# Model Recognition as Planning

**Diego Aineto** and **Sergio Jiménez** and **Eva Onaindia**
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València.
Camino de Vera s/n. 46022 Valencia, Spain
{dieaigar,serjice,onaindia}@dsic.upv.es

## Abstract

Given a set of possible action models and a partially observed plan execution, *model recognition* is the task of identifying which of these action models has the highest probability of producing the given plan execution. The *model recognition* task is relevant because it enables identifying algorithms by their execution and because, once an action model is recognized, the planning model-based machinery becomes applicable. This paper formalizes the *model recognition* task and proposes a method to assess the probability of a given STRIPS action model to produce a partially observed plan execution. This method, that we called *model recognition as planning*, is robust to missing data in the intermediate states and actions of the observed plan execution besides, it is computable with an off-the-shelf classical planner. The effectiveness of *model recognition as planning* is shown with sets of STRIPS models that represent different *Turing Machines*, all of them defined within the same tape alphabet and same machine states. We show that *model recognition as planning* effectively identifies the executed *Turing Machine* despite intermediate machine state, applied transitions or tape values, are unobserved.

## Introduction

*Plan recognition* is the task of predicting the future actions of an agent provided the observation of its current behaviour. *Goal recognition* is a subtask of plan recognition with the particular aim of discovering the final objectives of the observed agent. Diverse methods has been proposed for plan and goal recognition (**?**) such as *rule-based systems*, *parsing* (both conventional and stochastic), *graph-covering*, *Bayesian nets*, ...

*Plan recognition as planning* recently introduced a model-based approach for plan/goal recognition (**?**; **?**). This approach assumes that the action model of the observed agent is known and leverages it to compute the most likely goal according to the observed plan execution.

In this paper we introduce the *model recognition* task as the task of identifying the action model with the highest probability of producing an observed plan execution. This task is of interest because :

- Once the action model is recognized the planning model-based machinery (**?**; **?**) becomes applicable.
- It enables identifying algorithms by their execution. Many computational models are compilable into classical planning (**?**; **?**; **?**).

The paper introduces *model recognition as planning*, a novel method to assess the probability of a given STRIPS action model to produce an observed plan execution. This method is robust to missing data in the intermediate states and actions of the observed plan execution besides, it is computable with an off-the-shelf classical planner.

Last but not least, the paper evaluates the effectiveness of *model recognition as planning* with sets of STRIPS models that represent different *Turing Machines*, all of them defined within the same tape alphabet and same machine states. We show that *model recognition as planning* effectively identifies the executed *Turing Machine* despite intermediate machine state, applied transitions or tape values, are unobserved.

## Background

This section defines the classical planning model and the STRIPS action model that we use for formalizing the *model recognition as planning* method.

### Classical planning

We use $F$ to denote the set of *fluents* (propositional variables) describing a state. A *literal* $l$ is a valuation of a fluent $f \in F$; i.e. either $l = f$ or $l = \neg f$. A set of literals $L$ represents a partial assignment of values to fluents (without loss of generality, we will assume that $L$ does not contain conflicting values). We use $\mathcal{L}(F)$ to denote the set of all literal sets on $F$; i.e. all partial assignments of values to fluents.

A *state* $s$ is a full assignment of values to fluents; $|s| = |F|$, so the size of the state space is $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions but often we will abuse of notation by defining a state $s$ only in terms of the fluents that are true in $s$, as it is common in STRIPS planning.

A *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where $F$ is a set of fluents and $A$ is a set of actions. An action $a \in A$ is defined with *preconditions*, $\mathsf{pre}(a) \subseteq \mathcal{L}(F)$, *positive effects*, $\mathsf{eff}^+(a) \subseteq \mathcal{L}(F)$, and *negative effects* $\mathsf{eff}^-(a) \subseteq \mathcal{L}(F)$. We say that an action $a \in A$ is *applicable* in a state $s$

iff $\mathsf{pre}(a) \subseteq s$. The result of applying $a$ in $s$ is the *successor state* denoted by $\theta(s,a) = \{s \setminus \mathsf{eff}^-(a)) \cup \mathsf{eff}^+(a)\}$.

