Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 00 (2018) 1–38
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 FAMA, a novel approach for learning STRIPS action models from observations of plan executions that compiles the learning task into a classical planning task. Unlike all existing learning systems, FAMA is able to learn when the actions of the plan executions are partially or totally unobservable and information on intermediate states is partially provided. This flexibility makes FAMA an ideal learning approach in domains where only sensoring data are accessible. Additionally, we leverage the compilation scheme and extend it to come up with an evaluation method that allows us to assess the quality of a learned model syntactically, that is, with respect to the actual model; and, semantically, that is, with respect to a set of observations of plan executions. We also show that the extended compilation scheme can be used to lay the foundations of a framework for action model comparison. FAMA is exhaustively evaluated over a wide range of IPC domains and its performance is compared to ARMS, a state-of-the-art benchmark in action model learning.

Keywords: Action model learning, AI planning, Machine Learning

1. Introduction

There is common agreement in the planning community that the unavailability of an adequate 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 pioneer learning systems like ARMS [2], we have seen systems 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.

A system for learning planning action models receives as input observations of the agent's plan execution and outputs an approximation of the actions that embody the physics of the modeled domain. The primary underlying motivation for acquiring planning action models is to solve model-based planning tasks afterwards, but there exists as well a large variety of planning-related tasks that rely upon the existence of a planning model. Among these tasks we might cite: *goal and plan recognition* approaches based on a domain theory [10, 11, 12]; *transparent planning*, in which an agent implicitly communicates its true goal by making its intentions and its action selection *transparent* (recognizable) to observers [13]; or *deceptive path-planning*, which aims at finding a path such that the probability of an observer identifying the final destination is minimised [14]. Planning models are also used in *explainable AI planning* to form a common basis for communicating with users and facilitate the generation of transparent and explainable decisions [15] as well as explanations in terms of the differences with a human mental model [16]. *Counterplanning* requires also a model of the opponent agent in order to recognize its goals [17] and *model reconciliation* aims to conform the models of two agents with respect to an observation of a plan computed with one of the two models [18].

Motivated by recent advances on the use of classical planning for the generation of different types of planning models (regular automata, context-free grammars, finite-state machines) [19, 20, 21, 22], in this paper we claim

that a planning model is learnable even though an accurate representation of the agent's behavior 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 [23], under minimal observability.

Most learning approaches assume that the observation of the agent's plan execution (plan trace) encompasses the fully observed sequence of the executed actions; i.e, they assume all the actions performed by the agent are observable. This heavily restricts the applicability of the learning approaches to contexts where the behaviour of the agent is fully observable, which also commonly entails a human annotator that correctly labels the executed actions. On the other hand, learning approaches accept a varying degree of observability in the states traversed in the plan trace, ranging from fully observable to fully unobservable states (see section 2.2 for details). In contrast, FAMA allows for an incomplete or empty sequence of observable actions, and the minimum observability case acceptable by FAMA is when the algorithm is only fed with the initial and final state of a plan trace. Like many Machine Learning (ML) techniques, FAMA is able to operate with only input/output pairs of states (pairs of initial and final states) and an unknown (or 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 [24, 25].

FAMA is a new learning approach, based on AI planning technology, that automatically compiles the task of learning STRIPS actions into a classical planning task which is then solved with an off-the-shelf planner [26]. The construction of a STRIPS planning model starts out from a set of plan traces containing the observation of several plan executions. As mentioned above, a plan trace may comprise none of the actions executed by the agent but must include, at least, the initial state s_0 and final state s_f of the execution. The compilation scheme defines a planning task from the set of plan traces using a set of *building actions* that insert the preconditions and effects of a learned action, and a set of *validating actions* that validate the learned actions in the plan traces. Hence, a solution to this planning task is a plan that determines the preconditions and effects of the actions of a STRIPS planning model M. The rationale of this process draws its strength in that when the problem $\langle s_0, s_f \rangle$ is solved with the learned model M, a plan π is generated such that π will contain the observed actions of the plan trace, if any, and the states generated by the application of π to s_0 will encompass all the (possibly) partially observed states of the trace. In other words, the plan π generated when solving the problem $\langle s_0, s_f \rangle$ with M is *consistent* with the plan trace.

FAMA is thus a model-based approach that automatically builds its own planning model by logical inference from the input plan traces that contain the observations of the agent execution. This behaviour diverges from ML techniques, which aim to minimize an error function on the training data. Moreover, FAMA requires far less sample data (example plan traces) than typical ML algorithms, thus alleviating the dependency on the assumption that there are enough data for learning the action models [27].

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 proposes instead two novel semantic evaluation metrics that build upon two well-known ML metrics, *precision* and *recall* [28], to evaluate the learned action models with respect to observations of plan executions. Our semantic evaluation is generally more informative than counting the number of errors between two models and alleviates two important 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*, which is defined as the actual impossibility of listing all the preconditions required for a real world action to have its intended effects [29].

Our semantic evaluation method is built on the same compilation scheme for solving a learning task. In particular, FAMA also accepts an input initial action model M of the agent's behaviour, either complete or partially specified [8, 9], alongside the observation of the agent execution. In this case, FAMA returns a model M' that follows the input model M and is consistent with the observations. We designed an *edition* mechanism that serves to correct the input model to the output model, which in turn defines an assessment of the accuracy with which M explains the observations. Interpreting the edition measure as a distance-based concept between two models can also be exploitable in model reconciliation [30].

In summary, FAMA is a novel learning approach characterized by:

- Compiling the task of learning a STRIPS planning model into a planning task that is automatically built from a

set of input plan traces.

- The plan traces are correct (no noise is considered in the observations) but may be incomplete in the number of observed actions as well as in the number and contents of the observed states.
- The planning task resulting from the compilation comprises actions for programming the preconditions and effects of the STRIPS actions and actions for validating the learned STRIPS actions.
- A semantic evaluation proposal that enables to assess a learned model beyond a merely syntactic comparison to a reference model.

A first description of the FAMA compilation scheme already appeared in our previous conference paper [26]. This paper 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 minimum 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 15 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 formalizes the addressed learning task and Section 4 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 and define the classical planning model we aim to learn; secondly, we summarize the most relevant existing approaches to learn classical planning action models.

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). The complement of L is defined as $\neg L = \{\neg l : l \in L\}$. We use $\mathcal{L}(F)$ to denote the set of all literal sets on F, i.e. all partial assignments of values to fluents.

FAMA adopts the *open world assumption*; i.e., what is not known to be true in a state is unknown. Consequently, states will explicitly include positive literals (f) and negative literals ($\neg f$) such that literals that are not in a state are unknown or unobserved. Hence, a *state* s is a full assignment of values to fluents; i.e. $|s| = |F|$, so the size of the state space is $2^{|F|}$. Like in PDDL [31], 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 $a \in A$ has a set of preconditions $\mathsf{pre}(a) \in \mathcal{L}(F)$ and a set of effects $\mathsf{eff}(a) \in \mathcal{L}(F)$. An action $a \in A$ is applicable in a given state s iff $pre(a) \subseteq s$, i.e. if the literals $pre(a)$ hold in s. The result of executing an applicable action $a \in A$ in a state s is a new state $\theta(s, a) = \{s \setminus \neg\mathsf{eff}(a) \cup \mathsf{eff}(a)\}$. Note that subtracting the complement of $\mathsf{eff}(a)$ from s ensures that $\theta(s, a)$ remains a well-defined state with positive and negative literals. With this regard we say that:

- $\mathsf{eff}^+(a) \in \mathcal{L}(F)$ is the *positive effects* of a, the subset of action effects that assert a positive literal in the state resulting after the application of a
- $\mathsf{eff}^-(a) \in \mathcal{L}(F)$ is the *negative effects* of a, the subset of action effects that assert a negative literal in the state resulting after the application of a

Since we restrict our attention to learning STRIPS action models, we will assume the syntactic constraints imposed by STRIPS models, namely $\mathsf{eff}^-(a) \subseteq \mathsf{pre}(a)$, $\mathsf{eff}^-(a) \cap \mathsf{eff}^+(a) = \emptyset$ and $\mathsf{pre}(a) \cap \mathsf{eff}^+(a) = \emptyset$. Additionally, actions $a \in A$ are instantiated from given action schemas, as in PDDL.

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 partial observability in the plan traces. Let $\pi = \langle a_1, \ldots, a_n \rangle$ be the plan executed by an agent that induces the state trajectory $s = \langle s_0, s_1, \ldots, s_n \rangle$, and let $\tau = \langle s_0, \ldots, a_i, \ldots, s_j, \ldots, s_n \rangle$ be the plan trace observed from the plan execution. With regards to the observed states of τ, that we will refer to as τ_s, we identify two general cases of observability:

1. We say that τ_s is a fully-observable (FO) state trajectory if every observed intermediate state of τ_s is a full assignment of values to fluents, and there exists a single action that transitions from every state s_i to state s_{i+1} in τ_s; that is $\theta(s_i, \langle a \rangle) = s_{i+1}$. This case clearly states that $\tau_s = s$, meaning that $\forall s_i \in \tau_s$, s_i comprises all the literals of the corresponding state in the trajectory s of the plan π. Formally, $\forall i, 1 \leq i < n, |s_i| = |F|$.
2. We say that τ_s is a partially-observable (PO) state trajectory if at least one intermediate state of τ_s is a partial assignment of values to fluents. Formally, $\exists i, 1 \leq i < n, |s_i| < |F|$. This means that one or more literals are missing in the intermediate s_i, all of which may be missing. When all literals are missing, s_i is a *missing* or *empty state* ($s_i = \emptyset$).

The general definition of a PO state trajectory gives rise to two special cases:

1. When **all** of the $n - 1$ intermediate states of s are **missing** in τ_s, τ_s is a *non-observable* (NO) state trajectory. Formally, $\forall i, 1 \leq i < n, s_i = \emptyset$; i.e., $|s_i| = 0$.
2. When **none** of the $n - 1$ intermediate states of s are **missing** in τ_s, we will refer to τ_s as a PO* state trajectory. Formally, $\exists i, 1 \leq i < n, s_i \neq \emptyset$; i.e., $0 < |s_i| < |F|$.

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.

FAMA can also deal with partial observability in the observed actions of τ, that we will refer to as τ_a. We identify three levels of observability, from the greatest to the lowest:

1. When **all** of the actions of π appear in τ_a, we say that τ_a is a fully-observable (FO) action sequence; i.e., $\tau_a = \pi$. In this case, τ_a contains all the necessary actions to transit every state s_{i-1} to its corresponding successor state s_i, from s_0 to s_n. This is the type of input trace accepted by all the existing learning approaches (see section 2.2 for details).
2. When **some** of the actions of π appear in τ_a, we say that τ_a is a partially observable (PO) action sequence. In this case, at least one of the necessary actions of the plan π is missing in τ_a. Formally, $\exists i, 1 \leq i \leq n, a_i \in \pi \wedge a_i \notin \tau_a$.
3. When **none** of the actions of π appear in τ_a, we say that τ_a is a non-observable (NO) action sequence. Formally, $\forall i, 1 \leq i \leq n, a_i \in \pi \wedge a_i \notin \tau_a$. That is, $\tau_a = \emptyset$.

Plan traces can then be classified accordingly to the type of observed state trajectory (FO, PO*, PO or NO) and action sequence (FO, PO or NO).

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

Table 1: Classification of state trajectories accordingly to the observed information.

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 observability of the 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 used 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, say traces with PO state trajectories, will also enable learning 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*, two of them also allow the sequence of intermediate states to be empty (PO) and the majority accept NO state trajectories. Exceptionally, LOCM is the only approach capable of learning from a fully-empty state trajectory, with neither initial nor 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 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 [23] 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.

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

	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
LOUGA	NO states FO actions	STRIPS negative preconditions	Genetic algorithm
LOCM2	— 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

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 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

	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
LOUGA	cross-validation with a test set of plan traces	redundant effects differences wrt the test set	5 800 - 3200 actions (160 traces)
LOCM2	manual checking wrt GTM	—	—
FAMA	checking wrt GTM validation with a test set	precision and recall	15 20-50 actions

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

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**) [27] 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.

