Program Synthesis for Generalized Planning

Javier Segovia Aguas

TESI DOCTORAL UPF / 2018

Thesis advisor

Prof. Dr. Anders Jonsson,
Department of Information and Communication Technologies

To my family and specially to my beloved future wife

Acknowledgements

This thesis would be impossible to write without the help and support of many people. First I have to express my gratitude to my PhD supervisor Anders Jonsson for his infinite patience and being the light I needed in the darkest moments. You are an exceptional researcher and a better person in which I hope to become one day. I am also grateful to team-up with Sergio Jiménez for such a nice discussions about life, future and research. You are an amazing human being. We have all worked well and hard to explore new research lanes and building bridges between scientific communities. I have the feeling this is just the beginning, thank you both.

It has also been a pleasure to share these years of teaching duties, deadlines and contests with generous and wonderful people. Thanks in no special order to Filippos Kominis for giving always a different perspective about life; Damir Lotinac for offering everything he has at his disposal in every situation; Jonathan Ferrer for his happiness and kind answers of every single question I had; Guillem Francés for your insightful talks and showing that research is more than epsilon; Oussam Larkem for his resilience when things do not go as expected; Iulia Olkhovskaia for teaching us useful words in russian; and Miquel Junyent for his peaceful attitude even in situations of maximum stress.

I am very thankful to people from the Artificial Intelligence and Machine Learning group of Universitat Pompeu Fabra. I have learnt a lot from all of you Hector Palacios, Dimitri Ognibene, Martí Sánchez, Vicenç Gómez, Gergely Neu, Vladimir Estivill, Victor Dalmau, and Jorge Lobo. It has also been a privilege to share my time with one of best teachers and researchers I have met so far, Hector Geffner thank you for sharing your wisdom with us. I want to mention also to the ETIC secretary team for the nice work they do, that is hidden for the rest of people but we know is a big burden, so thank you for your efforts to ease our lives. On the other hand, I am thankful to all bachelor students I have been teaching at UPF and specially to those with the hunger of knowledge that decide to participate in extra events like programming competitions in their free time.

The experience of this PhD has been completed thanks to Adrian Pierce and Nir Lipovetzky that kindly accepted me as a visitor researcher in the University of Melbourne. I have been very lucky to find incredible human beings in Melbourne like Nir, Miquel, Sebastian, Angela, Nim, Meiling, Louis and many others that made my life easier and lovely during my stage.

This work has been feasible mostly because of the huge effort the planning community is doing to provide open access to research and tools for other re-

searchers to explore. This thesis is also dedicated to all those who have left their research careers due to the lack of funds.

Finally, I would like to thank to all my friends for showing me the beauty of life and sharing good moments practicing sports, meeting in restaurants and playing board games. To my parents Tere and Rafa, my sisters Ali and Laura, my parents in law Vicky and Fran, my loving dog Fosca and the rest of my family for their inconditional love, teaching me that happiness is about enjoying the path not the goal, and that great things are achieved step by step. In last instance, I want to mention my eternal gratitude to Patricia Fernández, this is partly her work too, for her love and support in the worst and the best moments; she is always doing the effort of understading that what I do goes beyond myself. I love you all.

Abstract

Generalized planning is the problem of finding an algorithm-like solution called *generalized plan* to multiple planning instances. The two main tasks to perform in generalized planning are *synthesizing* and *validating* generalized plans. In this thesis, we represent generalized plans as a *planning programs*, enhanced with *conditional goto* conditions, or *finite state controllers*. Then, we compile generalized planning problems to PDDL such that we can compute programs using off-the-shelf *classical planners*. Because solutions to generalized planning are similar to algorithms, we can build libraries of previous knowledge and reuse them if necessary using a *call stack*. This feature extends to *planning programs with procedures*, *hierarchical finite state controllers* and allows recursion. Finally, we introduce new application areas for planning, e.g. *unsupervised classification of instances* or *context-free grammar generation*, by defining *non-deterministic choice* functions for planning programs.

