τ follows after the apply actions.

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

In some contexts, it is reasonable to assume that some parts of the action model are known and so there is no need to learn the entire model from scratch [8]. In FAMA, when an action model ξ is partially specified, the known preconditions and effects are encoded as fluents $pre_p(\xi)$, $del_p(\xi)$ and $add_p(\xi)$ set to true in the initial state I_Λ. In this case, the corresponding programming actions, $\mathsf{programPre}_{p,\xi}$ and $\mathsf{programEff}_{p,\xi}$, become unnecessary and are removed from A_Λ, thereby making the classical planning task P_Λ easier to be solved.

So far we have explained the compilation for learning from a single input trace. However, the compilation is extensible to the more general case $\Lambda = \langle \mathcal{M}, \mathcal{T} \rangle$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_k\}$ is a set of plan traces. Taking this into account, a small modification is required in our compilation approach. In particular, the actions in P_Λ for *validating* the last state $s_m^t \in \tau_t$, $1 \leq t \leq k$ of a plan trace τ_t reset the current state and the current plan. These actions are now redefined as:

$$
\begin{aligned}
\mathsf{pre}(\mathsf{validate_j}) =& s_m^t \cup \{test_{j-1}\} \cup \{\neg mode_{prog}\},\\
\mathsf{cond}(\mathsf{validate_j}) =& \{\emptyset\} \rhd \{\neg test_{j-1}, test_j\} \cup\\
& \{\neg f\}_{\forall f \in s_m^t, f \notin s_0^{t+1}} \cup \{f\}_{\forall f \in s_0^{t+1}, f \notin s_m^t},\\
& \{\neg f\}_{\forall f \in F_{\pi_t}} \cup \{f\}_{\forall f \in F_{\pi_{t+1}}}.
\end{aligned}
$$