One of the latest entries to the family of action model learning algorithms is LOUGA. This system uses a genetic algorithm to learn the effects of actions. Each gene in the genome encodes whether a predicate is a positive effect, negative effect or none for a particular action, and the fitness of an individual is evaluated by reproducing the trace with the model encoded in the individual. After a solution for the effects is found, an ad-hoc algorithm is used to infer preconditions by finding those literals that are always present before the execution of an action. LOUGA evaluates the learned models via cross-validation using the same metrics they use in their fitness function. In more detail, they measure (1) redundant positive and negative effects, (2) preconditions not met, and (3) literals observed in the input trace but not in the corresponding execution of the plan with the learned model.

Finally, we present the Learning Object-Centred Models (**LOCM**), possibly the most distinctive learning system due to its ability of learning with minimal input knowledge. **LOCM** only requires the FO action sequence of the plan trace, without need for providing any information about the predicates or the state trajectory, not even the initial or final state [32, 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 [33].

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 [33]. It is worth noting the last contribution of the LOCM family, called **LOP** (LOCM with Optimized Plans), addresses the problem of inducing static predicates [34]. 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.

The distinctive feature of LOCM2 lies in the need to learn the state variables (fluents) because predicates are not either provided as input nor they are deducible from the plan trace as no state observability is allowed. In contrast, FAMA and the rest of approaches either assume the set of predicates are provided alongside the input traces or assume they are extractable from the observed states of the plan trace, in which case the plan trace must comprise at least a grounded sample of every predicate. Similarly, the syntax of an action header (the action name and its parameters) is either extractable from the action sequence of the plan trace or it must be explicitly provided.

3. Learning task

In this section we first define the addressed *learning task*. Subsequently, we examine the particularities of the learning task to the different types of input plan traces (according to the observed state trajectories and action sequences). The analysis will serve to justify the use of planning for solving the learning task as well as to highlight the principal distinctive features of our approach FAMA with respect to the related work reviewed in Section 2.2.

FAMA addresses the learning and evaluation of PDDL action models that follow the STRIPS requirement [35, 31]. 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 (i.e. $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ 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 [36] 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.

Our *learning task* consists in learning a classical domain model 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 **initial domain model** (set of 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 can be inferred from s_0.
  3. The header of an action model is either given by $\mathcal{M}$ or inferable 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 NO 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 Λ will contain the predicates Ψ as well as the headers of the actions models, either explicitly provided in $\mathcal{M}$ or deducible from τ. Figure 2 shows the learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ corresponding to the observation of the execution of the four-action plan $\pi = \langle (\texttt{unstack B A}), (\texttt{putdown B}), (\texttt{pickup A}), (\texttt{stack A B}) \rangle$ for inverting a two-block tower. In this example $\tau = \langle s_0, \texttt{(putdown B)}, \texttt{(stack A B)}, s_4 \rangle$. Therefore, τ contains a NO state trajectory because only the initial and final state are observed and the three intermediate states, s_1, s_2 and s_3, are missing; and a PO action sequence where actions a_2 and a_3 are observed while a_1 and a_4 are unknown. The initial domain model $\mathcal{M}$ only contains two of the four needed headers, but can be completed with the headers `(putdown ?v1)` and `(stack ?v1 ?v2)` inferred from τ.

A *solution* to a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ is a domain model $\mathcal{M}'$ that is consistent with the information of $\mathcal{M}$ and the observed plan trace τ. This means that the action sequence (plan) that solves the planning problem $\langle s_0, s_n \rangle$ with $\mathcal{M}'$ along with the state trajectory induced by this plan encompass the plan trace τ.

Our 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* domain model $\mathcal{M}'$ must be consistent with the input model $\mathcal{M}$ and every observed plan trace $\tau \in \mathcal{T}$.

3.1. On the use of planning for learning action models

Since τ is a (partial) observation of the execution of π, actions, fluents or states traversed by π may be missing in τ (we will use τ_s and τ_a to refer to the observed states and observed actions of τ, respectively). We distinguish two well differentiated cases:

```
;;;;;; 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 $\tau = \langle s_0, \texttt{(putdown B)}, \texttt{(stack A B)}, s_4 \rangle$

- τ **determines the length of** π. This happens in three different scenarios (1), when τ_a is a FO action sequence (we know the actions of π) (2), when τ_s is a FO state trajectory (the states induced by the actions of π are fully known) or (3), when τ is a PO* state trajectory (we have at least one fluent for every state induced by π, in which case we know there is a single action that transitions from every state of τ_s).

  (1) The common assumption of having FO action sequences in a learning task, as is the case of all the learning approaches presented in section 2.2 except for FAMA, is unrealistic in many domains as it commonly 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 [25].

  (2) The assumption of having FO state trajectories means that the sensors are able to capture every state change at every instant, which is also typically unrealistic. Normally, the process of 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.

  (3) The assumption of having PO* state trajectories seems more appropriate to reflect sensor readings but still requires that at least one fluent of every state traversed by π is captured by the sensors.

  In these three scenarios the length of π is given by τ, the learning task is **SAT compilable**, and it is known that a Boolean satisfiability problem is a NP-complete task [37]. This is the reason why SAT solvers are commonly used in the approaches presented in section 2.2.

- τ **does not identify the length of** π. This happens when τ_a and τ_s are both partially observed (PO) or non-observable (NO) action/state trajectories. In this case, we are unaware of the number of actions of π and the number of states induced by π. This gives rise to a completely different scenario and a more challenging learning task that brings one key difference: the transition between two given observed states of τ 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 unbounded, and so **the horizon of** π **is no longer known**. This justifies the use of **planning techniques** for solving the learning task, which can now be interpreted as *filling the gap* between two observable points of τ. As it is also well known, the worst case complexity of STRIPS planning is PSPACE-complete so the learning task we address in this paper is PSPACE-complete instead of NP-complete.

In this particular scenario, the actual number of plans that are consistent with the input plan trace is also unbounded and grows exponentially with the actual length of the plans (that is now unknown). Therefore, when we assume partial observability in both actions and states, a learning approach must consider that the length of the observed plan traces is not an indication of the actual length of the plan, which motivates and justifies the use of planning, as our proposal of compiling the learning task to a classical planning problem.

4. Learning action models from plan executions with classical planning

Our proposal to address a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ is to transform Λ into a planning task P_Λ. The intuition behind the compilation is that when P_Λ is solved with a planner, the solution plan π_Λ is a sequence of actions that build the action models of the output domain model $\mathcal{M}'$ and verify that $\mathcal{M}'$ complies with the actions and states of the observed plan trace $\tau = \langle s_0, \ldots, s_n \rangle$. Hence, π_Λ will comprise two differentiated blocks of actions: a first set of actions each defining the **insertion** of a fluent as a precondition, a positive effect or a negative effect of an action model $\xi \in \mathcal{M}'$; and a second set of actions that determine the **application** of the learned ξs while successively **validating** the effects of the action application in every observable point of τ, including that the final reached state comprises s_n. Roughly speaking, in the *blocksworld* domain, the format of the first set of actions of π_Λ will look like `(insert_pre_stack_holding_v1)`, `(insert_eff_stack_clear_v1)`, `(insert_eff_stack_clear_v2)`, where the first effect denotes a positive effect and the second one a negative effect to be inserted in $name(\xi)$ =`stack`; and the format of the second set of actions of π_Λ will be like `(apply_unstack blockB blockA)`, `(apply_putdown blockB)` and `(validate_1)`, `(validate_2)`, where the last two actions denote the points at which the data generated through the action application must be validated with the data of τ.

The specification of P_Λ requires a propositional encoding of the components of the action models ξs, which is explained in the following section. The compilation approach is fully detailed in section 4.2 and section 4.3 presents some theoretical properties of the compilation scheme.

4.1. A propositional encoding for STRIPS action models

In this section we formalize a propositional encoding of an STRIPS action model ξ. This encoding is at the core of the FAMA compilation approach for addressing the learning task defined in section 3.

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 bounded 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)}. 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 the model fluents of $pre(\xi)$, $del(\xi)$ and $add(\xi)$. For every ξ and $p \in \Psi_\xi$, we create:

- $pre_p(\xi)$: model fluent formed by the combination of the prefixes `pre` and $name(\xi)$ plus a fluent of arity 0 that results from appending the elements of p (e.g. `pre_stack_on_v1_v2`, for $name(\xi)$ = `stack` and p = `on`(v_1, v_2))
- $del_p(\xi)$: model fluent formed by the combination of the prefixes `del` and $name(\xi)$ plus a fluent of arity 0 that results from appending the elements of p (e.g. `del_stack_on_v1_v2`)
- $add_p(\xi)$: model fluent formed by the combination of the prefixes `add` and $name(\xi)$ 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 model fluents that represents the propositional encoding of the preconditions, negative effects and positive effects of the action model corresponding to 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*.

4.2. Compilation

Our compilation scheme builds upon the approach presented in [26] but FAMA comes up with a more general and flexible scheme able to capture any type of input plan trace.

A learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$ is compiled into a planning task P_Λ with conditional effects in the context of a planning frame $\Phi = \langle F, A \rangle$. We use conditional effects because they allow us to compactly define actions whose effects depend on the current state. 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; that is, $triggered(s, a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E$. The result of applying a in state s follows the same definition of successor state, $\theta(s, a)$, introduced in section 2.1 but applied to the conditional effects in $triggered(s, a)$.

A solution plan π_Λ to P_Λ induces the output domain model $\mathcal{M}'$ that solves the learning task Λ. Specifically, a solution plan π_Λ serves two purposes:

1. **To build the action models of $\mathcal{M}'$.** π_Λ comprises a first block of actions (plan *prefix*) that set the predicates $p \in \Psi_\xi$ of $pre(\xi)$, $del(\xi)$ and $add(\xi)$ for each $\xi \in \mathcal{M}$.
2. **To validate the action models of $\mathcal{M}'$.** π_Λ also comprises a second block of actions (plan *postfix*) which is aimed at validating of the observed plan trace τ with the built action models $\mathcal{M}'$.

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 extends F with the model fluents to represent the preconditions and effects of each $\xi \in \mathcal{M}$ as well as some other fluents to keep track of the validation of τ. Specifically, F_Λ contains:
  - The set of fluents obtained from s_0; i.e., F.
  - The model 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 4.1
  - 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 literals at_n and $test_m$. When these two goals are achieved by the solution plan π_Λ, we will be certain that the action models of $\mathcal{M}'$ are validated in all the actions and states observed in the input plan trace τ.

- A_Λ includes three types of actions that give rise to the actions of π_Λ.

1. Actions for *inserting* a component (precondition, positive effect or negative effect) in $\xi \in \mathcal{M}$ following the syntactic constraints of STRIPS models. These actions will form the prefix of the solution plan π_Λ. Among the *inserting* actions, we find:
   – Actions which support the addition of a *precondition* $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$. A precondition p is inserted in ξ when neither pre_p, del_p nor add_p exist in ξ.

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

   – Actions which support the addition of a *negative* or *positive* effect $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$. A positive effect is inserted in ξ under the same conditions of a precondition insertion, and a negative effect is inserted in ξ when neither del_p nor add_p appear in ξ but pre_p does.

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

   For instance, given $name(\xi)$ =`stack` and $C_{pre-stack}$ = {`(pre_stack_holding_v1)`, `(pre_stack_holding_v2)`, `(pre_stack_on_v1_v2)`, `(pre_stack_clear_v1)`, `(pre_stack_clear_v1)`,...}, the insertion of each item $c \in C_{pre-stack}$ in ξ will generate a different alternative in the search space when solving P_Λ as long as $c \notin pre(\xi)$, $c \notin add(\xi)$ and $c \notin del(\xi)$. The same applies to effects with respect to sets $C_{add-stack}$ and $C_{del-stack}$ that would include all fluents starting with prefix `add` and `del`, respectively.

   Note that executing an insert action, e.g. `(insert_pre_stack_holding_v1)`, will add the corresponding model fluent `(pre_stack_holding_v1)` to the successor state. Hence, the execution of the insert actions of π_Λ yield a state containing the valuation of the model fluents that shape every $\xi \in \mathcal{M}$. For example, executing the insert actions that shape the action model $name(\xi)$ =`putdown` leads to a state containing the positive literals `(pre_putdown_holding_v1)`, `(eff_putdown_holding_v1)`, `(eff_putdown_clear_v1)`, `(eff_putdown_ontable_v1)`, `(eff_putdown_handempty)`.

2. Actions for *applying* the action models $\xi \in \mathcal{M}$ built by the insert actions and bounded to objects $\omega \subseteq \Omega^{ar(\xi)}$. Since action headers are known, the variables $pars(\xi)$ are bounded to the objects in ω that appear in the same position.

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

These actions will be part of the postfix of the plan π_Λ and they determine the application of the learned action models according to the values of the model fluents in the current state configuration. Figure 4 shows the PDDL encoding of `(apply_stack)` for applying the action model of the *stack* operator. Let's assume the action `(apply_stack blockB blockA)` is in π_Λ. Executing this action in a state s implies activating the preconditions and effects of `(apply_stack)` according to the values of the model fluents in s. For example, if $\{$`(pre_stack_holding_v1)`, `(pre_stack_clear_v2)`$\} \subset s$ then it must be checked that positive literals `(holding blockB)` and `(clear blockA)` hold in s. Otherwise, a different set of precondition literals will be checked. The same applies to the conditional effects, generating the corresponding literals according to the values of the model fluents of s.
Note that executing an apply action, e.g. `(apply_stack blockB blockA)`, will add the literals `(on blockB blockA)`, `(clear blockB)`, `(not (clear blockA))`, `(handempty)` and `(not (clear blockB))` to the successor state if $name(\xi)$ =`stack` has been correctly programmed by the insert actions. Hence, while **insert actions** add the values of the **model fluents** that shape ξ, the **apply actions** add the values of the **fluents of** F that result from the execution of ξ.