Resumen

La *Planificación Generalizada* es el problema de encontrar una solución en forma de algoritmo llamada *plan generalizado* a múltiples instancias de planificación. Las principales tareas son la *síntesis* y la *validación* de planes generalizados. En esta tesis, representamos los planes generalizados como *programas de planificación* mejorados con *saltos condicionales*, o *controladores de estado finitos*. Después, compilamos problemas de planificación generalizada a PDDL tal que podamos computarlos utilizando cualquier *planificador clásico* que esté listo para usarse. Como las soluciones son algoritmos, podemos construir librerías de conocimiento previo y reutilizarlas con una *pila de llamadas*. Esta característica se extiende a los *programas de planificación con procedimientos*, a los *controladores de estado finitos jerárquicos* y permite recursividad. Finalmente, damos a conocer nuevas areas dónde aplicar planificación, por ejemplo la *clasificación no supervisada de instancias* o la *generación de gramática libre de contexto*, gracias a la definición de una *función de elección no determinista*.

Resum

La *Planificació Generalitzada* és el problema de trobar una solució en forma d'algorisme anomenat *pla generalitzat* a múltiples instàncies de planificació. Les principals tasques són la *síntesi* i la *validació* de plans generalizats. En aquesta tesi, representem els plans generalizats com *programes de planificació* millorats amb *salts condicionals*, o *controladors d'estat finits*. Després, compilem problemes de planificació generalitzada a PDDL per poder computar-los amb qualsevol *planificador clàssic* que estigui llest per utilitzar-se. Com les solucions són algorismes, podem construir llibreries de coneixement previ y reutilitzar-les amb una *pila de trucades*. Aquesta característica s'estén als *programes de planificació amb procediments*, als *controladors d'estat finits jeràrquics* i permet recursivitat. Finalmente, donem a conèixer noves àrees on aplicar planificació, per exemple la *classificació no supervisada d'instàncies* o la *generació de gramàtica lliure de context*, gràcies a la definició d'una *funció d'elecció no determinista*.

Preface

The interest of *general solvers* is rooted in the origins of Artificial Intelligence (Newell et al., 1959) with the motivation of mimicking the human thinking process. One possible interpretation of a general solver, is the capability of solving problems from any domain independently. Part I of the thesis starts with Classical Planning, whose solvers fall into the category of being domain independent, so instances from different planning domains can be solved using the same classical planning solver (see Chapter 1). Another possible interpretation of a general solver, is the generalization over a set of problems, also known as Generalized Planning (see Chapter 2). In other words, the general solver must find a solution that solves each problem from a set of planning problems (Hu and De Giacomo, 2011). The solution to a generalized planning problem is a generalized plan. In the rest of this thesis we show multiple representations and computations of generalized plans.

We mainly represent and compute generalized plans as planning programs in Part II. A planning program is a set of instructions enhanced with conditional gotos that can be programmed and executed (see Chapter 3). We show a methodology to compile a set of planning problems into a single PDDL domain and instance. Then, we can use any off-the-shelf classical planner to synthesize and validate programs.

We continue with an extension of planning programs with a call stack that simulates a computer stack in PDDL (see Chapter 4). This allows to include and call previous knowledge in form of planning programs as procedures. We can even synthesize recursive solutions when procedures call themselves, and if procedures have parameters they can find solutions like a depth-first search (DFS) algorithm for traversing a binary tree. In addition to planning program formalism and influenced by Baier et al. [2007], we show results when instructions are lifted such that they are used as Domain Control Knowledge (DCK). We define planning programs with lifted actions as *non-deterministic planning programs* (see Chapter 5).

Programs synthesis (Gulwani et al., 2017) is the problem of automatically finding a program given some input specification. These programs are represented in many different formalisms, i.e. from logical formulae (Cresswell and Coddington, 2004; Patrizi et al., 2011), to Finite State Controllers (FSCs) (Bonet et al., 2010; Hu and De Giacomo, 2013). Thus, in Part III we explore the connection of program synthesis as generalized planning with FSCs. In Chapter 6 we show how to synthesize FSCs following a similar approach to planning programs. Further-

more, the call stack already implemented in Chapter 4 can be used to generate a hierarchy of FSCs.

