The Knowledge Engineering Review, Vol. 34, e5, 1–28. © Cambridge University Press, 2019
doi:10.1017/S0269888918000231

A review of generalized planning

SERGIO JIMÉNEZ[1], JAVIER SEGOVIA-AGUAS[2] and ANDERS JONSSON[2]

[1]*Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Camino de Vera s/n. 46022 Valencia, Spain; e-mail: serjice@dsic.upv.es;*
[2]*Information and Communication Technologies, Universitat Pompeu Fabra, Roc Boronat 138, 08018 Barcelona, Spain; e-mail: javier.segovia@upf.edu, anders.jonsson@upf.edu*

Abstract

Generalized planning studies the representation, computation and evaluation of solutions that are valid for multiple planning instances. These are topics studied since the early days of AI. However, in recent years, we are experiencing the appearance of novel formalisms to compactly represent generalized planning tasks, the solutions to these tasks (called *generalized plans*) and efficient algorithms to compute generalized plans. The paper reviews recent advances in generalized planning and relates them to existing planning formalisms, such as *planning with domain control knowledge* and approaches for *planning under uncertainty*, that also aim at generality.

1 Introduction

Automated Planning (AP) can solve complex deliberative tasks in highly structured environments by exploiting models of the agents and their environment (Ghallab *et al*., 2004; Geffner and Bonet, 2013). Traditionally the solutions generated by automated planners are tied to a particular planning instance and hence, do not generalize.

Generalized planning goes one step further and studies the computation of planning solutions that generalize over a set of planning instances. In the worst case, each instance in the set may require a different solution. In many cases, however, it is possible to compute a single compact solution that exploits some common structure of multiple planning instances.

A *generalized plan* is an algorithm-like solution that is valid for a given set of planning instances. An illustrative example is the following generalized plan for the well-known *blocksworld* domain (Slaney and Thiébaux, 2001), where the goal is to stack the blocks on each other in a given pattern. This generalized plan solves any instance in the domain, regardless of the number of blocks and the names of the blocks.

(1) Put all the blocks on the table.
(2) Move a block X on top of a block Y whenever (1) X and Y are clear; (2) X is supposed to be on Y in its goal position; and (3) Y is already at its goal position.

Figure 1 depicts three different blocksworld instances (named Prob1, Prob2 and Prob3 with two, three and four blocks, respectively) that are solvable by the previous generalized plan. For each instance, the figure shows the blocks configuration for the initial state (left side) and the corresponding goal state (right side).

The problem of computing general solutions for complex decision-making tasks has been studied since the early days of AI (Newell *et al*., 1959). In recent years we are experiencing a renewed interest caused by the appearance of novel formalisms for representing families of planning solutions, as well as new algorithms to compute such solutions. These advances reveal the potential of techniques from generalized

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

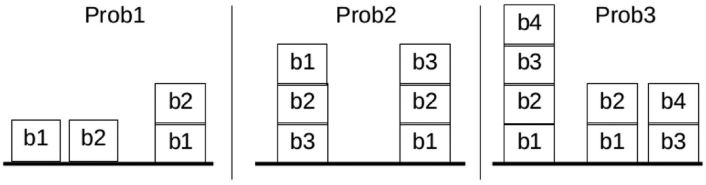

Figure 1 Three different example instances from *blocksworld*. Each instance shows the blocks configuration for the initial state (left side) and goal state (right side).

planning and encourage the application of planning to diverse areas of computer science such as *program synthesis*, *autonomous control*, *data wrangling* or *form recognition* (Torlak and Bodik, 2013; Alur *et al.*, 2015; Gulwani *et al.*, 2015).

This paper reviews these recent advances in generalized planning and relates them to existing formalisms that also aim at generality within AP, such as *planning with domain control knowledge* and different approaches for *planning under uncertainty*. First, the paper provides a background on AP, formalizes the generalized planning task, and introduces our criteria for reviewing the work on generalized planning. Second, the paper discusses different approaches for specifying sets of planning tasks. Third, the paper surveys diverse representation formalisms for generalized plans analyzing their strengths and weaknesses. Fourth, current algorithms for computing generalized plans are examined. Finally the paper ends discussing different implementations and identifying open research questions to encourage future research.

2 Background

This section introduces *classical planning* (the vanilla model for AP), proposes a formal model for generalized planning based on classical planning, and defines the framework we use to analyze the existing work on generalized planning.

2.1 Classical planning

The *classical planning model* is the most common model for AP, and is based on the following assumptions:

1. The planning task to solve has a finite and *fully observable* state space.
2. Actions are *deterministic* and cause instantaneous state transitions.
3. Goals are conditions referred to the last state reached by a solution plan.

Therefore, a solution to a classical planning instance is a sequence of applicable actions that transforms a given initial state into a goal state, that is, a state that satisfies a previously specified set of goal conditions (Geffner and Bonet, 2013).

Formally we use F to denote a set of propositional variables or *fluents* that together describe a state. A *literal* l is a valuation of a fluent $f \in F$, that is, $l = f$ or $l = \neg f$. A set of literals L on F is *well-defined* if there does not exist a fluent $f \in F$ such that $f \in L$ and $\neg f \in L$. Hence a well-defined literal set L assigns at most one value to each fluent in F, effectively representing a partial assignment of values to fluents. We use $\mathcal{L}(F)$ to denote the set of all well-defined literal sets on F. Given L, let $\neg L = \{\neg l : l \in L\}$ be its complement.

A *state* s is a literal set in $\mathcal{L}(F)$ such that $|s| = |F|$, that is, a total assignment of values to fluents. Explicitly including negative literals in states simplifies subsequent definitions, but we often abuse notation by defining a state s only in terms of the fluents that are true in s, as 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. Each action $a \in A$ has a precondition $\text{pre}(a) \in \mathcal{L}(F)$ and a set of effects $\text{eff}(a) \in \mathcal{L}(F)$. An action $a \in A$ is *applicable* in a given state s iff $pre(a) \subseteq s$, that is, if its precondition holds in s. The result of executing an applicable action $a \in A$ in a state s is a new state $\theta(s, a) = (s \backslash \neg\text{eff}(a)) \cup \text{eff}(a)$. Subtracting the complement of $\text{eff}(a)$ from s ensures that $\theta(s, a)$ remains a well-defined state.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

Given a frame $\Phi = \langle F, A \rangle$, a *classical planning instance* is a tuple $P = \langle F, A, I, G \rangle$, where $I \in \mathcal{L}(F)$ is an initial state (i.e. $|I| = |F|$) and $G \in \mathcal{L}(F)$ is a goal condition. A *plan* for P is an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ that induces a state sequence $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$ and, for each i such that $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The plan π *solves* P if and only if $G \subseteq s_n$, that is, if the goal condition is satisfied in the last state that is reached following the application of π in I.

The Planning Domain Definition Language (PDDL) (McDermott *et al.*, 1998) is the input language for the International Planning Competition (IPC) (Vallati *et al.*, 2015) and the *de facto* standard for representing classical planning instances. Besides classical planning, PDDL can represent more expressive planning models such as *temporal planning* or planning with *path constraints* and *preferences* (Fox and Long, 2003; Gerevini and Long, 2005).

PDDL separates the representation of a given planning instance into two parts, the *domain* and the *problem*:

- A PDDL *domain* defines *predicates* and *action schemas*, whose parameters are instantiated on objects to respectively form *fluents* and *ground actions*. Figure 2 shows the PDDL action schema unstack from blocksworld, whose effect is to unstack the top block from a tower of blocks (in PDDL a question mark denotes the start of a variable name and a semicolon denotes the start of a comment). Apart from unstack, the PDDL definition of the blocksworld domain includes three other action schemas: stack for stacking a block onto a tower of blocks and pick-up and put-down, for picking up a block from the table or putting a block down the table.
- A PDDL *problem* defines the *objects* of the planning instance, the *initial state* of these objects, and their *goal conditions*. Figure 3 shows the PDDL representation of the three classical planning instances illustrated in Figure 1.

Both the fluent set F and the action set A of a given planning problem are instantiated by assigning objects, from the *PDDL problem*, to the parameters of the predicates and action schemas (defined in the *PDDL domain*). For example, if the unstack action schema is instantiated with parameters $?x = b1$ and $?y = b2$, then pre(*unstack* $(b1, b2)$) = {(on b1 b2), (clear b1), (empty)}.

PDDL assumes that different instances belonging to the same domain share the same actions schemas, but this does not mean they share the same planning frame. For example, the three blocksworld instances shown in Figures 1 and 3 have different sets of objects, which induce different fluent and action sets.

2.2 Generalized planning

Generalized planning is often used as an umbrella term that refers to more general notions of planning, like the computation of plans with control flow structures, planning with domain control knowledge or diverse models for planning under uncertainty (such as conformant, contingent, Markov decision process (MDP) or partially observable Markov decision process (POMDP) planning (Geffner and Bonet, 2013)). This paper is a review of the work on generalized planning under the assumptions of *full state observability* and *deterministic actions*.

Definition 1. *A generalized planning instance is a finite set of classical planning instances* $\mathcal{P} = \{P_1, \ldots, P_T\}$ *that share some common structure.*

```
;;; PDDL definition of the unstack action schema
(:action unstack
  :parameters (?x ?y - block)
  :precondition (and (on ?x ?y) (clear ?x) (empty))
  :effect (and (hold ?x) (clear ?y)
               (not (clear ?x)) (not (empty)) (not (on ?x ?y))))
```

Figure 2 Action schema unstack from the blocksworld coded in Planning Domain Definition Language (PDDL).

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

```
;;; PDDL definition of problem Prob1
 (define (problem Prob1)
  (:domain blocks)
  (:objects b1 b2 - block )
  (:init (clear b1) (ontable b1) (clear b2) (ontable b2) (empty))
  (:goal (and (on b2 b1))))

;;; PDDL definition of problem Prob2
(define (problem Prob2)
  (:domain blocks)
  (:objects b1 b2 b3 - block )
  (:init (clear b1) (on b1 b2) (on b2 b3) (ontable b3) (empty))
  (:goal (and (on b3 b2) (on b2 b1))))

;;; PDDL definition of problem Prob3
(define (problem Prob3)
  (:domain blocks)
  (:objects b1 b2 b3 b4 - block )
  (:init (clear b4) (on b4 b3) (on b3 b2) (on b2 b1) (ontable b1) (empty))
  (:goal (and (ontable b1) (on b2 b1) (ontable b3) (on b4 b3))))
```

Figure 3 Three planning problems from the blocksworld domain coded in Planning Domain Definition Language (PDDL).

Previous approaches to compute general knowledge for AP, such as macro-actions (Fikes *et al*., 1972), case-based planners (Borrajo *et al*., 2015) or even the learning track of the IPC (Fern *et al*., 2011), assumed that the given set of classical planning instances shares the same predicates and action schemes.

