

# Model Recognition as Planning

Diego Aineto and Sergio Jiménez and Eva Onaindia and Miquel Ramírez

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 partially observed plan execution and a set of possible planning models, *model recognition* is the task of identifying the model in the set that produced the observed plan execution. The paper formalizes the *model recognition* task and proposes a novel method to assess the probability of a STRIPS model to produce a given 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 in a set of STRIPS models encoding different *Turing Machines*. We show that *model recognition as planning* succeeds to identify the executed *Turing Machine* despite the actual applied transitions, internal machine state or the values of several tape cells, are unknown.

## Introduction

*Plan recognition* is the task of predicting the future actions of an agent provided observations of its current behaviour. Plan recognition is considered *automated planning* in reverse; while automated planning aims to compute a sequence of actions that accounts for a given goals, plan recognition aims to compute the goals that account for an observed sequence of actions (Geffner and Bonet 2013).

Diverse approaches has been proposed for plan recognition such as *rule-based systems*, *parsing*, *graph-covering*, *Bayesian nets*, etc (Carberry 2001). *Plan recognition as planning* is the model-based approach for plan recognition (Ramírez 2012; Ramírez and Geffner 2009). This approach assumes that the action model of the observed agent is known and leverages it to compute the most likely goal of the agent according to the observed plan execution.

This paper introduces the task of *model recognition*. Given a partially observed plan execution and a set of possible planning models, *model recognition* is the task of identifying which model in the set has the highest probability of producing the input observation. *Model recognition* is of interest because:

- Once the planning model is recognized with certainty,

then the model-based machinery for automated planning becomes applicable (Ghallab, Nau, and Traverso 2004).

- It enables the recognition of algorithms by observing their execution given that diverse algorithm representations, like *GOLOG programs*, *finite state controllers* or *push-down automata* can be encoded as classical planning models (Baier, Fritz, and McIlraith 2007; Bonet, Palacios, and Geffner 2010; Segovia-Aguas, Jiménez, and Jonsson 2017).

The paper introduces also *model recognition as planning*; a novel method to assess the probability of a given STRIPS model to produce an observed plan execution. The 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. The paper evaluates the effectiveness of *model recognition as planning* with a set of STRIPS models that represent different *Turing Machines* (all of them defined within the same *tape alphabet* and same *machine states* but different *transition functions*). We show that *model recognition as planning* succeeds to identify the executed *Turing Machine* despite the actual applied transitions, the internal machine state or the values of several tape cells, are unknown.

## Background

This section formalizes classical planning and the observation of the execution of a classical plan.

### 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. The dynamics of the actions in  $A$  are specified with two functions:  $App(s)$  that denotes the subset of actions applicable in a state  $s$  and  $\theta(s, a)$  that denotes the *successor state* that results of applying  $a$  in  $s$ .

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  $\pi = \langle a_1, \dots, a_n \rangle$  that, when executed starting from the initial state  $I$ , induces the *state trajectory*  $\langle s_0, s_1, \dots, s_n \rangle$  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 execution of the plan  $\pi$  in the initial state  $I$ . A solution plan is *optimal* if it has minimum length.

### The observation model

Given classical planning problem  $P = \langle F, A, I, G \rangle$ , the observation of the execution of a plan  $\pi$  that solves  $P$  is an interleaved combination  $\tau = \langle s_1, a_1, \dots, a_n, s_m \rangle$  of a *partially-observed* sequence of actions and a sequence of *partially-observed* states. Because of the partial observability  $0 \leq n \leq |\pi|$  and  $0 \leq m \leq |\pi| + 1$  and hence, the transitions between two consecutive observed states in  $\tau$  may involve the execution of more than a single action. Formally  $\theta(s_i, \langle a_1, \dots, a_k \rangle) = s_{i+1}$ , where  $k \geq 1$  is unknown and unbound. This means that having  $\tau$  does not implies knowing the actual length of  $\pi$ .

We say that a given observation  $\tau$  is *compliant* with a given plan  $\pi$  iff:

- The sequence of actions  $\langle a_1, \dots, a_n \rangle$  in  $\tau$  is a subsequence of  $\pi$ .
- The sequence of states  $\langle s_1, \dots, s_m \rangle$  in  $\tau$  is the same sequence of states traversed by  $\pi$  but (1), with certain states omitted and/or (2), the value of certain fluents omitted in that states, i.e.  $|s_i| \leq |F|$  for every  $0 \leq i \leq m$ .