In Part IV, we extend planning programs with the concept of *choice instruction*. The first approach includes the formalism of planning programs with procedures, and adds a non-deterministic choice function that can call online any of the previous procedures to solve a planning instance. This problem of choosing the correct procedure to execute is connected to plan recognition (Ramírez and Geffner, 2010). Thus, we call *unsupervised classification of planning instances* (see Chapter 7) the process of automatically determining the procedure that correctly solves a planning problem. In the second approach, the non-determinist choice function is programmed on the first line of each procedure, and the planner has to choose online a valid program line to jump and execute a piece of code. This representation is equivalent to a context-free grammar (CFG) from which we can perform the tasks of string parsing and production using a classical planner (see Chapter 8).

The last part of the thesis (Part V) is to discuss the related work (see Chapter 9) of the previous chapters. The idea of dedicating a chapter to this topic is to avoid repeating citations because some previous work is connected to many chapters. The last chapter is to wrap-up the dissertation (see Chapter 10) and explain some future work to move towards a general AI.

Most of the content of this dissertation is under review or has been published in the following articles:

- Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. *Generalized planning with procedural domain control knowledge*. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling*, pages 285-293, 2016. [Chapter 4]

- Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. *Hierarchical finite state controllers for generalized planning*. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 3235-3241, 2016. (winner of *IJCAI-16 Distinguished Paper Award*) [Chapter 6]

- Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. *Unsupervised Classification of Planning Instances*. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling*, pages 452-460, 2017. [Chapter 7]

- Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. *Generating context-free grammars using classical planning*. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 4391-4397, 2017. [Chapter 8]

- Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. *Computing Hierarchical Finite State Controllers with Classical Planning*. In *Journal of Artificial Intelligence Research*, 2018. (accepted for publication) [Chapter 6]

- Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. *Computing programs for generalized planning using a classical planner*. In *Artificial Intelligence*, 2018. (major revision submitted) [Chapter 3, Chapter 4, and Chapter 5]

- Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. *A review of generalized planning*. In *The Knowledge Engineering Review*, 2018. (major revision submitted) [Chapter 2 and Chapter 9]

Other publications not included in the contents of the thesis are:

- Damir Lotinac, Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. *Automatic generation of high-level state features for generalized planning*. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 3199-3205, 2016.

- Javier Segovia-Aguas, Jonathan Ferrer-Mestres and Anders Jonsson. *Planning with Partially Specified Behaviors*. In *Proceedings of the 19th International Conference of the Catalan Association for Artificial Intelligence*, pages 263-272, 2016.

Contents

List of Figures

List of Tables

PART I

Background

CHAPTER 1

Classical Planning

In this chapter we introduce the state model and factored representation of Classical Planning. We then explain how the model can be extended with conditional effects and conclude with a review of its complexity for plan existence and bounded plan existence.

1.1 Introduction

Planning is one of the branches of Artificial Intelligence, also called Automated Planning and Scheduling or AI Planning. The agents or systems are in charge of solving the planning problems by following a strategy or a sequence of actions.

One particular case of AI Planning is Classical Planning (Russell et al., 1995), where the environment and the actions are deterministic. On one hand, classical planning requires a language to specify the problem. The most extended language in the planning community is called Planning Domain Definition Language (PDDL). On the other hand, it requires solvers or planners to compute solutions or plans.