More recent work imposes a stronger constraint on the classical planning instances in a given generalized planning task, they must share the set of fluents and the set of actions. Formally, the $\{P_1, \ldots, P_T\}$ instances in $\mathcal{P}$ belong to the same planning frame Φ and hence, $P_1 = \langle F, A, I_1, G_1 \rangle, \ldots, P_T = \langle F, A, I_T, G_T \rangle$ share the same set of fluents and actions and differ only in the initial state and goals. This constraint forces the set of planning instances in a generalized planning task to share the same *state space*. Note that this definition of generalized planning still makes it possible to encode instances $P \in \mathcal{P}$ with different number of objects fixing their irrelevant fluents to False. For instance, when defining the two-block planning task illustrated in Figure 1, any fluent referred to blocks $b3$ and $b4$ is set to False.

A *generalized plan* can be viewed as a procedural representation of the instances in a generalized planning task. *Generalized plans* are then generative models that may have diverse forms. Each form with its own expressiveness capacity and own computation and validation complexity. Generalized plans range from programs (Winner and Veloso, 2003; Segovia-Aguas *et al*., 2016a) and *generalized polices* (Martn and Geffner, 2004) to *Finite State Controllers* (FSCs) (Bonet *et al*., 2010; Segovia-Aguas *et al*., 2016b), AND/OR graphs, formal grammars (Ramirez and Geffner, 2016) or hierarchical task networks (HTNs) (Nau *et al*., 2003). We can classify generalized plans according to their specification of *the action to apply next*:

- *Fully specified* solutions, that *unambiguously specify the action to apply next*, for solving every instance in a given generalized planning task. Programs, generalized policies or deterministic FSCs belong to this class. Conformant, contingent or POMDP plans belong also to this class (if we consider that the possible initial states represent different classical planning instances all sharing the same state variables, actions and goals (Hu and Giacomo, 2011)).
- *Non specified*. In this case the action to apply next is not explicitly specified. For instance, a classical planner provided with a domain model is a *non-specified* generalized plan. Such a plan is very general (covers any instance representable in the classical planner's input language) but has an inefficient execution mechanism (running the classical planner to produce a fully specified solution for every instance in the generalized planning task).

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

- *Partially specified*. Between these two extremes we find generalized plans that share elements of both:

1. A planner is still required to produce a fully specified solution for a particular instance.
2. Some general knowledge is exploited to constrain the possible solutions.

The different approaches for *planning with domain-specific control knowledge* (DCK) belong to this class. This class includes planning with partially specified programs, non-deterministic FSCs, formal grammars, AND/OR graphs or HTNs that do not exactly capture the action to apply next.

Despite the different forms of generalized plans, we can define the conditions under which a generalized plan is considered a solution to a given generalized planning task.

Definition 2. *The execution of a generalized plan* $\prod$ *in a classical planning instance* $P = \langle F, A, I, G\rangle$ *is a classical plan, denoted as* $exec(\prod, P) = \langle a_1, \ldots, a_n\rangle$, *that induces a state sequence* $\langle s_0, s_1, \ldots, s_n\rangle$ *such that* $s_0 = I$ *and, for each* $1 \leq i \leq n$, a_i *is applicable in* s_{i-1} *and generates the successor state* $s_i = \theta(s_{i-1}, a_i)$.

Definition 3. *A generalized plan* $\prod$ *is a solution to a given generalized planning instance* $\mathcal{P} = \{P_1, \ldots, P_T\}$ *iff the execution of* $\prod$ *on every classical planning instance* P_t, $1 \leq t \leq T$ *produces a classical plan that solves* P_t.

In the remainder of the paper we analyze, criticize and compare different approaches to generalized planning following the abstract framework shown in Figure 4:

- The *problem generator* box refers to a generative model of the instances in the generalized planning task. A generalized planning task comprises a set of individual planning tasks to be solved. This set of planning tasks can either be finite or infinite. Likewise it can be specified in different ways, for example, an explicit enumeration of classical planning instances or implicitly by using logic formulae, a probabilistic distribution, a problem generation program, etc. Problem generation is skipped when an explicit specification of the planning tasks is provided.
- The *generalized planner* box refers to an algorithm fed with an *input-output* specification of the instances to solve and that generates a solution to these instances. The algorithms for generalized planning range from pure *top-down* approaches, that search in the space of generalized plans a solution that covers all the input instances, to *bottom-up* approaches, that compute a solution to a single instance, generalizing it and merging it with previously found solutions to widen the coverage of the generalized plan.

To illustrate our use of this abstract framework, we follow it here to look at *classical planning* as if it were a generalized planning approach.

1. In classical planning the planner only receives as input a single and ground planning instance.
2. The state-of-the-art algorithms for classical planning are heuristic search in the state space (Helmert, 2006; Frances *et al.*, 2017) or compilation to other forms of problem solving such as SAT (Rintanen, 2012).

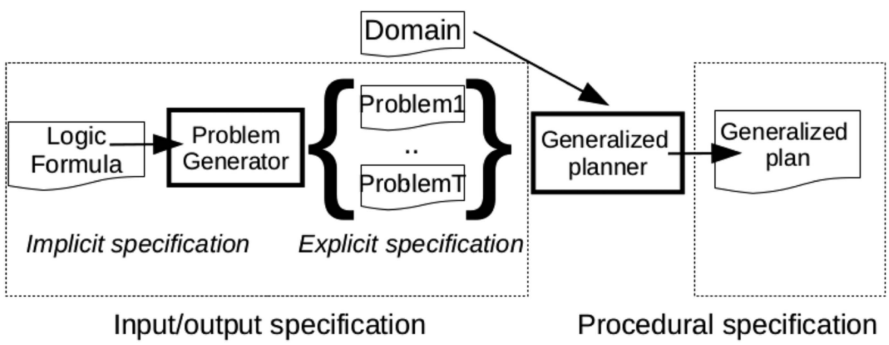

Figure 4 Abstract framework for generalized planning.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

3. A classical plan is a sequence of actions and both the execution and validation of a classical plan are linear in the length of the plan. Nevertheless actions with conditional effects, variables and control-flow structures can be used to compactly represent solutions to classical planning tasks (Hector, 2005; Scala *et al.*, 2016).

3 Representing sets of planning tasks

This section analyzes different formalisms for representing sets of planning tasks within generalized planning.

3.1 Representing actions

Compact and general task representations usually require an action model where different effects can occur depending on the current state of the world. An example is the agent-centered action model of the ATARI video-game (Mnih *et al.*, 2015), where the 18 possible actions have different effects according to the current state of the video-game. Here, we review extensions to the classical planning action model that aim more compact and general representations of the planning tasks and the planning solutions.

3.1.1 Conditional effects

The model of *classical planning with conditional effects* is more expressive than the basic *classical planning* model. Conditional effects cannot be compiled away if plan size should grow only linearly (Bernhard, 2000). A classical planning task with conditional effects is a tuple $P = \langle F, A, I, G \rangle$, as defined for classical planning, except for the set of actions A. Now, each action $a \in A$ with conditional effects is defined as:

- The *preconditions*, a set of literals $\text{pre}(a) \subseteq \mathcal{L}(F)$.
- The set of *conditional effects*, $\text{cond}(a)$. Each conditional effect $C \triangleright E \in \text{cond}(a)$ is composed of two sets of literals, $C \subseteq \mathcal{L}(F)$ (the condition) and $E \subseteq \mathcal{L}(F)$ (the effect).

An action $a \in A$ with conditional effects is applicable in a state s if and only if $\text{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is,

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

that is, effects whose conditions hold in s. The result of applying a in s is a new state $\theta(s, a) = (s \backslash \neg \text{eff}(s, a)) \cup \text{eff}(s, a)$. The definition of a plan, and a solution plan, is analogous to that for planning problems without conditional effects.

PDDL supports the definition of conditional effects with the when keyword. In PDDL the condition of a given conditional effect has the same expressiveness as action *preconditions* and *goals*, so it can either be a negation, a conjunction, a disjunction or a quantified formula, as defined in the action description language formalism (Pednault, 1989; Fox and Long, 2003). Many classical planners natively cope with conditional effects without compiling them away. In fact since 2014, the support of PDDL conditional effects is a requirement for participating at IPC (Vallati *et al.*, 2015).

The model of *classical planning with conditional effects* makes it possible to repeatedly refer to the same action while the precise effects of the action are determined by the state where the action is applied. For instance, the execution of the six-action sequence (unstack, put-down, unstack, put-down, unstack, put-down) can unstack a single tower of either four, three or two blocks if unstack and put-down are actions with conditional effects as defined in Figure 5. The effects of these actions are defined using the universally quantified variables ?x and ?y of type block. Quantified variables do not increase the expressiveness of the actions but allow a more succinct representation. Note that these actions are defined with a single tower of blocks otherwise, if there is more than one tower, these actions would represent the unstacking of multiple blocks at the same time.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

```
(:action unstack
  :parameters ()
  :precondition (and)
  :effect
    (forall (?x ?y - block)
      (and
        (when (and (clear ?x) (on ?x ?y) (empty))
              (and (hold ?x) (clear ?y)
                   (not (clear ?x)) (not (on ?x ?y)) (not (empty)))))))

(:action put-down
  :parameters ()
  :precondition (and)
  :effect
    (forall (?x - block)
       (and (when (and (hold ?x))
                   (and (ontable ?x) (clear ?x) (empty)
                        (not (hold ?x)))))))
```

Figure 5 Planning Domain Definition Language (PDDL) actions from a blocksworld version to unstack a single tower of blocks using *conditional effects* and *universally quantified variables*.

```
(:derived (above ?x ?y - block)
   (or (on ?x ?y)
       (exists (?z - block)
               (and (on ?x ?z) (above ?z ?y)))))
```

Figure 6 Planning Domain Definition Language derived predicate with one existentially quantified variable ?z that leverages recursion to capture when a block ?x is above another block ?y.

3.1.2 Update formulas and high-level state features

The series of work by Srivastava *et al*. (2011a) on generalized planning encodes the effects of actions with update formulas. An update formula is an arbitrary first-order logic (FOL) formulae, that includes *transitive closure*, defining the new value of a given predicate after an action application. The *transitive closure* allows compact representation of connectivity properties such as the *above* concept in blocksworld.

In more detail, a particular update formula for a predicate p has the form $p' \equiv [\neg p \wedge \Delta^{+}_{p,a}] \vee [p \wedge \neg\Delta^{-}_{p,a}]$ where:

- p' denotes the value of the predicate p after the application of action a.
- $\Delta^{+}_{p,a}$ denotes the conditions under which p is changed to *true* by action a.
- $\Delta^{-}_{p,a}$ denotes the conditions under which p is changed to *false* by action a.

While this model for the action effects encodes more expressive state transitions than simple *conditional effects*, it is not supported by off-the-shelf PDDL classical planners.

Arbitrary FOL formulae, that include *transitive closure*, can be represented in PDDL using *derived predicates*. Derived predicates can later be included in action preconditions, conditional effects and goals. Figure 6 shows how PDDL defines the above derived predicate that models whether a block ?x is *above* another block ?y in a *blocksworld* tower.

