




# Automated Acquisition of Control Knowledge for Classical Planners

Marta Vomlelová<sup>1</sup><sup>a</sup>, Jindřich Vodrážka<sup>1</sup>, Roman Barták<sup>1</sup><sup>b</sup> and Lukáš Chrpa<sup>1,2</sup><sup>c</sup>

<sup>1</sup>Faculty of Mathematics and Physics, Charles University, Czech Republic

<sup>2</sup>Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

**Keywords:** Automated Planning, Control Knowledge, Acquisition, Finite State Automata, PDDL.

**Abstract:** Attributed transition-based domain control knowledge (ATB-DCK) has been proposed as a simple way to express expected (desirable) sequences of actions in a plan with constraints going beyond physics of the environment. This knowledge can be compiled to Planning Domain Description Language (PDDL) to enhance an existing planning domain model and hence any classical planner can exploit it. In the paper, we propose a method to automatically acquire this control knowledge from example plans. First, a regular expression representing provided plans is found. Then, this expression is extended with attributes expressing extra relations among the actions and hence going beyond regular languages. The final expression is then translated, through ATB-DCK, to PDDL to enhance a planning domain model. We will empirically demonstrate that such an enhanced domain model improves efficiency of existing state-of-the-art planning engines.


## 1 INTRODUCTION


Domain Control Knowledge (DCK) is an approach to describe useful information for planning that is going beyond the description of physical interaction among the actions. Automated planners can be built to exploit domain control knowledge, for example in the form of control rules (Kabanza et al., 1997) or hierarchical task networks (HTNs) (Erol et al., 1996), or DCK can be compiled to the domain model, for example in Planning Domain Description Language (PDDL) (McDermott et al., 1998), so any automated planner (using PDDL as its input) can exploit it (Chrpa and Barták, 2016). Control rules, represented by, for instance, Linear Temporal Logic (Kvarnström and Doherty, 2000; Bacchus and Kabanza, 2000), describe desired evolution of states of the environment and state sequences violating the rules are pruned by a forward-search algorithm. The idea behind HTNs is that the planning domain model is described as a hierarchy of tasks, where the top task represents the goal to reach and the task is solved by decomposing it to sub-tasks and so on until primitive tasks – actions – are obtained (alternative decompositions might be used). Since HTN planning is more expressive than classical planning, only a fragment HTN formalism such as tasks consisting of fully ordered primitive tasks (actions) and simple recursive


tasks can be compiled into classical planning (Alford et al., 2009; Alford et al., 2015).

The major motivation behind DCK is speeding-up the planing process by providing useful guidelines, for example, in the form of recommended action sequences to achieve specific goals. One of the major challenges here is how to obtain useful DCK. Currently, DCK is usually constructed by a human modeller, which is a tedious task as it requires expert knowledge not only about the domain but also about the planning techniques. It would be beneficial, if DCK is acquired automatically by using example plans that include typical sequencing of actions to achieve the goal. The DCK can encode such a knowledge that can be used later for other problems in the same domain.

Automated acquisition of domain models have a history with works such as ARMS (Wu et al., 2007) and LOCM (Cresswell et al., 2013). With regards to automated acquisition of DCK, in order to enhance the planning process, there are several techniques that are being leveraged. Probably the best known technique is generating macro-operators that represent sequences of ordinary operators (Botea et al., 2005; Chrpa et al., 2014; Chrpa and Vallati, 2019). Macro-operators, roughly speaking, stand for “short-cuts” in the state space, therefore, planning engines can find plans in fewer steps. On the other hand, only static operator sequences that frequently occur in plans can be encoded as macro-operators. Another work concerns learning control rules, which describe what states or

<sup>a</sup>  <https://orcid.org/0000-0001-9104-804X>

<sup>b</sup>  <https://orcid.org/0000-0002-6717-8175>

<sup>c</sup>  <https://orcid.org/0000-0001-9713-7748>

their sequences are admissible (Yoon et al., 2008). Closer to our proposal, there is a technique that learns planning programs, which encode routines in form of sequences of instructions for plan generation (Aguas et al., 2019). Similarities can be found also with workflow mining research (Yaman et al., 2009).