Finally, we will detail the composition of a solution plan π_Λ to a planning task P_Λ and the mechanism to extract the output set of action models $\mathcal{M}'$ from π_Λ. The plan of Figure 5 shows a solution to the task P_Λ that encodes a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ for obtaining the action models of the *blocksworld* domain, where the models for `pickup`, `putdown` and `unstack` are already specified in $\mathcal{M}$. Therefore, this plan programs and validates the action model for `stack`, using the input plan trace of Figure 2. Plan steps 00 – 01 program the preconditions of the `stack` model, steps 02 – 06 program the action model effects and steps 07 – 11 is the plan postfix that validates the programmed model following the plan trace of Figure 2.

```
00 :  (program_pre_stack_holding_v1)
01 :  (program_pre_stack_clear_v2)
02 :  (program_eff_stack_clear_v1)
03 :  (program_eff_stack_clear_v2)
04 :  (program_eff_stack_handempty)
05 :  (program_eff_stack_holding_v1)
06 :  (program_eff_stack_on_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* action model (using the plan trace τ of Figure 2) as well as previously specified action models for *pickup*, *putdown* and *unstack*.

Given a solution plan π_Λ that solves P_Λ, the set of action models $\mathcal{M}'$ that solves $\Lambda = \langle \mathcal{M}, \tau \rangle$ are computed in linear time and space. In order to do so, π_Λ is executed in the initial state I_Λ and the action model $\mathcal{M}'$ will be given by the fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$ that are set to true in the last state reached by π_Λ, $s_g = \theta(I_\Lambda, \pi_\Lambda)$. For each $\xi \in \mathcal{M}'$, we build the sets of preconditions, positive effects and negative effects as follows:

$$
\begin{aligned}
pre(\xi) =& \{p \mid pre_p(\xi) \in s_g\}_{\forall p \in \Psi_\xi},\\
add(\xi) =& \{p \mid add_p(\xi) \in s_g\}_{\forall p \in \Psi_\xi},\\
del(\xi) =& \{p \mid del_p(\xi) \in s_g\}_{\forall p \in \Psi_\xi}.
\end{aligned}
$$

The logical inference process we base our approach on has trouble learning preconditions that are not, at the same time, negative effects, since no change can be observed between the pre-state and post-state of an action. This is specially the case for static predicates that never change and, hence, can only be preconditions. In order to address this shortcoming, we apply a post-process based on the one proposed in [35]. In their proposal, they obtain a full trace by applying the sequence of actions of the input trace and infer the preconditions from this fully observable trace. In our case, since the sequence of actions of the input trace might not be fully observable, we produce the traces by applying the actions found in the validation part of the solution plan. For instance, in the example of the figure 5, the sequence of actions used to produce the full trace would be (unstack blockB blockA), (put-down blockB), (pick-up blockA), and (stack blockA blockB). This post-process allows FAMA to learn more complete lists of preconditions and deal with the always problematic static predicates.

4.5. Properties of the compilation

Lemma 1. *Soundness. Any classical plan π_Λ that solves P_Λ induces a set of action models $\mathcal{M}'$ that solves $\Lambda = \langle \mathcal{M}, \tau \rangle$.*

Proof sketch. Once action models $\mathcal{M}'$ 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 state reached by π_Λ. By the definition of the $\mathsf{apply}_{\xi,\omega}$ and the $\mathsf{validate}_\mathsf{j}$ actions, these goals can only be achieved executing an applicable sequence of programmed action models that reaches every state $s_j \in \tau$, starting in the corresponding initial state and following the sequence of n observed actions of τ. This means that the programmed action model $\mathcal{M}'$ complies with the provided input knowledge and hence, that $\mathcal{M}'$ is a solution to Λ. □

Lemma 2. *Completeness. Any set of action models $\mathcal{M}'$ that solves $\Lambda = \langle \mathcal{M}, \tau \rangle$ is computable solving the corresponding classical planning task P_Λ.*

Proof sketch. By definition, $\Psi_\xi \subseteq \Psi_v$ fully captures the set of elements that can appear in an action model $\xi \in \mathcal{M}$. The compilation does not discard any possible set of action models $\mathcal{M}'$ definable within Ψ_v that satisfies the observed state trajectory and action sequence of τ. This means that for every $\mathcal{M}'$ that solves Λ, there exists a plan π_Λ that can be built selecting the appropriate programming, apply and validate actions from the P_Λ compilation. □

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

- The arity of the actions and the fluents in τ given as input in Λ. The larger the arity, the larger the size of the Ψ_ξ sets. This is the term that dominates the compilation size because it defines the $pre_p(\xi)/del_p(\xi)/add_p(\xi)$ fluents and the corresponding set of *programming* actions.
- The length of the observed action sequence and state trajectory of τ. The larger the number of observed actions, $a_i \in \tau$ s.t. $1 \leq i \leq n$, the more $\{at_i\}$ fluents. The larger the number of observed states, $s_j \in \tau$ s.t. $1 \leq j \leq m$, the more $\{test_j\}$ fluents and $\{\mathsf{validate}_\mathsf{j}\}$ actions in P_Λ.

An interesting aspect of our approach is that when a *fully* or *partially specified* STRIPS action model $\mathcal{M}$ is given in Λ, the P_Λ compilation also serves to validate whether the observed τ follows the given model $\mathcal{M}$:

- $\mathcal{M}$ is proved to be a *valid* 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* 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.

The validation capacity of our compilation is beyond the functionality of VAL (the plan validation tool [36]) because our P_Λ compilation is able to address *model validation* of a partial (or even an empty) action model with a partially observed plan trace. On the other hand, VAL requires (1) a full plan and (2), a full action model for plan validation.

5. Evaluation of action models

In this section we introduce the metrics used by FAMA to evaluate the action models that result from solving a learning task Λ. First, we will describe two standard syntactic metrics (*precision* and *recall*) and then section 5.1 will define a semantic evaluation measure that builds upon *precision* and *recall*. Finally, section 5.2 explains how FAMA computes our novel semantic-based metrics.

When the planning reference model of the input observations (i.e., the GTM) is available, the quality of the learned action models is measurable using two well-studied syntax-based metrics, *precision* and *recall*, commonly used in tasks such as information retrieval and recommender systems [19]. These two syntactic metrics are generally more informative than counting the number of errors between the learned action models and the GTM. 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 particular 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).

Introducing semantic-based evaluation metrics can be justified on two grounds:

1. When the GTM is unknown. This is the most common scenario in ML, where models are both learned and evaluated with respect to datasets.
2. When the GTM is known but a test-based evaluation on a dataset is preferable (or needed as complementary to a syntactic evaluation). As a rule of thumb, it is preferable to evaluate the learned models wrt a dataset because a learned model can be semantically correct though syntactically incorrect (different from the GTM). We refer to this phenomenon as *model reformulation*.

An example of *model reformulation* is the swapping of the roles of two *comparable* action models. Two action models ξ and ξ' are comparable if both have the same parameters (iff $pars(\xi) = pars(\xi')$) and so they share the same space of possible models. Hence, the *blocksworld* operator `stack` could be *learned* with the preconditions and effects of the `unstack` operator, and viceversa, because they are comparable. On the contrary, this reformulation will not happen between the `stack` and `pickup` because they are not comparable. In the same way, the roles of two action parameters that share the same type can also be swapped (e.g., interchanging the role of the two parameters of the operator `stack` or the opreator `unstack`) and yet the learned models would be semantically correct with respect to the given input observations. A more complex kind of reformulation occurs when two or more action models are learned in a single *macro-action*.

These semantic alterations typically appear in the learned models when the observed input data given in τ is scarce. Defining a proper semantic evaluation is key because the application of syntax-based metrics may report low scores for learned models that are actually *sound* and *complete* but correspond to *reformulations* of the GTM. In the following sections, we introduce a novel evaluation metric that is robust to different types of reformulation.

5.1. Semantic-based precision and recall

The ARMS system showed that a semantic evaluation can be done via validation of a set of plan traces with the learned model [2]. The underlying idea is that an error indication of the learned action models is obtained by counting the number of preconditions that are not satisfied during the execution of the plan trace with the learned models, similarly to the functionality provided by the automatic validation tool VAL [36] used in the IPCs. This approach can be understood as modifying the plan trace (by adding the necessary preconditions to the intermediate states) so as to allow the execution of the observed actions using the learned models. In other words, modifying the plan trace to fit the model. Inspired by this approach, we present here an alternative evaluation that, instead, determines the modifications required by a learned model to explain the given plan traces.

The rationale behind our novel metrics lies in counting the *edit operations* that need to be applied in a set of action models $\mathcal{M}$ to fit the plan traces. Given a set of action models $\mathcal{M}$, the two allowed edit operations are:

- *Deletion*. A fluent $pre_p(\xi)/del_p(\xi)/add_p(\xi)$ is removable from $\xi \in \mathcal{M}$.
- *Insertion*. A fluent $pre_p(\xi)/del_p(\xi)/add_p(\xi)$ can be added to $\xi \in \mathcal{M}$.

We now provide formal definitions of $INS(\mathcal{M}, \mathcal{M}')$ and $DEL(\mathcal{M}, \mathcal{M}')$, the sets of insertions and deletions, respectively, that are needed to transform a set of action models $\mathcal{M}$ into a new set of action models $\mathcal{M}'$.

Definition 3. *Let* $PRE(\xi) = \bigcup_{\forall p \in pre(\xi)} pre_p(\xi)$, $ADD(\xi) = \bigcup_{\forall p \in add(\xi)} add_p(\xi)$, *and* $DEL(\xi) = \bigcup_{\forall p \in del(\xi)} del_p(\xi)$ *be the set of propositional fluents that represent preconditions, positive and negative effects of a given action model* ξ*. We define:*

$$\begin{aligned} INS(\mathcal{M}, \mathcal{M}') = & PRE(\xi')\backslash PRE(\xi) \cup \\ & ADD(\xi')\backslash ADD(\xi) \cup \\ & DEL(\xi')\backslash DEL(\xi), \forall \xi \in \mathcal{M}, \xi' \in \mathcal{M}' s.t.\ name(\xi) = name(\xi') \end{aligned}$$

$$\begin{aligned} DEL(\mathcal{M}, \mathcal{M}') = & PRE(\xi)\backslash PRE(\xi') \cup \\ & ADD(\xi)\backslash ADD(\xi') \cup \\ & DEL(\xi)\backslash DEL(\xi'), \forall \xi \in \mathcal{M}, \xi' \in \mathcal{M}' s.t.\ name(\xi) = name(\xi') \end{aligned}$$

With these ingredients in mind, we first adapt the definitions of syntactic precision and recall to sets of action models. Let $\mathcal{M}$ be a set of learned action models and let $\mathcal{M}'$ be the GTM. We know that $size(\mathcal{M}) = |pre(\xi)| + |add(\xi)| + |del(\xi)|\, \forall \xi \in \mathcal{M}$ and by definition the number of preconditions and effects of the learned action models is equal to the sum of *true positives* and *false positives*; that is, $size(\mathcal{M}) = tp + fp$.

The number of *deletions* required to transform $\mathcal{M}$ into $\mathcal{M}'$ ($|DEL(\mathcal{M}, \mathcal{M}')|$) matches our previous definition of the number of *false positives*; and $|INS(\mathcal{M}, \mathcal{M}')|$, the number of *insertions* required to transform $\mathcal{M}$ into $\mathcal{M}'$, corresponds to the number of *false negatives* of $\mathcal{M}$. Then we can affirm that $size(\mathcal{M}') = size(\mathcal{M}) - |DEL(\mathcal{M}, \mathcal{M}')| + |INS(\mathcal{M}, \mathcal{M}')|$.

Definition 4. *The precision of* $\mathcal{M}$ *relative to the GTM is defined as the fraction of the common predicates and effects between* $\mathcal{M}$ *and the GTM among all predicates and effects of* $\mathcal{M}$*. This gives an intuitive measure of the soundness of* $\mathcal{M}$*.*

$$Precision = \frac{tp}{tp + fp} = \frac{size(\mathcal{M}) - |DEL(\mathcal{M}, GTM)|}{size(\mathcal{M})}$$

Definition 5. *The recall of* $\mathcal{M}$ *relative to the GTM is defined as the fraction of the common predicates and effects between* $\mathcal{M}$ *and the GTM among all predicates and effects of the GTM. This gives an intuitive measure of the completeness of* $\mathcal{M}$*.*

$$Recall = \frac{tp}{tp + fn} = \frac{size(\mathcal{M}) - |DEL(\mathcal{M}, GTM)|}{size(\mathcal{M}) - |DEL(\mathcal{M}, GTM)| + |INS(\mathcal{M}, GTM)|}$$

Definitions 4 and 5 are syntax-based metrics to evaluate $\mathcal{M}$ with respect to the GTM. A semantic metric, on the other hand, evaluates $\mathcal{M}$ with respect to a given set of plan traces.

We interpret the semantic evaluation of action models as a learning task $\Lambda = \langle \mathcal{M}, \mathcal{T} \rangle$, where:

- $\mathcal{M}$ is a **set of learned action models** obtained using any learning approach such as FAMA; in general, $\mathcal{M}$ can be any given input set of action models even manually encoded.
- $\mathcal{T}$ is a set of plan traces used for **testing**.

A solution to this task is an **edited set of action models** $\mathcal{M}'$ such that (1) $\mathcal{M}'$ is obtained by exclusively applying a finite sequence of *deletion* and *insertion* operations to $\mathcal{M}$ and (2) $\mathcal{M}'$ explains $\mathcal{T}$; i.e. $\mathcal{M}'$ is *compliant* with every plan trace $\tau \in \mathcal{T}$. It is always recommended for the test set to be different from the one used during learning and this is specially important for satisfying approaches such as FAMA; otherwise $\mathcal{M}' = \mathcal{M}$ since $\mathcal{M}$ would be able to explain $\mathcal{T}$ without any modification.

Since we are defining the semantic evaluation task in terms of a learning task Λ, there might exist potentially many edited models $\mathcal{M}'$ which are solution to this task. Although the actual GTM is included among the solution set, it is impossible to identify it, so we define the best solution based on its proximity to the input model.

Definition 6. *Given a set of action models $\mathcal{M}$, and all the sets of action models $\mathcal{M}'$ able to explain the plan traces $\mathcal{T}$. The* **closest compliant set of action models**, $\mathcal{M}^*$, *is the comparable set of action models closest to $\mathcal{M}$ (in terms of editions) that is able to explain $\mathcal{T}$;*

$$\mathcal{M}^* = \underset{\forall \mathcal{M}' \rightarrow \mathcal{T}}{\arg\min} \left|INS(\mathcal{M}, \mathcal{M}') \cup DEL(\mathcal{M}, \mathcal{M}')\right|$$

The closest compliant set of action models $\mathcal{M}^*$ allows us to define a semantic version of *precision* and *recall* following definitions 4 and 5.

$$\begin{aligned} \textit{sem-Precision} &= \frac{size(\mathcal{M}) - |DEL(\mathcal{M}, \mathcal{M}^*)|}{size(\mathcal{M})} \\ \textit{sem-Recall} &= \frac{size(\mathcal{M}) - |DEL(\mathcal{M}, \mathcal{M}^*)|}{size(\mathcal{M}) - |DEL(\mathcal{M}, \mathcal{M}^*)| + |INS(\mathcal{M}, \mathcal{M}^*)|} \end{aligned}$$

Proposition 7. *When the closest compliant set of action models $\mathcal{M}^*$ of an evaluation task $\Lambda = \langle \mathcal{M}, \mathcal{T} \rangle$ is the GTM, the syntactic and semantic evaluation of $\mathcal{M}$ return the same values; that is, Precision = sem-Precision = and Recall = sem-Recall.*

The intuition behind this evaluation is to *semantically* assess how well the learned action models $\mathcal{M}$ explain a set of given observations of plan executions according to the amount of *edition* required by $\mathcal{M}$ to induce the observations. Unlike the semantic metric defined by ARMS, our novel semantic definitions of precision and recall are not sensitive to flaws that appear more than once in the plan traces since the flaws are corrected only once in the learned models instead of at every intermediate state of the plan traces.

5.2. Semantic evaluation with classical planning

The compilation scheme presented in section 4.4 is extensible to address the evaluation task $\Lambda = \langle \mathcal{M}, \mathcal{T} \rangle$ defined in section 5.1. In this extended task, $\mathcal{M}$ represents a set of previously learned action models; therefore, rather than learning the action models from scratch, we simply edit $\mathcal{M}$ until it satisfies the given test set of plan traces $\mathcal{T}$. A solution to the classical planning task resulting from the extended compilation is a plan that:

1. **Edits the action models $\mathcal{M}$ 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 the two *edit operations* defined above, *deletion* and *insertion*.
2. **Validates the edited model $\mathcal{M}'$ in the observed plan traces**. The solution plan continues with a postfix that validates the edited model $\mathcal{M}'$ on the given observations $\mathcal{T}$, as explained in Section 4.4 for the models that are programmed from scratch.

Given $\Lambda = \langle \mathcal{M}, \mathcal{T} \rangle$, the output of the extended compilation is a 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 $pre_p(\xi)/del_p(\xi)/add_p(\xi)$, $p \in \Psi_\xi$, 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 action model and the actions for *validating* an observation are defined exactly as in the previous compilation. The only difference here is that the *programming actions* now implement the two editing operations (i.e., they also include the actions for *deleting* a precondition or negative/positive effect from an action model).