```
(: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 they appear 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 observable data of the input plan trace τ follows after the execution of the apply actions.

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

There will be a validate action in π_Λ per observed state in τ. The position of the validate actions in π_Λ will be determined by the planner checking that the state resulting after the execution of an apply action comprises the observed state $s_j \in \tau$.

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 insert actions, $\mathsf{insertPre_{p,\xi}}$ and $\mathsf{insertEff_{p,\xi}}$, 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 $s^t_m \in \tau_t$, $1 \leq t \leq k$ of a plan trace τ_t reset the current state and the current plan. These actions are now redefined as:

$$\begin{aligned}\mathsf{pre}(\mathsf{validate_j}) =& s^t_m \cup \{test_{j-1}\} \cup \{\neg mode_{prog}\},\\ \mathsf{cond}(\mathsf{validate_j}) =& \{\emptyset\} \rhd \{\neg test_{j-1}, test_j\} \cup\\ & \{\neg f\}_{\forall f \in s^t_m, f \notin s^{t+1}_0} \cup \{f\}_{\forall f \in s^{t+1}_0, f \notin s^t_m},\\ & \{\neg f\}_{\forall f \in F_{\pi_t}} \cup \{f\}_{\forall f \in F_{\pi_{t+1}}}.\end{aligned}$$

Finally, we will detail the composition of a solution plan π_Λ to a planning task P_Λ and the mechanism to extract the action models of $\mathcal{M}'$ 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, the plan shows the insert actions and validate action for the action model `stack` using the input plan trace of Figure 2. Plan steps 00 – 01 insert the preconditions of the `stack` model, steps 02 – 06 insert the action model effects, and steps 07 – 11 form the plan postfix that applies the action models (only the `stack` model is learned) and validates the result in the plan trace of Figure 2.

```
00 : (insert_pre_stack_holding_v1)
01 : (insert_pre_stack_clear_v2)
02 : (insert_eff_stack_clear_v1)
03 : (insert_eff_stack_clear_v2)
04 : (insert_eff_stack_handempty)
05 : (insert_eff_stack_holding_v1)
06 : (insert_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:

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

The logical inference process our approach is based on has trouble learning preconditions that are not, simultaneously, negative effects as well, since in this case 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 [38]. 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.

4.3. 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 [39]) 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. VAL, however, requires a full plan and 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 motivate the uso of two standard syntactic metrics (*precision* and *recall*). Then, section 5.1 will define these metrics for planning models, and section 5.2 will introduce a semantic evaluation measure that builds upon *precision* and *recall*. Finally, section 5.3 explains how FAMA computes our novel semantic-based metrics.

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 [28]. These two syntactic metrics are generally more informative than counting the number of errors between the learned action models and the GTM:

- $Precision = \frac{tp}{tp+fp}$, 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).

5.1. Syntactic-based precision and recall for planning models

The rationale behind the adaptation of precision and recall for planning models lies in counting the *edit operations* that need to be applied in domain model $\mathcal{M}$ to transform it into the reference model $\mathcal{M}'$. 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 domain model $\mathcal{M}$ into a new domain model $\mathcal{M}'$.

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

$$INS(\mathcal{M}, \mathcal{M}') = \bigcup_{\xi \in \mathcal{M}, \xi' \in \mathcal{M}' \text{ s.t. } name(\xi)=name(\xi')} PRE(\xi')\backslash PRE(\xi) \cup ADD(\xi')\backslash ADD(\xi) \cup DEL(\xi')\backslash DEL(\xi)$$

$$DEL(\mathcal{M}, \mathcal{M}') = \bigcup_{\xi \in \mathcal{M}, \xi' \in \mathcal{M}' \text{ s.t. } name(\xi)=name(\xi')} PRE(\xi)\backslash PRE(\xi') \cup ADD(\xi)\backslash ADD(\xi') \cup DEL(\xi)\backslash DEL(\xi')$$

With these ingredients in mind, we adapt the definitions of syntactic precision and recall to domain models. Let $\mathcal{M}$ be a domain model and let $\mathcal{M}'$ be the GTM. We know that $size(\mathcal{M}) = \sum_{\xi \in \mathcal{M}} |pre(\xi)| + |add(\xi)| + |del(\xi)|$ 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 preconditions and effects between $\mathcal{M}$ and the GTM among all prediconditions and effects 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 preconditions and effects between $\mathcal{M}$ and the GTM among all preconditions and effects of the GTM.*

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

Intuitively, precision gives a notion of *soundness* while recall gives a notion of the *completeness* of the learned models. We interpret a sound learned model as one in which all preconditions and effects are correct with respect to the GTM, and so there is no need to remove anything. A complete model is one in which no precondition or effect is missing. As an example, a precision of 0.5 means that only half of the predicates that make up the learned domain model are present in the GTM, while a recall of 0.5 means that only half of the predicates that make up the GTM are present in the learned domain model.

5.2. Semantic-based precision and recall for planning models

Pure syntax-based evaluation metrics 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. These metrics measure how well a model can reproduce a given plan execution, so the use of the word "semantic" here is in reference to the degree to which the learned model is able to capture the physics of the domain. Semantic metrics are appropriate for scenarios where:

1. The GTM is unknown. This is the most common scenario in ML, where models are both learned and evaluated with respect to datasets.
2. We are interested in measuring the ability of a model to explain a given plan trace, which is a good indicator of how the model will perform in actual planning tasks. 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.

The ARMS system was the first to show 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 [39] 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 novel semantic-based error measure that builds upon the *precision* and *recall* metrics, and instead, determines the modifications required by a learned model to explain the given 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 **learned domain model** obtained using any learning approach such as FAMA; in general, $\mathcal{M}$ can be any given input domain model even manually encoded.
- $\mathcal{T}$ is a set of plan traces used for **testing**.

A solution to this task is an **edited domain model** $\mathcal{M}'$ 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 domain model $\mathcal{M}$, and all the domain models $\mathcal{M}'$ able to explain the plan traces $\mathcal{T}$. The* **closest compliant domain model***, $\mathcal{M}^*$, is the comparable domain model 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 domain model $\mathcal{M}^*$ allows us to define a semantic version of *precision* and *recall* following definitions 4 and 5.

$$\textit{sem-Precision} = \frac{size(\mathcal{M}) - |DEL(\mathcal{M}, \mathcal{M}^*)|}{size(\mathcal{M})}$$

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

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 domain model $\mathcal{M}$ 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 in the action model that manifest 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.

5.3. Semantic evaluation with classical planning

The compilation scheme presented in section 4.2 is extensible to address the evaluation task $\Lambda = \langle \mathcal{M}, \mathcal{T} \rangle$ defined in section 5.2. In this extended task, $\mathcal{M}$ represents a previously learned domain model; 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 domain model $\mathcal{M}$ to build $\mathcal{M}'$**. A solution plan starts with a prefix that modifies the preconditions and effects of the action schemes in $\mathcal{M}$ using 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 4.2 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).

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.

```
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.

Assuming we are using an optimal planner to solve P'_Λ, the solution plan of this problem will induce the *closest compliant domain model* $\mathcal{M}^*$. 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 model will be the closest compliant one, but a classical planner will always try to minimize the solution plan length and hence the number of edit operations applied to the input model.

6. Experimental evaluation

This section presents several experiments to evaluate the performance of FAMA and the quality of the learned models. Whenever applicable, we will consider scenarios with both known and unknown plan horizon to draw conclusions at both levels of complexity. We also devote one experiment to show the suitability of the novel evaluation metrics introduced in section 5.

6.1. Setup

We evaluate FAMA on 15 IPC domains that satisfy the STRIPS requirement [31], all taken from the PLANNING.DOMAINS repository [40]. Table 4 presents the features of the tested domains that affect the size of the planning task P_Λ that results from the compilation. For each domain, the columns report, from left to right, the number of actions, the number of predicates, the maximum arity of the actions, and the maximum arity of the predicates.

The details of our experimental setup are the following:

- **Plan traces**. For each domain, we generated 10 plan traces, each with 10 actions and 10 intermediate states, using random walks. Depending on the experiment, the traces are used for training or testing purposes (more details on this issue are provided at the particular experiment).

- **Planner**. The classical planner we used to solve the instances of P_Λ that result from our compilations is MADAGASCAR [41]. We used MADAGASCAR for several reasons:

  1. Other planners such as FASTDOWNWARD were also tested but provided worse experimental results
  2. A SAT-based planner like MADAGASCAR is particularly suitable for tasks where the observed plan trace τ determines the length of the solution plan (case 1 presented in section 3). In this case, the prefix of the solution plan is solvable in two steps (planning horizon length 2) because the actions for inserting preconditions can be applied in parallel in a single step and the same for the actions inserting the effects.
  3. When the length of the plan is unknown (case 2 presented in section 3), the ability of MADAGASCAR to deal with instances populated with dead-ends is very helpful [42].

- **Hardware**. All experiments were 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
Rovers	9	25	6	3
Satellite	5	8	4	2
Transport	3	5	5	2
Visitall	1	3	2	2
Zenotravel	5	4	5	2

Table 4: Feature description of the domains used in the experiments.

6.2. *Impact of the size of the input knowledge*

This experiment evaluates the impact of $|\mathcal{T}|$, the size of the input knowledge, on the performance of FAMA in order 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 and the precision and recall of the learned models wrt the GTM as $|\mathcal{T}|$ increases from 1 to 10 plan traces. To keep the experiment practicable, we introduced a 1000s timeout, after which the learning process is killed and a score of 0 is given to both the precision and recall of the learned model. We defined two case studies:

- **FO action sequence and PO state trajectory**: This is the common case addressed by most of the state-of-the-art learning approaches, which corresponds to a scenario where the plan horizon is given by the action sequence. In this experiment we assume a degree of observability of only 10% for the state trajectory, meaning that each literal of a state has a 10% chance of being observed.

- **NO action sequence and NO state trajectory**: In this case study, both the input action sequence and state trajectory are *empty* and the length of the plan is unknown. 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 first case study. The values plotted in these figures are averages over the 15 domains. In Figure 7 we see that, after three traces, precision stabilizes at 0.84 whereas recall stabilizes at 0.95. These results show that FAMA does, in fact, not need big amounts of input knowledge to learn sound and complete models as opposite to other approaches in the literature where models are learned using around 100 traces (see Table 3).

Figure 8 displays the scalability of FAMA. Interestingly, we can observe an exponential increase in computation time for input sizes beyond five traces. Until an input size of four traces the computation time is below 1 sec but it reaches 166 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 since each trace has 10 actions and 10 intermediate states).