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

A *classical planning problem* is a tuple $P = \langle F, A, I, G\rangle$, where $I$ is an initial state and $G \subseteq \mathcal{L}(F)$ is a goal condition. A *plan* for $P$ is an action sequence $\pi = \langle a_1, \ldots, a_n\rangle$ that induces the *state trajectory* $\langle s_0, s_1, \ldots, s_n\rangle$ such that $s_0 = I$ and $a_i$ $(1 \leq i \leq n)$ is applicable in $s_{i-1}$ and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The *plan length* is denoted with $|\pi| = n$ . A plan $\pi$ *solves* $P$ iff $G \subseteq s_n$; i.e. if the goal condition is satisfied in the last state resulting from the application of the plan $\pi$ in the initial state $I$.

### STRIPS action schemas

This work addresses the learning and evaluation of PDDL action schemas that follow the STRIPS requirement (**?**; **?**). Figure 1 shows the *stack* action schema, coded in PDDL, from a four-operator *blocksworld* (**?**).

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

Figure 1: STRIPS operator schema coding, in PDDL, the *stack* action from a four-operator *blocksworld*.

To formalize the target of the learning and evaluation tasks, we assume that fluents $F$ are instantiated from a set of *predicates* $\Psi$, as in PDDL. Each predicate $p \in \Psi$ has an argument list of arity $ar(p)$. Given a set of *objects* $\Omega$, the set of fluents $F$ is induced by assigning objects in $\Omega$ to the arguments of predicates in $\Psi$, i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ s.t. $\Omega^k$ is the $k$-th Cartesian power of $\Omega$.

Let $\Omega_v = \{v_i\}_{i=1}^{\max_{a\in A} ar(a)}$ be a new set of objects $(\Omega \cap \Omega_v = \emptyset)$, denoted as *variable names*, and that is bound by the maximum arity of an action in a given planning frame. For instance, in a three-block *blocksworld* $\Omega = \{block_1, block_2, block_3\}$ while $\Omega_v = \{v_1, v_2\}$ because the operators with the maximum arity, `stack` and `unstack`, have arity two.

We define $F_v$, a new set of fluents s.t. $F \cap F_v = \emptyset$, produced instantiating $\Psi$ using only *variable names*, and that defines the elements that can appear in the action schemes. In *blocksworld* this set contains 11 elements, $F_v$=`{handempty, holding(`$v_1$`), holding(`$v_2$`), clear(`$v_1$`), clear(`$v_2$`), ontable(`$v_1$`), ontable(`$v_2$`), on(`$v_1,v_1$`), on(`$v_1,v_2$`), on(`$v_2,v_1$`), on(`$v_2,v_2$`)}`.

In more detail, for a given operator schema $\xi$, we define $F_v(\xi) \subseteq F_v$ as the subset of elements that can appear in that action schema. For instance, for the *stack* action schema $F_v$(`stack`) $= F_v$ while $F_v$(`pickup`)=`{handempty, holding(`$v_1$`), clear(`$v_1$`), ontable(`$v_1$`), on(`$v_1,v_1$`)}` excludes the elements from $F_v$ that involve $v_2$ because `pickup(`$v_1$`)` has arity one.

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

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

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

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

## Model Recognition

Given a set of *comparable* action models $M = \{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$ and a partially observed plan execution $\mathcal{T} = \langle s_0, a_{,1}, s_1, \ldots, a_n, s_n\rangle$ that is obtained watching the execution of a plan $\pi = \langle a_1, \ldots, a_n\rangle$ s.t., for each $1 \leq i \leq n$, $a_i$ is applied in $s_{i-1}$ and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. *Model recognition* is the task of identifying which model $\mathcal{M} \in M$ has the highest probability of producing $\mathcal{T}$.