```
00 :  (insert_add_stack_handempty)
01 :  (insert_add_stack_clear_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 and validating the action model `stack` in which the positive effects `(handempty)` and `(clear ?v1)` are missing.

Figure 6 shows the plan for editing the action model of the operator `stack` of the *blocksworld* domain where only the two positive effects `(handempty)` and `(clear ?v1)` are missing. In this case the edited action model is again validated in the plan trace shown in Figure 2.

Assuming we are using an optimal planner to solve P'_Λ, the solution plan of this problem will induce the *closest compliant set of action models* $\mathcal{M}^*$. Therefore, our compilation enables the straightforward computation of the semantic versions of *precision* and *recall*. An argument can be made, however, that solving optimally P'_Λ may turn the evaluation process very time consuming. Considering this, *sem-Precision* and *sem-Recall* can be approximated if P'_Λ is solved with a satisfying planner. In this case, no guarantees can be made that the edited models will be the closest compliant ones, but a classical planner will always try to minimize the solution plan length and hence the number of edit operations applied to the input models.

6. Experimental results

This section evaluates different aspects of FAMA that range, from CPU-time performance to the quality of the learned models. Whenever applicable, we will consider both NP-complete and PSPACE-complete scenarios to draw conclusions at both levels of worst case complexity. We will also devote one experiment to show the suitability of the brand new evaluation metrics introduced in this paper.