In this paper, we address the problem of fully automated acquisition of domain control knowledge from example plans. Even a single example plan may be used for this purpose, which is a great advantage over machine learning techniques dependent on huge amount of data. The learned control knowledge is encoded in the form of attributed finite state automaton. The non-deterministic automaton models the suggested sequences of actions and the attributes are used to pass additional information between the actions (for example, about the objects participating in the actions). The acquired knowledge can then be compiled back to the PDDL domain model and hence any PDDL-based planner can immediately exploit it.

The paper is organized as follows. We will first give necessary background on automated planning. Then we will describe a method of constructing finite-state automaton (regular expression) capturing sequencing of actions in example plans. After that we enhance this automaton by adding attributes to actions that pass information about common objects between the actions. Finally, we will empirically justify that the proposed method of generating control knowledge indeed improves performance of classical planners.

## 2 BACKGROUND ON PLANNING

Classical STRIPS planning (Fikes and Nilsson, 1971) deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal conditions. World states are modelled as sets of propositions that are true in those states, and actions are modelled to change the validity of certain propositions.

Formally, let  $P$  be a set of all propositions modelling properties of world states. Then a state  $S \subseteq P$  is a set of propositions that are true in that state (every other proposition is false).

Each action  $a$  is described by three sets of propositions  $(B_a^+, A_a^+, A_a^-)$ , where  $B_a^+, A_a^+, A_a^- \subseteq P, A_a^+ \cap A_a^- = \emptyset$ . Set  $B_a^+$  describes positive preconditions of action  $a$ , that is, propositions that must be true right before applying the action  $a$ . Some modeling approaches allow also negative preconditions, but these preconditions can be compiled away. Action  $a$  is applicable to state  $S$  iff  $B_a^+ \subseteq S$ . Sets  $A_a^+$  and  $A_a^-$  describe positive and negative effects of action  $a$ , that is, propositions that

will become true and false in the state right after executing the action  $a$ . If an action  $a$  is applicable to state  $S$  then the state right after the action  $a$  is:

$$\gamma(S, a) = (S \setminus A_a^-) \cup A_a^+. \quad (1)$$

while  $\gamma(S, a)$  is undefined if an action  $a$  is not applicable to state  $S$ .

The classical planning problem, also called a STRIPS problem, consists of a set of actions  $A$ , a set of propositions  $P$ , an initial state  $S_0$ , and a set of goal propositions  $G^+$  describing the propositions required to be true in the goal state (again, negative goal is not assumed as it can be compiled away). A solution to the planning problem is a sequence of actions  $a_1, a_2, \dots, a_n$  such that  $S = \gamma(\dots\gamma(\gamma(S_0, a_1), a_2), \dots, a_n)$  and  $G^+ \subseteq S$ . This sequence of actions is called a *plan*.

Rather than specifying each action separately, it is common to specify an action schema or an operator with attributes describing specific objects that the action deals with. The action is then obtained by substituting constants for these attributes. Figure 1 shows an example of such an operator in PDDL (Planning Domain Description Language that is a de-facto standard language to model planning problems).

```
(:action drive_tent
:parameters ( ?x1 - person ?x2 - place ?x3 - place
?x4 - car ?x5 - tent )
:precondition (and (at_tent ?x5 ?x2)
(at_car ?x4 ?x2)
(at_person ?x1 ?x2)
(down ?x5) )
:effect (and (not (at_person ?x1 ?x2))
(not (at_car ?x4 ?x2))
(not (at_tent ?x5 ?x2))
(at_tent ?x5 ?x3)
(at_person ?x1 ?x3)
(at_car ?x4 ?x3) )
)
```

Figure 1: Example of planning operator in PDDL.

A part of a plan, which can be generated by domain-independent planning engines, is presented in Figure 2. Only a sequence of actions and the attribute assignment is provided. The full body with preconditions and effect can be derived from this information and the operator definition (Figure 1) and stays not listed until compiling the DCK to PDDL in Section 4.3.

```
(put_down guy0 place1 TENT0)
(drive_tent guy2 place1 place2 car0 TENT0)
(put_up guy2 place2 tent0)
(drive guy0 place1 place2 CAR1)
(drive_passenger guy0 place2 place1 car0 GUY2)
(walk_together tent0 place2 guy2 place1 girl2 COUPLE2)
```

Figure 2: Example of (a part of) a plan.