### The STRIPS edit distance

The intuition of our method for *model recognition* is to assess how well an action model $\mathcal{M} \in M$ explains $\mathcal{T}$ according to the amount of *edition* required by the model $\mathcal{M}$ to induce the observations $\mathcal{T}$. In this work we assume that all the action models in $M$ are represented using the STRIPS formalism. We define here the two allowed *operations* to edit a given STRIPS action model $\mathcal{M}$:

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

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

**Definition 1.** *Let $\mathcal{M}$ and $\mathcal{M}'$ be two* comparable STRIPS *action models. The* **edit distance***, denoted as $\delta(\mathcal{M}, \mathcal{M}')$, is the minimum number of* edit operations *that is required to transform $\mathcal{M}$ into $\mathcal{M}'$.*

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

**Definition 2.** *The* ***maximum edit distance*** *of an* STRIPS *action model $\mathcal{M}$ built from the set of possible elements $F_v$ is $\delta(\mathcal{M}, *) = \sum_{\xi \in \mathcal{M}} 3 \times |F_v(\xi)|$.*

We define now an edit distance to asses the matching of a learned action model with respect to a plan trace $\mathcal{T}$.

**Definition 3.** *Given $\mathcal{M}$, a* STRIPS *action model built from $F_v$, and a plan trace $\mathcal{T}$ whose state observation are built with fluents in $F$. The* **observation edit distance***, denoted by $\delta(\mathcal{M}, \mathcal{T})$, is the minimal edit distance from $\mathcal{M}$ to any* comparable *model $\mathcal{M}'$, such that $\mathcal{M}'$ can produce a valid plan trace $\mathcal{T}$;*

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

Note that the distance of an action model $\mathcal{M}$ with respect to a plan trace $\mathcal{T}$ could also be defined quantifying the amount of edition required by the observations of the plan execution to match the given model. This would imply defining *edit operations* that modify the fluents in the state observations instead of the *edit operations* that modify the action schemes (**?**). Our definition of the *observation edit distance* is more practical since normally, $F_v$ is smaller than $F$ because the number of *variable objects* is smaller than the number of objects in the state observations.

### The STRIPS probability distribution

According to the *Bayes* rule, the probability of an hypothesis $\mathcal{H}$ provided the observation $\mathcal{O}$ is given by the expression $P(\mathcal{H}|\mathcal{O}) = \frac{P(\mathcal{O}|\mathcal{H})P(\mathcal{H})}{P(\mathcal{O})}$.

In our scenario, the hypotheses are about the set of possible STRIPS action models in $M$ while the observation is a partially observable input plan trace $\mathcal{T}$. Given the set of predicates $\Psi$ and the given a set of operator headers (in other words, given the $F_v(\xi)$ sets) the size of the set of possible STRIPS models set is $\prod_\xi 2^{2|F_v(\xi)|}$. If we assume that a priori all models are equiprobable this means that $P(\mathcal{M}) = \frac{1}{\prod_\xi 2^{2|F_v(\xi)|}}$.

With respect to the observations, given $\Psi$ and a set of objects $\Omega$, the number of possible state observations of length $n$ in a trace $\mathcal{T}$ is $2^{n \times |F|}$ while the number of possible action observations of length $n$ in a trace $\mathcal{T}$ is $2^{n \times |A|}$. If we assume that a priori all traces are equiprobable, this means $P(\mathcal{T}) = \frac{1}{2^{n \times |F|} \times 2^{n \times |A|}}$.

With this regard, we call $P(\mathcal{M}|\mathcal{T})$ the STRIPS *probability distribution* to the probability distribution of the possible STRIPS models (within the $F_v(\xi)$ sets) provided that the plan trace $\mathcal{T}$ is observed. The STRIPS *probability distribution* can be estimated by:

1. Estimating the a priori probabilities $P(\mathcal{T})$ and $P(\mathcal{H})$, as explained.
2. Computing the *observation edit distance* $\delta(\mathcal{M}, \mathcal{T})$ for every possible model $\mathcal{M} \in M$ and mapping the *observation edit distance* into a $P(\mathcal{O}|\mathcal{H})$ likelihood with the following expression $1 - \frac{\delta(\mathcal{M}, \mathcal{T})}{\delta(\mathcal{M}, *)}$.
3. Applying the Bayes rule to obtain the normalized posterior probabilities, these probabilities must sum 1.