6.1. Setup

We evaluate FAMA on fourteen IPC domains that satisfy the STRIPS requirement [26], all taken from the PLANNING.DOMAINS repository [37]. Table 5 compiles the features of the domains, all these features affect the size of our compilation. From left to right columns report, for each domain, its number of actions, number of predicates, maximum arity of the actions, and maximum arity of the predicates.

Now we explain the details of our experimental setup:

- **Plan traces**. For each domain, we generated 10 plan traces (with 10 actions/intermediate states each) using random walks. Depending on the experiment, these traces are used for training or testing purposes (more details on this are provided at the particular experiment).
- **Planner**. The classical planner we use to solve the P_Λ instances that result from our compilations is MADAGASCAR [38]. We use MADAGASCAR for two reasons:
  1. Its ability to deal with planning instances populated with dead-ends [39].
  2. It can leverage the horizon of the solution plan, when it is known in advance. When the length of plan traces is known (e.g. for FO action sequences or FO state trajectories) the horizon of the solution plan is also known. SAT-based planners can apply the actions for programming preconditions in parallel during a single planning step (and the same for actions programming effects) because these actions do not interact. Hence, we also know that the programming prefix of a solution plan can be solved in two steps, independently of the number of programming actions applied.
- **Hardware**. All experiments are run on an Intel Core i5 3.50 GHz x 4 with 16 GB of RAM.

- **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.

	Domain features			
	# actions	**# predicates**	**max action arity**	**max predicate arity**
Blocks	4	5	2	2
Driverlog	6	5	4	2
Ferry	3	5	2	2
Floortile	7	10	4	2
Grid	5	9	4	2
Gripper	3	4	3	2
Hanoi	1	3	3	2
Miconic	4	6	2	2
Npuzzle	1	3	3	2
Parking	4	5	3	2
Satellite	5	8	4	2
Transport	3	5	5	2
Visitall	1	3	2	2
Zenotravel	5	4	5	2

Table 5: Feature description of the domains used in the experiments.

6.2. Impact of the size of the input knowledge

This experiment evaluates how the size of the input knowledge affects the performance of FAMA to:

1. Identify the minimal amount of input knowledge required by FAMA to learn *sound* and *complete* models,
2. Evaluate the scalability of FAMA with respect to the size of the input knowledge.

The experiment analyzes the evolution of the CPU-time, as well as the *precision* and *recall* of the learned models wrt the GTM, as $|\mathcal{T}|$ increases from 1 to 10 (each trace in $\mathcal{T}$ comprises 10 actions/intermediate states). To keep the experiment practicable, we introduce a 1000s timeout after which the learning process is killed and 0 scores, for both *precision* and *recall*, are given to the learned model. We defined two case studies:

- **FO action sequence and PO state trajectory**: This is the typical case addressed by most of previous work, which corresponds to NP-complete scenarios. In this experiment we assume a degree of state observability of just 10%, meaning that each literal of a state has a 10% chance of being observed.

- **NO action sequence and NO state trajectory**: A PSPACE-complete scenario where both the input action sequence and state trajectory are *empty*. Only the initial and final states are observed, i.e., $\tau = \langle s_0, s_m \rangle, \forall \tau \in \mathcal{T}$.

Figures 7 and 8 show the quality of the models and computation time, respectively, for the NP-complete scenario. In Figure 7 we see precision stabilizes at 0.86 after 5 traces while recall stabilizes at 0.95 after 3 traces. These results show that FAMA does, in fact, not need big amounts of input to learn sound and complete models as opposite to other approaches in the literature where models are learned using a hundred of traces (see Table 3).

With regards to the scalability of FAMA, Figure 8 paints an exponential increase in computation time. Until an input size of 4 traces the computation time is below 1 sec, but it reaches 153 secs when the input is composed of 10 traces. These results match the expected performance of MADAGASCAR, since this planner is known to struggle with plan horizons beyond 150-200 steps (in our case 160 steps corresponds to 8 traces).

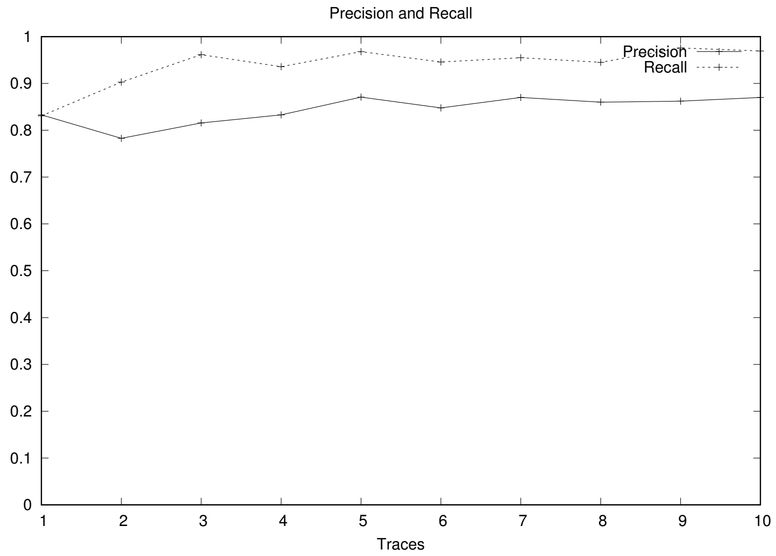

Figure 7: *Precision* and *recall* when learning from [1-10] plan traces with FO action sequences and PO state trajectories with 10% observability.

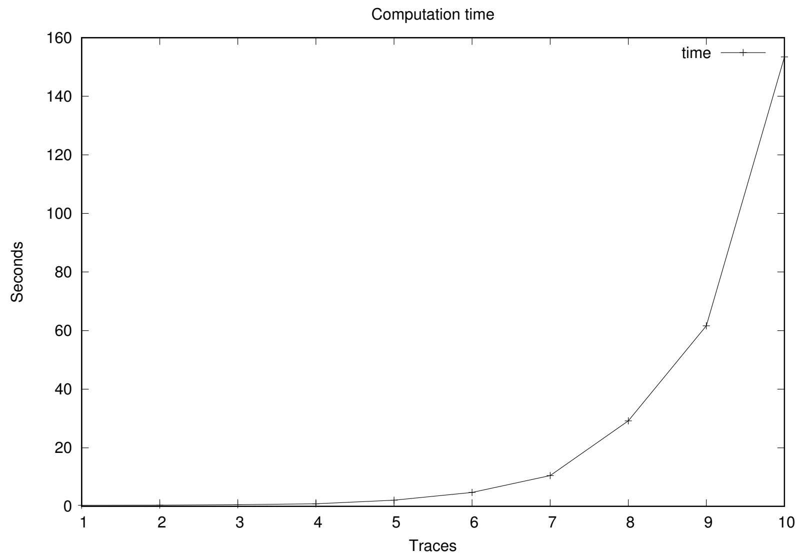

Figure 8: Computation time when learning from [1-10] plan traces with FO action sequences and PO state trajectories with 10% observability.