Derived predicates can represent expressive state queries including hierarchies over the state variables and recursion (Thiébaux *et al*., 2005). This has proven useful for compactly representing planning tasks and also for more effective planning (Ivankovic and Haslum, 2015). Figure 7 shows a PDDL derived predicate, with a quantified variable ?b, that represents the set of *blocksworld* states where all blocks are on the table. Apart from derived predicates, diverse formalisms have been used to represent state queries in planning, ranging from first order clauses (Veloso *et al*., 1995) to description logic formulae (Martn and

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

```
(:derived (all-ontable)
   (forall (?b - block)
           (and (clear ?b) (ontable ?b))))
```

Figure 7 Example of a Planning Domain Definition Language derived predicate, with one universally quantified variable ?b, that captures when all the blocks are on the table.

```
(:action sense-block-color
    :parameters (?b - block)
    :precondition (and (hold ?b)
                       (color ?b unknown))
    :effect (and (not (color ?b unknown))
                 (oneof (color ?b red)
                        (color ?b green)
                        (color ?b yellow)
                        (color ?b blue)))
```

Figure 8 Non-deterministic action for sensing the color of a given block.

Geffner, 2004), or even linear temporal logic formulae to define queries about sequence of states (Cresswell and Alexandra, 2004).

3.1.3 Sensing and non-deterministic actions

When states are fully observable, explicit sensing actions are not necessary given that any state information is obtained via state queries. Sensing actions are then suitable for *planning under partial observability* and they do not model state transitions, but the observation of some piece of information from the current state that is *unknown*. Planners apply sensing actions when the lack of information about the current state prevents them from generating a plan that achieves the goals with certainty.

If we assume that uncertainty about the current state decreases monotonically (i.e. once the value of a state variable is *known* it can change, but cannot become *unknown* again) sensing actions can be encoded as non-deterministic actions (Muise *et al*., 2014). Figure 8 shows an example of a sensing action for observing the color of a block encoded as a non-deterministic action. The oneof effect represents a kind of state constraint (often called *state invariant*) expressing that as a result of the sensing action, a block can only have one of these four colors: red, green, yellow or blue.

Sensing actions generate *contingent plans*, that is, plans with decision points predicated on the different sensing outcomes (Albore *et al*., 2009). *Contingent plans* generalize noise-free *decision trees*. A decision tree can be defined as a particular kind of contingent plan: whose internal nodes contain only sensing actions and its leaf nodes only contain actions that set a particular class label (Segovia-Aguas *et al*., 2017b).

The action in Figure 8 assumes that there is no knowledge about the likelihood of the different sensing outcomes (e.g. because this knowledge is non-stationary). When this knowledge is available, it can be encoded with probabilistic effects using, for instance, PPDDL, the probabilistic version of PDDL (Younes and Littman, 2004). Figure 9 shows a PPDDL action for sensing the color of a given block s.t. observing a red block is twice as probable. Planning with probabilistic actions becomes an *optimization* task where the planner aims at maximizing the probability of reaching the goals. Both non-deterministic and probabilistic actions can also encode non-deterministic state transitions, like in Fully Observable Non-Deterministic (FOND) or MDP planning (Mausam and Kolobov, 2012; Geffner and Bonet, 2013).

3.2 Representing initial and goal states

A set of states can be defined *explicitly*, enumerating each state in the set, or *implicitly*, defining the constraints that a state has to satisfy to belong to the set.

The set of instances in a generalized planning task can also be explicitly specified, enumerating the individual classical planning instances in the generalized planning task. An example of this is the set of three blocksworld instances, shown in Figure 1, plus their shared domain model with the action schemes

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

```
(:action probabilistic-sense-block-color
    :parameters (?b - block)
    :precondition (and (hold ?b)
                       (color ?b unknown))
    :effect (and (not (color ?b unknown))
                 (probabilistic 0.4 (color ?b red)
                                0.2 (color ?b green)
                                0.2 (color ?b yellow)
                                0.2 (color ?b blue)))
```

Figure 9 probabilistic version of Planning Domain Definition Language action for sensing the color of a given block coded.

```
(define (problem conformant-b2)
  (:domain blocks)
  (:objects A B - block)
  (:init
    (and (oneof (empty) (hold A) (hold B))
         (oneof (hold A) (clear A) (on B A))
         (oneof (hold A) (ontable A) (on A B))
         (oneof (hold B) (clear B) (on A B))
         (oneof (hold B) (ontable B) (on B A))

         (or (not (empty)) (not (hold A)))
         (or (not (empty)) (not (hold B)))
         (or (not (hold A)) (not (hold B)))

         (or (not (hold A)) (not (clear A)))
         (or (not (hold A)) (not (on B A)))
         (or (not (clear A)) (not (on B A)))

         (or (not (hold A)) (not (ontable A)))
         (or (not (hold A)) (not (on A B)))
         (or (not (ontable A)) (not (on A B)))

         (or (not (hold B)) (not (clear B)))
         (or (not (hold B)) (not (on A B)))
         (or (not (clear B)) (not (on A B)))

         (or (not (hold B)) (not (clear B)))
         (or (not (hold B)) (not (on A B)))
         (or (not (clear B)) (not (on A B)))

         (or (not (on A B)) (not (on B A)))))
  (:goal (and (ontable A) (on B A))))
```

Figure 10 Conformant planning task for a 2-block *blocksworld*. A single goal condition is defined for the different possible initial configurations of the two blocks.

for the unstack, stack, pick-up and put-down actions. Implicit representations of generalized planning tasks define two sets of constraints, one that defines the set of possible initial states, and a second one defining the set of goal states. An example of this is the conformant planning task shown in Figure 10 taken from IPC-2008.

Here we review different formalisms for representing a set of planning instances according to the language used for specifying these constraints:

- *Propositional logic*. In this case the sets of possible initial and goal states are represented exclusively using literals and the three basic logical connectives (and, to indicate a conjunction of literals or, to indicate a disjunction of literals and not, to indicate negation). Examples of sets of planning instances represented with propositional logic are conformant, contingent or POMDPs planning tasks that define the different possible initial states of the task as a disjunction on the problem literals (goals are shared for all the possible initial states in the planning task) (Bonet *et al*., 2010).

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

- *First-order logic*. The benefit of *first-order logic* constraints is that they can contain quantified variables, include the transitive closure and represent unbounded sets of states. These features make first-order formulae achieve compact representations of sets of planning instances as well as to represent planning tasks of unbounded size (Srivastava *et al*., 2011a). For a given finite set of objects, a first-order representation can be transformed straightforward into a propositional logic representation.

- *Constraint programming*. The previous representations restricted themselves to Boolean (two-valued) state variables. In this case sets of states are defined by a set of *finite-domain variables* $X = \{x_1, \ldots, x_n\}$ (where each variable x_i, $1 < i < n$ has an associated finite domain $D(x_i)$) and a set of constrains C that determines when a state is part of the set.

Finite-domain variables are already *de facto* being used by classical planners: a standard preprocess to extract a many-valued representation from Boolean state variables. This preprocess can be quite expensive but it is completely unnecessary if states are represented with *finite-domain variables* (Rintanen, 2015). In addition, constraint programming languages offer a great representation flexibility and off-the-shelf constraint satisfaction problem (CSP) solvers can be used in this case to solve the generalized planning tasks (Pralet *et al*., 2010). This representation can be transformed into a first order representation with a given set of objects representing the domain of the variables.

- *Three-valued logic*. In this logic language there are three truth values 1 (true), 0 (false), or $\frac{1}{2}$ (unknown). Srivastava *et al*. (2011a) use three-valued logic for state abstraction, to compactly represent unbounded sets of concrete states. Three-valued logic has also been useful to represent and solve conformant and contingent tasks (Petrick and Bacchus, 2004; Albore *et al*., 2009; Palacios and Geffner, 2009).

Apart from the sets of initial and goals states, further information can be used to specify a set of planning instances such as *domain invariants* (Srivastava *et al*., 2011a), or even classified execution histories including *positive* and *negative* examples (Hu and Giacomo, 2013), similar to what is done in Inductive Logic Programming (ILP) (Ross Quinlan, 1990).

4 Generalized plans

With respect to classical plans, generalized plans have two benefits, *compactness* and *generality*. In other words, generalized plans can be more succinct and be valid for solving multiple classical planning instances.

4.1 Representation

As mentioned in Section 2, generalized plans may have diverse forms, each with different expressiveness capacities and different execution mechanisms. The syntax and semantics of the formalism chosen for representing generalized plans defines the space of solutions that can be computed as well as the worst case computation complexity.

4.1.1 Control flow

Control-flow structures augment the flexibility of generalized plans with respect to classical plans:

- *Branching*: the execution of the plan branches according to the result of the evaluation of a given expression in the current state. Examples of planning solutions with branching structures are AND/OR tree-like *contingent plans* (Albore *et al*., 2009) or *K-fault tolerant plans* (Domshlak, 2013).
- *Loops*: the execution of a plan segment is repeated until a given condition holds in the current state. Examples of planning solutions with loops include the *policy-like* plans used for representing solutions to MDPs (Kolobov, 2012), and FOND planning tasks (Muise *et al*., 2014).

The size of a solution plan containing only branching constructs can be exponential in the number of possible state observations. Combining *branching* and *loops* is often helpful to compress generalized plans. In some solution representations, like DSPlanners (Winner and Veloso, 2003), branching and loops

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

correspond to different control-flow constructs but often, they are implemented with the same construct (e.g. conditional transitions) to keep the solution space tractable. This is what happens in FSCs (Bonet *et al*., 2010), *generalized policies* (Martn and Geffner, 2004), or with the *conditional gotos* used in *planning programs* (Segovia-Aguas *et al*., 2016a).

Figure 11 shows a generalized plan, in the form of a *planning program*, for unstacking a single tower of blocks no matter its height. The plan contains actions unstack and put-down (as defined in Figure 5) for unstacking the block at the top of the tower and putting it down on the table, respectively. The control flow instruction 2.goto (0, ! (empty)) jumps back to the first step, 0. put-down, when the robot hand is not empty. Note that such a compact and general plan is definable because the actual effects of put-down and unstack depend on the current state (put-down and unstack have the conditional effects shown in Figure 5).

4.1.2 Variables

Unstacking multiple towers of blocks is more challenging than unstacking a single tower of blocks (there can be an arbitrary number of towers, each with different height). A general solution to this task cannot be compactly represented branching and looping over the ground values of the given state variables.

DSPlanners address this issue representing solutions with *quantified variables* (Winner and Veloso, 2003). Quantified variables makes it possible to identify objects with particular features and to apply selective actions to the identified objects. Figure 12 shows a generalized plan for unstacking multiple towers of blocks that uses two existential variables, ?b1 and ?b2. These variables capture the *block to move next*, linking the parameters of lifted predicates and actions. The generalized plan in Figure 12 has the form of a DSPlanner and its execution in a given planning instance requires unifying variables ?b1 and ?b2 with actual blocks in the current state of the planning task.