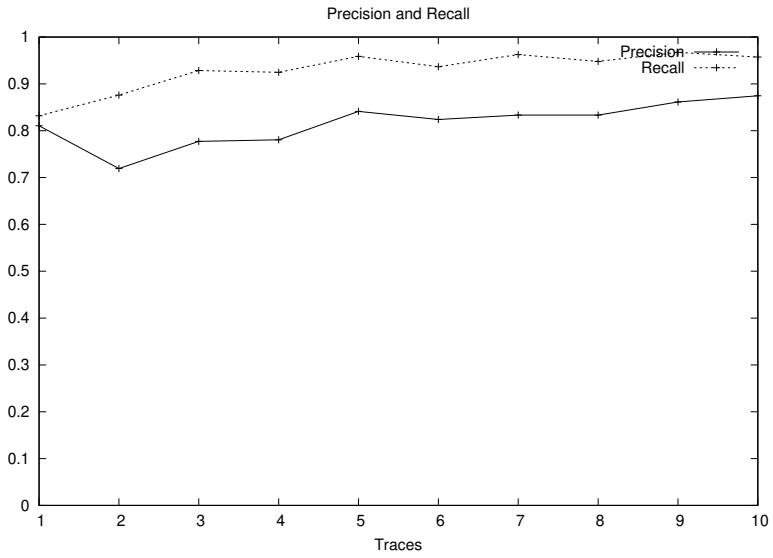

Figure 7: Precision and recall when learning from [1-10] plan traces with FO action sequences and PO state trajectories with 10% observability.

Figure 9 displays the average precision and recall of the 15 learned models in the scenario with unknown plan horizon. As expected, the quality of the learned models is lower than when the horizon of the plan is known. The higher complexity of this setting is also reflected in the appearance of some timeouts when solving the learning task. The first timeout is found at $|\mathcal{T}| = 3$ in the *grid* domain, and the number of timeouts increases until it reaches 6 with $|\mathcal{T}| = 10$. We can observe in Figure 9 the opposite behaviour to Figure 7; that is, we find a drop of the quality as the input knowledge increases. The drop in the score is caused by the increasing number of timeouts, meaning that no solution is found in many tasks within the given time-bound, and consequently a value of 0 for precision and recall is assigned to these experiments. Figure 10, on the other hand, reflects that the computation time of the first case study is also higher than in the second case, which is explained by both the higher complexity and the large number of timeouts.

The conclusions we draw from these experiments is that, when the plan horizon is known, learning with few input samples yield action models that contain 95% of the preconditions and effects of the GTM plus some extra ones as

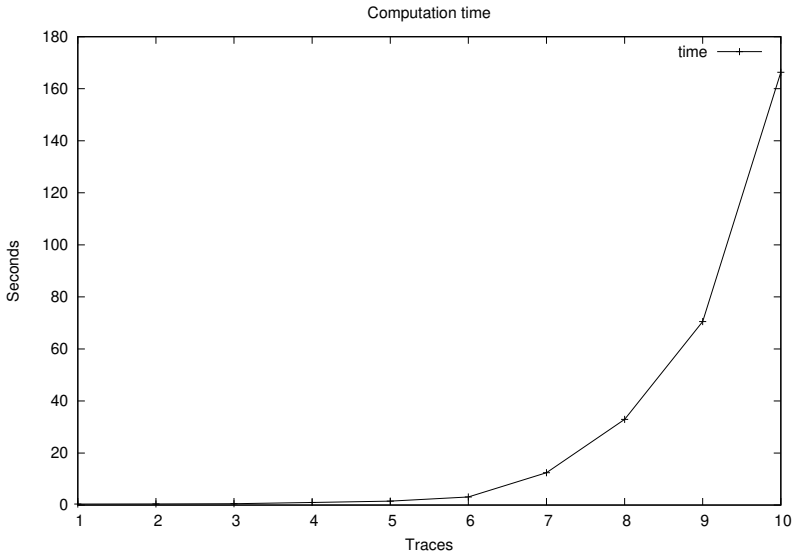

Figure 8: Computation time when learning from [1-10] plan traces with FO action sequences and PO state trajectories with 10% observability.

indicated by the values of precision and recall. With unkown plan horizons, on the other hand, the learned models are generally more different from the GTM; while timeouts are the main cause of the drop in the score, we must point out that pure syntax-based metrics are not adequate to evaluate such under-constrained tasks since the phenomenon of *reformulation* occurs and this largely impacts the results. We will provide experimental evidence of this in section 6.4.

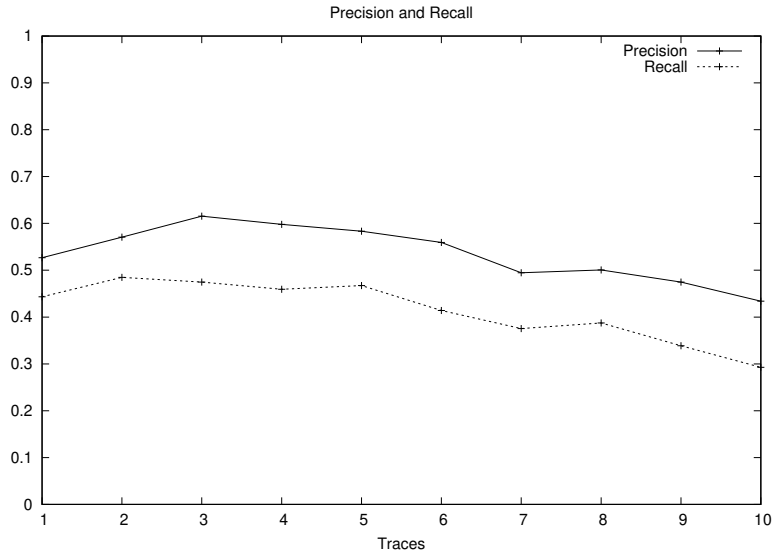

Figure 9: Precision and recall when learning from [1-10] plan traces with NO action sequences and NO state trajectories.

6.3. Comparison with ARMS

In this section we analyze the performance of FAMA compared to ARMS, one of the most well-known approaches to learning planning models. ARMS, as well as most of the existing current learning systems, works under the assump-

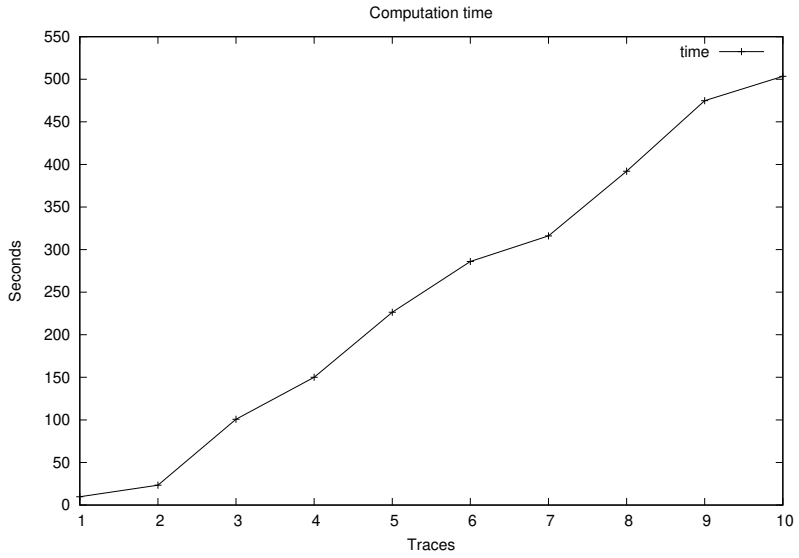

Figure 10: Computation time when learning from [1-10] plan traces with NO action sequences and NO state trajectories.

tion of plan traces with FO action sequences and NO state trajectories and therefore is not able to handle the scenarios where the plan horizon is unknown. We will thereby restrict the experimentation to the cases manageable by ARMS.

In this experiment, we defined a *degree of observability* σ for the state trajectory, ranging from 0% to 100%, that measures the probability of observing a literal, and evaluated both FAMA and ARMS for increasing values of σ using five traces as input knowledge. 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. The horizontal axes represent the degree of observability and vertical axes show the average precision (Figure 11) and recall (Figure 12) computed over the 15 tested domains. Remarkably, FAMA dominates in terms of precision in all cases except for the FO state trajectories. Particularly, the models learned by FAMA are between 13% to 34% more precise than those learned by ARMS. A similar trend is observed for recall (Figure 12), where the difference is even larger, meaning that our learned models are more complete.

The results highlight that FAMA outperforms ARMS when very few plan traces are available. This by no means is conclusive that FAMA is overall better in NP-complete scenarios but only that it is able to learn better with very limited input knowledge (actually, Figure 8 reflects the exponential behaviour of FAMA with more than five traces).

6.4. Learning with minimal input knowledge

In this section, we will take a closer look at the action models learned from minimal input knowledge. To that end, we will limit the input to only two plan traces and analyze the results under different degrees of observability. We evaluate three case studies:

- **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 5.
- **PO action sequence and PO state trajectory**: In this case study we are assuming a degree of observability of 30% for both the action sequence and state trajectory. Results are shown in Table 6.
- **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 7.

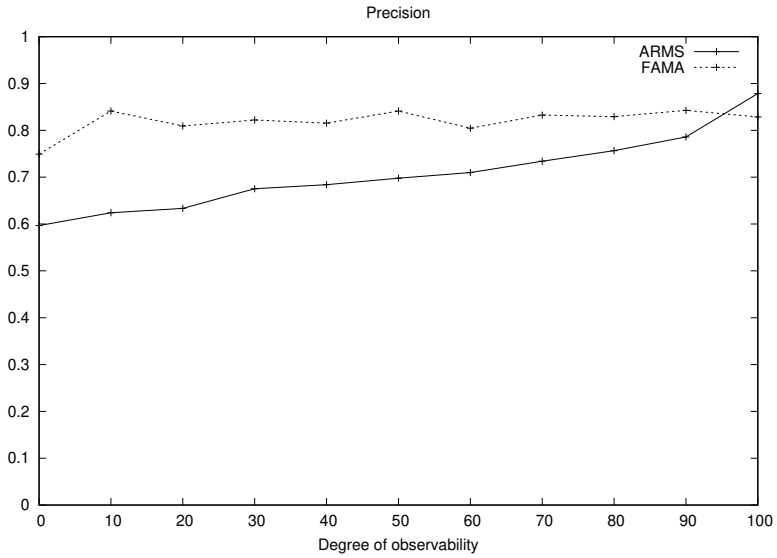

Figure 11: Precision comparison between FAMA and ARMS for different *degrees of observability*.

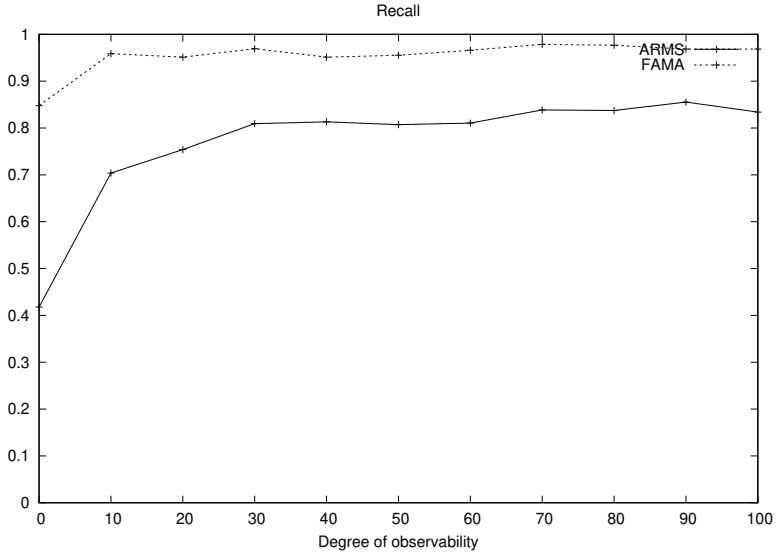

Figure 12: Recall comparison between FAMA and ARMS for different *degrees of observability*.

All tables in this section (Tables 5, 6 and 7) 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 within a 1800s timeout.

Table 5 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 15 domains were perfectly complete. Although precision is overall lower, it is interesting to notice that the learned sets of negative effects are mostly flawless. With regards to the computation time, we can observe times are below one second in most cases except for some of the more complex domains.