Figure 9 reports *precision* and *recall* for the models learned by our approach in the PSPACE-complete scenario (NO action sequence and NO state trajectory). The *precision* and *recall* values are now lower and they drop as the amount of input knowledge increases. For the PSPACE-complete scenario, *precision* and *recall* scores have to be taken with a grain of salt because **pure syntax-based metrics are not informative when reformulations appear**. This is the case of such an under-constrained learning task and we provide experimental evidence of this in Section 6.4. Further, the score drop reflects an increasing number of timeouts meaning that, no solution is found for several domains within the given time-bound (the worst case complexity of this scenario is larger). Computation time (Figure 10) is also higher explained again by this larger number of timeouts.

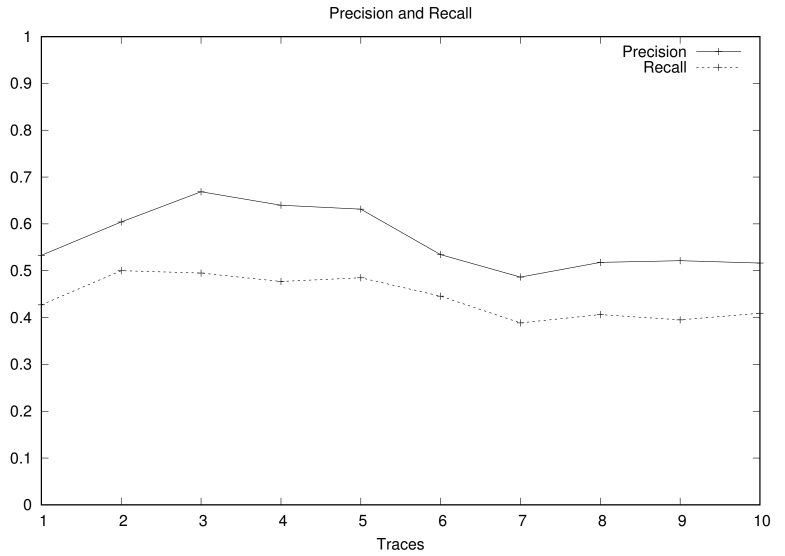

Figure 9: *Precision* and *recall* when learning from [1-10] plan traces with NO action sequences and NO state trajectories.

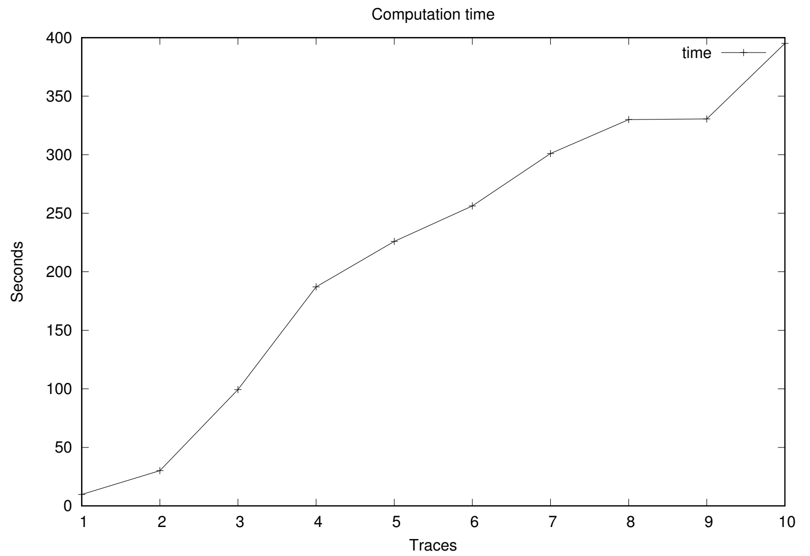

Figure 10: Computation time when learning from [1-10] plan traces with NO action sequences and NO state trajectories.

6.3. Comparison with ARMS

Now we compare the performance of FAMA to that of ARMS. ARMS, as any other previous approaches for learning planning models cannot handle the PSPACE-complete scenario. On the other hand, ARMS assumes input plan traces with FO action sequences and NO state trajectories so here, we will also adopt this assumption.

We define a *degree of observability* σ for the state trajectory, ranging from 0% to 100%, that measures the probability of observing a literal, and evaluate both approaches as σ increases using 5 plan traces as input. When $\sigma = 0$ we have a NO state trajectory, when $\sigma = 100$ we have a FO state trajectory and all cases in-between correspond to the PO scenario. Figures 11 and 12 compare FAMA and ARMS in terms of *precision* and *recall*. Horizontal axes represents the degree of observability, while vertical axes show the average *precision* (Figure 11) or *recall* (Figure 12). FAMA dominates in terms of *precision* in all cases (the models learned by our approach are 15% to 24% more precise than those learned by ARMS) except for FO state trajectories. A similar trend is drawn for *recall* (Figure 12), where the difference is even larger, meaning that our learned models are more complete. These results evidence the FAMA capacity to learn better models when few plan traces are available but do not imply that our approach is overall better for NP-complete scenarios (Figure 8 showed that FAMA cannot handle so much input as ARMS).

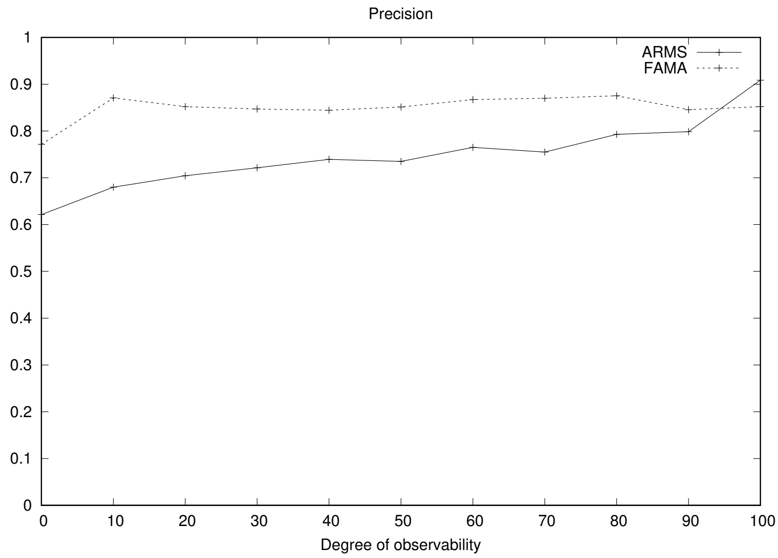

Figure 11: *Precision* comparison between FAMA and ARMS for different *degrees of observability*.