In the different formalisms for representing generalized plans, *quantified variables* appear as:

- *Existential variables*. An existential variable is a variable that asserts that a given property, or relation, holds for at least one possible variable value. Besides DSPlanners, existential variables also appear in *choice actions* (Srivastava *et al*., 2011a), that is, actions instantiated during the execution of the plan and as a result of evaluating a FOL formula in the current state. Another example are *generalized policies*, whose rules contain variables to be unified with the current state (Khardon, 1999). PDDL can represent policies with derived predicates (Ivankovic and Haslum, 2015), Figure 13 shows the PDDL derived predicates representing a two-rule policy for unstacking multiple towers of blocks (the encoding requires that these derived predicates are added as extra preconditions of the corresponding actions to make up the policy). More recently *conjunctive queries*, including existential variables, are used to address classification tasks that are modeled as generalized planning (Lotinac *et al*., 2016).

```
0. put-down
1. unstack
2. goto(0,!(empty))
3. end
```

Figure 11 Example of a generalized plan for unstacking a single tower of blocks.

```
plan ← {}
while(in_current_state(on(?b1:block ?b2:block)) and
      in_current_state(clear(?b1:block))) do
   operator ← move-block-to-table(?b1:block ?b2:block)
   execute operator
   plan ← plan + operator
```

Figure 12 *DSPlanner* for unstacking multiple towers of blocks.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

- *Universal variables*. A universal variable asserts that a given property or relation holds for all the possible variable values. Figure 14 shows that the program in Figure 11, for unstacking a single tower of blocks, can be rewritten using the (all-ontable) derived predicate with universal variables defined in Figure 7.

The use of derived predicates, that evaluate a given expression over quantified variables, applies also to other forms of generalized plans such as FSCs. Figure 15 shows a FSC for collecting a green block in a tower of blocks that achieve generalization because observations *H* and *G* are the result of evaluating an expression over quantified variables. In particular *H* holds when a block is being held and *G* when the top block is green. These two observations are defined using the derived predicates shown in Figure 16. The state queries depend on the joint variant *HG* with four possible values (true-true, true-false, false-true and false-false). Related to universal/existential variables are also the *angelic/devilish* concepts used in the literature to define planning hierarchies (Marthi *et al*., 2007).

```
(:derived (apply-putdown ?x - block)
          (and (hold ?x)))

(:derived (apply-unstack ?x ?y - block)
          (and (empty) (clear ?x) (on ?x ?y)))
```

Figure 13 Two-rule policy for unstacking towers of blocks represented with two Planning Domain Definition Language derived predicates.

```
0. unstack
1. put-down
2. goto(0,!(all-ontable))
3. end
```

Figure 14 Example of a generalized plan for unstacking a single tower of blocks using the (all-ontable) derived predicate.

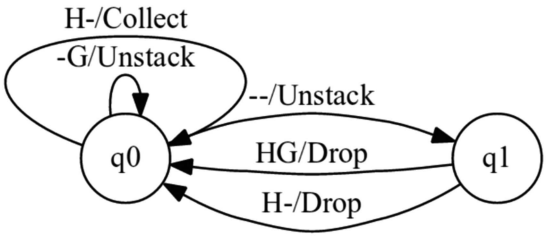

Figure 15 *Finite State Controllers* for collecting a green block in a tower of blocks by observing whether a block is being held (H), and whether the top block is green (G).

```
(:derived (H)
          (exists (?b - block) (and (hold ?b))))

(:derived (G)
          (exists (?b - block)
                  (and (clear ?b)
                       (color ?b green))))
```

Figure 16 Derived predicates with existential variables that capture when a block is being held (H), and when the top block is green (G).

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

4.1.3 Call stack

A *call stack* is another artifact borrowed from programming to make generalized plans more flexible (Fritz *et al*., 2008). Although one can explicitly encode a call stack using only basic control-flow and variables, more compact solutions are often derived with a call stack (e.g. tasks with recursive solutions (Segovia-Aguas *et al*., 2016b)). Figure 17 shows a generalized plan for visiting all the nodes of a binary tree implementing a recursive Depth First Search with one procedural parameter. The instructions call(0, *node*) are recursive calls assigning argument *node* to the only parameter of the program and restarting the execution from its first line, 0. visit (current).

A given generalized plan can also be represented as a formal grammar with a *call stack* (Ramirez and Geffner, 2016; Segovia-Aguas *et al*., 2017a). For instance, Figure 18 shows a grammar that encodes a generalized plan for *blocksworld*. The first instruction of this plan, 0. choose (1|5|8), is a *choice instruction*, that jumps to one of these three possible targets, line 1, line 5 or line 8 and hence, represents a rule selection. This action is non-deterministic since the rule selection is initially unknown and determined only during the execution of the program. Lines 1–4 encode the first grammar rule while lines 5–8 encode the second grammar rule. The third grammar rule parses the empty string and is encoded implicitly by line 8.

Furthermore, the benefits of using *a call stack* in generalized planning come from the reuse of existing generalized plans and the incremental building of hierarchical generalized plans (Segovia-Aguas *et al*., 2016b). Figure 19 illustrates these benefits showing a two-module generalized plan for sorting lists of numbers that implements the *selection sort* algorithm and reuses Π^1, a previously generated generalized plan for finding the minimum number in a list. Here Π^0, left side, is the main program and its instruction call (1) invokes the execution of the auxiliary program Π^1, right side, from its first line, 0. inc-pointer (inner).

```
0. visit(current)
1. goto(6,!isInternal(current))
2. left(current,child)
3. right(current,current)
4. call(0,current)
5. call(0,child)
6. end
```

Figure 17 Example of a generalized plan $\pi_{DFS}(node)$ implementing a recursive Depth First Search (DFS) for traversing a binary tree of arbitrary size.

```
0. choose(1|5|8)
1. unstack(X1,X2)
2. putdown(X1)
3. call(0)
4. end
5. pickup(X1)
6. stack(X1,X2)
7. call(0)
8. end
```

Figure 18 Grammar encoding a partially specified generalized plan for *blocksworld*.

(a)

```
0. call(1)
1. swap(*mark,*outer)
2. inc-pointer(outer)
3. goto(0,!(eq(inner,itermax)))
4. end
```

(b)

```
0. inc-pointer(inner)
1. goto(3,!(lt(*inner,*mark)))
2. assign(mark,inner)
3. goto(0,!(eq(inner,itermax)))
4. end
```

Figure 19 Generalized plan with a procedure call for sorting list of numbers of arbitrary size and that corresponds to the selection sort algorithm. (a) Π^0: Main program that repeatedly selects the minimum value and swap contents. (b) Π^1: Auxiliary program that selects the minimum value from current position outer.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

As a rule of thumb, a call stack allows the execution of a given generalized plan to jump to another generalized plan:

- Keeping different contexts. Each generalized plan can have different local state/variables.
- Sharing information. Passing information between the generalized plans as procedural parameters or using global state/variables.

4.2 Execution and validation

Generalized plans can branch, loop and have variables so executing a generalized plan on a particular classical planning instance requires specific machinery, different from the one traditionally used in classical planning:

- *Branching*. The execution of a generalized plan with different possible execution branches requires a mechanism for selecting the corresponding execution branch according to the current value of the state variables. The execution of several generalized plan that branch (like an HTN, an AND/OR tree-like plan or a policy) can be compiled into classical planning (Albore *et al*., 2009; Alford *et al*., 2009; Ivankovic and Haslum, 2015). As explained in Section 3, different possible execution outcomes according to different values of the state variables can be effectively modeled in classical planning via *conditional effects* (Bernhard, 2000).
- *Loops*. Executing generalized plans that explicitly represent loops, like programs (or FSCs), requires to keep track of the current program line (or controller state). The execution of FSCs and programs can be compiled into classical planning by encoding the corresponding automata (its states and possible transitions) as extra state variables (Baier *et al*., 2007; Segovia-Aguas *et al*., 2016a; 2016b).
- *Variables*. If the generalized plan contains *quantified variables*, then plan execution requires unification mechanisms that assign possible values to these variables. Early planning systems implemented variable binding algorithms for matching control rules (Veloso *et al*., 1995). Nowadays Fast-Downward evaluates derived predicates with quantified variables implementing the *marking algorithm* (Helmert, 2006). A different approach is to leverage external solvers, such as *Answer Set Programming* (Ivankovic and Haslum, 2015) or CSP (Francès and Geffner, 2016) solvers, to ground quantified variables. The compilation approach has also been followed for binding existential variables in *conjunctive queries* (Lotinac *et al*., 2016) and to evaluate FOL state queries with the transitive closure (Porco *et al*., 2011). Unfortunately most current off-the-shelf planners only effectively support simple conditions as conjunctions of propositional atoms and compiling away existentially quantified formulas have an exponential cost (Francès and Geffner, 2015).

The simplest desired property for the execution of a generalized plan on a given planning instance is *termination*, also referred in literature as the *halting problem*. In the worst case, the number of actions of a generalized plan execution has an upper bound given by the total number of possible states of the generalized plan. Infinite loops can then be detected counting the number of actions during plan execution and checking if this count exceeds the previous upper-bound (Bäckström *et al*., 2014).

A second property for the execution of a generalized plan is guaranteeing that the plan solves a given instance. Testing this property is called *validation*, proving validation subsumes the proof of termination and is implicitly required as a part of plan generation. Plan validation in classical planning is linear, since either a validation proof or a failure proof is straightforward obtained by *executing* the plan starting from the initial state of the classical planning task. VAL (Howey *et al*., 2004), introduced in the third IPC, is the standard plan validation tool for classical planning.

The execution of a generalized plan (that can branch, loop and have variables) on a given planning instance can fail to solve that instance because:

1. The plan is *unsound*:

- The generalized plan does not satisfy the *termination* condition because it enters into an infinite loop.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

- The action-to-apply-next (according to the generalized plan) cannot be applied. For instance, the preconditions of the recommended action do not hold in the current state.
- The execution of the plan ended but it did not achieve the goals of the planning task. Some forms of generalized plans include explicit termination actions (or states) so goal testing is only done when such actions are applied (or such states reached) (Srivastava *et al*., 2011b; Jiménez and Jonsson, 2015). If the generalized plan lacks these termination actions (or states) goal achievement has to be tested after executing every plan step (Bonet *et al*., 2010).

2. The plan is *incomplete*. There is no action-to-apply-next for the current state (e.g. a policy with no applicable rule at the current state).

The execution of a generalized plan on a given classical planning task can be compiled into another classical planning task (Baier *et al*., 2007; Segovia-Aguas *et al*., 2016a, 2016b) and hence, an off-the-shelf classical planner can be used to effectively check the previous validation conditions. In that case, the validation of a generalized plan is as complex as the synthesis of a classical plan. When actions have non-deterministic effects plan validation becomes more complex since it requires proving that all the possible plan executions reach the goals (Cimatti *et al*., 2003). In such scenario *model checking* (Clarke *et al*., 1999) and *non-deterministic planning* are suitable approaches (Hoffmann, 2015).

Unlike classical plans, that are tied to a particular planning instance, generalized plans can also be executed on a set of different planning instances. The validation of a generalized plan on a generalized planning task requires executing the plan in all the instances comprised in the task and verifying that the plan solves them all. This means that the validation of a given generalized plan in a given instance should be polynomial in the size of the plan to be effective. The execution of a generalized plan in a set of planning instances can be implemented following two different approaches:

- *Sequential*, that is, executing the generalized plan at each instance separately one after the other. This is the approach followed for executing generalized plans using a classical planner that sequentially executes the plan in each of the individual planning instances comprised in a given generalized planning task (Segovia-Aguas *et al*., 2016a).
- *Parallel*. The generalized plan is executed simultaneously in the set of instances of a generalized planning task (like in conformant, contingent or POMDP planning where the execution of an action progresses a set of states (Geffner and Bonet, 2013)).