### 3 CONSTRUCTING REGULAR EXPRESSION FROM EXAMPLE PLANS

In this Section, we consider a plan as a sequence of action names only (i.e., without attributes). Such a plan can be seen as a string of a regular language.

Common algorithms learning deterministic regular grammars use either full set of positive and negative examples or a stochastic language (Carrasco and Oncina, 1999) represented by positive examples that are assumed to be random samples. In our case, we do not assume rich set of examples that guarantees stochastic properties. We do not assume complete list of positive examples therefore we are not able to identify negative examples either. On the other hand, we do not aim to perfectly distinguish positive and negative examples. We aim to restrict the search space of the planner. That is, to find a domain knowledge that covers all positive examples tight. In brief, we heuristically select a 'milestone' action and assume the automaton state after this action is always the same with the exception of the first and the last occurrence of the action. If no such action can be found, we allow any of the remaining actions and leave the search on the planner.

Our method for learning domain control knowledge starts with capturing typical action sequences in the form of a regular expression (equivalent to a finite state automaton). At this stage we use action names only and we ignore attributes of actions.

We are given a set of example plans – a single plan per problem is sufficient – these plans should describe typical sequences of actions to achieve the goal. We select an action that appears in all input plans and based on that action we split each plan into three parts. One part is a piece of plan before the first occurrence of that action (the same action may appear at several positions in the plan). One part is a piece of plan after the last occurrence of the action. The final part consists of continuous sub-plans between subsequent occurrences of the action. Note that all these sub-plans do not contain the selected action. Figure 3 shows how the splitting is done for a single plan.

The plans before the selected action are put together, and similarly the plans after the action, and finally the plans between occurrences of the action. On each of these sets of plans we apply recursively the same learning process. The process is stopped when there is no action that appears in all the plans. Let  $J$  be all actions from these plans, we construct a regular expression  $J^+$  for these plans (if there is an empty plan among the input plans, the expression is  $J^*$ ; if there is exactly one action in each plan, the expression is  $J$ ).

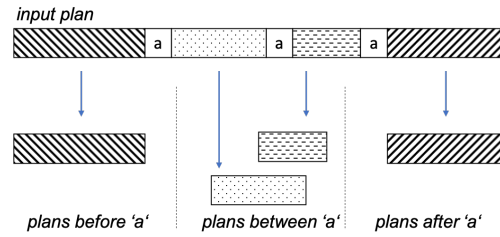


Figure 3: Splitting the plan to three sections based on selected action  $a$ .

Let  $I$  be the regular expression obtained for plans before the selected action  $a$ ,  $E$  be the regular expression for plans after  $a$ , and  $M$  be the regular expression for plans between the occurrences of  $a$ . Then we construct a regular expression  $I.a.(M.a)^*.E$  for plans with  $a$  (under several conditions a simplified expression is generated, see Algorithm 1. For example, for a plan 'drive,walk,walk,walk,drive' an expression 'drive.walk.(walk)<sup>+</sup>.drive' is generated.

Algorithm 2 is a formal description of the regular expression construction process. One may ask, which action to split the plans is selected in case there are more actions appearing in all plans. Based on preliminary experiments we use the action with the smallest number of occurrences, but a deeper study about which action to select is needed.

Algorithm 1: An expression generated from a split action  $a$  and sup-expressions  $I, M, E$ . Specific number of occurrences of  $a$  in the plans  $Plans$  enables simplified versions of the expression.

```

1: procedure EXP( $a, I, M, E, Plans$ )
2:    $o \leftarrow$  the number of occurrence of  $a$  in all plans
   and switch
3:   if  $o$  is always 1 then
4:     return  $I.a.E$ 
5:   else if  $o$  is always 2 then
6:     return  $I.a.M.a.E$ 
7:   else if  $o$  is always more than 1 then
8:     return  $I.a.(M.a)^+.E$ 
9:   else
10:    return  $I.a.(M.a)^*.E$ 
11:  end if
12: end procedure

```

Figure 4 shows a regular expression generated for plans in the classical domain Hiking used in International Planning Competitions. We added attributes to actions as we shall now show how to connect these attributes from different actions (actually, the regular expression already presents attributes shared by actions). The first action selected to split is `put_down`, then `put_up`, then in some branches `drive_tent` in other branches `drive_passenger`.