## Model Recognition as Planning

Our method for addressing the *model recognition* task is compiling it into a classical planning task with conditional effects. This approach follows the following assumptions:

1. The initial state $s_0 \in \mathcal{T}$ is *fully observable*.
2. Intermediate actions $a_i \in \mathcal{T}$ and states $s_i \in \mathcal{T}$ s.t. $1 \leq i \leq n$, can be *partially observed*. This means that some fluents in $s_i$ may be missing because it is unknown whether their value is either positive or negative. In the extreme, entire states $s_i$ can be missing.
3. Observations in $\mathcal{T}$ are *noiseless*, meaning that if a fluent or an action is observed it is the correct value.

### Conditional effects

Conditional effects 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 $s$:

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

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

### The compilation

The compilation allow us to estimate the STRIPS edit distance of a given model $\mathcal{M} \in M$ with regard to the partially observed plan execution $\mathcal{T}$. The intuition behind the compilation is that a solution to the resulting classical planning task is a sequence of actions that:

1. **Edits the action model $\mathcal{M}$ to build $\mathcal{M}'$.** A solution plan starts with a *prefix* that modifies the preconditions and effects of the action schemes in $\mathcal{M}$ using to the two *edit operations* defined above, *deletion* and *insertion*.
2. **Validates the edited model $\mathcal{M}'$ in the observed plan trace**. The solution plan continues with a postfix that validates the edited model $\mathcal{M}'$ on the given observations $\mathcal{T}$.

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

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

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

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

Given $\langle \mathcal{M}, \Psi, \mathcal{T} \rangle$ the compilation outputs a classical planning task $P = \langle F, A, I, G \rangle$:

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

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

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

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

     Besides these actions $A$ also contains the actions for *inserting* a precondition and for *deleting* a negative/-positive effect.
  2. Actions for *applying* a programmed operator schema $\xi \in \mathcal{M}$ bound with objects $\omega \subseteq \Omega^{ar(\xi)}$. Since operators headers are given as input, the variables $pars(\xi)$ are bound to the objects in $\omega$ that appear at the same position. Figure 3 shows the PDDL encoding of the action for applying a programmed operator *stack* from *blocksworld*.

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

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

```
(:action apply_stack
  :parameters (?o1 - object ?o2 - object)
  :precondition
   (and (or (not (pre_on_stack_v1_v1)) (on ?o1 ?o1))
        (or (not (pre_on_stack_v1_v2)) (on ?o1 ?o2))
        (or (not (pre_on_stack_v2_v1)) (on ?o2 ?o1))
        (or (not (pre_on_stack_v2_v2)) (on ?o2 ?o2))
        (or (not (pre_ontable_stack_v1)) (ontable ?o1))
        (or (not (pre_ontable_stack_v2)) (ontable ?o2))
        (or (not (pre_clear_stack_v1)) (clear ?o1))
        (or (not (pre_clear_stack_v2)) (clear ?o2))
        (or (not (pre_holding_stack_v1)) (holding ?o1))
        (or (not (pre_holding_stack_v2)) (holding ?o2))
        (or (not (pre_handempty_stack)) (handempty)))
  :effect
   (and (when (del_on_stack_v1_v1) (not (on ?o1 ?o1)))
        (when (del_on_stack_v1_v2) (not (on ?o1 ?o2)))
        (when (del_on_stack_v2_v1) (not (on ?o2 ?o1)))
        (when (del_on_stack_v2_v2) (not (on ?o2 ?o2)))
        (when (del_ontable_stack_v1) (not (ontable ?o1)))
        (when (del_ontable_stack_v2) (not (ontable ?o2)))
        (when (del_clear_stack_v1) (not (clear ?o1)))
        (when (del_clear_stack_v2) (not (clear ?o2)))
        (when (del_holding_stack_v1) (not (holding ?o1)))
        (when (del_holding_stack_v2) (not (holding ?o2)))
        (when (del_handempty_stack) (not (handempty)))
        (when (add_on_stack_v1_v1) (on ?o1 ?o1))
        (when (add_on_stack_v1_v2) (on ?o1 ?o2))
        (when (add_on_stack_v2_v1) (on ?o2 ?o1))
        (when (add_on_stack_v2_v2) (on ?o2 ?o2))
        (when (add_ontable_stack_v1) (ontable ?o1))
        (when (add_ontable_stack_v2) (ontable ?o2))
        (when (add_clear_stack_v1) (clear ?o1))
        (when (add_clear_stack_v2) (clear ?o2))
        (when (add_holding_stack_v1) (holding ?o1))
        (when (add_holding_stack_v2) (holding ?o2))
        (when (add_handempty_stack) (handempty))
        (when (modeProg) (not (modeProg)))))
```

Figure 3: PDDL action for applying an already programmed schema *stack* (implications are coded as disjunctions).