The implementation of the *sequential* approach is simpler but its utility is limited by the number of instances. The *parallel* approach allows to handle larger sets of instances (or instances with unbound number of objects) but it requires elaborated state progression techniques, such as belief tracking (Bonet and Geffner, 2014) or the application of action updates on abstract states (Srivastava *et al*., 2011a). Evaluating expressive goals, or derived fluents, becomes more complex in a parallel execution since it implies formulae evaluation over sets of states (Geffner and Bonet, 2013). A third way to validate generalized plans is to show that some property holds before and after execution, like in program validation with *Hoare triples* (Schwinghammer *et al*., 2009).

4.3 Evaluation

Given a set of planning instances, different generalized plans can be *consistent* with it, for example, the different generalized plans shown in Figures 11, 12 and 14 can unstack a tower of blocks of any height. It is then necessary to define methods that quantify the aptitude of a given generalized plan to articulate preferences among the possible solutions.

The aptitude of a generalized plan can be assessed with regard to different metrics:

- *Coverage*. The *domain coverage* of a generalized plan can be assessed as the ratio of the number of problem instances with size n that the generalized plan can solve, divided by the total number of solvable problem instances with this same size (Srivastava *et al*., 2011a). In practice knowing these numbers implies solving large sets of planning tasks, so it is often intractable. Statistical Machine

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

Learning (ML) techniques estimate the quality of a solution according to how well the solution performs on a *representative* sample of the domain instances, called *test set* (Mitchell, 1997). In generalized planning one could also define a test set of instances and count how many are covered by a solution. If we view a classical planner as a particular form of a generalized plan, this is what is done at the sequential-optimal track of the IPC (Vallati *et al*., 2015) where planners are awarded according to the amount of unseen instances they solve.

- *Complexity*. Because generalized plans are algorithm-like solutions their complexity can be theoretically assessed, for example, using asymptotic analysis that characterize how their running time and space requirements grow according to the size of the input tasks. In practice the complexity of a generalized plan can be quantified by the length of the sequence of actions produced by the execution of the plan on a given input instance. Then a generalized plan is *optimal* for a given instance when the length of this sequence is minimum for that instance. This is somehow related to what is done in the sequential-satisfying track of the IPC (Vallati *et al*., 2015) where the final value of a planner is reported as the accumulated quality of the solutions in the instances of a testing set.
- *Succinctness*. The size of a given generalized plan can be assessed regarding to its number of program lines, controller states, policy rules or quantified variables. Similar metrics were already introduced in ILP systems to quantify the *compactness* and *readability* of solutions and to prefer models with the least number of rules and rules with the smallest size (Muggleton, 1999). Note that in classical planning the execution complexity of a plan directly corresponds to its size.

5 Computing generalized plans

This section describes the two main approaches for the computation of generalized plans and reviews different *plan reuse* techniques to avoid computing generalized plans from scratch. The section ends reviewing particular implementations of different approaches for generalized planning.

5.1 Top-down/bottom-up generalized planning

The *top-down* approach for generalized planning searches for a solution that covers all the instances in the generalized planning task. On the other hand, the *bottom-up* approach computes a solution to a single instance (or to a subsets of the instances in the generalized planning task) and widens the coverage of the solution until covering all the instances in the generalized planning task. With respect to ML, the top-down approach relates to *off-line* ML algorithms that compute a model to cover, in a single iteration, the full set of input instances, for example, the induction of decision trees (Mitchell, 1997). The bottom-up approach is related to the *on-line* versions of ML algorithms, that iterate to incrementally adapt the model as more input instances are presented to the learning algorithm (Utgoff, 1989).

Top-down algorithms for generalized planning typically search for a solution in the space of possible generalized plans. The initial state of this search is the *empty* generalized plan and the search operators build a single step in the generalized plan (e.g. adding an instruction to a program, a new state or transition to a FSC, a new rule to a policy, etc.). The set of goal states of the search includes any state where the built generalized plan solves the given set of instances.

Examples of this approach are compilations of generalized planning into other forms of problem solving such as *classical planning* (Jiménez and Jonsson, 2015), *conformant planning* (Bonet *et al*., 2010), *CSP* (Pralet *et al*., 2010) or a *Prolog program* (Hu and Giacomo, 2013). These compilations implement a search space as described above, and benefit from off-the-shelf solvers (with efficient search algorithms and heuristics) to complete the search for a generalized plan. The main limitation of the compilation approach is scalability. In practice, it is common to bound the size of the possible generalized plans (e.g. the maximum number of program lines, controller states, policy rules or quantified variables) with the aim of keeping the search tractable. This is similar to what is done in SATPLAN approaches that fix a maximum plan length (Rintanen, 2012) and iteratively increments it until a solution is found.

Off-line algorithms for contingent (Albore *et al*., 2009), conformant (Palacios and Geffner, 2009) and POMDP planning (Geffner and Bonet, 2013) can also be understood as *top-down* algorithms for

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

generalized planning if we consider that the possible initial states represent different planning instances all sharing the same goals. In this case the search for a solution is not typically carried out in the space of possible generalized plans but in the space of reachable *belief states* (a belief state is a probability distribution over the states that are deemed possible). Here the scalability limitations come from the fact that the set of reachable belief states grows quickly (it is then key to exploit techniques for uncertainty reduction to keep the set of possible states tractable) and from the difficulty of defining effective heuristics that provide *informative* estimates with belief states.

Bottom-up generalized planning refers to an iterative and incremental approach where (1) a single planning instance (or a subset of the instances in the generalized planning task) is chosen, (2) a solution to it is computed, (3) generalized and finally (4), merged with the previously found and generalized solutions. This four-step process is repeated until all the instances in the generalized planning task are covered. The *bottom-up* approach is related to *plan repair* (Fox *et al*., 2006), *case-based planning (CBP)* (Borrajo *et al*., 2015) and *transfer learning* (Pan and Yang, 2010) because it also requires mechanisms to identify why a given solution does not cover a given instance (in this case validation mechanisms for generalized plans are suitable to find the cause of an assertion failure in a given plan (Winner and Veloso, 2003)) as well as mechanisms to adapt a given solution to a new scenario (this kind of adaptation mechanisms are also present when planning with imperfect control knowledge (Yoon *et al*., 2008)).

While *top-down* approaches can be implemented as compilations to other forms of problem solving, *bottom-up* approaches require specific techniques to lift and merge plans. On the other hand, *bottom-up* approaches provide anytime behavior and may be able to automatically build a small set of instances that achieve generalization (Srivastava *et al*., 2011b).

5.2 Reusing generalized plans

An alternative to the computation of generalized plans starting from scratch is to reuse existing solutions. Even when a certain generalized plan is unsound (in the sense that it fails to solve a given instance) or incomplete (it does not define *the action to apply next* for the given instance), it may contain useful knowledge. For example, the plan may be able to solve a sub-problem of the given generalized planning task, or similar instances (e.g. it is a solution but for objects of a different type), or solve new instances after tuning the conditions in the control-flow structures. In these cases adapting a previously existing generalized plan can pay off.

With this regard, *bottom-up* approaches for generalized planning are equipped with mechanisms to adapt a plan to unseen instances and incrementally increase its coverage (Srivastava *et al*., 2011b). On the other hand, *top-down* approaches can start with a partially specified solution instead of with the *empty* generalized plan. This has shown useful to narrow the search space and/or focus the search process making possible to address more challenging generalized planning tasks (Segovia-Aguas *et al*., 2016a, 2016b).

Next, we review different techniques for reusing previously found plans:

- *Compilations*. When the existing generalized plan has the form of a generalized policy it can be compiled into a set of PDDL derived predicates, one for each rule in the policy, that captures the different situations where the actions should be applied (Ivankovic and Haslum, 2015). Figure 13 illustrated this approach showing an example of PDDL derived predicates representing a two-rule policy for unstacking towers of blocks. Existing generalized plans in the form of programs, FSCs or AND/OR graphs, can be encoded into a classical PDDL planning task by computing the cross product between the corresponding automata and the original planning task (Baier *et al*., 2007; Ramirez and Geffner, 2016; Segovia-Aguas *et al*., 2016a). In this case, new extra state variables are added to the original planning task to represent the states and transitions of the automata corresponding to the program, FSC or AND/OR graph.
- *Planning actions*. Actions in classical planning do not only represent primitive actions but can also represent a generalized plan themselves. Figure 20 shows a classical planning action that corresponds to a generalized plan for unstacking any block in a blocksworld, that is, the first step in a general solution for solving any blocksworld instance.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

```
(:action unstack-all
  :parameters ()
  :precondition (and)
  :effect (and (forall (?x - block ?y - block)
                 (when (and (on ?x ?y))
                       (and (not (on ?x ?y))
                            (clear ?x)
                            (ontable ?x))))))
```

Figure 20 Action for unstacking all the blocks in a blocksworld task. The action is encoded in Planning Domain Definition Language (PDDL) using universally quantified variables and conditional effects.

```
;;; Primitive action
(:action unstack
  :parameters (?x - block ?y - block)
  :precondition (and (on ?x ?y) (clear ?x) (empty))
  :effect (and (hold ?x) (clear ?y)
               (not (clear ?x)) (not (empty)) (not (on ?x ?y))))

;;; Primitive action
(:action drop
  :parameters (?x - block)
  :precondition (hold ?x)
  :effect (and (clear ?x) (empty) (ontable ?x)
               (not (hold ?x))))

;;; Macro-action
(:action unstack-drop
  :parameters (?x - block ?y - block)
  :precondition (and (on ?x ?y) (clear ?x) (empty))
  :effect (and (clear ?y) (ontable ?x)
               (not (on ?x ?y))))
```

Figure 21 Two primitive actions from the blocksworld described in Planning Domain Definition Language (PDDL) and the macro-action resulting from assembling them.

The compilation of existing solutions into new planning actions is well-studied for the particular case of *macro-actions*. A macro-action can be viewed as a parameterized generalized plan, without control flow, that can be reused in a straightforward way to enrich a domain theory (macro-actions have the form of standard classical planning actions). Figure 21 shows the classical planning actions unstack and drop from the blocksworld domain, for unstacking a block X from another block Y and for putting a block X onto the table, as well as the macro-action unstack-drop resulting from assembling them. The assembly of these two actions can, for example, be computed following the early planning algorithm of the *triangle-table* (Fikes *et al.*, 1972).

A limitation of macro-actions is that their structure is too rigid so many generalized plans cannot be encoded as macro-actions. For instance, the generalized plan introduced in Section 1 for solving any blocksworld instance, cannot be encoded as a macro-action. Further research is necessary to automatically compile arbitrary generalized plans into planning actions, that can be directly included in a domain theory, without adding extra state variables. Recent work on *planning with simulators* opens the door to more effective approaches for reusing existing procedural solutions as black-box actions (Frances *et al.*, 2017).