Definition 1.1 (Classical Planning)**.** *Classical planning can be stated as a path finding problem on a directed graph, where nodes are states and the directed edges are the actions. Applying an action in a state always leads to the same new state. The solution is a sequence of actions called plan, that applied in the initial state reaches a goal state.*

Because classical planning is a path finding problem, we can use any search algorithm to solve it, but the main limitation is that problems may have huge

state spaces, e.g. 2^{1000} states. In that case we need some improvements over the classic search algorithms such as Breadth First Search (BrFS), Best First Search (BFS), A* among others, that either are too slow or require too much memory. The two main techniques we are going to consider in this work are planning with Heuristic Search (Bonet and Geffner, 2001) and Structure Search (Lipovetzky and Geffner, 2012). These techniques find solutions in practice to problems that are theoretically intractable.

Definition 1.2 (Heuristic Search)**.** *Heuristic Search is the process of using heuristics in search algorithms to speed up solution finding. A heuristic is a function to estimate the path cost from the current state to a goal state.*

Heuristics use information of a goal condition to estimate a value in the current state that represent the cost-to-go for reaching a goal. For this reason heuristic search is goal oriented. Heuristics can be computed automatically in domain independent solvers, but also can be chosen for domain-specific solvers following a trade-off criteria of optimality, completeness, and time complexity. The most commonly used heuristic search solver is the Fast-Downward (FD) system (Helmert, 2006a) with the LAMA-2011 setting (Richter and Westphal, 2010).

Definition 1.3 (Structure Search)**.** *Structure Search is a blind search technique that exploits the changes in the state when an action is performed. In order to decide if a new state should be expanded or pruned, it computes a value based on the changes in the new state and checks this value against a given bound.*

In contrast with heuristic search, structure search only uses information of the current and the next state, so it is not goal oriented. Serialized Iterative Width (IW) (Lipovetzky and Geffner, 2012) is a planner that can solve a significant amount of problems in the plan satisfiability track from the International Planning Competition (IPC) (Vallati et al., 2015) but in most of the problems heuristic search perfoms better.

The state-of-the-art solver is called Best First Width Search (BFWS) (Lipovetzky and Geffner, 2017). It is a heuristic search algorithm that prioritizes states by the value provided of the computation of structure search techniques. In other words, it combines the advantages of both techniques into a heuristic search approach.

1.2 The State Model

The planning state model is represented by a tuple $\mathcal{S} = \langle S, s_0, S_G, A, \theta, c \rangle$ where:

- S is the finite set of discrete states.

- $s_0 \in S$ is the initial state.
- $S_G \subseteq S$ is the set of goal states.
- A is the set of actions, and $A(s) \subseteq A$ is the set of actions that are applicable in state $s \in S$.
- $\theta(s, a)$ is the deterministic transition function that maps state and action pairs into a state, $\theta : S \times A \to S$.
- $c(s, a)$ is the cost function that maps state and action pairs into a value, $c : S \times A \to \mathbb{R}^+$. By default this value is always 1.

The transition and cost functions are defined over the set of applicable actions. Thus $s' = \theta(s, a)$ and $c(s, a)$ exist when $a \in A(s)$, also denoted as $a(s)$.

A *plan* π is a sequence of applicable actions $\langle a_1(s_0), ..., a_n(s_{n-1})\rangle \in A$ where each step $i \in [1, n]$ genetares a new state $s_i = \theta(s_{i-1}, a_i(s_{i-1}))$. A *solution* is a plan that solves the planning problem that when applied in the initial state $s_0 \in S$ reaches a goal state $s_n \in S_G$ in n steps.

Definition 1.4 (Plan Space)**.** *Plan Space is the set of all possible plans that solve a planning problem. So any of these plans applied in the initial state reaches a goal state.*

Equation 1.1 shows the general cost computation of a plan. When actions have default costs, the cost of a plan is equivalent to the size of the plan $|\pi|$.

$$cost(\pi) = \sum_{i=1}^{n} c(s_{i-1}, a_i). \tag{1.1}$$

A plan is considered optimal if its cost is the minimum among all plans in the plan space. There could be multiple optimal plans for a problem, for instance a 2×2 grid where an agent should move from the bottom-left corner to the top-right corner through adjacent tiles (or tiles that share one side), has two possible optimal plans: either i) it moves up and then right; or ii) it moves right and then up.