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

   $\mathsf{pre}(\mathsf{validate_i}) = s_i \cup \{test_j\}_{j \in 1 \leq j < i} \cup \{\neg test_j\}_{j \in i \leq j \leq n} \cup \{mode_{val}\}$,

   $\mathsf{cond}(\mathsf{validate_i}) = \{\emptyset\} \rhd \{test_i, \neg mode_{val}\}$.

## Evaluation

To evaluate the empirical performance of our method for *planning model recognition* we defined a set of possible STRIPS models, each representing a different *Turing Machines*, but all sharing the same tape alphabet and same set of machine states.

## Modeling *Turing Machines* with STRIPS

A *Turing machine* is a tuple $M = \langle Q, q_o, Q_\perp, \mathcal{T}, \square, \Sigma, \delta \rangle$:

- $Q$, is a finite and non-empty set of machine states such that $q_0 \in Q$ is the initial state of the machine and $Q_\perp \subseteq Q$ is the subset of acceptor states.
- $\mathcal{T}$ is the *tape alphabet*, that is a finite non-empty set of symbols that includes the *blank symbol* $\square \in \mathcal{T}$ (the only symbol allowed to occur on the tape infinitely often) and that contains $\Sigma \subseteq \mathcal{T}$, the set of symbols allowed to initially appear in the tape (also called the *input alphabet*).
- $\delta : (Q \setminus Q_\perp) \times \mathcal{T} \rightarrow Q \times \{left, right\} \times \mathcal{T}$ is the *transition function*. If $\delta$ is not defined for the current pair of machine state and tape symbol, then the machine halts.

| | $q_0$ | $q_1$ | $q_2$ | $q_3$ | $q_4$ | $\underline{q_5}$ |
|---|---|---|---|---|---|---|
| a | x,r,$q_1$ | a,r,$q_1$ | - | a,l,$q_3$ | - | - |
| b | - | y,r,$q_2$ | b,r,$q_2$ | b,l,$q_3$ | - | - |
| c | - | - | z,l,$q_3$ | - | - | - |
| x | - | - | - | x,r,$q_0$ | - | - |
| y | y,r,$q_4$ | y,r,$q_1$ | - | y,l,$q_3$ | y,r,$q_4$ | - |
| z | - | - | z,r,$q_2$ | z,l,$q_3$ | z,r,$q_4$ | - |
| $\square$ | - | - | - | - | $\square$,r,$q_5$ | - |

Figure 4: Example of a seven-symbol six-state *Turing Machine* for recognizing the $\{a^n b^n c^n : n \geq 1\}$ language ($\underline{q_5}$ is the only acceptor state).

A table is the most common convention to represent the transitions defined by $\delta$, where the table rows are indexed by the current tape symbol, while the table columns are indexed by the current machine state. For each possible pair of tape symbol and machine state, there is a table entry that defines: (1) the tape symbol to print at the current position of the header (2) whether the header is shifted *left* or *right* after the print operation and (3), the new state of the machine after the print operation. For instance, Figure 4 shows the table that represents the $\delta$ function of a *Turing Machine* for recognizing the $\{a^n b^n c^n : n \geq 1\}$ language. In this example the tape alphabet is $\Sigma = \{a, b, c, x, y, z, \square\}$ while the possible machine states are $Q = \{q_0, q_1, q_2, q_3, q_4, \underline{q_5}\}$ where $\underline{q_5}$ is the only acceptor state.