- *Domain-specific heuristics*. Incorrect and/or incomplete generalized plans have also been used to improve the performance of a classical planner working as domain-specific heuristics (Yoon *et al.*,

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

2008; Rosa *et al*., 2011). This approach is specially useful at planning tasks where domain independent heuristics have flaws, for example, due to strong goal interactions.

5.3 Implementations

Now we review particular approaches for generalized planning. We analyze how they represent the tasks to solve, the representation of the generalized plans and the algorithms for computing them.

5.3.1 Computing macro-actions

Macro-actions are one of the first suggestions to compute general knowledge valid for solving different planning tasks (Fikes *et al*., 1972). There are diverse approaches in the literature for computing macro-actions (Botea *et al*., 2005; Coles and Smith, 2007; Jonsson, 2009; Lukás, 2010) but the most common approach is to: (1) solve a training set of classical planning instances that share the same domain theory with an off-the-shelf classical planner and (2), identify, in the solution plans, sub-sequences of actions that are frequently used together.

The strength of macro-actions is that they have the form of standard classical planning actions so they can be added straightforward to the domain theory without requiring extra state variables. This makes *macro-actions* a practical and robust approach for reusing general planning knowledge: On the one hand, either planning with macro-actions or the execution and validation of plans that contain macro-actions do not require specific algorithms. On the other hand, adding incomplete or incorrect macro-actions will not prevent a planner to find a solution to a solvable task because the planner can always build a solution using the original actions.

The main limitation of macro-actions for defining general planning strategies is its sequential execution flow, that is too rigid. A solution involving macros may not be applicable to other problems, even when macro-actions are parameterized.

5.3.2 Computing generalized policies

A *generalized policy* is a set of rules that defines a mapping of state and goals, into the preferred *action to execute next*. Like macro-actions, generalized policies also allow parameters and can be induced from a set of solutions to classical planning instances that share the same domain theory (Martn and Geffner, 2004; Yoon *et al*., 2008; Rosa *et al*., 2011). Generalized policies are, however, more flexible than macro-actions since they can define execution flows with branching and loops.

Computing a good generalized policy is complex and, nowadays, the success of this approach is still limited to a reduced number of benchmarks. On the one hand, correct and complete *generalized policies* are not computable for many domains using the given representation for the states, actions and goals. In the past, the limitations of the given representation language has been addressed by hand-coding high-level state features that increase the expressiveness of the given representation (Khardon, 1999) or changing the representation language to reason better about classes of objects (Martn and Geffner, 2004; Fern *et al*., 2006; Yoon *et al*., 2008). On the other hand, the algorithms for computing generalized policies traditionally consider planning and generalization as two separated phases. This separation produces noisy examples difficult to be generalized due to the high number of symmetries and transpositions that typically appear in solution plans.

If a correct *generalized policy* is available, it can be added to a domain theory using derived predicates that capture the states where an action should be applied (Ivankovic and Haslum, 2015), as shown in Figure 13. If the policy is incomplete or incorrect, adding it to the domain theory can turn solvable planning instances into unsolvable (adding the policy means adding new constraints to the original planning task). A more robust approach to reuse imperfect policies is to consider them as domain-specific heuristics that guide the search for a solution plan. Exploiting policies in such way requires the modification of the planner (Yoon *et al*., 2008; Rosa *et al*., 2011).

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

5.3.3 Computing Finite State Controllers

FSCs generalize policies with a finite amount of memory (Bonet and Geffner, 2015). A FSC with a single state represents a policy, that is, a memory-less controller. The additional controller states of FSCs provide them with memory that allows different actions to be taken given the same observation. The FSC formalism can also be extended with a *call stack* to represent hierarchical and recursive solutions (Segovia-Aguas *et al*., 2016b).

The existing algorithms for computing FSCs for generalized planning follow a *top-down* approach that interleaves *programming* the FSC with validating it and hence, they tightly integrate planning and generalization. To keep the computation of FSCs tractable, they limit the space of possible solutions bounding the maximum size of the FSC. In addition, they impose that the instances to solve share, not only the domain theory (actions and predicates schemes) but the set of fluents (Segovia-Aguas *et al*., 2016a) or a subset of *observable* fluents (Bonet *et al*., 2010).

The computation of FSCs for generalized planning includes works that compile the generalized planning task into another forms of problem solving so they benefit from the last advances on off-the-shelf solvers (e.g. *classical planning* (Segovia-Aguas *et al*., 2016a), *conformant planning* (Bonet *et al*., 2010), *CSP* (Pralet *et al*., 2010) or a *Prolog program* (Hu and Giacomo, 2013)). This last case requires a behavior specification of the FSC consisting on classified execution histories that (1) accept all legal execution histories leading to a goal-satisfying state, and (2) reject those that contain repeated configurations (indicating an infinite loop) and that cannot be extended (indicating a dead end) (Hu and Giacomo, 2013).

5.3.4 Computing programs

Programs increase the readability of FSCs separating the control-flow structures from the primitive actions. Like FSCs, programs can also be computed following a *top-down* approach, for example, exploiting compilations that program and validate the program on instances with the same state and action space (Segovia-Aguas *et al*., 2016a). Since these *top-down* approaches search in the space of solutions, it is helpful to limit the set of different control-flow instructions. For instance, using only *conditional gotos* that can both implement branching and loops (Jiménez and Jonsson, 2015).

One of the first attempts to represent generalizes plans as programs are *DSPlanners* (Winner and Veloso, 2003; Winner and Veloso, 2007). A *DSPlanner* is a domain-specific program that can contain if-then-else and while constructs. These constructs branch and loop the execution control flow of the program according to FOL queries on the current state and/or the goals of the planning task.

The algorithm to compute DSPlanners is called Distill and implements a *bottom-up* approach on a set of classical planning instances that share the same domain theory. Given an instance, Distill computes a partially ordered plan for that instance and integrates it into an existing DSPlanner as follows. First, Distill lifts the partially ordered plan choosing a parameterization that matches the existing DSPlanner. If no such parameterization exists, Distill randomly assigns variable names to the objects in the plan. Then Distill attempts to identify *if statements* and unrolled *loop iterations* in the solution to replace them by the corresponding control-flow structure.

The work on generalized planning by Srivastava et al. introduces a powerful and compact structure to programs, called *choice actions*, that combines existential variables and control flow (Srivastava *et al*., 2011a, 2011b). Input instances in this work are expressed as an abstract FOL representation with the transitive closure. This formalism allows to represent planning tasks with an unbounded number of objects and to guarantee the generalization of solutions for such tasks.

The generalized planning algorithm by Srivastava et al. implements also a *bottom-up* strategy. The algorithm starts with an empty generalized plan, and incrementally increases its coverage by identifying an instance that it cannot solve, invoking a classical planner to solve that instance, generalizing the obtained solution and merging it back into the generalized plan. The process is repeated until producing a generalized plan that covers the entire desired class of instances (or when a predefined limit of the computation resources is reached).

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

Table 1 Summary of the diverse approaches for generalized planning according to the solution representations

	Variables	Control-flow	Execution
Classical plan	–	–	Ground actions
Macro-actions	Action parameters	–	Lifted actions
Generalized policy	Rule parameters	Branching and loops	Lifted rules
DSPlanners	Existential	Branching and loops	Lifted predicates and lifted actions
FSCs	Quantified	Branching and loops	Derived predicates
Hierarchical FSCs	Quantified and parameters	Branching, loops and call stack	Derived predicates and parameter passing
Programs	Quantified and parameters	Branching, loops and call stack	Derived predicates and parameter passing

FSC = Finite State Controllers.

Both programs and FSCs can be compiled into a planning domain theory (Baier *et al*., 2007; Segovia-Aguas *et al*., 2016a, 2016b). Like happens with policies, this compilation is *safe* (is not turning solvable planning instances into unsolvable) when the given program (or FSC) is correct.

Table 1 is a summary of the reviewed approaches for generalized planning. The table indicates whether a given solution representation allows the use of *variables*, the kind of *control-flow* and whether the *execution* of the solution requires particular machinery.

6 Related work

Here we review other forms of planning and problem solving that are related to the generalized planning approaches reviewed in the paper.

6.1 Planning under partial observability: conformant, contingent and POMDP planning

Conformant planning computes sequences of actions whose execution is consistent with a set of different initial states (Palacios and Geffner, 2009). The difference to the classical planning model is the uncertainty in the initial state, which is described by means of clauses. A *conformant plan* is a sequence of actions that solves all the classical planning tasks given by the set of possible initial states that satisfy these clauses. The execution of same sequence of actions can produce different outcomes for different initial states because actions have conditional effects. The main approaches for conformant planning are:

- *Uncertainty reduction*. Compiling the conformant planning into classical planning to compute:

1. A plan *prefix* that removes any relevant uncertainty. In other words, only a single state (or at least a single partial state for the subset of state variables that are relevant for achieving the goals) is possible after the prefix application (Palacios and Geffner, 2009).
2. A plan *postfix* that transforms the state (or partial state), where the relevant uncertainty is removed, into a state that achieves the goals of the conformant planning task.

- *Belief propagation*. Searching in the space of belief states where: the *root* belief state represents the set of possible initial states and the *goals* are the belief states s.t., all the possible states in the belief state satisfy the goal condition of the planning task (Cimatti *et al*., 2004; Hoffmann and Brafman, 2006). While the previous approach leverages the classical planning machinery, this approach requires (1) mechanisms for the compact representation and update of beliefs states and (2), effective heuristics to guide the search in the space of belief states.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

Contingent planning extends the conformant planning model with a sensing model. This sensing model is a function that maps state-action pairs (the true state of the system and the last action done) into a non-empty set of observations (Albore *et al*., 2009; Albore *et al*., 2011). Observations provide only partial information about the true state of the system because the same observation may be possible in different states. A *contingent plan* must satisfy that:

- Its execution reaches a goal belief state (all the states in the belief satisfy the goal condition of the planning task) in a finite number of steps.
- The conditions for branching and looping refer to the observations (or the subset of state variables that are observable).

Like generalized plans, contingent plans can have different forms such as policies, AND/OR graphs, FSCs or programs (Bonet *et al*., 2010).

POMDP planning extends the contingent planning model allowing to encode uncertainty through probability distributions, rather than with sets of possible initial states and with sets of possible observations (Geffner and Bonet, 2013). With this regard, the *Bayes*' rule is used to update belief states after an action application or after an observation of the current state. The aim of a POMDPs solution is to maximize the expected cost to the goals, so POMDP planning becomes an optimization task. An optimal conformant/contingent/POMDP plan is the one that minimizes the cost of achieving the goals in the worst case.

Generalized planning can be seen as a particular example of contingent/POMDP planning: (1) the initial states and goals of the different instances comprised in a generalized planning task can be encoded as the different possible initial states of a POMDP task and (2) our definition of generalized planning assumed deterministic actions and full observability (the conditions for branching and looping can refer to the value of any state variable).

6.2 Planning with control knowledge

Since the beginning of research in planning, *control knowledge* has shown effective to improve the scalability of planners (Bacchus and Kabanza., 2000; Nau *et al*., 2003). This was evidenced at IPC-2002 where planners exploiting DCK performed orders of magnitude faster than state-of-the-art planners (Long and Fox, 2003).