1.3 Factored Representation and Languages

A factored representation describes the state with a set of variables that have finite and discrete values. It is used to simplify the model description with huge state-spaces and be more informative for planning techniques.

In the year 1971, the automated planner Stanford Research Institute Problem Solver (STRIPS) (Fikes and Nilsson, 1971) was developed as a planning solver to find sequences of actions of world models in conjunction with robotics research. Later, the same name was used as a formal input language for classical planning solvers. From now on we refer to STRIPS as the language and not the planner. In STRIPS , the factored representation of the state is a set of boolean variables that can be either true or false. The STRIPS model is a 4-tuple $\mathcal{P} = \langle F, A, I, G \rangle$ where:

- F is the set of propositional variables
- A is the set of actions
- $I \subseteq F$ is the initial state, that is a subset of the propositions
- $G \subseteq F$ is the goal condition and is another subset of the propositions

The set of *fluents* F are propositional variables that are used to describe states. We define propositional variables as instantiations of *predicates* with *arguments*. A predicate p belongs to the set of predicate symbols Ψ, and has an associated argument list of arity $\mathsf{ar}(p)$. Given the set of objects Ω, the set of fluents F is induced from the assignment of objects to each predicate, i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{\mathsf{ar}(p)}\}$, where $\Omega^{\mathsf{ar}(p)}$ is the $\mathsf{ar}(p)$-th cartesian product of set Ω.

In order to simplify definitions throughout this thesis we refer to valuations of fluents $f \in F$ as *literals*. A literal $l = f$ is used to denote that l assigns true to f, while $l = \neg f$ assigns false to f. The set of literals L is *well-defined* if it does not assign conflicting values to any fluent, i.e. for some fluent $f \in F$, the set of literals L does not contain f and $\neg f$ at the same time where $f = \neg(\neg f)$. Also it is *fully-assigned* if it assigns values to all fluents $f \in F$ where $|L| = |F|$. We use $\neg L$ to denote the complement of L such that $\neg L = \{\neg l : l \in L\}$.

Now we describe a state s as a well-defined and fully-assigned set of literals L such that $|s| = |F|$, resulting in a state-space S bounded in size by $2^{|F|}$. Then, we have a well-defined and fully-assigned initial state described as $s_0 = \{f : f \in I\} \cup \{\neg f : f \notin I\}$, and set of goal states $S_G = \{s \in S : G \subseteq s\}$. In case we only consider a subset of fluents $F' \subseteq F$, we denote $s|_{F'}$ as the *projection* of s onto F', defined as $s|_{F'} = (s \cap F') \cup (s \cap \neg F')$.

Every action $a \in A$ is composed of preconditions $\mathsf{pre}(a) \subseteq F$, negative effects $\mathsf{del}(a) \subseteq F$ and positive effects $\mathsf{add}(a) \subseteq F$. An action is applicable if and only if the preconditions of the action holds in the state s, i.e. $\mathsf{pre}(a) \subseteq s$, and the transition function $\theta(s, a)$ produces a new well-defined state removing first the negative effects and then adding the positive effects, as shown in Equation 1.2.

$$\theta(s, a) = (s \setminus \mathsf{del}(a)) \cup \mathsf{add}(a) \tag{1.2}$$

Planning Languages

The Action Description Language (ADL) (Pednault, 1994) extended STRIPS allowing actions to have conditional effects, so some effects are triggered depending on the state. In contrast to STRIPS , ADL allows existential quantification, negative literals, goals with disjunctions, different object types, and while variables in STRIPS are either true or false, in ADL they can be true, false or undefined.

In the year 1998, with the aim to make a standard representation of planning languages, the Planning Domain Definition Language (PDDL) (McDermott et al., 1998) was published. The IPC has been used to compare planning solvers performance but also to include new features to the language. These are the different published versions:

- **PDDL1.2** is the version used in the first international planning competition IPC-98 (Long et al., 2000) where the planning problem model is splitted into a *domain* and a *problem* description.