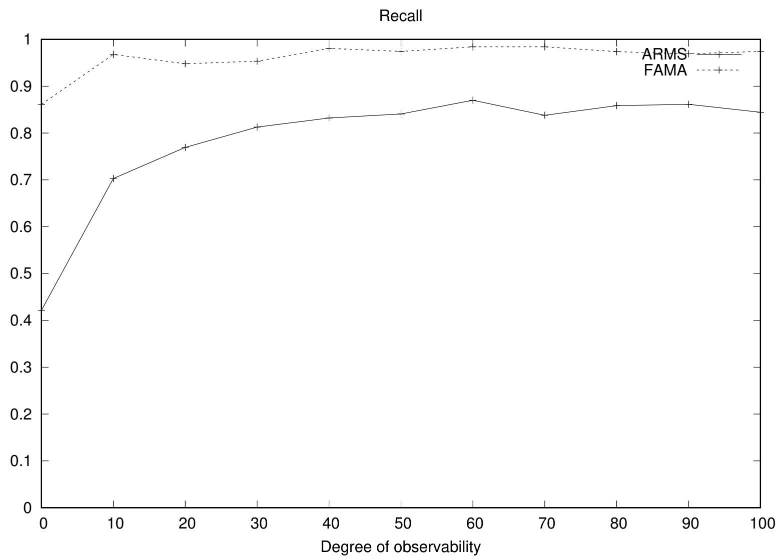

Figure 12: *Recall* comparison between FAMA and ARMS for different *degrees of observability*.

6.4. Learning with minimal input knowledge

Let us take a closer look at the action models learned from minimal input knowledge. To that end, we will limit the input to only 2 plan traces and analyze the results under different degrees of observability. In this experiment we evaluate three cases of study, the first one being an NP-complete scenario and the other two being PSPACE-complete:

- **FO action sequence and PO state trajectory**: We are, once again, assuming a degree of observability of 10% for the state trajectory. Results of this case study are detailed in Table 6.
- **PO action sequence and PO state trajectory**: In this case study (results shown in Table 7) we are assuming a degree of observability of 30% for both the action sequence and state trajectory.
- **NO action sequence and NO state trajectory**: Both the action sequence and state trajectory are completely empty so only the initial and final states are observed, i.e., $\tau = \langle s_0, s_m \rangle, \forall \tau \in \mathcal{T}$. Results of this case study are reported in Table 8.

All tables in this section (Tables 6, 7 and 8) follow the same structure. *Precision* (**P**) and *Recall* (**R**) scores are computed separately for the preconditions (**Pre**), positive effects (**Add**) and negative Effects (**Del**), and also globally (**Global**). The last column reports the computation time (in seconds) needed to obtain the learned models. Missing values in the tables (reported as -) correspond to domains where no solution was found given a 1800 seconds timeout.

Table 6 shows the results of the first case study. Recall scores are generally higher than the precision ones, and, in fact, the models learned for six out of the fourteen domains were perfectly complete. Although precision is overall lower, it is interesting that most learned sets of negative effects are flawless. With regards to the computation time, we find times below one second in all cases except for the *floor-tile* domain.

	Pre		Add		Del		Global		
	P	R	P	R	P	R	P	R	Time
blocks	0.86	0.67	0.86	0.67	0.8	0.44	0.84	0.59	0.21
driverlog	0.68	0.93	0.5	0.57	0.83	0.71	0.67	0.79	0.27
ferry	0.78	1.0	0.5	1.0	1.0	1.0	0.71	1.0	0.4
floor-tile	0.69	1.0	0.45	0.91	1.0	0.82	0.65	0.93	1.42
grid	0.76	0.94	0.55	0.86	1.0	1.0	0.74	0.94	0.7
gripper-strips	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.16
hanoi	0.67	1.0	0.67	1.0	1.0	1.0	0.73	1.0	0.31
miconic	1.0	1.0	0.5	1.0	1.0	1.0	0.8	1.0	0.29
n-puzzle	0.75	1.0	1.0	1.0	1.0	1.0	0.88	1.0	0.28
parking	0.73	0.79	0.7	0.78	1.0	0.78	0.78	0.78	0.65
satellite	0.93	1.0	0.63	1.0	1.0	0.75	0.85	0.96	0.36
transport	1.0	1.0	0.71	1.0	1.0	0.6	0.9	0.9	0.22
grid-visit-all	0.67	1.0	0.5	1.0	1.0	1.0	0.63	1.0	0.87
zeno-travel	0.92	0.79	0.6	0.86	0.8	0.57	0.78	0.75	0.26
	0.82	0.94	0.66	0.90	0.96	0.83	0.78	0.90	0.46

Table 6: *Precision* and *recall* scores for learning tasks with FO action sequences and PO state trajectories with 10% observability.

In Table 7, which collects the result of the PSPACE case study with 30% observability, the scores for some domains are missing. This is the case for *floor-tile*, and *parking* both considered *puzzle-like* domains and that put a strain on the planners. Interestingly enough, *hanoi* also qualifies as a *puzzle-like* domain, and we can see how this is reflected on the computation time. With regards to the learned models, we find that they retain similar level of soundness, while their completeness is lower than in the previous case study. This is specially the case for the preconditions, which have gone from 0.94 recall to 0.54. The explanation for these scores is that input actions are strong constraints that play a key role on how close the learned model are with respect to the GTM. The more these actions are missing the more present is reformulation and pure syntax-based metrics become less relevant.

The effect of missing input actions is more evident in our last case study with NO action sequences and state trajectories (Table 8). In this case we find solutions for all domains. Search is even less constrained so there are far more possible solutions for this learning task. This wider range of solutions is also evidenced by the kind of the learned models, which in spite of being compliant with the input data, are further from the GTM. Both *precision* and *recall* decreased to 0.6 and 0.5, respectively. Syntax-based metrics are not appropriate for scenarios with minimal observability as they cannot cope with the *reformulations* that frequently occur in these circumstances.

To illustrate this, Figure 13 shows the PDDL encoding of the action model of the `stack` operator learned from plan traces with NO action sequences and state trajectories. This learned action model removes a block from on top of another block and puts it down on the table. There are two main differences with respect to the GTM: (1), the learned

	Pre		Add		Del		Global		
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**Time**
blocks	1.0	0.89	0.9	1.0	1.0	0.89	0.96	0.93	3.51
driverlog	0.3	0.21	0.31	0.57	0.29	0.29	0.3	0.32	24.41
ferry	0.83	0.71	0.8	1.0	1.0	1.0	0.87	0.87	4.39
floor-tile	-	-	-	-	-	-	-	-	-
grid	-	-	-	-	-	-	-	-	-
gripper-strips	1.0	0.67	0.8	1.0	1.0	1.0	0.92	0.86	1.31
hanoi	1.0	0.5	1.0	1.0	1.0	1.0	1.0	0.75	1566.44
miconic	0.75	0.33	0.5	0.75	0.5	0.67	0.57	0.5	0.86
n-puzzle	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	62.82
parking	-	-	-	-	-	-	-	-	-
satellite	0.78	0.5	0.57	0.8	0.33	0.25	0.63	0.52	22.76
transport	0.5	0.3	0.43	0.6	0.5	0.4	0.47	0.4	31.07
grid-visit-all	1.0	0.5	0.5	0.5	1.0	1.0	0.75	0.6	3.11
zeno-travel	0.67	0.29	0.75	0.43	0.75	0.43	0.71	0.36	494.2
	0.80	0.54	0.69	0.79	0.76	0.72	0.74	0.65	201.35

Table 7: *Precision* and *recall* when learning with PO action sequences and PO state trajectories, 30% observability in both cases.

action is unstacking a block instead of stacking it and (2), the block ends on the table after the action is applied, working as an `unstack&putdown` *macro-action*. We refer to the first difference as *role swapping* and it happens when there are missing actions in the plan trace. If no actions are present in the input traces, the name of actions becomes meanless (actions are effectively anonymous), meaning that they can interchange their behavior with any other comparable action model. The second issue happens when there are missing states in the input traces.