### Model Recognition

The *model recognition* task is a tuple  $\langle P, M, \tau \rangle$  where:

- $P = \langle F, A, I, G \rangle$  is a classical planning problem such that the dynamics of the actions in  $A$  is unknown because the  $App(s)$  and  $\theta(s, a)$  functions are undefined.
- $M = \{\mathcal{M}_1, \dots, \mathcal{M}_m\}$  is a finite and non-empty set of action models, each defining a different dynamics (a different  $App(s)$  and  $\theta(s, a)$  pair) for the actions in  $A$ .
- $\tau$  is the observation of the execution of a solution plan  $\pi$  for  $P$ .

A solution to the *model recognition* task is the discrete probability distribution  $P(\mathcal{M}|\tau)$  that expresses, for each model  $\mathcal{M} \in M$ , its probability of producing the observation  $\tau$ .

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 the *model recognition* task, hypotheses are about the possible action models  $\mathcal{M} \in M$

```
(: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 action schema coded in PDDL.

while the given observation is the partially observed plan execution  $\tau$ . The  $P(\mathcal{M}|\tau)$  probability distribution can then be estimated in three steps:

1. Computing the *a priori* probabilities.  $P(\tau)$ , that measures how surprising is the given observation and  $P(\mathcal{M})$ , that expresses if one model is a priori more likely than the others.
2. Computing the conditional probability  $P(\tau|\mathcal{M})$ . Our approach is to estimate this value according to the minimum amount of *edition* that is required by  $\mathcal{M}$  to produce a plan  $\pi_\tau^*$  such that:
  - (a)  $\pi_\tau^*$  is an optimal solution for the classical planning problem  $P$ .
  - (b)  $\pi_\tau^*$  is compliant with the given observation  $\tau$ .
3. Applying the Bayes rule to obtain the normalized posterior probabilities (the  $P(\mathcal{M}|\tau)$  probabilities must sum 1 for all the  $\mathcal{M} \in M$ ).

Note that the second step assumes that the observed agent is acting rationally, like in *plan recognition as planning* (Ramírez 2012; Ramírez and Geffner 2009). A related approach is recently followed for *model reconciliation* (Chakraborti et al. 2017) where model edition is used to conform the different PDDL models of two different agents.

### Recognition of STRIPS models

Here we analyze the particular instantiation of the *model recognition* task where the dynamics of the actions  $A \in P$  (i.e. the  $App(s)$  and  $\theta(s, a)$  functions) are specified with STRIPS action schemas. Figure 1 shows the PDDL code for the *stack* actions, taken from a four-operator *blocksworld* (Slaney and Thiébaux 2001).

### The space of STRIPS models

Like in PDDL (McDermott et al. 1998; Fox and Long 2003), we assume that fluents  $F$  are instantiated from a set of *predicates*  $\Psi$ . 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 an additional set of objects ( $\Omega \cap \Omega_v = \emptyset$ ), that we denote 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 four-operator *blocksworld*  $\Omega = \{block_1, block_2, block_3\}$  while  $\Omega_v = \{v_1, v_2\}$  because the *stack* and *unstack* actions have arity two. We define  $F_v$ , a new set of fluents,  $F \cap F_v = \emptyset$ , produced instantiating  $\Psi$  using