Table 6 gathers the results of the case study with 30% observability. We can see in the table that the scores of some domains are missing. This is the case of *floor-tile* and *grid*, which not only are fairly complex domains, but also

	Pre		Add		Del		Global		
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**Time**
Blocks	0.86	0.67	1.0	0.67	0.8	0.44	0.89	0.59	0.24
Driverlog	0.6	0.86	0.36	0.57	0.67	0.29	0.53	0.64	0.58
Ferry	0.7	1.0	0.36	1.0	1.0	1.0	0.6	1.0	0.37
Floortile	0.69	1.0	0.55	1.0	1.0	0.82	0.69	0.95	1.38
Grid	0.68	0.88	0.5	0.86	0.88	1.0	0.67	0.9	0.65
Gripper	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.18
Hanoi	0.67	1.0	1.0	1.0	1.0	1.0	0.8	1.0	0.36
Miconic	1.0	1.0	0.57	1.0	1.0	1.0	0.84	1.0	0.3
Npuzzle	0.75	1.0	1.0	1.0	1.0	1.0	0.88	1.0	0.26
Parking	0.78	1.0	0.69	1.0	1.0	1.0	0.8	1.0	0.24
Rovers	0.54	1.0	0.3	0.76	1.0	0.46	0.48	0.85	2.14
Satellite	0.93	1.0	0.56	1.0	1.0	0.75	0.81	0.96	0.4
Transport	0.83	1.0	0.5	1.0	0.6	0.6	0.67	0.9	0.19
Visitall	1.0	1.0	0.25	0.5	1.0	1.0	0.57	0.8	1.31
Zenotravel	0.9	0.64	0.5	0.71	0.83	0.71	0.73	0.68	0.25
	0.8	0.94	0.61	0.87	0.92	0.8	0.73	0.88	0.59

Table 5: Precision and recall scores for learning tasks with FO action sequences and PO state trajectories with 10% observability.

categorized as *puzzle-like* domains, a feature that is known for putting a strain in the planners. Interestingly enough, we note the high computation time of *hanoi* and *parking*, which also qualify as a *puzzle-like* domains. Regarding quality, we find that the learned models retain a level of soundness similar to Table 5 but the completeness is lower than in the previous case study. This is specially noticeable in the preconditions, where recall values drop from from 0.88 to 0.64. This is because the input actions act as strong constraints playing a key role on the closeness of the learned model to the GTM. The more actions are missing in the input knowledge, the more likely the occurrence of reformulations.

	Pre		Add		Del		Global		
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**Time**
Blocks	0.89	0.89	0.8	0.89	0.83	0.56	0.84	0.78	0.77
Driverlog	0.57	0.29	0.31	0.57	0.4	0.29	0.4	0.36	7.35
Ferry	0.83	0.71	0.36	1.0	1.0	1.0	0.62	0.87	3.38
Floortile	-	-	-	-	-	-	-	-	-
Grid	-	-	-	-	-	-	-	-	-
Gripper	1.0	1.0	0.8	1.0	1.0	1.0	0.93	1.0	0.17
Hanoi	0.67	0.5	1.0	1.0	1.0	1.0	0.86	0.75	132.69
Miconic	1.0	0.33	1.0	1.0	1.0	0.67	1.0	0.56	0.71
Npuzzle	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.86	13.48
Parking	0.83	0.36	1.0	0.89	0.83	0.56	0.9	0.56	160.64
Rover	0.43	0.73	0.24	0.47	0.56	0.38	0.39	0.61	31.13
Satellite	0.6	0.21	0.56	1.0	0.5	0.5	0.56	0.43	11.55
Transport	1.0	0.2	0.57	0.8	1.0	0.4	0.73	0.4	23.39
Visitall	0.67	1.0	0.5	0.5	1.0	1.0	0.67	0.8	3.13
Zenotravel	1.0	0.36	0.4	0.29	1.0	0.43	0.77	0.36	226.27
	0.81	0.56	0.66	0.8	0.86	0.68	0.74	0.64	47.28

Table 6: Precision and recall when learning with PO action sequences and PO state trajectories, 30% observability in both cases.

We now analyze the case study with NO action sequences and NO state trajectories (Table 7). A first outstanding observation is that, contrary to what might be expected by looking at the previous table, we are able in this case to find solutions for all the domains. This happens because the search is less constrained and consequently there are far more possible solutions for this learning task. This broader space of solutions is also stressed in a diminished quality of the learned models. Thus, despite the learned models being compliant with the input data, they are further from the original GTM. In Table 7 we can observe the global values of precision and recall drop to 0.57 and 0.48, respectively.

We argue, however, that 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 NO state trajectories. This learned action model removes a block from on top of another block and puts it down on the table in a single step. There are two main differences with respect to the model of the `stack` operator of the GTM: (1) the learned action is actually *unstacking* a block instead of stacking it and (2) the block on the top ends on the table, not held by the robot arm. We refer to the first difference as *role swapping* and it happens when there are missing actions in the input plan trace. If no actions are present in the input traces, the names of actions become meaningless, in which case the effectively anonymous actions can interchange their behaviour with any other comparable action model. The second difference indeed reveals that the learned action model is working as an `unstack+put-down` *macro-action*. This happens when there are missing states in the input traces since a *macro-action* can be seen as the application of more than one action in a single step, thus skipping some intermediate states.

Reformulated action models, like the one in Figure 13, are indeed sound models that 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`). Hence, the NO/NO case study features all the conditions for reformulation to happen, and this is the reason why scenarios such as this one are better evaluated using *semantic-based metrics*.

	Pre		**Add**		**Del**		**Global**		
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**Time**
Blocks	0.6	0.67	0.33	0.22	0.67	0.44	0.55	0.44	0.26
Driverlog	0.5	0.29	0.33	0.57	0.0	0.0	0.38	0.29	0.88
Ferry	0.5	0.43	0.5	0.25	0.67	0.5	0.55	0.4	0.45
Floortile	0.48	0.45	0.27	0.36	0.46	0.55	0.41	0.45	58.38
Grid	0.25	0.24	0.33	0.43	0.14	0.14	0.25	0.26	234.63
Gripper	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.86	0.16
Hanoi	0.6	0.75	1.0	1.0	1.0	1.0	0.78	0.88	6.32
Miconic	0.63	0.56	0.6	0.75	0.25	0.33	0.53	0.56	0.25
Npuzzle	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.52
Parking	0.57	0.29	0.2	0.11	0.8	0.44	0.53	0.28	17.43
Rovers	0.38	0.64	0.07	0.24	0.13	0.54	0.22	0.53	1.74
Satellite	0.86	0.43	0.43	0.6	0.75	0.75	0.67	0.52	3.15
Transport	0.29	0.2	0.38	0.6	0.67	0.4	0.39	0.35	1.06
Visitall	0.0	0.0	1.0	0.5	0.0	0.0	1.0	0.2	1.21
Zenotravel	0.4	0.29	0.25	0.29	0.2	0.14	0.3	0.25	17.48
	0.54	0.46	0.51	0.53	0.52	0.48	0.57	0.48	22.99

Table 7: 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.

6.5. Syntactic versus semantic evaluation

Our last experiment is devoted to compare the scores provided by the syntactic and semantic versions of precision and recall. For that purpose, we will evaluate the models learned in Section 6.4 both syntactically, using the GTM, and semantically, computing the set of action models closest to the learned models that is compliant with a testing set of five traces (see Definition 6). We must note that since we are using MADAGASCAR, a satisficing planner, the solution to the model evaluation may not be the closest compliant domain model, so the scores of sem-Precision and sem-Recall are approximate values. Our goal with this experiment is to gauge the suitability of the semantic metrics proposed in section 5 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 experimentation 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	0.89	0.64
Driverlog	0.53	0.64	0.71	0.83
Ferry	0.6	1.0	0.96	1.0
Floortile	0.69	0.95	0.97	0.95
Grid	0.67	0.9	0.95	0.98
Gripper	1.0	1.0	1.0	1.0
Hanoi	0.8	1.0	0.9	1.0
Miconic	0.84	1.0	1.0	1.0
Npuzzle	0.88	1.0	1.0	1.0
Parking	0.8	1.0	0.98	1.0
Rovers	0.31	0.72	0.94	0.99
Satellite	0.81	0.96	0.93	1.0
Transport	0.67	0.9	1.0	1.0
Visitall	0.57	1.0	1.0	1.0
Zenotravel	0.73	0.68	0.85	0.88
	0.72	0.88	0.94	0.95

Table 8: Syntactic and semantic scores when learning with FO action sequences and PO state trajectories with 10% observability.

Table 8 shows the results of the first case study. Looking at the high scores of the syntactic metrics, specially the value of 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 of sem-Precision and sem-Recall are also very high across the table, which is exactly the desired behavior for these metrics given that solutions are very close to the GTM. In comparison, recall and sem-Recall show similar scores, while sem-Precision is significantly higher than precision, thus showing that the sem-Precision is more lenient towards extra preconditions or effects. This is in line with the results of the previous experiments, where the common appearance of redundant or implicit preconditions in the learned models is penalized by the precision metric. We can interpret this phenomenon as a manifestation of the qualification problem [29]. For instance, the model learned 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 syntactically different from the GTM and hence penalized by the precision metric.

Table 9 details the results of the NO/NO case study. One first observation is the impossibility of applying a semantic evaluation in some of the most complex domains with five traces. Contrary to the previous case study, the difference between the syntactic and semantic metrics is larger in this scenario with unknown plan horizon. Comparing the scores of both versions, we find that learned models that achieved mediocre scores when using the GTM as reference (syntactic metrics), are in fact reasonably sound and complete, reaching overall scores of 0.92 and 0.89 in sem-Precision and sem-Recall. This is an indication that the models learned by our approach, despite syntactically different from the GTM, require very few editions to explain the testing set of traces.

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	0.44	0.77	0.77
Driverlog	0.38	0.29	0.86	0.86
Ferry	0.55	0.4	0.82	0.53
Floortile	0.41	0.45	-	-
Grid	0.25	0.26	-	-
Gripper	1.0	0.86	1.0	1.0
Hanoi	0.78	0.88	0.89	1.0
Miconic	0.53	0.56	1.0	0.89
Npuzzle	1.0	1.0	1.0	1.0
Parking	0.53	0.28	-	-
Rovers	0.22	0.53	-	-
Satellite	0.67	0.52	-	-
Transport	0.39	0.35	0.94	1.0
Visitall	1.0	0.2	1.0	1.0
Zenotravel	0.57	0.48	-	-
	0.57	0.48	0.92	0.89

Table 9: Syntactic and semantic metric scores for learning tasks with NO action sequences and NO state trajectories.

7. Conclusions

In this paper we have presented FAMA, an approach for learning STRIPS action models based on the compilation of the learning task into a planning task. The distinctive characteristic of FAMA over other state-of-the-art approaches is its ability to learn from minimal input knowledge and, more particularly, from minimal observability plan traces with no observed actions and no observed intermediate states. By relaxing the input constraints, FAMA opens up the way for action model learning to operate on real-world problems, as opposite to current approaches where the heavy input restrictions have limited their applicability to synthetic benchmarks. Our approach is thus very well suited for learning action models in data-based applications where the only observable information is a possibly incomplete sequence of partially observed states.

Besides its capacity of working with minimal observability, FAMA is also able to learn from very small amounts of input knowledge, a clear advantage in domains where obtaining enough training samples is difficult or costly. While the experimental evaluation shows in general an exponential increase of the computation time as the number of training traces augments, FAMA is able to learn action models more accurate than those of ARMS with very limited input knowledge. Unlike extensive-data ML approaches, our work explores an alternative research direction that exploits logic reasoning to learn sound models from minimal input observability.

This great flexibility of FAMA to different amounts of minimal input knowledge impacts the complexity of the planning tasks. When the length of the input plan traces is unknown, that is, we ignore the number of steps of the decision-making process of the agent, the planning task to be solved becomes a PSPACE-complete scenario. This is one of the reasons that justifies our compilation-to-planning scheme in contrast to the most extended SAT-solving scheme used in many existing learning systems. Additionally, our planning-based solving scheme allows us to leverage *off-the-shelf* classical planners and benefit from the multiple advances in classical planning research.

A key contribution of this work is the proposal of two novel metrics to semantically evaluate the learned action models. These two metrics build upon the well-known syntax-based metrics *precision* and *recall*. Our semantic evaluation mitigates the common issue known as *reformulation* that appears when training sets of minimal observability are used. Due to the lack of observable information, FAMA can learn semantically valid models that are syntactically different from the reference model. The application of the semantic-based precision and recall allows us to assess the validity of the learned models even in domains where a reference model is not available.

We highlight that the semantic evaluation of FAMA is done via the same compilation-to-planning scheme that we use to learn the action models. Given that the input of our learning task definition accepts an initial action model of the agent's behaviour, either complete or partially specified, this solving scheme is exploitable to computing a model that follows the initial model and is compliant with a test set of plan traces. In other words, FAMA is applicable in model reconciliation tasks by defining a distance metric that measures how close the two models are [30].