	Pre		Add		Del		Global		
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**Time**
blocks	0.5	0.56	0.5	0.33	0.75	0.33	0.55	0.41	0.27
driverlog	0.13	0.07	0.38	0.71	0.0	0.0	0.25	0.21	0.98
ferry	0.5	0.29	0.5	0.5	0.67	0.5	0.55	0.4	0.47
floor-tile	0.34	0.64	0.5	0.36	0.44	0.73	0.39	0.59	165.92
grid	0.47	0.41	0.38	0.43	0.25	0.29	0.39	0.39	214.87
gripper-strips	1.0	0.83	1.0	1.0	1.0	1.0	1.0	0.93	0.2
hanoi	0.6	0.75	1.0	1.0	1.0	1.0	0.78	0.88	3.23
miconic	0.4	0.44	0.6	0.75	0.25	0.33	0.42	0.5	0.25
n-puzzle	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	4.15
parking	0.67	0.57	0.43	0.33	0.5	0.44	0.56	0.47	7.61
satellite	0.5	0.14	0.5	0.6	0.75	0.75	0.57	0.35	2.06
transport	0.6	0.3	0.38	0.6	0.5	0.2	0.47	0.35	0.83
grid-visit-all	0.0	0.0	1.0	0.5	0.0	0.0	1.0	0.2	1.24
zeno-travel	0.86	0.43	0.29	0.29	0.33	0.14	0.53	0.32	28.4
	0.54	0.46	0.60	0.60	0.53	0.48	0.60	0.50	30.75

Table 8: *Precision* and *recall* scores for learning tasks with NO action sequences and NO state trajectories.

```
(:action stack
 :parameters (?o1 - object ?o2 - object)
 :precondition (and (on ?o1 ?o2)(handempty ))
 :effect (and (not (on ?o1 ?o2))(clear ?o1)(clear ?o2)(ontable ?o1)))
```

Figure 13: PDDL encoding of the learned action model of the *stack* operator from the four-operator *blocksworld* domain.

Reformulated action models, like the one in Figure 13, can be used to solve planning tasks. For instance, any blocks-world problem can be solved unstacking all the blocks to the table (unstack + put-down) and then stacking them to meet the goal conditions (pick-up + stack). The NO/NO case study meets all the conditions where reformulations happens, and this is the reason why scenarios such as this are better evaluated using *semantic-based metrics*.

6.5. Syntactic versus semantic evaluation

The last experiment compares the scores provided by the syntactic and semantic versions of *precision* and *recall*. For that purpose, we will evaluate the models learned in 6.4 both syntactically, using the GTM, and semantically, using a testing set of 5 plan traces. Our goal with this experiment is to gauge the suitability of the semantic metrics we propose with respect to their well-known counterparts. With that in mind, we define two case studies:

- **FO action sequence and PO state trajectory**: In this case study the full sequence of actions is known and no states are missing, which makes it practically impossible for reformulated models to appear. In fact, in all our experiments with FAMA and other approaches we never observed reformulations when the full sequence of actions is known.
- **NO action sequence and NO state trajectory**: This is a case study that favors reformulations in the learned models, as previously discussed.

	Precision	Recall	sem-Precision	sem-Recall
blocks	0.84	0.59	0.84	0.64
driverlog	0.67	0.79	0.70	0.92
ferry	0.71	1.0	1.0	1.0
floor-tile	0.65	0.93	0.95	0.95
grid	0.74	0.94	1.0	0.98
gripper-strips	1.0	1.0	1.0	1.0
hanoi	0.73	1.0	0.91	1.0
miconic	0.8	1.0	1.0	1.0
n-puzzle	0.88	1.0	1.0	1.0
parking	0.78	0.78	0.97	0.84
satellite	0.85	0.96	0.96	1.0
transport	0.9	0.9	1.0	1.0
grid-visit-all	0.63	1	1.0	1.0
zeno-travel	0.78	0.75	0.85	0.96
	0.78	0.9	0.94	0.95

Table 9: Syntactic and semantic scores when learning with FO action sequences and PO state trajectories with 10% observability.

Table 9 shows the results of the first case study. Looking at the high scores of the syntactic metrics, specially *recall*, we can conclude that the learned models are, in fact, fairly similar to the GTM. This supports our conclusion that no reformulation occurs in this case study, which also means that the space of possible solutions is restricted to models close to the GTM. The values for *sem-Precision* and *sem-Recall* are also high across the table, which is exactly the desired behavior for these metrics given that solutions are close to the GTM. In comparison, both versions of *recall* show similar scores, while the difference is bigger for the precision metrics. This means that the semantic version is more lenient towards extra preconditions or effects. This matches what we have observed during the experiments for instance, the learned model for the `move` action of the *hanoi* domain, specifies that both the origin and destination disks must be bigger than the one moving, but the GTM contains only one of these preconditions. This learned model is semantically correct but sintactically different from the GTM and hence, penalized by the *precision* metric.

Finally Table 10 details the results of the NO/NO case study. Contrary to the previous case study, the difference between the syntactic and semantic metrics is larger here. Comparing the scores of both versions, we find that learned models that achieved mediocre scores when using the GTM as reference, are in fact reasonable sound and complete, reaching and overall score of 0.91 for both *sem-Precision* and *sem-Recall*. This means that the learned models by our approach, despite syntactically different from the GTM, require very few editions to explain the testing traces given as input.

Looking at the results of both case studies we can draw two conclusions with regards to the semantic metrics proposed in this paper. The first one is that, when no reformulation occurs, these metrics behave similarly to their syntactic counterparts, which means they are a good substitute when the GTM is not available. The second conclusion is that *sem-Precision* and *sem-Recall* are better suited to evaluate reformulated models than the original syntactic metrics, since they contemplate valid solutions outside the GTM that successfully explain the given input data.