```

(pre_holding_stack_v1) (pre_clear_stack_v2)
(del_holding_stack_v1) (del_clear_stack_v2)
(add_hanempty_stack) (add_clear_stack_v1)
(add_on_stack_v1_v2)

```

Figure 2: Propositional encoding for the *stack* schema from a four-operator *blocksworld*.

only *variable names*. This set contains eleven elements for the mentioned *blocksworld*,  $F_v = \{\text{handempty}, \text{holding}(v_1), \text{holding}(v_2), \text{clear}(v_1), \text{clear}(v_2), \text{ontable}(v_1), \text{ontable}(v_2), \text{on}(v_1, v_1), \text{on}(v_1, v_2), \text{on}(v_2, v_1), \text{on}(v_2, v_2)\}$ .

For a given action schema  $\xi$ , then  $F_\xi \subseteq F_v$  defines the subset of elements that can appear in the preconditions and effects of that action schema. For instance  $F_{\text{stack}} = F_v$  while  $F_{\text{pickup}} = \{\text{handempty}, \text{holding}(v_1), \text{clear}(v_1), \text{ontable}(v_1), \text{on}(v_1, v_1)\}$  excludes any element in  $F_v$  that involves  $v_2$  because *pickup* actions have arity one.

Now we are ready to define the fluents  $\text{pre}_f(\xi)$ ,  $\text{del}_f(\xi)$  and  $\text{add}_f(\xi)$ , for every  $f \in F_\xi$ , that represent the propositional encoding for the preconditions, negative and positive effects of an action schema  $\xi$ . If a fluent  $\text{pre}_f(\xi)/\text{del}_f(\xi)/\text{add}_f(\xi)$  holds, it means that  $f$  is a precondition/negative/positive effect in that schema. For instance, Figure 2 shows the conjunction of fluents that represents the propositional encoding for the preconditions, negative and positive effects of the *stack* schema shown in Figure 1.

The size of the space of possible STRIPS models for an action schema  $\xi$  is given by the expression,  $2^{|F_\xi|}$  (STRIPS constraints require negative effects appearing as preconditions and that they cannot be positive effects and also, that positive effects cannot appear as preconditions). For the mentioned *blocksworld*,  $2^{|F_{\text{stack}}|} = 4194304$  while  $2^{|F_{\text{pickup}}|} = 1024$ . We say that two STRIPS schemes  $\xi$  and  $\xi'$  are *comparable* iff both share the same space of possible STRIPS models. For instance, we claim that *blocksworld* schemas *stack* and *unstack* are *comparable* while *stack* and *pickup* are not. Last but not least, two STRIPS models  $\mathcal{M}$  and  $\mathcal{M}'$  are *comparable* iff there exists a bijective function  $\mathcal{M} \mapsto \mathcal{M}^*$  that maps every action schema  $\xi \in \mathcal{M}$  to a comparable schema  $\xi' \in \mathcal{M}'$  and vice versa.

## The STRIPS edit distance

We define two edit *operations* on a STRIPS model  $\mathcal{M} \in M$ :

- *Deletion*. A fluent  $\text{pre}_f(\xi)/\text{del}_f(\xi)/\text{add}_f(\xi)$  is removed from the operator schema  $\xi \in \mathcal{M}$ , such that  $f \in F_\xi$ .
- *Insertion*. A fluent  $\text{pre}_f(\xi)/\text{del}_f(\xi)/\text{add}_f(\xi)$  is added to the operator schema  $\xi \in \mathcal{M}$ , s.t.  $f \in F_\xi$ .

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**  $\delta(\mathcal{M}, \mathcal{M}')$  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.

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

We define now an edit distance to assess the matching of a STRIPS model with respect to an observation of a plan execution.

**Definition 3.** Given  $\tau$ , an observation of the execution of a plan for solving  $P$  and  $\mathcal{M}$ , a STRIPS action model built from  $F_v$ . The **observation edit distance**,  $\delta(\mathcal{M}, \tau)$ , is the minimal edit distance from  $\mathcal{M}$  to any comparable model  $\mathcal{M}'$  s.t.  $\mathcal{M}'$  produces a plan  $\pi_\tau^*$  optimal for  $P$  and compliant with  $\tau$ ;

$$\delta(\mathcal{M}, \tau) = \min_{\forall \mathcal{M}' \rightarrow \tau} \delta(\mathcal{M}, \mathcal{M}')$$

Our  $\delta(\mathcal{M}, \tau)$  distance could also be defined assessing the edition required by the observed plan execution to match the given model. This implies defining *edit operations* that modify  $\tau$  instead of  $\mathcal{M}$  (Sohrabi, Riabov, and Udrea 2016). Our definition of the *observation edit distance* is more practical since normally  $F_v$  is smaller than  $F$ . In practice, the number of *variable objects* is normally smaller than the number of objects in a planning problem.

## The $P(\mathcal{M}|\tau)$ probability for STRIPS models

Since we assume that the dynamics of the actions  $A \in P$  is specified with STRIPS action schemas, we can estimate the  $P(\mathcal{M}|\tau)$  probability as follows:

1. Assuming that a priori, all action models  $\mathcal{M} \in M$  are equiprobable because there are no reasons to assume that one is a priori more likely than the others. Therefore,  $P(\mathcal{M}) = \frac{1}{\prod_{\xi \in \mathcal{M}} 2^{|F_\xi|}}$ .
2. Assuming that all the observations of plan executions with a maximum of  $n$  observed actions and  $m$  observed states are equiprobable then  $P(\tau) = \frac{1}{2^{n \times |A| \times 2^{m \times |F|}}}$ .
3. To estimate the conditional probability  $P(\tau|\mathcal{M})$  we compute the *observation edit distance*  $\delta(\mathcal{M}, \tau)$  and map it into a  $[0, 1]$  likelihood with the expression,  $1 - \frac{\delta(\mathcal{M}, \tau)}{\delta(\mathcal{M}, *)}$ .

## Model recognition as planning

This section shows that, when the dynamics of the actions  $A \in P$  are specified with STRIPS action schemas, then  $\delta(\mathcal{M}, \tau)$  can be computed with a compilation of a classical planning with conditional effects. The intuition behind this 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 the two *edit operations* defined above, *deletion* and *insertion*.