Algorithm 2: Algorithm for constructing a regular expression describing a set of plans.

```

procedure REGEXPGEN(Plans - list of plans)
2:   if exists an action in all plans then
      a ← select an action in all plans
4:   for each plan in Plans do
      parts ← plan.split(a)
6:   I ← RegExpGen([parts.first()])
      E ← RegExpGen([parts.last()])
8:   M ← RegExpGen(parts[1:len-1])
      return Exp(a,I,M,E,Plans)
10:  end for
12:  else
      join present actions to a set J
      return J, (J)+ or (J)* as appropriate
14:  end if
end procedure

```

```

0 drive_tent {0: 'guy0', 2: 'place1', 4: 'tent3'}
1 put_up {0: 'guy0', 1: 'place1', 2: 'tent3'}
2 drive {1: 'place0', 2: 'place1'}
3 drive_passenger {1: 'place1', 2: 'place0'}
4 walk_together {0: 'E', 1: 'place1', 3: 'place0'}
5 (walk_together)+ {0: 'E', 1: 'place1', 3: 'place0'}
6 drive_passenger {1: 'place1', 2: 'place0'}
7 drive {1: 'place0', 2: 'place1'}
8 drive {1: 'place0', 2: 'place1'}
9 put_down {0: 'girl0', 1: 'place1', 2: 'F'}
10 (drive_tent {0: 'girl0', 1: 'place1', 2: 'place2', 4: 'F'})
11 put_up {0: 'girl0', 1: 'place2', 2: 'F'}
12 drive {0: 'I', 1: 'H', 2: 'place2'}
13 drive_passenger {1: 'place2', 2: 'H', 4: 'I'}
14 walk_together {0: 'D', 1: 'place2', 3: 'H'}
15 (walk_together)+ {0: 'D', 1: 'place2', 3: 'H'}
16 drive_passenger {1: 'place2', 2: 'H'}
17 drive {1: 'H', 2: 'place2'}
18 drive {1: 'H', 2: 'place2'}
19 put_down)+ {1: 'place2', 2: 'C'}

```

Figure 4: Attributed Regular Expression Example.

## 4 CONNECTING ATTRIBUTES IN ACTIONS

Assume now, that we got a regular expression with sequence of actions `put_down`, `drive_tent`, `put_up` describing tent transport. If action attributes are ignored then these actions may deal with different tents and different locations and, hence, the action sequence may be meaningless. The meaning of the control knowledge was that we `put_down` some `?tent` at certain location `?from`, then we drive that tent to another location `?to`, where we put that tent up. This information can be encoded via shared attributes of the actions, for example:

```

(put_down ? ?from ?tent)
(drive_tent ? ?from ?to ? ?tent)
(put_up ? ?to ?tent)

```

In this section we address the problem how to connect the attributes of actions or, in other words, how to find equivalence classes of actions' attributes.

Figure 4 shows the final equivalence classes as part of the regular expression, so we will use it to explain the notation used. Each line contains exactly one action (and possibly some delimiters '(', ')', '+'). We number the attributes of each action starting with 0, Figure 4 shows only the attributes that are shared between the actions (for example, attributes 1 and 3 are missing in the first action `drive_tent` as they are not shared with other actions). The group of shared attributes is identified by a specific label. For example, the label `place1` tells: the attribute 2 of action 0 should be passed as attribute 1 to action 1 and then as attribute 2 in action 2 etc. The same attribute is used up to the line 10 as the attribute 1 of action `drive_tent` and then forgotten (not used in the rest of the plan).

This shared attribute is represented by the label `place1` and the set of pairs  $\{(0,2), (1,1), (2,2), (3,1), (4,2), \dots, (10,1)\}$ , where each pair means (*actionLine*, *attributePosition*). We will call the label with the corresponding set an *equivalence class*. A set of equivalence classes is called a *pattern*.

While we construct the underlying regular expression top-down, the pattern is constructed bottom-up, starting from the elementary parts of the regular expression.