Algorithm-like representations of DCK (Baier *et al*., 2007) bear strong resemblances with generalized plans. Indeed both DCK and generalized plans represent general strategies that are valid for solving different planning instances. Despite the distinction between them is thin, one can claim that a generalized plan is a *fully specified solution*, that does not require a planner to be applied in a particular instance. On the other hand, DCK corresponds to *partially specified solutions* (contain non-deterministic constructs and missing/open segments to be determined by a planner at the time of the plan generation). Therefore DCK requires a planner to produce a fully specified solution to a given classical planning instance.

A different approach to define DCK is with a database of solved instances. In fact an alternative view of a generalized plan is as a compact library of plans. CBP is the approach to AP that aims saving computational effort by reusing previously found solutions (Borrajo *et al*., 2015). A CBP system implements *retrieval* mechanisms that identify instances similar to the one to solve as well as *adaptation* mechanisms that repair flaws in a retrieved solution to make it applicable to another instance. Retrieval and adaptation mechanisms of CBP are relevant to *bottom-up* algorithms for generalized planning since they identify when a given generalized plan does not cover an instance and adapt the plan to cover it (Winner and Veloso, 2003; Srivastava *et al*., 2011a). The development of such mechanisms for large case libraries following a domain-independent approach is still a challenge.

Another formalism for representing and exploiting DCK is *hierarchical planning*. Like classical planning, hierarchical planning deals with deterministic and fully observable planning tasks but uses a different task representation. While in classical planning actions are characterized in terms of their pre and postconditions, and their choice and ordering is computed automatically by a planner, *hierarchical planning* specifies a sketch of the solution with extra information about (1) which subgoal to pursue

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

(Shivashankar *et al*., 2012), and/or (2) which actions can be applied for achieving a given subgoal (Nau *et al*., 2003).

In hierarchical planning the separation between the representation of the task to be solved and the strategy for solving it is not as clear as in classical planning. A hierarchical planning task can be understood as a partially specified generalized plan (or a domain-specific planner) where the missing parts of the plan are determined during its execution, by running a hierarchical planner. While a classical planner aims to compute a sequence of applicable actions that transforms a given initial state into a goal state, the hierarchical planner computes a sequence of applicable actions that: (1) transforms a given initial state into a goal state and (2), this transformation is compliant with the given hierarchy.

6.3 GOLOG

The Golog family of action languages has proven to be a useful mean for the high-level control of autonomous agents (Levesque *et al*., 1997). Apart from conditionals, loops and recursive procedures, an interesting feature of Golog programs is that they can contain non-deterministic parts. A Golog program does not need to represent a fully specified solution, but a sketch of it, where the non-deterministic parts are gaps to be filled by the system. This feature provides the Golog programmer with the flexibility to chose the right balance between:

- Determine predefined behavior, which normally implies larger programs.
- Leave certain parts to be solved by the system by means of search, which normally implies larger computation times.

The basic Golog interpreter uses the PROLOG back-tracking mechanism to resolve the search. This mechanism basically amounts to do a blind search so, when addressing planning tasks, it soon becomes unfeasible for all but the smallest instance sizes. IndiGolog (Sardina *et al*., 2004) extends Golog to contain a number of built-in planning mechanisms. Furthermore, the semantics compatibility between Golog and PDDL (Röger *et al*., 2008) can be exploited and a PDDL planner can be embedded (Claßen *et al*., 2008) to address the sub-problems that are combinatorial in nature.

6.4 Program synthesis

Program synthesis is the task of automatically generating a program that satisfies a given high-level specification. Many ideas from this research field are relevant to generalized planning but they are not immediately applicable since generalized planning follows a domain-independent approach and handles its own specific representation for states, actions and goals. Here we review two of the most successful approaches for program synthesis:

- *Programming by Example* (PbE), computes a set of programs consistent with a given set of input-output examples. Input-output examples are intuitive for non-programmers to create programs moreover, this type of specification makes program synthesis more tractable than reasoning with abstract program states. PbE techniques have already been deployed in the real world and are part of the Flash Fill feature of Excel in Office 2013 that generates programs for string transformation (Gulwani, 2011). In this case the set of synthesized programs are represented succinctly in a restricted Domain-Specific Language using a data-structure called version space algebras (Mitchell, 1982). The programs are computed with a domain-specific search that implements a divide and conquer approach.
- In *Programming by Sketching* (PbS) programmers provide a partially specified program, that is, a program that expresses the high-level structure of an implementation but that leaves low level details undefined to be determined by the synthesizer (Solar-Lezama *et al*., 2006). This form of program synthesis relies on a programming language called SKETCH, for sketching partial programs. PbS implements a counterexample-driven iteration over a synthesize-validate loop built from two communicating SAT solvers, the inductive synthesizer and the validator, to automatically generate test inputs and ensure that the program satisfy them. Despite, in the worst case, program synthesis is

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

harder than NP-complete, this counterexample-driven search terminates on many real problems after solving only a few SAT instances (Lake *et al*., 2015).

7 Conclusions

Generalized plans are able to solve planning tasks beyond the scope of classical planning: they can address planning tasks that comprise multiple instances or with an unbound number of objects, as well as planning tasks with partial observability and non-deterministic actions (Bonet *et al*., 2010; Hu and Levesque, 2011; Srivastava *et al*., 2011b; Hu and Giacomo, 2013). Generalized planning is then a promising paradigm for problem solving but further research is needed to effectively address arbitrary planning tasks.

- *Representation of generalized planning tasks*. Implicit representations allow to handle large sets of planning instances. However, these representations require specific mechanisms for state progression, as well as for testing goals and action preconditions, that are different from the ones traditionally implemented in off-the-shelf planners.

Apart from the representation formalism, the given set of instances in a generalized planning task affects to the performance of the different approaches for computing a plan that generalizes. Sometimes small sets of representative instances can be built using *corner cases*. Corner cases push state variables to their minimum or maximum value so the plan behavior, is only considered on those specific states as opposed to consider all the possible input instances. For the general case, it is complex to automatically identify a small number of representative instances so often, the selection of representative instances in a generalized planning task, is still done by hand.

A first step towards automatically determining the instances to compute a solution that generalizes is characterizing the conditions under which a policy generalizes to other problems (Bonet and Geffner, 2015). This approach opens the door to the development of methods for automatically generating the simplest set of instances that is required to compute a solution that generalizes.

- *Computation of generalized plans*. Current algorithms for generalized planning are only able to address relatively small tasks. Further research on specific heuristics for generalized planning, the automatic identification of relevant state variables (e.g. finding the subset of state variables that could appear in the conditions of the loops and branches) or the automatic serialization of goals, can help to increase the scalability of generalized planners.

Domain-specific decompositions allow also to address more challenging generalized planning tasks (Segovia-Aguas *et al*., 2016a). Unfortunately these decompositions are currently done by hand and it is still an open issue how to automatically compute them from the representation of the generalized planning task. With this regard, *planning landmarks* can be an interesting research direction (Hoffmann *et al*., 2004). An alternative work-line to improve the scalability of generalized planners is to explore the transformation of a given planning task into a smaller one that (1) is solvable by the same generalized plan and (2) has a more tractable search space (Bonet and Geffner, 2015).

With regard to the reuse of generalized plans, key issues are the evaluation of the suitability of a given generalized plan for a given planning instance (like similarity metrics from *CBP*), and the reuse of incomplete or incorrect generalized plans. In this case, reusing existing generalized plans as *domain-specific heuristics* or *preferences*, is a safer approach that forcing to follow the generalized plans at every moment.

- *Representation of generalized plans*. Generalized plans that include variables and control-flow require more sophisticated execution mechanisms than a plan that just comprises a sequence of ground actions but, they may be able to represent more tasks. The same claim applies for partially specified solutions (whose execution is more complex because it requires a planner) with respect to fully specified solutions. Given a generalized planning task, identifying the kind of solution that is more suitable for solving it, is also an open issue.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

The computation of a generalized plan is constrained by the given instances in the generalized planning task but also by the given representation for coding the states, actions and goals. The automatic derivation of alternative representations that allow more effective computation of generalized plans is a promising research direction with multiple links to previous research on AI such as ILP *predicate invention* (Craven and Slattery, 2001) or *feature generation* in ML.

Last but not least, generalized plans are generative models that can address tasks beyond planning. For instance, given a generalized plan and an execution trace, the *parsing task* can be defined as the task of determining whether that execution trace could be generated with the given generalized plan. This approach is useful for object classification (Lotinac *et al*., 2016) but also for goal recognition (Ramírez and Geffner, 2010) and task classification (Segovia-Aguas *et al*., 2017b). Furthermore, solutions to these tasks can be implemented with techniques very similar to the ones used for the computation of generalized plans. With this same regard, there are previous works using PbE techniques to synthesize a parser from input/output examples (Leung *et al*., 2015). This task have been addressed with classical planning for small context-free grammars (Segovia-Aguas *et al*., 2017a), however, further research has to be done for building more challenging parsers.

Acknowledgment

This work is partially supported by grant TIN-2015-67959 and the Maria de Maeztu Units of Excellence Programme MDM-2015-0502, MEC, Spain. S. J. is supported by the *Ramon y Cajal* program, RYC-2015-18009, funded by the Spanish government.

References

Albore, A., Palacios, H. & Geffner, H. 2009. A translation-based approach to contingent planning. In *IJCAI*.

Albore, A., Ramrez, M. & Geffner, H. 2011. Effective heuristics and belief tracking for planning with incomplete information. In *ICAPS*.

Alford, R., Kuter, U. & Nau, D. S. 2009. Translating htns to PDDL: a small amount of domain knowledge can go a long way. In *IJCAI*.

Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E. & Udupa, A. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* **40**, 1–25.

Bacchus, F. & Kabanza., F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* **116**(1), 123–191.

Bäckström, C., Jonsson, A. & Jonsson, P. 2014. Automaton plans. *Journal of Artificial Intelligence Research* **51**, 255–291.

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

Bernhard, N. 2000. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research* **12**, 271–315.

Bonet, B. & Geffner, H. 2014. Belief tracking for planning with sensing: width, complexity and approximations. *Journal of Artificial Intelligence Research* **50**, 923–970.

Bonet, B. & Geffner, H. 2015. Policies that generalize: solving many planning problems with the same policy. In *IJCAI*.

Bonet, B., Palacios, H. & Geffner, H. 2010. Automatic derivation of finite-state machines for behavior control. In *AAAI*.

Borrajo, D., Roubickova, A. & Serina, I. 2015. Progress in case-based planning. *ACM Computing Surveys (CSUR)* **47** (2), 35.

Botea, A., Enzenberger, M., Müller, M. & Schaeffer, J. 2005. Macro-ff: improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* **24**, 581–621.

Cimatti, A., Pistore, M., Roveri, M. & Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* **147**(1–2), 35–84.

Cimatti, A., Roveri, M. & Bertoli, P. 2004. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence* **159**(1–2), 127–206.

Claßen, J., Engelmann, V., Lakemeyer, G. & Röger, G. 2008. Integrating golog and planning: an empirical evaluation. In *Non-Monotonic Reasoning Workshop*.