```

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 3: Plan for editing (steps [0-1]) and validating (steps [2-6]) a given STRIPS planning model for the *blocksworld*.

## 2. Validates the edited model $\mathcal{M}'$ in the observation $\tau$ .

The solution plan continues with a postfix that validates the edited model  $\mathcal{M}'$  on the given observations  $\tau$ .

Figure 3 shows the plan for editing a given *blocksworld* model where the positive effects (`handempty`) and (`clear ?v1`) of the `stack` schema are missing. The edited action model is validated at the observation of a four action plan for inverting a two-block tower where the intermediate states,  $s_1, s_2$  and  $s_3$ , are unobserved.

Note that our interest is not in  $\mathcal{M}'$ , the edited model resulting from the compilation, but in the number of required *edit operations* (insertions and deletions) required by  $\mathcal{M}'$  to be validated in the given observation, e.g.  $\delta(\mathcal{M}, \tau) = 2$  for the example in Figure 3. In this case  $\delta(\mathcal{M}, *) = 3 \times 2 \times (11 + 5)$  since there are 4 action schemes (`pickup`, `putdown`, `stack` and `unstack`) and  $|F_v| = |F_{\text{stack}}| = |F_{\text{unstack}}| = 11$  while  $|F_{\text{pickup}}| = |F_{\text{putdown}}| = 5$ .

## Conditional effects

Conditional effects allow us to compactly define our compilation. An action  $a \in A$  with conditional effects is defined as a set of *preconditions*  $\text{pre}(a) \in \mathcal{L}(F)$  and a set of *conditional effects*  $\text{cond}(a)$ . Each conditional effect  $C \triangleright E \in \text{cond}(a)$  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  $\text{pre}(a) \subseteq s$ , and the *triggered effects* resulting from the action application are the effects whose conditions hold in  $s$ :

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

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

## The compilation formalization

Given a STRIPS model  $\mathcal{M} \in M$  and the observation  $\tau$  of the execution of a plan for solving  $P = \langle F, A, I, G \rangle$ , our compilation outputs a classical planning task  $P' = \langle F', A', I', G' \rangle$  such that:

- $F'$  contains:
  - The original fluents  $F$ .
  - Fluents  $\text{pre}_f(\xi)/\text{del}_f(\xi)/\text{add}_f(\xi)$  modeling the space of STRIPS models.

- The fluents  $F_\pi = \{\text{plan}(\text{name}(a_i), \Omega^{ar(a_i)}, i)\}_{1 \leq i \leq n}$  to code the  $i^{\text{th}}$  action in  $\tau$ . The static facts  $\text{next}_{i,i+1}$  and the fluents  $\text{at}_i, 1 \leq i < n$ , are also added to iterate through the  $n$  steps of  $\tau$ .
- The fluents  $\{\text{test}_j\}_{1 \leq j \leq m}$ , indicating the state observation  $s_j \in \tau$  where the action model is validated.
- The fluents  $\text{mode}_{\text{edit}}$  and  $\text{mode}_{\text{val}}$  to indicate whether the operator schemas are edited or validated.

- $I'$  extends the original initial state  $I$  with the fluent  $\text{mode}_{\text{edit}}$  set to true as well as the fluents  $F_\pi$  plus fluents  $\text{at}_1$  and  $\{\text{next}_{i,i+1}\}, 1 \leq i < n$ , for tracking the plan step where the action model is validated. Our compilation assumes that initially  $\mathcal{M}'$  is defined as  $\mathcal{M}$ . Therefore fluents  $\text{pre}_f(\xi)/\text{del}_f(\xi)/\text{add}_f(\xi)$  hold as given by  $\mathcal{M}$ .
- $G' = G \cup \{\text{at}_n, \text{test}_m\}$ .
- $A'$  comprises three kinds of actions with conditional effects:

### 1. Actions for *editing* operator schema $\xi \in \mathcal{M}$ :

- Actions for adding a *precondition*  $f \in F_v(\xi)$  from the action schema  $\xi \in \mathcal{M}$ .