The same recursive tree structure as in Algorithm 1 is followed. Compared to Algorithm 1, the sub-plan is extended by the immediate predecessor and the immediate successor (if they exist).

Consider the example in Table 1. The first action to split is `put_down`. Assume the action appears four times in the plan, therefore three sub-plans  $P_1^M, P_2^M, P_3^M$  are sent to the middle part in the brackets. The second split in the middle part is `put_up`, a single remaining  $I_2$  is action `drive_tent`. The sequence of three actions, '`put_down, drive_tent, put_up`', is sent to the leaf '`drive_tent`' for each of the three sub-plans  $P_1^M, P_2^M, P_3^M$ . These action triples are listed in Table 2 (we will denote them as  $P_1, P_2, P_3$ ).

Table 1: Regular expression evolution: at the first level, action `put_down` is chosen for the split. On the second level in the middle is the split `put_up`.  $I_2$  contains a single action `drive_tent` in all plans.

$I_1$ .	<code>put_down.</code>	(	$M_1$ .	<code>put_down</code> )*.	$E_1$
$I_1$ .	<code>put_down.</code>	(	$I_2$ .	<code>put_up.</code>	$E_2$ .
$I_1$ .	<code>put_down.</code>	(	<b>drive_tent.</b>	<code>put_up.</code>	$E_2$ .
				<code>put_down</code> )*.	$E_1$

Table 2: Elementary Pattern Generation.

$P_1$	(put-down g0 p0 t0)	(drive-tent g0 p0 p1 c0 t0)	(put-up g0 p1 t0)
$P_2$	(put-down g0 p1 t0)	(drive-tent g0 p1 p2 c0 t0)	(put-up g0 p2 t0)
$P_3$	(put-down g0 p2 t4)	(drive-tent g0 p2 p3 c0 t0)	(put-up g0 p3 t0)
$\mathcal{P}$	(put-down 0:G 1:X)	(drive-tent 0:G 1:X 2:Y 4:T)	(put-up 0:G 1:Y 2:T)

First, we construct an elementary pattern from the sub-plan  $P_1$ . Then, we apply function `AddToPattern` to all remaining sub-plans  $P_2, P_3$ . The same is done in the leaf  $E$ . Then, patterns from both leaves are combined together by the function `ExtendPattern`. Further, patterns are recursively extended to the top level.

#### 4.1 Elementary Pattern Construction

Assume that we have three plans in the plan list,  $P_1, P_2$  and  $P_3$  as depicted in Table 2. For each plan, we identify the equivalence classes by tracing the attributes with the same constant. In plan  $P_1$  we have constants (objects)  $g0, p0, \dots, p4, c0$  and  $t0$ . Four of them –  $g0, p0, p1$ , and  $t0$  – appear more times among the attributes of actions so we will get a pattern with four equivalence classes ' $G$ ':(put-down,0)(drive-tent,0),(put-up,0), ' $X$ ':(put-down,1)(drive-tent,1), ' $Y$ ':(drive-tent,2),(put-up,2) and ' $T$ ':(put-down,2)(drive-tent,4),(put-up,3) (for clarity, we use the action names there rather than the action numbers). We can construct a similar pattern for the second plan, the only difference is that it uses different constants ( $p1$  and  $p2$ ), but the pattern structure is identical. So when we intersect the patterns, we preserve the equivalence classes. Now, we add  $P_3$ . Since the object ' $p2$ ' covers the equivalence class (put-down,1)(drive-tent,1), the original group ' $X$ ' is preserved. On the other hand, the objects at positions (put-down,2),(drive-tent,4) differ and therefore the group ' $T$ ' is split to a class ' $T$ ':(drive-tent,4),(put-up, 3) a singleton (put-down,2) (which is removed as it does not represent any information passing between actions). The last line in Table 2 shows the regular expression with the pattern found. The reader may notice that constant  $g0$  appears in all plans as the second argument of the first action. This can also be reflected in the pattern (meaning that a given constant is always used at given attributes).

Algorithm 3 shows how the patterns of two plans are combined together. Basically, we intersect the equivalence classes constructed for both plans.