Clarke, E. M., Grumberg, O. & Peled, D. 1999. *Model checking*. MIT press.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

Coles, A. & Smith, A. 2007. Marvin: a heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research* **28**, 119–156.

Craven, M. & Slattery, S. 2001. Relational learning with statistical predicate invention: better models for hypertext. *Machine Learning* **43**(1), 97–119.

Cresswell, S. & Alexandra, M. 2004. Coddington. Compilation of LTL goal formulas into PDDL. In *ECAI*.

Domshlak, C. 2013. Fault tolerant planning: complexity and compilation. In *ICAPS*.

Fern, A., Khardon, R. & Tadepalli, P. 2011. The first learning track of the international planning competition. *Machine Learning* **84**(1–2), 81–107.

Fern, A., Yoon, S. & Givan, R. 2006. Approximate policy iteration with a policy language bias: solving relational markov decision processes. *Journal of Artificial Intelligence Research* **25**, 75–118.

Fikes, R. E., Hart, P. E. & Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial intelligence* **3**, 251–288.

Fox, M., Gerevini, A., Long, D. & Serina, I. 2006. Plan stability: replanning versus plan repair. In *ICAPS*.

Fox, M. & Long, D. 2003. Pddl2. 1: an extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research* **20**, 61–124.

Francès, G. & Geffner, H. 2015. Modeling and computation in planning: better heuristics from more expressive languages. In *ICAPS*.

Francès, G. & Geffner, H. 2016. E-strips: existential quantification in planning and constraint satisfaction. In *IJCAI*.

Frances, G., Ramrez, M., Lipovetzky, N. & Geffner, H. 2017. Purely declarative action representations are overrated: classical planning with simulators. In *IJCAI*.

Fritz, C., Baier, J. A. & McIlraith, S. A. 2008. Congolog, sin trans: compiling congolog into basic action theories for planning and beyond. In *KR*.

Geffner, H. & Bonet, B. 2013. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* **8**(1), 1–141.

Gerevini, A. & Long, D. 2005. Plan constraints and preferences in pddl3. The language of the fifth international planning competition. Technical Report, Department of Electronics for Automation, University of Brescia, 75.

Ghallab, M., Nau, D. & Traverso, P. 2004. *Automated Planning: Theory and Practice*. Elsevier.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices* **46**, 317–330. ACM.

Gulwani, S., Hernandez-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U. & Zorn, B. 2015. Inductive programming meets the real world. *Communications of the ACM* **58**, 90–99.

Hector, J. 2005. Levesque. Planning with loops. In *IJCAI*.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* **26**, 191–246.

Hoffmann, J. 2015. Simulated penetration testing: From dijkstra to turing test++. In *ICAPS*.

Hoffmann, J. & Brafman, R. I. 2006. Conformant planning via heuristic forward search: a new approach. *Artificial Intelligence* **170**(6–7), 507–541.

Hoffmann, J., Porteous, J. & Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* **22**, 215–278.

Howey, R., Long, D. & Fox, M. 2004. Val: automatic plan validation, continuous effects and mixed initiative planning using pddl. In *ICTAI*.

Hu, Y. & Giacomo, G. D. 2011. Generalized planning: synthesizing plans that work for multiple environments. In *IJCAI*.

Hu, Y. & Giacomo, G. D. 2013. A generic technique for synthesizing bounded finite-state controllers. In *ICAPS*.

Hu, Y. & Levesque, H. J. 2011. A correctness result for reasoning about one-dimensional planning problems. In *IJCAI*.

Ivankovic, F. & Haslum, P. 2015. Optimal planning with axioms. In *IJCAI*.

Jiménez, S. & Jonsson, A. 2015. Computing plans with control flow and procedures using a classical planner. In *SOCS*.

Jonsson, A. 2009. The role of macros in tractable planning. *Journal of Artificial Intelligence Research* **36**, 471–511.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence* **113**(1), 125–148.

Kolobov, A. 2012. Planning with markov decision processes: an AI perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning* **6**(1), 1–210.

Lake, B. M., Salakhutdinov, R. & Tenenbaum, J. B. 2015. Human-level concept learning through probabilistic program induction. *Science* **350**(6266), 1332–1338.

Leung, A., Sarracino, J. & Lerner, S. 2015. Interactive parser synthesis by example. In *ACM SIGPLAN Notices*, **50**, 565–574. ACM.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F. & Scherl, R. B. 1997. Golog: a logic programming language for dynamic domains. *The Journal of Logic Programming* **31**(1–3), 59–83.

Long, D. & Fox, M. 2003. The 3rd international planning competition: results and analysis. *Journal of Artificial Intelligence Research* **20**, 1–59.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

Lotinac, D., Segovia-Aguas, J., Jiménez, S. & Jonsson, A. 2016. Automatic generation of high-level state features for generalized planning. In *IJCAI*.

Lukás, C. 2010. Generation of macro-operators via investigation of action dependencies in plans. *The Knowledge Engineering Review* **25**(3), 281–297.

Marthi, B., Russell, S. J. & Wolfe, J. A. 2007. Angelic semantics for high-level actions. In *ICAPS*.

Martn, M. & Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Applied Intelligence* **20**(1), 9–19.

Mausam & Kolobov, A. 2012. Planning with markov decision processes: an AI perspective. *Morgan & Claypool Publishers*.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. & Wilkins, D. 1998. Pddl-the planning domain definition language.

Mitchell, T. M. 1982. Generalization as search. *Artificial Intelligence* **18**, 203–226.

Mitchell, T. M. 1997. *Machine Learning*, 1st edition. McGraw-Hill Inc.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G. & Petersen, S. 2015. Human-level control through deep reinforcement learning. *Nature* **518**(7540), 529–533.

Muggleton, S. 1999. Inductive logic programming: issues, results and the challenge of learning language in logic. *Artificial Intelligence* **114**(1), 283–296.

Muise, C. J., Belle, V. & McIlraith, S. A. 2014. Computing contingent plans via fully observable non-deterministic planning. In *AAAI*.

Muise, C., McIlraith, S. A. & Belle, V. 2014. Non-deterministic planning with conditional effects. In *ICAPS*.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D. & Yaman, F. 2003. Shop2: An HTN planning system. *Journal of Artificial Intelligence Research* **20**, 379–404.

Newell, A., Shaw, J. C. & Simon, H. A. 1959. A general problem-solving program for a computer. *Computers and Automation* **8**(7), 10–16.

Palacios, H. & Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research* **35**, 623–675.

Pan, S. J. & Yang, Q. 2010. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering* **22** (10), 1345–1359.

Pednault, E. P. D. 1989. Adl: Exploring the middle ground between strips and the situation calculus. In *KR*.

Petrick, R. P. A. & Bacchus, F. 2004. Extending the knowledge-based approach to planning with incomplete information and sensing. In *ICAPS*.

Porco, A., Machado, A. & Bonet, B. 2011. Automatic polytime reductions of np problems into a fragment of strips. In *ICAPS*.

Pralet, C., Verfaillie, G., Lematre, M. & Infantes, G. 2010. Constraint-based controller synthesis in non-deterministic and partially observable domains. In *ECAI*.

Ross Quinlan, J. 1990. Learning logical definitions from relations. *Machine Learning* **5**, 239–266.

Ramírez, M. & Geffner, H. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *AAAI*.

Ramirez, M. & Geffner, H. 2016. Heuristics for planning, plan recognition and parsing. *arXiv preprint arXiv:1605.05807*.

Rintanen, J. 2012. Planning as satisfiability: heuristics. *Artificial Intelligence Journal* **193**, 45–86.

Rintanen, J. 2015. Impact of modeling languages on the theory and practice in planning research. In *AAAI*, 4052–4056.

Röger, G., Helmert, M. & Nebel, B. 2008. On the relative expressiveness of adl and golog: the last piece in the puzzle. In *KR*.

Rosa, T. D. L., Jiménez, S., Fuentetaja, R. & Borrajo, D. 2011. Scaling up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research* **40**, 767–813.

Sardina, S., Giacomo, G. D., Lespérance, Y. & Levesque, H. J. 2004. On the semantics of deliberation in indigolog from theory to implementation. *Annals of Mathematics and Artificial Intelligence* **41**(2–4), 259–299.

Scala, E., Ramirez, M., Haslum, P. & Thiebaux, S. 2016. Numeric planning with disjunctive global constraints via smt. In *ICAPS*.

Schwinghammer, J., Birkedal, L., Reus, B. & Yang, H. 2009. Nested hoare triples and frame rules for higher-order store. In *International Workshop on Computer Science Logic*.

Segovia-Aguas, J., Jiménez, S. & Jonsson, A. 2016a. Generalized planning with procedural domain control knowledge. In *ICAPS*.

Segovia-Aguas, J., Jiménez, S. & Jonsson, A. 2016b. Hierarchical finite state controllers for generalized planning. In *IJCAI*.

Segovia-Aguas, J., Jiménez, S. & Jonsson, A. 2017a. Generating context-free grammars using classical planning. In *IJCAI*.

Segovia-Aguas, J., Jiménez, S. & Jonsson, A. 2017b. Unsupervised classification of planning instances. In *ICAPS*.

Shivashankar, V., Kuter, U., Nau, D. & Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*.

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

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S. & Saraswat, V. 2006. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review* **40**, 404–415.

Srivastava, S., Immerman, N. & Zilberstein, S. 2011a. A new representation and associated algorithms for generalized planning. *Artificial Intelligence* **175**(2), 615–647.

Srivastava, S., Immerman, N., Zilberstein, S. & Zhang, T. 2011b. Directed search for generalized plans using classical planners. In *ICAPS*.

Thiébaux, S., Hoffmann, J. & Nebel, B. 2005. In defense of pddl axioms. *Artificial Intelligence* **168**(1), 38–69.

Torlak, E. & Bodik, R. 2013. Growing solver-aided languages with rosette. In *ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 135–152. ACM.

Utgoff, P. E. 1989. Incremental induction of decision trees. *Machine Learning* **4**(2), 161–186.

Vallati, M., Chrpa, L., Grzes, M., McCluskey, T. L., Roberts, M. & Sanner, S. 2015. The 2014 international planning competition: progress and trends. *AI Magazine* **36**(3), 90–98.

Veloso, M., Carbonell, J., Perez, A., Borrajo, D., Fink, E. & Blythe, J. 1995. Integrating planning and learning: the prodigy architecture. *Journal of Experimental & Theoretical Artificial Intelligence* **7**(1), 81–120.

Winner, E. & Veloso, M. 2003. Distill: learning domain-specific planners by example. In *ICML*.

Winner, E. & Veloso, M. 2007. Loopdistill: Learning looping domain-specific planners from example plans. In *ICAPS, Workshop on Artificial Intelligence Planning and Learning*.

Yoon, S., Fern, A. & Givan, R. 2008. Learning control knowledge for forward search planning. *The Journal of Machine Learning Research* **9**, 683–718.

Younes, H. L. S. & Littman, M. L. 2004. PPDDL1. 0: an extension to pddl for expressing planning domains with probabilistic effects. *Technical Report*, *CMU-CS-04-162*.

https://doi.org/10.1017/S0269888918000231 Published online by Cambridge University Press