$$\begin{aligned} \text{pre}(\text{programPre}_{f,\xi}) &= \{\neg \text{pre}_f(\xi), \neg \text{del}_f(\xi), \neg \text{add}_f(\xi), \text{mode}_{\text{edit}}\}, \\ \text{cond}(\text{programPre}_{f,\xi}) &= \{\emptyset\} \triangleright \{\text{pre}_f(\xi)\}. \end{aligned}$$

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

$$\begin{aligned} \text{pre}(\text{programEff}_{f,\xi}) &= \{\neg \text{del}_f(\xi), \neg \text{add}_f(\xi), \text{mode}_{\text{edit}}\}, \\ \text{cond}(\text{programEff}_{f,\xi}) &= \{\text{pre}_f(\xi)\} \triangleright \{\text{del}_f(\xi)\}, \\ &\quad \{\neg \text{pre}_f(\xi)\} \triangleright \{\text{add}_f(\xi)\}. \end{aligned}$$

Besides these actions, the  $A'$  set also contains the actions for *deleting* a precondition and a negative/positive effect.

- 2. Actions for *applying* an edited operator schema  $\xi \in \mathcal{M}$  bound with objects  $\omega \subseteq \Omega^{ar(\xi)}$ . Since operators headers are given as input, the variables  $\text{pars}(\xi)$  are bound to the objects in  $\omega$  that appear at the same position. Figure 4 shows the PDDL encoding of the action for applying a programmed operator *stack* from *blocksworld*.

$$\begin{aligned} \text{pre}(\text{apply}_{\xi,\omega}) &= \{\text{pre}_f(\xi) \implies p(\omega)\}_{\forall p \in \Psi, f=p(\text{pars}(\xi))} \\ &\quad \cup \{\neg \text{mode}_{\text{val}}\}, \\ \text{cond}(\text{apply}_{\xi,\omega}) &= \{\text{del}_f(\xi)\} \triangleright \{\neg p(\omega)\}_{\forall p \in \Psi, f=p(\text{pars}(\xi))}, \\ &\quad \{\text{add}_f(\xi)\} \triangleright \{p(\omega)\}_{\forall p \in \Psi, f=p(\text{pars}(\xi))}, \\ &\quad \{\text{mode}_{\text{edit}}\} \triangleright \{\neg \text{mode}_{\text{edit}}\}, \\ &\quad \{\emptyset\} \triangleright \{\text{mode}_{\text{val}}\}. \end{aligned}$$

When the observation  $\tau$  includes observed actions, then the extra conditional effects  $\{\text{at}_i, \text{plan}(\text{name}(a_i), \Omega^{ar(a_i)}, i)\} \triangleright \{\neg \text{at}_i, \text{at}_{i+1}\}_{\forall i \in [1,n]}$  are included in the  $\text{apply}_{\xi,\omega}$  actions to validate that actions are applied, exclusively, in the same order as they appear in  $\tau$ .

```

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

```

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

	$q_0$	$q_1$	$q_2$	$q_3$	$q_4$	$q_5$
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 5: Seven-symbol six-state *Turing Machine* for recognizing the  $\{a^n b^n c^n : n \geq 1\}$  language ( $q_5$  is the only acceptor state).

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

$$\begin{aligned} \text{pre}(\text{validate}_j) &= s_j \cup \{\text{test}_{j-1}\}, \\ \text{cond}(\text{validate}_j) &= \{\emptyset\} \triangleright \{\neg \text{test}_{j-1}, \text{test}_j, \neg \text{mode}_{val}\}. \end{aligned}$$

## Evaluation

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

## Modeling Turing Machines with STRIPS

A *Turing machine* is a tuple  $\mathcal{M} = \langle Q, q_0, Q_\perp, \Sigma, \Upsilon, \square, \delta \rangle$ :

- $Q$ , 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.
- $\Sigma$  is the *tape alphabet*, that is a finite non-empty set of symbols that contains the *input alphabet*  $\Upsilon \subseteq \Sigma$  (the subset of symbols allowed to initially appear in the tape) and the *blank symbol*  $\square \in \Upsilon$  (the only symbol allowed to occur on the tape infinitely often).
- $\delta : \Sigma \times (Q \setminus Q_\perp) \rightarrow \Sigma \times Q \times \{\text{left}, \text{right}\}$  is the *transition function*. For each possible pair of tape symbol and non-terminal machine state  $\delta$  defines (1), the tape symbol to print at the current position of the header (2), the new state of the machine and (3), whether the header is shifted *left* or *right* after the print operation. If  $\delta$  is not defined for the current pair of tape symbol and machine state, the machine halts.