More importantly, FAMA is applicable not only in environments where the domain model is unknown but also in

environments where the executable actions are unknown as well. This ability broadens the range of application to goal reasoning tasks such as plan recognition under imperfect observability [12], planning for transparency [13] or counterplanning [17]. The application of FAMA to these tasks offers a plan-based solution where the assumption of a known domain model can be removed. In other words, FAMA opens up the way towards domain-free goal reasoning.

Finally, we would like to add a note on the open issue of generating *informative* plan traces for learning planning action models. Planning actions include preconditions that are only satisfied by specific sequences of actions which have low probability of being chosen by chance [43]. The success of recent algorithms for exploring planning tasks [44] 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 towards 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-12, 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. Ramírez, H. Geffner, Plan recognition as planning, in: International Joint conference on Artifical Intelligence, (IJCAI-09), AAAI Press, 2009, pp. 1778–1783.
[11] M. Ramírez, Plan recognition as planning, Ph.D. thesis, Universitat Pompeu Fabra (2012).
[12] 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.
[13] A. M. MacNally, N. Lipovetzky, M. Ramírez, A. R. Pearce, Action selection for transparent planning, in: International Conference on Autonomous Agents and MultiAgent Systems, (AAMAS-18), 2018, pp. 1327–1335.
[14] P. Masters, S. Sardiña, Deceptive path-planning, in: International Joint Conference on Artificial Intelligence, (IJCAI-17), 2017, pp. 4368–4375.
[15] M. Fox, D. Long, D. Magazzeni, Explainable planning, in: First Workshop on Explainable AI (XAI), IJCAI-13, 2017.
[16] 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.
[17] A. Pozanco, Y. E.-Martín, S. Fernández, D. Borrajo, Counterplanning using goal recognition and landmarks, in: International Joint Conference on Artificial Intelligence, (IJCAI-18), 2018, pp. 4808–4814.
[18] T. Chakraborti, S. Sreedharan, Y. Zhang, S. Kambhampati, Plan explanations as model reconciliation: Moving beyond explanation as soliloquy, in: International Joint Conference on Artificial Intelligence, (IJCAI-17), 2017, pp. 156–163.
[19] 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-09), AAAI Press, 2009.
[20] J. S. Aguas, S. J. Celorrio, A. Jonsson, Generalized Planning with Procedural Domain Control Knowledge, in: International Conference on Automated Planning and Scheduling, (ICAPS-16), AAAI Press, 2016, pp. 285–293.
[21] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Generating Context-Free Grammars using Classical Planning, in: International Joint Conference on Artificial Intelligence, (IJCAI-17), AAAI Press, 2017, pp. 4391–4397.
[22] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Computing hierarchical finite state controllers with classical planning, Journal of Artificial Intelligence Research 62 (2018) 755–797.
[23] 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.

[24] 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.
[25] M. Asai, A. Fukunaga, Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary, in: National Conference on Artificial Intelligence, AAAI-18, 2018.
[26] D. Aineto, S. Jiménez, E. Onaindia, Learning STRIPS action models with classical planning, in: International Conference on Automated Planning and Scheduling, (ICAPS-18), AAAI Press, 2018, pp. 399–407.
[27] H. H. Zhuo, Crowdsourced action-model acquisition for planning, in: National Conference on Artificial Intelligence, AAAI-15, 2015, pp. 3439–3446.
[28] J. Davis, M. Goadrich, The relationship between precision-recall and ROC curves, in: International Conference on Machine learning, ACM, 2006, pp. 233–240.
[29] M. L. Ginsberg, D. E. Smith, Reasoning about action II: the qualification problem, Artificial Intelligence 35 (3) (1988) 311–342.
[30] 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.
[31] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains., Journal of Artificial Intelligence Research 20 (2003) 61–124.
[32] 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), AAAI Press, 2009.
[33] S. Cresswell, P. Gregory, Generalised Domain Model Acquisition from Action Traces, in: International Conference on Automated Planning and Scheduling, ICAPS-11, AAAI Press, 2011.
[34] 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.
[35] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL – The Planning Domain Definition Language (1998).
[36] J. Slaney, S. Thiébaux, Blocks world revisited, Artificial Intelligence 125 (1-2) (2001) 119–153.
[37] S. A. Cook, The complexity of theorem-proving procedures, in: Proceedings of the third annual ACM symposium on Theory of computing, ACM, 1971, pp. 151–158.
[38] J. Kucera, R. Barták, LOUGA: learning planning operators using genetic algorithms, in: Pacific Rim Knowledge Acquisition Workshop, PKAW-18, 2018, pp. 124–138.
[39] 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.
[40] C. Muise, Planning.domains, ICAPS system demonstration.
[41] J. Rintanen, Madagascar: Scalable planning with SAT, in: International Planning Competition, (IPC-2014), 2014.
[42] C. L. López, S. J. Celorrio, Á. G. Olaya, The deterministic part of the seventh international planning competition, Artificial Intelligence 223 (2015) 82–119.
[43] 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, AAAI Press, 2004, pp. 191–199.
[44] 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

Compiled PDDL domain file for learning the blocksworld action models from two initial and final states.

```
(define (domain blocks)
 (:requirements :strips)
 (:types object - None )
 (:constants a -  object b -  object c -  object d -  object e -  object f -  object g -  object)
 (:predicates (add_clear_pick-up_var1) (add_clear_put-down_var1) (add_clear_stack_var1) (add_clear_stack_var2)
(add_clear_unstack_var1) (add_clear_unstack_var2) (add_handempty_pick-up) (add_handempty_put-down)
(add_handempty_stack) (add_handempty_unstack) (add_holding_pick-up_var1) (add_holding_put-down_var1)
(add_holding_stack_var1) (add_holding_stack_var2) (add_holding_unstack_var1) (add_holding_unstack_var2)
(add_on_stack_var1_var1) (add_on_stack_var1_var2) (add_on_stack_var2_var1) (add_on_stack_var2_var2)
(add_on_unstack_var1_var1) (add_on_unstack_var1_var2) (add_on_unstack_var2_var1) (add_on_unstack_var2_var2)
(add_ontable_pick-up_var1) (add_ontable_put-down_var1) (add_ontable_stack_var1) (add_ontable_stack_var2)
(add_ontable_unstack_var1) (add_ontable_unstack_var2) (clear ?o1 - object) (del_clear_pick-up_var1)
(del_clear_put-down_var1) (del_clear_stack_var1) (del_clear_stack_var2) (del_clear_unstack_var1)
(del_clear_unstack_var2) (del_handempty_pick-up) (del_handempty_put-down) (del_handempty_stack)
(del_handempty_unstack) (del_holding_pick-up_var1) (del_holding_put-down_var1) (del_holding_stack_var1)
(del_holding_stack_var2) (del_holding_unstack_var1) (del_holding_unstack_var2) (del_on_stack_var1_var1)
(del_on_stack_var1_var2) (del_on_stack_var2_var1) (del_on_stack_var2_var2) (del_on_unstack_var1_var1)
(del_on_unstack_var1_var2) (del_on_unstack_var2_var1) (del_on_unstack_var2_var2) (del_ontable_pick-up_var1)
(del_ontable_put-down_var1) (del_ontable_stack_var1) (del_ontable_stack_var2) (del_ontable_unstack_var1)
(del_ontable_unstack_var2) (handempty) (holding ?o1 - object) (modeProg) (on ?o1 - object ?o2 - object)
(ontable ?o1 - object) (pre_clear_pick-up_var1) (pre_clear_put-down_var1) (pre_clear_stack_var1)
(pre_clear_stack_var2) (pre_clear_unstack_var1) (pre_clear_unstack_var2) (pre_handempty_pick-up)
(pre_handempty_put-down) (pre_handempty_stack) (pre_handempty_unstack) (pre_holding_pick-up_var1)
(pre_holding_put-down_var1) (pre_holding_stack_var1) (pre_holding_stack_var2) (pre_holding_unstack_var1)
```

```
(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.

```
(pre_holding_unstack_var2) (pre_on_stack_var1_var1) (pre_on_stack_var1_var2) (pre_on_stack_var2_var1)
(pre_on_stack_var2_var2) (pre_on_unstack_var1_var1) (pre_on_unstack_var1_var2) (pre_on_unstack_var2_var1)
(pre_on_unstack_var2_var2) (pre_ontable_pick-up_var1) (pre_ontable_put-down_var1) (pre_ontable_stack_var1)
(pre_ontable_stack_var2) (pre_ontable_unstack_var1) (pre_ontable_unstack_var2) (test1) (test2) (test3))

 (:action pick-up
   :parameters (?o1 - object)
   :precondition (and (not (modeProg ))(or (not (pre_ontable_pick-up_var1 ))(ontable ?o1))
(or (not (pre_clear_pick-up_var1 ))(clear ?o1))(or (not (pre_handempty_pick-up ))(handempty ))
(or (not (pre_holding_pick-up_var1 ))(holding ?o1)))
   :effect (and (when (and (del_ontable_pick-up_var1 ))(not (ontable ?o1)))
(when (and (add_ontable_pick-up_var1 ))(ontable ?o1))(when (and (del_clear_pick-up_var1 ))(not (clear ?o1)))
(when (and (add_clear_pick-up_var1 ))(clear ?o1))(when (and (del_handempty_pick-up ))(not (handempty )))
(when (and (add_handempty_pick-up ))(handempty ))(when (and (del_holding_pick-up_var1 ))(not (holding ?o1)))
(when (and (add_holding_pick-up_var1 ))(holding ?o1))))

 (:action put-down
   :parameters (?o1 - object)
   :precondition (and (not (modeProg ))(or (not (pre_ontable_put-down_var1 ))(ontable ?o1))
(or (not (pre_clear_put-down_var1 ))(clear ?o1))(or (not (pre_handempty_put-down ))(handempty ))
(or (not (pre_holding_put-down_var1 ))(holding ?o1)))
   :effect (and (when (and (del_ontable_put-down_var1 ))(not (ontable ?o1)))
(when (and (add_ontable_put-down_var1 ))(ontable ?o1))(when (and (del_clear_put-down_var1 ))(not (clear ?o1)))
(when (and (add_clear_put-down_var1 ))(clear ?o1))(when (and (del_handempty_put-down ))(not (handempty )))
(when (and (add_handempty_put-down ))(handempty ))(when (and (del_holding_put-down_var1 ))(not (holding ?o1)))
(when (and (add_holding_put-down_var1 ))(holding ?o1))))

 (:action stack
   :parameters (?o1 - object ?o2 - object)
   :precondition (and (not (modeProg ))(or (not (pre_on_stack_var1_var1 ))(on ?o1 ?o1))
(or (not (pre_on_stack_var1_var2 ))(on ?o1 ?o2))(or (not (pre_on_stack_var2_var1 ))(on ?o2 ?o1))
(or (not (pre_on_stack_var2_var2 ))(on ?o2 ?o2))(or (not (pre_ontable_stack_var1 ))(ontable ?o1))
(or (not (pre_ontable_stack_var2 ))(ontable ?o2))(or (not (pre_clear_stack_var1 ))(clear ?o1))
(or (not (pre_clear_stack_var2 ))(clear ?o2))(or (not (pre_handempty_stack ))(handempty ))
(or (not (pre_holding_stack_var1 ))(holding ?o1))(or (not (pre_holding_stack_var2 ))(holding ?o2)))
   :effect (and (when (and (del_on_stack_var1_var1 ))(not (on ?o1 ?o1)))(when (and (add_on_stack_var1_var1 ))(on ?o1 ?o1))
(when (and (del_on_stack_var1_var2 ))(not (on ?o1 ?o2)))(when (and (add_on_stack_var1_var2 ))(on ?o1 ?o2))
(when (and (del_on_stack_var2_var1 ))(not (on ?o2 ?o1)))(when (and (add_on_stack_var2_var1 ))(on ?o2 ?o1))
(when (and (del_on_stack_var2_var2 ))(not (on ?o2 ?o2)))(when (and (add_on_stack_var2_var2 ))(on ?o2 ?o2))
(when (and (del_ontable_stack_var1 ))(not (ontable ?o1)))(when (and (add_ontable_stack_var1 ))(ontable ?o1))
(when (and (del_ontable_stack_var2 ))(not (ontable ?o2)))(when (and (add_ontable_stack_var2 ))(ontable ?o2))
(when (and (del_clear_stack_var1 ))(not (clear ?o1)))(when (and (add_clear_stack_var1 ))(clear ?o1))
(when (and (del_clear_stack_var2 ))(not (clear ?o2)))(when (and (add_clear_stack_var2 ))(clear ?o2))
(when (and (del_handempty_stack ))(not (handempty )))(when (and (add_handempty_stack ))(handempty ))
(when (and (del_holding_stack_var1 ))(not (holding ?o1)))(when (and (add_holding_stack_var1 ))(holding ?o1))
(when (and (del_holding_stack_var2 ))(not (holding ?o2)))(when (and (add_holding_stack_var2 ))(holding ?o2))))
```