	Precision	Recall	sem-Precision	sem-Recall
blocks	0.55	0.41	0.9	0.86
driverlog	0.25	0.21	0.54	0.72
ferry	0.55	0.4	1.0	0.79
floor-tile	0.39	0.59	-	-
grid	0.39	0.39	-	-
gripper-strips	1.0	0.93	1.0	1.0
hanoi	0.78	0.88	0.89	1.0
miconic	0.42	0.5	0.89	0.85
n-puzzle	1.0	1.0	1.0	1.0
parking	0.56	0.47	-	-
satellite	0.57	0.35	-	-
transport	0.47	0.35	0.93	0.93
grid-visit-all	1.0	0.2	1.0	1.0
zeno-travel	0.53	0.32	-	-
	0.6	0.5	0.91	0.91

Table 10: Syntactic and semantic metric scores for learning tasks with NO action sequences and NO state trajectories.

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 that exploits logic reasoning to learn sound models from minimal input knowledge. Remarkably, the action models of the *gripper*, *npuzzle* or *visitall* domains were perfectly learned from only observations of the initial and final states.

To the best of our knowledge, this is the first work on learning and evaluating STRIPS action models from state observations, using exclusively an *off-the-shelf* classical planner, and tested over a wide range of different domains. 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, our approach can learn 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 [16]. 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.

When example plans are available, our approach is also able to compute accurate action models from small sets of learning examples. Our experimental results report that FAMA achieves *precision* and *recall* scores that are over 0.7 for ten out of the fourteen IPC domains, using only two plan traces (with ten actions each) and in little computation time (less than a second). Our experimental evaluation also shows that, in the particular setting of minimal input data, FAMA outperforms ARMS.

On the other hand, this work also introduced a semantic version of the *precision* and *recall* metrics. These two metrics, widely used in ML, separately measure the soundness and completeness of the learned models, which facilitate the identification of model flaws. Our brand new semantic version of *precision* and *recall* do not require a GTM and evaluate STRIPS action models with respect to a given set of observations. In addition, when no reformulation occurs, these new metrics behave similarly to their syntactic counterparts.

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 [41]. The success of recent algorithms for exploring planning tasks [42] motivates the development of novel techniques that enable to autonomously collect informative learning examples. The combination of such exploration techniques with our learning approach is an appealing research direction that opens up the door to the bootstrapping of planning action models.

Acknowledgment

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

References

[1] 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.
[2] Q. Yang, K. Wu, Y. Jiang, Learning action models from plan examples using weighted MAX-SAT, Artificial Intelligence 171 (2-3) (2007) 107–143.
[3] E. Amir, A. Chang, Learning partially observable deterministic action models, Journal of Artificial Intelligence Research 33 (2008) 349–402.
[4] 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.
[5] 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.
[6] 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.
[7] S. N. Cresswell, T. L. McCluskey, M. M. West, Acquiring planning domain models using LOCM, The Knowledge Engineering Review 28 (02) (2013) 195–213.
[8] H. H. Zhuo, T. A. Nguyen, S. Kambhampati, Refining incomplete planning domain models through plan traces, in: International Joint Conference on Artificial Intelligence, IJCAI-13, 2013, pp. 2451–2458.
[9] H. H. Zhuo, S. Kambhampati, Model-lite planning: Case-based vs. model-based approaches, Artificial Intelligence 246 (2017) 1–21.
[10] M. Fox, D. Long, D. Magazzeni, Explainable planning, in: First Workshop on Explainable AI (XAI), IJCAI-13, 2017.
[11] T. Chakraborti, S. Sreedharan, S. Kambhampati, Explicability versus explanations in human-aware planning, in: International Conference on Autonomous Agents and MultiAgent Systems, AAMAS-18, 2018, pp. 2180–2182.
[12] R. Sheh, Different XAI for different HRI, in: AAAI Fall Symposium Series: Artificial Intelligence for Human-Robot Interaction, 2017.
[13] M. Roberts, I. Monteath, R. Sheh, D. W. Aha, P. Jampathom1, K. Akins, E. Sydow, V. Shivashankar, C. Sammut, What *was* I planning to do?, in: Workshop on Explainable AI Planning (XAIP), ICAPS-18, 2018.
[14] 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.
[15] G. Konidaris, L. P. Kaelbling, T. Lozano-Pérez, From skills to symbols: Learning symbolic representations for abstract high-level planning, Journal of Artificial Intelligence Research 61 (2018) 215–289.
[16] M. Asai, A. Fukunaga, Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary, in: National Conference on Artificial Intelligence, AAAI-18, 2018.
[17] D. Aineto, S. Jiménez, E. Onaindia, Learning strips action models with classical planning, in: International Conference on Automated Planning and Scheduling, ICAPS-18, 2018, pp. 399–407.
[18] H. H. Zhuo, Crowdsourced action-model acquisition for planning, in: National Conference on Artificial Intelligence, AAAI-15, 2015, pp. 3439–3446.
[19] J. Davis, M. Goadrich, The relationship between precision-recall and ROC curves, in: International Conference on Machine learning, ACM, 2006, pp. 233–240.
[20] M. L. Ginsberg, D. E. Smith, Reasoning about action II: the qualification problem, Artificial Intelligence 35 (3) (1988) 311–342.
[21] A. Kulkarni, T. Chakraborti, Y. Zha, S. G. Vadlamudi, Y. Zhang, S. Kambhampati, Explicable robot planning as minimizing distance from expected behavior, CoRR abs/1611.05497.
[22] 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.
[23] J. Segovia, S. Jiménez, A. Jonsson, Generalized planning with procedural domain control knowledge, 2016.
[24] 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.
[25] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Generating context-free grammars using classical planning, in: International Joint Conference on Artificial Intelligence, ICAPS-17, 2017.
[26] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains., Journal of Artificial Intelligence Research 20 (2003) 61–124.
[27] 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.
[28] S. Cresswell, P. Gregory, Generalised domain model acquisition from action traces, in: International Conference on Automated Planning and Scheduling, ICAPS-11, 2011.
[29] 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.
[30] S. J. Russell, P. Norvig, Artificial intelligence: a modern approach, Pearson Education Limited,, 2016.
[31] 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.

[32] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL – The Planning Domain Definition Language (1998).
[33] J. Slaney, S. Thiébaux, Blocks world revisited, Artificial Intelligence 125 (1-2) (2001) 119–153.
[34] 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.
[35] D. Long, J. Dolejsi, M. Fox, Building support for pddl as a modelling tool, KEPS 2018 78.
[36] 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.
[37] C. Muise, Planning. domains, ICAPS system demonstration.
[38] J. Rintanen, Madagascar: Scalable planning with sat, Proceedings of the 8th International Planning Competition (IPC-2014).
[39] C. L. López, S. J. Celorrio, Á. G. Olaya, The deterministic part of the seventh international planning competition, Artificial Intelligence 223 (2015) 82–119.
[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] 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.
[42] 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 14: 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 15: 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 16: 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 17: Compiled PDDL problem file for the blocksworld domain.