Figure 5 shows the  $\delta$  function of a *Turing Machine* for recognizing the  $\{a^n b^n c^n : n \geq 1\}$  language. The *tape alphabet* is  $\Sigma = \{a, b, c, x, y, z, \square\}$ , the *input alphabet*  $\Upsilon = \{a, b, c, \square\}$  and the possible machine states are  $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$  where  $q_5$  is the only acceptor state.

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

- Fluents  $F$  are instantiated from a set of four *predicates*  $\Psi$ : (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

```

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

state. Given a set of *objects*  $\Omega$  that represent the cells in the tape of the given Turing Machine, the set of fluents  $F$  is induced by assigning objects in  $\Omega$  to the arguments of the predicates in  $\Psi$ .

- Actions  $A$  are instantiated from STRIPS operator schema. For each transition in  $\delta$ , a STRIPS schema is defined:
  - The **header** is `transition-id(?xl ?x ?xr)` where *id* uniquely identifies the transition in  $\delta$ . Parameters *?xl*, *?x* and *?xr* are tape cells.
  - The **preconditions** are `(head ?x)` and `(next ?xl ?x) (next ?x ?xr)` to make *?x* the tape cell pointed by the header and *?xl*/*?xr* its left/right neighbors. Preconditions `(symbol-σ ?x)` and `(state-q)` are also included to capture the symbol currently pointed by the header and the current machine state.
  - The **delete effects** remove the symbol currently pointed by the header and the current machine state while the **positive effects** set the new symbol pointed by the header and the new machine state according to  $\delta$ .

The STRIPS action schema of Figure 6 models the rule  $a, q_0 \rightarrow x, r, q_1$  of the *Turing Machine* defined in Figure 5 (the full encoding of the *Turing Machine* of Figure 5 produces a total of sixteen STRIPS action schema).

Since the *transition function* of a *Turing Machine* can be encoded as a classical planning frame  $\Phi = \langle F, A \rangle$ , executions of that *Turing Machine* are definable as an action sequence  $\langle a_1, \dots, a_n \rangle$  that induces the *state trajectory*  $\langle s_0, s_1, \dots, s_n \rangle$  such that  $a_i$  ( $1 \leq i \leq n$ ) is applicable in  $s_{i-1}$  and generates the successor state  $s_i = \theta(s_{i-1}, a_i)$ .

## Experimental setup

We randomly generated a  $M = \{\mathcal{M}_1, \dots, \mathcal{M}_{100}\}$  set of one-hundred different *Turing Machines* where each  $\mathcal{M} \in M$  is a seven-symbol six-state *Turing Machine*. The classical planning frame  $\Phi = \langle F, A \rangle$  is the same for all the *Turing Machines*, there is an  $a \in A$  action for each pair of tape symbol and non-terminal state machine, while the  $\theta(s, a)$  function is defined differently for each the machines (using a different STRIPS action model).

We randomly choose a machine  $\mathcal{M} \in M$  and produce the observation  $\tau$  of a fifty-step execution plan. Finally, we follow our *model recognition as planning* method to identify the *Turing Machine* that produced  $\tau$ . This experiment is repeated for different amounts of missing information in the input trace  $\tau$ : unknown applied transitions, unknown internal machine state and unknown values of several tape cells.

**Reproducibility** MADAGASCAR is the classical planner we used to solve the instances that result from our compilations for its ability to deal with dead-ends (Rintanen 2014). Due to its SAT-based nature, MADAGASCAR can apply the actions for editing preconditions in a single planning step (in parallel) because there is no interaction between them. Actions for editing effects can also be applied in a single planning step, thus significantly reducing the planning horizon.

The compilation source code, evaluation scripts and benchmarks (including the used training and test sets) are fully available at this anonymous repository so any experimental data reported in the paper can be reproduced.

## Results

## Conclusions

## 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.; 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.
- Chakraborti, T.; Sreedharan, S.; Zhang, Y.; and Kambhampati, S. 2017. Plan explanations as model reconciliation: Moving beyond explanation as soliloquy. In *IJCAI*.
- 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 Artificial Intelligence*, 1778–1783.
- Ramírez, M. 2012. *Plan recognition as planning*. Ph.D. Dissertation, Universitat Pompeu Fabra.
- Rintanen, J. 2014. Madagascar: Scalable planning with sat. *Proceedings of the 8th International Planning Competition (IPC-2014)*.
- Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2017. Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence, ICAPS-17*.
- 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.