Algorithm 3: `AddToPattern: Pattern - pattern, Plan - partial plan.`

---

```

1: procedure ADDTOPATTERN(Pattern, Plan)
2:   partition  $\mathcal{P}_1 \leftarrow$  Pattern partition
3:   partition  $\mathcal{P}_2 \leftarrow$  Plan partition
4:   partition  $\mathcal{P}_{new} \leftarrow \{u \cap v; \text{ for } u \in \mathcal{P}_1, u \in \mathcal{P}_2\}$ 
5:   return  $\mathcal{P}_{new}$ 
6: end procedure

```

---

#### 4.2 Pattern Concatenation

Consider Table 1 again. We have a pattern for the leaf 'drive-tent'. Assume we have also a pattern for the sub-tree  $E$  representing a general sub-tree. We need to concatenate the patterns. In particular, in the bottom-up traverse of the regular expression, initial, middle and the end parts are concatenated in Algorithm 1. These patterns share the split action  $a$ , so we specifically combine equivalence classes containing this action. See Table 3 for an example.

Table 3: Example of Pattern Concatenation.

$\mathcal{P}_1$	(put-up X1 t4 X2)	(walk X3 X1 X4)	
$\mathcal{P}_2$		(walk Y1 Y2 Y3)	(drive Y2 Y4)
	(put-up Y2 t4 X2)	(walk Y1 Y2 Y3)	(drive Y2 Y4)

There is an equivalence class X1: (put-up, 0), (walk, 1) from the first expression and an equivalence class Y2: (walk, 1), (drive, 0) from the second expression. By joining them, we get Y2: (put-up, 0), (walk, 1), (drive, 0) (any label of the class can be chosen). Algorithm 4 describes the concatenation process.

Algorithm 4: `ExtendPattern:  $P_{left}, P_{right}$  - Patterns.`

---

```

1: procedure EXTENDPATTERN:( $P_{left}, P_{right}$ )
2:    $A \leftarrow$  first action of  $P_{right}$ 
3:    $P_{new} \leftarrow$  new pattern, initially empty
4:   shift all action indexes in  $P_{right}$  by the last action of  $P_{left}$ 
5:   for each attribute of  $A$  do
6:      $right \leftarrow$  class of ( $A, attribute$ ) in  $P_{right}$ 
7:      $left \leftarrow$  class of ( $A, attribute$ ) in  $P_{left}$ 
8:      $join \leftarrow left \cup right$ ; use the right label
9:     add  $join$  to  $P_{new}$ 
10:  end for
11:  rename classes in  $P_{left}$  to avoid duplicates
12:  add to  $P_{new}$  all classes from  $P_{left}, P_{right}$  not used for any join
13:  return  $P_{new}$ 
14: end procedure

```

---

#### 4.3 Compiling Attributed Regular Expressions to PDDL

To compile the attributed regular expression to a standard PDDL domain, we are inspired by the DCK representation (Chrapa and Barták, 2016). First, some pruning is performed. Assume an expression  $I.a.(M.a)^*.E$ . If  $E$  is an initial sub-sequence of  $M$  then  $E$  can be omitted. The accepting condition is a correct plan of the planning engine, the DCK has no final states and may end in the middle of  $M$  expression. Similarly, if  $I$  is a final sub-string of  $M$ , the

initial action of the DCK may 'jump' into  $M$  avoiding separate listing of  $I.a$ . This way, we make the automaton smaller, which is important for efficiency of planning.

Then, for each line  $ID$  in the regular expression a new predicate 'DCK\_ $(ID)$ ' is introduced. The attributes to be carried from the previous action and to the next action can be recognized from the pattern. For a pattern in Table 3, class  $Y2$ , that is (put-up,0) is taken into walk operator and class  $Y2:(walk,1)$  is stored for the next action *drive*. In general, more than one attribute is stored.