```
 (:action unstack
   :parameters (?o1 - object ?o2 - object)
   :precondition (and (not (modeProg ))(or (not (pre_on_unstack_var1_var1 ))(on ?o1 ?o1))
(or (not (pre_on_unstack_var1_var2 ))(on ?o1 ?o2))(or (not (pre_on_unstack_var2_var1 ))(on ?o2 ?o1))
(or (not (pre_on_unstack_var2_var2 ))(on ?o2 ?o2))(or (not (pre_ontable_unstack_var1 ))(ontable ?o1))
(or (not (pre_ontable_unstack_var2 ))(ontable ?o2))(or (not (pre_clear_unstack_var1 ))(clear ?o1))
(or (not (pre_clear_unstack_var2 ))(clear ?o2))(or (not (pre_handempty_unstack ))(handempty ))
(or (not (pre_holding_unstack_var1 ))(holding ?o1))(or (not (pre_holding_unstack_var2 ))(holding ?o2)))
   :effect (and (when (and (del_on_unstack_var1_var1 ))(not (on ?o1 ?o1)))
(when (and (add_on_unstack_var1_var1 ))(on ?o1 ?o1))(when (and (del_on_unstack_var1_var2 ))(not (on ?o1 ?o2)))
(when (and (add_on_unstack_var1_var2 ))(on ?o1 ?o2))(when (and (del_on_unstack_var2_var1 ))(not (on ?o2 ?o1)))
(when (and (add_on_unstack_var2_var1 ))(on ?o2 ?o1))(when (and (del_on_unstack_var2_var2 ))(not (on ?o2 ?o2)))
(when (and (add_on_unstack_var2_var2 ))(on ?o2 ?o2))(when (and (del_ontable_unstack_var1 ))(not (ontable ?o1)))
(when (and (add_ontable_unstack_var1 ))(ontable ?o1))(when (and (del_ontable_unstack_var2 ))(not (ontable ?o2)))
(when (and (add_ontable_unstack_var2 ))(ontable ?o2))(when (and (del_clear_unstack_var1 ))(not (clear ?o1)))
(when (and (add_clear_unstack_var1 ))(clear ?o1))(when (and (del_clear_unstack_var2 ))(not (clear ?o2)))
(when (and (add_clear_unstack_var2 ))(clear ?o2))(when (and (del_handempty_unstack ))(not (handempty )))
(when (and (add_handempty_unstack ))(handempty ))(when (and (del_holding_unstack_var1 ))(not (holding ?o1)))
(when (and (add_holding_unstack_var1 ))(holding ?o1))(when (and (del_holding_unstack_var2 ))(not (holding ?o2)))
(when (and (add_holding_unstack_var2 ))(holding ?o2))))

 (:action program_pre_ontable_pick-up_var1
   :parameters ()
  :precondition (and (modeProg )(not (pre_ontable_pick-up_var1 ))(not (del_ontable_pick-up_var1 ))
  (not (add_ontable_pick-up_var1 )))
   :effect (and (pre_ontable_pick-up_var1 )))

 (:action program_eff_ontable_pick-up_var1
   :parameters ()
   :precondition (and (modeProg )(not (del_ontable_pick-up_var1 ))(not (add_ontable_pick-up_var1 )))
  :effect (and (when (pre_ontable_pick-up_var1 )(del_ontable_pick-up_var1 ))
  (when (not (pre_ontable_pick-up_var1 ))(add_ontable_pick-up_var1 ))))

 (:action program_pre_clear_pick-up_var1
   :parameters ()
  :precondition (and (modeProg )(not (pre_clear_pick-up_var1 ))(not (del_clear_pick-up_var1 ))
  (not (add_clear_pick-up_var1 )))
   :effect (and (pre_clear_pick-up_var1 )))

 (:action program_eff_clear_pick-up_var1
   :parameters ()
   :precondition (and (modeProg )(not (del_clear_pick-up_var1 ))(not (add_clear_pick-up_var1 )))
  :effect (and (when (pre_clear_pick-up_var1 )(del_clear_pick-up_var1 ))
  (when (not (pre_clear_pick-up_var1 ))(add_clear_pick-up_var1 ))))

 (:action program_pre_handempty_pick-up
   :parameters ()
  :precondition (and (modeProg )(not (pre_handempty_pick-up ))(not (del_handempty_pick-up ))
  (not (add_handempty_pick-up )))
   :effect (and (pre_handempty_pick-up )))

 (:action program_eff_handempty_pick-up
   :parameters ()
   :precondition (and (modeProg )(not (del_handempty_pick-up ))(not (add_handempty_pick-up )))
  :effect (and (when (pre_handempty_pick-up )(del_handempty_pick-up ))
  (when (not (pre_handempty_pick-up ))(add_handempty_pick-up ))))

 (:action program_pre_holding_pick-up_var1
   :parameters ()
  :precondition (and (modeProg )(not (pre_holding_pick-up_var1 ))(not (del_holding_pick-up_var1 ))
  (not (add_holding_pick-up_var1 )))
   :effect (and (pre_holding_pick-up_var1 )))

 (:action program_eff_holding_pick-up_var1
   :parameters ()
   :precondition (and (modeProg )(not (del_holding_pick-up_var1 ))(not (add_holding_pick-up_var1 )))
  :effect (and (when (pre_holding_pick-up_var1 )(del_holding_pick-up_var1 ))
  (when (not (pre_holding_pick-up_var1 ))(add_holding_pick-up_var1 ))))

 (:action program_pre_ontable_put-down_var1
   :parameters ()
  :precondition (and (modeProg )(not (pre_ontable_put-down_var1 ))(not (del_ontable_put-down_var1 ))
  (not (add_ontable_put-down_var1 )))
   :effect (and (pre_ontable_put-down_var1 )))

 (:action program_eff_ontable_put-down_var1
   :parameters ()
```

```
  :precondition (and (modeProg )(not (del_ontable_put-down_var1 ))(not (add_ontable_put-down_var1 )))
 :effect (and (when (pre_ontable_put-down_var1 )(del_ontable_put-down_var1 ))
 (when (not (pre_ontable_put-down_var1 ))(add_ontable_put-down_var1 ))))

(:action program_pre_clear_put-down_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_clear_put-down_var1 ))(not (del_clear_put-down_var1 ))
 (not (add_clear_put-down_var1 )))
  :effect (and (pre_clear_put-down_var1 )))

(:action program_eff_clear_put-down_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_clear_put-down_var1 ))(not (add_clear_put-down_var1 )))
 :effect (and (when (pre_clear_put-down_var1 )(del_clear_put-down_var1 ))
 (when (not (pre_clear_put-down_var1 ))(add_clear_put-down_var1 ))))

(:action program_pre_handempty_put-down
  :parameters ()
 :precondition (and (modeProg )(not (pre_handempty_put-down ))(not (del_handempty_put-down ))
 (not (add_handempty_put-down )))
  :effect (and (pre_handempty_put-down )))

(:action program_eff_handempty_put-down
  :parameters ()
  :precondition (and (modeProg )(not (del_handempty_put-down ))(not (add_handempty_put-down )))
 :effect (and (when (pre_handempty_put-down )(del_handempty_put-down ))
 (when (not (pre_handempty_put-down ))(add_handempty_put-down ))))

(:action program_pre_holding_put-down_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_holding_put-down_var1 ))(not (del_holding_put-down_var1 ))
 (not (add_holding_put-down_var1 )))
  :effect (and (pre_holding_put-down_var1 )))

(:action program_eff_holding_put-down_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_holding_put-down_var1 ))(not (add_holding_put-down_var1 )))
 :effect (and (when (pre_holding_put-down_var1 )(del_holding_put-down_var1 ))
 (when (not (pre_holding_put-down_var1 ))(add_holding_put-down_var1 ))))

(:action program_pre_on_stack_var1_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_on_stack_var1_var1 ))(not (del_on_stack_var1_var1 ))
 (not (add_on_stack_var1_var1 )))
  :effect (and (pre_on_stack_var1_var1 )))

(:action program_eff_on_stack_var1_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_on_stack_var1_var1 ))(not (add_on_stack_var1_var1 )))
 :effect (and (when (pre_on_stack_var1_var1 )(del_on_stack_var1_var1 ))
 (when (not (pre_on_stack_var1_var1 ))(add_on_stack_var1_var1 ))))

(:action program_pre_on_stack_var1_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_on_stack_var1_var2 ))(not (del_on_stack_var1_var2 ))
 (not (add_on_stack_var1_var2 )))
  :effect (and (pre_on_stack_var1_var2 )))

(:action program_eff_on_stack_var1_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_on_stack_var1_var2 ))(not (add_on_stack_var1_var2 )))
 :effect (and (when (pre_on_stack_var1_var2 )(del_on_stack_var1_var2 ))
 (when (not (pre_on_stack_var1_var2 ))(add_on_stack_var1_var2 ))))

(:action program_pre_on_stack_var2_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_on_stack_var2_var1 ))(not (del_on_stack_var2_var1 ))
 (not (add_on_stack_var2_var1 )))
  :effect (and (pre_on_stack_var2_var1 )))

(:action program_eff_on_stack_var2_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_on_stack_var2_var1 ))(not (add_on_stack_var2_var1 )))
 :effect (and (when (pre_on_stack_var2_var1 )(del_on_stack_var2_var1 ))
 (when (not (pre_on_stack_var2_var1 ))(add_on_stack_var2_var1 ))))

(:action program_pre_on_stack_var2_var2
```

```
  :parameters ()
 :precondition (and (modeProg )(not (pre_on_stack_var2_var2 ))(not (del_on_stack_var2_var2 ))
 (not (add_on_stack_var2_var2 )))
  :effect (and (pre_on_stack_var2_var2 )))

(:action program_eff_on_stack_var2_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_on_stack_var2_var2 ))(not (add_on_stack_var2_var2 )))
 :effect (and (when (pre_on_stack_var2_var2 )(del_on_stack_var2_var2 ))
 (when (not (pre_on_stack_var2_var2 ))(add_on_stack_var2_var2 ))))

(:action program_pre_ontable_stack_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_ontable_stack_var1 ))(not (del_ontable_stack_var1 ))
 (not (add_ontable_stack_var1 )))
  :effect (and (pre_ontable_stack_var1 )))

(:action program_eff_ontable_stack_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_ontable_stack_var1 ))(not (add_ontable_stack_var1 )))
 :effect (and (when (pre_ontable_stack_var1 )(del_ontable_stack_var1 ))
 (when (not (pre_ontable_stack_var1 ))(add_ontable_stack_var1 ))))

(:action program_pre_ontable_stack_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_ontable_stack_var2 ))(not (del_ontable_stack_var2 ))
 (not (add_ontable_stack_var2 )))
  :effect (and (pre_ontable_stack_var2 )))

(:action program_eff_ontable_stack_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_ontable_stack_var2 ))(not (add_ontable_stack_var2 )))
 :effect (and (when (pre_ontable_stack_var2 )(del_ontable_stack_var2 ))
 (when (not (pre_ontable_stack_var2 ))(add_ontable_stack_var2 ))))

(:action program_pre_clear_stack_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_clear_stack_var1 ))(not (del_clear_stack_var1 ))
 (not (add_clear_stack_var1 )))
  :effect (and (pre_clear_stack_var1 )))

(:action program_eff_clear_stack_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_clear_stack_var1 ))(not (add_clear_stack_var1 )))
 :effect (and (when (pre_clear_stack_var1 )(del_clear_stack_var1 ))
 (when (not (pre_clear_stack_var1 ))(add_clear_stack_var1 ))))

(:action program_pre_clear_stack_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_clear_stack_var2 ))(not (del_clear_stack_var2 ))
 (not (add_clear_stack_var2 )))
  :effect (and (pre_clear_stack_var2 )))

(:action program_eff_clear_stack_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_clear_stack_var2 ))(not (add_clear_stack_var2 )))
 :effect (and (when (pre_clear_stack_var2 )(del_clear_stack_var2 ))
 (when (not (pre_clear_stack_var2 ))(add_clear_stack_var2 ))))