A classical planning frame $\Phi = \langle F, A \rangle$ encodes the *transition function* $\delta$ of a *Turing Machine* $M$ as follows:

- The fluents $F$ are instantiated from the set of *predicates* $\Psi$ that contains: `(head ?x)` that encodes the current position of the header in the tape. `(next ?x1 ?x2)` encoding that the cell `?x2` follows cell `?x1` in the tape. `(symbol-`$\sigma$ `?x)`, encoding that the tape cell `?x` contains the symbol $\sigma \in \Sigma$. `(state-`$q$`)` encoding that $q \in Q$ is the current machine state. Given a set of *objects* $\Omega$ that represent the cells in the tape of the given Turing Machine $M$, the set of fluents $F$ is induced by assigning objects in $\Omega$ to the arguments of the predicates in $\Psi$.
- The actions $a \in A$ are instantiated from STRIPS operator schema. For each transition in $\delta$, a STRIPS action schema is defined such that:
  - The **header** of the schema is `transition-id(?xl ?x ?xr)` where $id$ uniquely identifies the transition in $\delta$ and the parameters $?xl$, $?x$ and $?xr$ are tape cells.
  - The **preconditions** of the schema includes `(head ?x)` and `(next ?xl ?x) (next ?x ?xr)` to force that $?x$ is the tape cell currently pointed by the header and that $?xl$ and $?xr$ respectively are its left and right neighbours. Additionally the schema includes preconditions

`(symbol-`$\sigma$ `?x)` and `(state-`$q$`)` to capture the symbol pointed by the header and the currrent machine state.

- The **delete effects** remove the symbol pointed by the header and the currrent machine state while the **positive effects** set the new symbol pointed by the header and the new machine state.

The STRIPS action schema of Figure 5 models the rule $a, q_0 \rightarrow x, r, q_1$ of the Turing Machine defined in Figure 4. The full encoding of the Turing Machine defined in Figure 4 produces a total of sixteen STRIPS action schema with the same structure as the one of Figure 5.

```
(:action transition-1    ;;; a,q0 → x,r,q1
  :parameters (?xl ?x ?xr)
  :precondition (and (head ?x) (symbol-a ?x) (state-q0)
                     (next ?xl ?x) (next ?x ?xr))
  :effect (and (not (head ?x))
               (not (symbol-a ?x)) (not (state-q0))
               (head ?xr) (symbol-x ?x) (state-q1)))
```

Figure 5: STRIPS action schema that models the transition $a, q_0 \rightarrow x, r, q_1$ of the Turing Machine defined in Figure 4.

The execution of a *Turing Machine* can then be defined as a *plan trace* $\mathcal{T} = \langle s_0, a_1, s_1, \ldots, a_n, s_n \rangle$ such that $s_0$ encodes the initial state of the tape plus the initial machine state and, for each $1 \leq i \leq n$, $a_i$ is applicable in $s_{i-1}$ and generates the successor state $s_i = \theta(s_{i-1}, a_i)$.

## Experimental setup

We randomly generated $M = \{\mathcal{M}_1, \ldots, \mathcal{M}_{100}\}$ set of one-hundred different *Turing Machines* where each $\mathcal{M} \in M$ is a two-symbol three-state *Turing Machines*. We randomly choose a machine $\mathcal{M} \in M$ and produce an fifty-step execution plan trace $\mathcal{T} = \langle s_0, a_1, s_1, \ldots, a_{50}, s_{50} \rangle$.

## Conclussions

## References


Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS*, 26–33.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Bonet, B.; Palacios, H.; and Geffner, H. 2010. Automatic derivation of finite-state machines for behavior control. In *AAAI Conference on Artificial Intelligence*.

Carberry, S. 2001. Techniques for plan recognition. *User Modeling and User-Adapted Interaction* 11(1-2):31–48.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Geffner, H., and Bonet, B. 2013. A concise introduction to models and methods for automated planning.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Elsevier.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language.

Ramírez, M., and Geffner, H. 2009. Plan recognition as planning. In *International Joint conference on Artifical Intelligence*, 1778–1783.

Ramírez, M. 2012. *Plan recognition as planning*. Ph.D. Dissertation, Universitat Pompeu Fabra.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016. Hierarchical finite state controllers for generalized planning. In *International Joint Conference on Artificial Intelligence, IJCAI-16*, 3235–3241. AAAI Press.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125(1-2):119–153.

Sohrabi, S.; Riabov, A. V.; and Udrea, O. 2016. Plan recognition as planning revisited. In *IJCAI*, 3258–3264.