For example, the state 'DCK\_10' derived from Figure 4 has three attributes, 'girl0', 'place1' and 'F'. This state is used as a precondition of action at the given line, *drive\_tent* in this case. Note also, that we include the line index in the name of new operator *drive\_tent10* as that operator may appear at several locations in the regular expression. The same action deletes the old state and introduces the next state among its effects – in this example, the next state 'DCK\_11' is used with attributes 'girl0', 'place2' and 'F'. Figure 5 shows such a modified operator.

```
(:action drive_tent10
  :parameters ( ?x1 - person ?x2 - place ?x3 - place
                ?x4 - car ?x5 - tent )
  :precondition (and (at_tent ?x5 ?x2)
                    (at_car ?x4 ?x2)
                    (at_person ?x1 ?x2)
                    (down ?x5) (DCK_10 ?x1 ?x2 ?x5) )
  :effect (and (not (at_person ?x1 ?x2))
              (not (at_car ?x4 ?x2))
              (not (at_tent ?x5 ?x2))
              (not (DCK_10 ?x1 ?x2 ?x5))
              (DCK_11 ?x1 ?x3 ?x5)
              (at_tent ?x5 ?x3)
              (at_person ?x1 ?x3)
              (at_car ?x4 ?x3) )
)
```

Figure 5: Example of translated planning operator in PDDL.

## 5 EMPIRICAL EVALUATION

To evaluate the proposed technique we did a preliminary experiment with a single domain and several planners. The aim of this experiment is verifying whether the learned domain control knowledge helps planners to improve efficiency and what type of control knowledge contributes most.

We used the Hiking domain for which we manually constructed domain control knowledge that we then compiled to the domain model. In the experiments, we used the original domain model without any control knowledge (called 'original'). The handwritten attributed domain control knowledge added to the domain model is named 'atbDck'. The model called 'RegExp' is the original domain model extended just with the regular expression (the first stage

of learning) without any attributes connected. The full learned regular expression with all identified attributes is denoted 'fullLearned'. Finally, we constructed a domain model, where only the attributes in the regular expression that last more than two subsequent actions were kept. We name this model 'Restricted'. We used 20 plans to learn the domain control knowledge.

We used three different planners to compare their efficiency on constructed domain models: FF, Madagascar, and Probe. FF has been selected as it is a standard heuristic-search planner and many other planners are built on top of it. Madagascar is a very different type of planner that is doing parallel planning (tries to plan several actions in a single parallel step) and uses SAT-based type of planning. Probe is also a different style of planner based on novelty search. The computer used for experiments was equipped with Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz and the time limit for solving each problem was 900 seconds.

We used 20 problems from the same domain to evaluate different versions of control knowledge. Three measures were evaluated: *coverage* counts the number of solved problems, *IPC Score* takes the shortest solution of a given problem  $N_p^*$  and for any configuration  $C$  reports  $N_p^*/N_p(C)$  if the problem was solved, 0 otherwise. *PAR10* takes the time of the solution, if the problem was solved, and ten times the time limit, that is 9000, otherwise. IPC Score combines information about the number of solved problems and their quality (shorter plans preferred) and hence provides slightly more information than coverage only. For each problem we calculated the above scores and in Table 4 we report their sums over all 20 problems. For PAR10, the lower values are better (less runtime needed), for the other two scores, the higher values are better (more problems solved in better quality).

From these preliminary results, we can already draw some conclusions. First, the added control knowledge indeed helps (to some planners) and this knowledge can be learned automatically from a small number of example plans. Second, the contribution of control knowledge depends a lot on a planner used. For Madagascar, the parallel planner, the sequential control knowledge actually hurts performance. This is not surprising, as Madagascar tries to put as many parallel actions as possible to a single step, while, the learned control knowledge consists of additional sequencing constraints and hence forces more parallel steps. For forward-planning based planners, this control knowledge is however beneficial. The automatically learned knowledge seems even comparable to carefully designed manual control knowledge, which is a particularly promising result. The third observa-

Table 4: Empirical results: PDDL solvers Madagascar, FF and Probe on models with different control knowledge compared using PAR10, Coverage and IPC scores.