(:action program_pre_handempty_stack
  :parameters ()
 :precondition (and (modeProg )(not (pre_handempty_stack ))(not (del_handempty_stack ))
 (not (add_handempty_stack )))
  :effect (and (pre_handempty_stack )))

(:action program_eff_handempty_stack
  :parameters ()
  :precondition (and (modeProg )(not (del_handempty_stack ))(not (add_handempty_stack )))
 :effect (and (when (pre_handempty_stack )(del_handempty_stack ))
 (when (not (pre_handempty_stack ))(add_handempty_stack ))))

(:action program_pre_holding_stack_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_holding_stack_var1 ))(not (del_holding_stack_var1 ))
 (not (add_holding_stack_var1 )))
  :effect (and (pre_holding_stack_var1 )))
```

```
(:action program_eff_holding_stack_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_holding_stack_var1 ))(not (add_holding_stack_var1 )))
 :effect (and (when (pre_holding_stack_var1 )(del_holding_stack_var1 ))
 (when (not (pre_holding_stack_var1 ))(add_holding_stack_var1 ))))

(:action program_pre_holding_stack_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_holding_stack_var2 ))(not (del_holding_stack_var2 ))
 (not (add_holding_stack_var2 )))
  :effect (and (pre_holding_stack_var2 )))

(:action program_eff_holding_stack_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_holding_stack_var2 ))(not (add_holding_stack_var2 )))
 :effect (and (when (pre_holding_stack_var2 )(del_holding_stack_var2 ))
 (when (not (pre_holding_stack_var2 ))(add_holding_stack_var2 ))))

(:action program_pre_on_unstack_var1_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_on_unstack_var1_var1 ))(not (del_on_unstack_var1_var1 ))
 (not (add_on_unstack_var1_var1 )))
  :effect (and (pre_on_unstack_var1_var1 )))

(:action program_eff_on_unstack_var1_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_on_unstack_var1_var1 ))(not (add_on_unstack_var1_var1 )))
 :effect (and (when (pre_on_unstack_var1_var1 )(del_on_unstack_var1_var1 ))
 (when (not (pre_on_unstack_var1_var1 ))(add_on_unstack_var1_var1 ))))

(:action program_pre_on_unstack_var1_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_on_unstack_var1_var2 ))(not (del_on_unstack_var1_var2 ))
 (not (add_on_unstack_var1_var2 )))
  :effect (and (pre_on_unstack_var1_var2 )))

(:action program_eff_on_unstack_var1_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_on_unstack_var1_var2 ))(not (add_on_unstack_var1_var2 )))
 :effect (and (when (pre_on_unstack_var1_var2 )(del_on_unstack_var1_var2 ))
 (when (not (pre_on_unstack_var1_var2 ))(add_on_unstack_var1_var2 ))))

(:action program_pre_on_unstack_var2_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_on_unstack_var2_var1 ))(not (del_on_unstack_var2_var1 ))
 (not (add_on_unstack_var2_var1 )))
  :effect (and (pre_on_unstack_var2_var1 )))

(:action program_eff_on_unstack_var2_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_on_unstack_var2_var1 ))(not (add_on_unstack_var2_var1 )))
 :effect (and (when (pre_on_unstack_var2_var1 )(del_on_unstack_var2_var1 ))
 (when (not (pre_on_unstack_var2_var1 ))(add_on_unstack_var2_var1 ))))

(:action program_pre_on_unstack_var2_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_on_unstack_var2_var2 ))(not (del_on_unstack_var2_var2 ))
 (not (add_on_unstack_var2_var2 )))
  :effect (and (pre_on_unstack_var2_var2 )))

(:action program_eff_on_unstack_var2_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_on_unstack_var2_var2 ))(not (add_on_unstack_var2_var2 )))
 :effect (and (when (pre_on_unstack_var2_var2 )(del_on_unstack_var2_var2 ))
 (when (not (pre_on_unstack_var2_var2 ))(add_on_unstack_var2_var2 ))))

(:action program_pre_ontable_unstack_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_ontable_unstack_var1 ))(not (del_ontable_unstack_var1 ))
 (not (add_ontable_unstack_var1 )))
  :effect (and (pre_ontable_unstack_var1 )))

(:action program_eff_ontable_unstack_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_ontable_unstack_var1 ))(not (add_ontable_unstack_var1 )))
 :effect (and (when (pre_ontable_unstack_var1 )(del_ontable_unstack_var1 ))
 (when (not (pre_ontable_unstack_var1 ))(add_ontable_unstack_var1 ))))
```

```
(:action program_pre_ontable_unstack_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_ontable_unstack_var2 ))(not (del_ontable_unstack_var2 ))
 (not (add_ontable_unstack_var2 )))
  :effect (and (pre_ontable_unstack_var2 )))

(:action program_eff_ontable_unstack_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_ontable_unstack_var2 ))(not (add_ontable_unstack_var2 )))
 :effect (and (when (pre_ontable_unstack_var2 )(del_ontable_unstack_var2 ))
 (when (not (pre_ontable_unstack_var2 ))(add_ontable_unstack_var2 ))))

(:action program_pre_clear_unstack_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_clear_unstack_var1 ))(not (del_clear_unstack_var1 ))
 (not (add_clear_unstack_var1 )))
  :effect (and (pre_clear_unstack_var1 )))

(:action program_eff_clear_unstack_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_clear_unstack_var1 ))(not (add_clear_unstack_var1 )))
 :effect (and (when (pre_clear_unstack_var1 )(del_clear_unstack_var1 ))
 (when (not (pre_clear_unstack_var1 ))(add_clear_unstack_var1 ))))

(:action program_pre_clear_unstack_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_clear_unstack_var2 ))(not (del_clear_unstack_var2 ))
 (not (add_clear_unstack_var2 )))
  :effect (and (pre_clear_unstack_var2 )))

(:action program_eff_clear_unstack_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_clear_unstack_var2 ))(not (add_clear_unstack_var2 )))
 :effect (and (when (pre_clear_unstack_var2 )(del_clear_unstack_var2 ))
 (when (not (pre_clear_unstack_var2 ))(add_clear_unstack_var2 ))))

(:action program_pre_handempty_unstack
  :parameters ()
 :precondition (and (modeProg )(not (pre_handempty_unstack ))(not (del_handempty_unstack ))
 (not (add_handempty_unstack )))
  :effect (and (pre_handempty_unstack )))

(:action program_eff_handempty_unstack
  :parameters ()
  :precondition (and (modeProg )(not (del_handempty_unstack ))(not (add_handempty_unstack )))
 :effect (and (when (pre_handempty_unstack )(del_handempty_unstack ))
 (when (not (pre_handempty_unstack ))(add_handempty_unstack ))))

(:action program_pre_holding_unstack_var1
  :parameters ()
 :precondition (and (modeProg )(not (pre_holding_unstack_var1 ))(not (del_holding_unstack_var1 ))
 (not (add_holding_unstack_var1 )))
  :effect (and (pre_holding_unstack_var1 )))

(:action program_eff_holding_unstack_var1
  :parameters ()
  :precondition (and (modeProg )(not (del_holding_unstack_var1 ))(not (add_holding_unstack_var1 )))
 :effect (and (when (pre_holding_unstack_var1 )(del_holding_unstack_var1 ))
 (when (not (pre_holding_unstack_var1 ))(add_holding_unstack_var1 ))))

(:action program_pre_holding_unstack_var2
  :parameters ()
 :precondition (and (modeProg )(not (pre_holding_unstack_var2 ))(not (del_holding_unstack_var2 ))
 (not (add_holding_unstack_var2 )))
  :effect (and (pre_holding_unstack_var2 )))

(:action program_eff_holding_unstack_var2
  :parameters ()
  :precondition (and (modeProg )(not (del_holding_unstack_var2 ))(not (add_holding_unstack_var2 )))
 :effect (and (when (pre_holding_unstack_var2 )(del_holding_unstack_var2 ))
 (when (not (pre_holding_unstack_var2 ))(add_holding_unstack_var2 ))))

(:action validate_1
  :parameters ()
  :precondition (and (modeProg ))
 :effect (and (not (modeProg ))(clear a)(not (clear b))(clear c)(not (clear d))(not (clear e))(not (clear f))
```

```
  (not (clear g))(handempty )(not (holding a))(not (holding b))(not (holding c))(not (holding d))
  (not (holding e))(not (holding f))(not (holding g))(not (on a a))(not (on a b))(not (on a c))(not (on a d))
  (not (on a e))(not (on a f))(on a g)(not (on b a))(not (on b b))(not (on b c))(not (on b d))(on b e)
  (not (on b f))(not (on b g))(not (on c a))(not (on c b))(not (on c c))(on c d)(not (on c e))(not (on c f))
  (not (on c g))(not (on d a))(on d b)(not (on d c))(not (on d d))(not (on d e))(not (on d f))(not (on d g))
  (not (on e a))(not (on e b))(not (on e c))(not (on e d))(not (on e e))(on e f)(not (on e g))(not (on f a))
  (not (on f b))(not (on f c))(not (on f d))(not (on f e))(not (on f f))(not (on f g))(not (on g a))
  (not (on g b))(not (on g c))(not (on g d))(not (on g e))(not (on g f))(not (on g g))(not (ontable a))
  (not (ontable b))(not (ontable c))(not (ontable d))(not (ontable e))(ontable f)(ontable g)(test1 )))

 (:action validate_2
   :parameters ()
  :precondition (and (not (modeProg ))(clear a)(clear b)(clear c)(clear d)(clear e)(not (clear f))(not (clear g))
  (handempty )(not (holding a))(not (holding b))(not (holding c))(not (holding d))(not (holding e))
  (not (holding f))(not (holding g))(not (on a a))(not (on a b))(not (on a c))(not (on a d))(not (on a e))
  (not (on a f))(on a g)(not (on b a))(not (on b b))(not (on b c))(not (on b d))(not (on b e))(not (on b f))
  (not (on b g))(not (on c a))(not (on c b))(not (on c c))(not (on c d))(not (on c e))(not (on c f))(not (on c g))
  (not (on d a))(not (on d b))(not (on d c))(not (on d d))(not (on d e))(not (on d f))(not (on d g))(not (on e a))
  (not (on e b))(not (on e c))(not (on e d))(not (on e e))(on e f)(not (on e g))(not (on f a))(not (on f b))
  (not (on f c))(not (on f d))(not (on f e))(not (on f f))(not (on f g))(not (on g a))(not (on g b))(not (on g c))
  (not (on g d))(not (on g e))(not (on g f))(not (on g g))(not (ontable a))(ontable b)(ontable c)(ontable d)
  (not (ontable e))(ontable f)(ontable g)(test1 ))
   :effect (and (test2 )(not (test1 ))))

 (:action validate_3
   :parameters ()
  :precondition (and (not (modeProg ))(clear a)(clear b)(clear c)(clear d)(clear e)(clear f)(clear g)(handempty )
  (not (holding a))(not (holding b))(not (holding c))(not (holding d))(not (holding e))(not (holding f))
  (not (holding g))(not (on a a))(not (on a b))(not (on a c))(not (on a d))(not (on a e))(not (on a f))
  (not (on a g))(not (on b a))(not (on b b))(not (on b c))(not (on b d))(not (on b e))(not (on b f))(not (on b g))
  (not (on c a))(not (on c b))(not (on c c))(not (on c d))(not (on c e))(not (on c f))(not (on c g))(not (on d a))
  (not (on d b))(not (on d c))(not (on d d))(not (on d e))(not (on d f))(not (on d g))(not (on e a))(not (on e b))
  (not (on e c))(not (on e d))(not (on e e))(not (on e f))(not (on e g))(not (on f a))(not (on f b))(not (on f c))
  (not (on f d))(not (on f e))(not (on f f))(not (on f g))(not (on g a))(not (on g b))(not (on g c))(not (on g d))
  (not (on g e))(not (on g f))(not (on g g))(ontable a)(ontable b)(ontable c)(ontable d)(ontable e)(ontable f)
  (ontable g)(test2 ))
   :effect (and (not (test2 ))(test3 )))

)



(define (problem learning_problem)
  (:domain blocks)
  (:objects g - object c - object d - object e - object a - object b - object f - object )
  (:init (modeProg ) )
  (:goal (and (test3 ))))
```

Figure 15: Compiled PDDL problem file for learning the blocksworld action models from two initial and final states.