- **PDDL2.1** (Fox and Long, 2003) introduced *numeric fluents* so resources can be represented. In previous versions, actions were directly applied in discrete time, but in this version actions can be described and performed in continuous space, so they are described as *temporal* or *durative actions*.

- **PDDL2.2** (Edelkamp and Hoffmann, 2004) introduced *derived predicates* that can represent dependency among literals through *transitive closures*. This version also introduces *timed initial literals* where some literals are triggered by independent events at different times.

- **PDDL3.0** (Gerevini and Long, 2006) introduced hard constraints called *state-trajectory constraints* that must be true along the execution of a plan, and soft constraints called *preferences* where plans that satisfy them are considered of better quality.

- **PDDL3.1** introduced *object-fluents* where any function can be an object type. This feature is a reformulation of Functional STRIPS (Geffner, 2000).

In this dissertation we are going to focus on PDDL2.2, that is the language supported by the most common planning systems like Fast-Forward (FF) (Hoffmann, 2001), Fast-Downward (FD) (Helmert, 2006a), and Best First Width Search (BFWS) (Lipovetzky and Geffner, 2017) developed under the Lightweight Automated Planning Toolkit (LAPKT) (Ramirez et al., 2015). There are also other planning languages like situation calculus (McCarthy, 1963), SAS+ (Bäckström and Nebel, 1995), and PDDL extensions for multiagent planning called MAPL

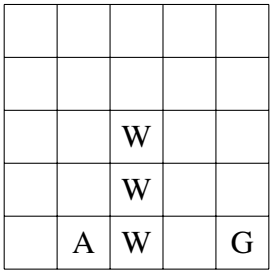


Figure 1.1: Example of a navigation task in a 5×5 grid where the agent is blocked by a wall when it tries to move right. The agent position is specified with letter A, the walls placed in the grid are represented with Ws, and the goal to achieve is to reach the tile marked G.

(Brenner, 2003), PPDDL (Younes and Littman, 2004) for probabilistic planning among others.

In PDDL, the planning problem model is splitted into domain and problem. Under this formalism, the domain is considered a *classical planning frame* defined by the tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions. Then, given a classical planning frame Φ, the problem is instantiated from an initial state I and a goal condition G. Thus, we can define multiple planning problems with the same planning frame Φ, e.g. $P_1 = \langle F, A, I_1, G_1 \rangle$, $P_2 = \langle F, A, I_2, G_2 \rangle$, and so on.

1.4 Planning with Conditional Effects

Conditional effects have been a requirement since the first IPC, where the effects of the action to apply are triggered if their conditions hold in the state. This is a more expressive representation of actions in the model and a basic requirement for generalized planning. This is also feature already shown in confomant planning (Palacios and Geffner, 2009) where a sequence of actions can be reused for many initial states, e.g. the `right` action of a grid domain like in Figure 1.1 has two possible outcomes, either the agent *moves* to the right cell if it is empty or the agent *stays* in the same cell if it is in the rightmost cell of the grid or there is a wall to the right cell.

We consider the fragment of classical planning with conditional effects that includes negative conditions and goals. A planning problem with conditional effects is still a tuple $P = \langle F, A, I, G \rangle$ on a planning frame $\Phi = \langle F, A \rangle$, and the only thing that changes is the definition of actions in A. Each action $a \in A$ has a well-defined literal set $\mathsf{pre}(a)$ called the *precondition* and a set of conditional

effects $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two well-defined literal sets C (the condition) and E (the effect).

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

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

i.e. effects whose conditions hold in s. We assume that $\mathsf{eff}(s, a)$ is a well-defined literal set for each state s and action a. The result of applying a in s is a new state $\theta(s, a) = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$.

Lemma 1.1. *For any state s and action a, $\theta(s, a)$ is a well-defined state.*

Proof. By assumption, $\mathsf{eff}(s, a)$ is a well-defined literal set. Consider any fluent $f \in F$. There are three cases:

1. $\mathsf{eff}(s, a)$ does not mention f. Then the set operations have no effect on f, so $\theta(s, a)$ assigns the same value to f as s.