<b>PAR10</b>	ff	mada	probe
original	5052.58	<b>6751.06</b>	609.64
atbDck	<b>0.04</b>	8550.09	2261.26
RegExp	6346.84	8550.09	2805.19
fullLearned	1.50	7200.99	6339.99
Restricted	8.91	7200.77	<b>83.41</b>
<b>Coverage</b>	ff	mada	probe
original	9	<b>5</b>	19
atbDck	<b>20</b>	<b>5</b>	15
RegExp	6	1	14
fullLearned	<b>20</b>	4	6
Restricted	<b>20</b>	4	<b>20</b>
<b>IPC Score</b>	ff	mada	probe
original	0.45	0.13	0.80
atbDck	0.89	<b>0.18</b>	0.63
RegExp	0.14	0.03	0.34
fullLearned	<b>0.90</b>	0.14	0.20
Restricted	<b>0.90</b>	0.14	<b>0.88</b>

tion is that the granularity of control knowledge is also important. Using just the finite state automaton (regular expression) actually hurts performance as it brings additional constraints to action sequencing, but (probably) does not guide the planner regarding which objects participate in the actions. On the other side, adding too much additional connections between the attributes of actions may add extra overhead that does not pay-off for some planners (Probe).

## 6 CONCLUSIONS

In this paper we proposed a fully automated method to acquire domain control knowledge from example plans. The control knowledge is expressed in the form attributed regular expression and it can be compiled back to the PDDL model and hence used by any contemporary automated planner. The regular expression is constructed in two steps. First, we build the core of the regular expression consisting of action names only. Second, we connect the attributes of actions to model information passing between the actions. The biggest advantage of the proposed method is that it does not require a large number of examples to learn from; even one plan is enough to construct the control knowledge.

The preliminary empirical evaluation confirmed that the obtained control knowledge indeed helps to improve efficiency of planning. It also showed that the effect is different for different planners and that the right granularity of control knowledge is also important.

The open problems to study further are which action to use for splitting the plans during construction of regular expressions and which attribute equivalence classes should be represented in control knowledge.

## ACKNOWLEDGEMENTS

This research was funded by the Czech Science Foundation under the project 18-07252S and by the OP VVV funded project CZ.02.1.01/0.0/0.0/16.019/0000765 “Research Center for Informatics”.

## REFERENCES

- Aguas, J. S., Celorrio, S. J., and Jonsson, A. (2019). Computing programs for generalized planning using a classical planner. *Artif. Intell.*, 272:52–85.
- Alford, R., Bercher, P., and Aha, D. W. (2015). Tight bounds for HTN planning. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, pages 7–15.
- Alford, R., Kuter, U., and Nau, D. S. (2009). Translating htns to PDDL: A small amount of domain knowledge can go a long way. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1629–1634.
- Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191.
- Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-ff: Improving AI planning with automatically learned macro-operators. *J. Artif. Intell. Res.*, 24:581–621.
- Carrasco, R. and Oncina, J. (1999). Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO - Theoretical Informatics and Applications*, 33.
- Chrpa, L. and Barták, R. (2016). Guiding planning engines by transition-based domain control knowledge. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016.*, pages 545–548.
- Chrpa, L. and Vallati, M. (2019). Improving domain-independent planning via critical section macro-operators. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 7546–7553.

- Chrapa, L., Vallati, M., and McCluskey, T. L. (2014). MUM: A technique for maximising the utility of macro-operators by constrained generation and use. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*.
- Cresswell, S., McCluskey, T. L., and West, M. M. (2013). Acquiring planning domain models using *LOCM*. *Knowledge Eng. Review*, 28(2):195–213.
- Erol, K., Hendler, J. A., and Nau, D. S. (1996). Complexity Results for HTN Planning. *Ann. Math. Artif. Intell.*, 18(1):69–93.
- Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2nd international joint conference on Artificial intelligence, IJCAI'71*, pages 608–620, San Francisco, CA, USA.
- Kabanza, F., Barbeau, M., and St-Denis, R. (1997). Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67 – 113.
- Kvarnström, J. and Doherty, P. (2000). TALplanner: a temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):119–169.
- McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL - The Planning Domain Definition Language. Technical Report TR-98-003, Yale Center for Computational Vision and Control.
- Wu, K., Yang, Q., and Jiang, Y. (2007). ARMS: an automatic knowledge engineering tool for learning action models for AI planning. *Knowledge Eng. Review*, 22(2):135–152.
- Yaman, F., Oates, T., and Burstein, M. (2009). A context driven approach for workflow mining. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI'09*, page 1798–1803, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Yoon, S. W., Fern, A., and Givan, R. (2008). Learning control knowledge for forward search planning. *J. Mach. Learn. Res.*, 9:683–718.