2. $\mathsf{eff}(s, a)$ and s assign the same value to f. Then subtracting $\neg\mathsf{eff}(s, a)$ from s has no effect on f, and taking the union with $\mathsf{eff}(s, a)$ adds the valuation already present in s, so $\theta(s, a)$ assigns the same value to f as s.

3. $\mathsf{eff}(s, a)$ and s assign different values to f. Then subtracting $\neg\mathsf{eff}(s, a)$ from s eliminates any mention of f, and taking the union with $\mathsf{eff}(s, a)$ causes $\theta(s, a)$ to assign the same value to f as $\mathsf{eff}(s, a)$.

We have shown that $\theta(s, a)$ assigns precisely one value to each fluent f, which is the definition of a state. □

1.5 Complexity

The computational complexity in planning has been deeply studied (Bylander, 1994). In that research, Bylander analyzed the complexity for plan existence and bounded plan existence of a planning problem $\mathcal{S}$ in its factored representation. We refer to plan existence as a decision problem $\mathsf{PE}(\mathcal{S})$ that is defined by the question, *does a plan π exists for planning problem $\mathcal{S}$?*. And to bounded plan existence as a decision problem $\mathsf{PC}(\mathcal{S}, k)$ that is defined by the question, *does a plan π exists for planning problem $\mathcal{S}$ with $cost(\pi) \leq k$?*. Bylander prooved that plan existence $\mathsf{PE}(\mathcal{S})$ and bounded plan existence $\mathsf{PC}(\mathcal{S}, k)$ problems are PSPACE-complete, which means they are worst case intractable and require polynomial memory space. Then, he shows for constant k, the bounded plan existence $\mathsf{PC}(\mathcal{S}, k)$ is

an NP-complete problem that can be used to find an optimal plan π^* for planning problem $\mathcal{S}$. Despite the fact that classical planning problems are worst case intractable, in practice the state-of-the-art solvers are able to find solutions in reasonable time.

1.6 Examples

The classical planning domain we use as an example is Visit-All, where an agent must visit the whole grid starting in the bottom-left corner. This domain was proposed by Geffner and Lipovetzky for the International Planning Competition of 2011. They expose the limits of state-of-the-art heuristics for multiple goal problems, when moving towards a goal is in conflict to reach the others.

In Figure 1.2 there is an agent in the bottom-right corner, of a 3×3 grid, that has already visited two cells.

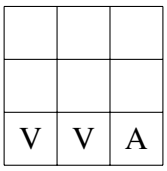


Figure 1.2: Example of a Visit-All task in a 3×3 grid. The agent is the A, the visited cells are Vs, and the rest are the cells the agent must visit.

So to describe last example as PDDL instance, we have to reset agent position to the bottom-left corner or coordinate $(1, 1)$, and no cells visited as shown in Figure 1.3.

The Visit-All PDDL domain is in Figure 1.4 which consists of a domain name, a set of language requirements, types of objects, a set of predicates, and four actions to move the agent in a cardinal direction and a fifth to visit the current cell. Then for the example used in Figure 1.2 we represent the model in PDDL as the conjunction of domain and instance.

Figure 1.5 shows how the previous Visit-All domain can be translated into a domain with conditional effects. The key idea is to avoid parameters in the actions, keeping the preconditions if they do not depend in the parameters and adding all the execution logic of the action in the effects. Because of the size of the domain we just provide the move right action that corresponds to increase by one the x-coordinate.

```
(define (problem grid-3)
 (:domain visit-all)
 (:objects v1 v2 v3 - value )
 (:init
   (consec v1 v2) (consec v2 v3)
   (xpos v1) (ypos v1)
 )
 (:goal (and
   (visited v1 v3) (visited v2 v3) (visited v3 v3)
   (visited v1 v2) (visited v2 v2) (visited v3 v2)
   (visited v1 v1) (visited v2 v1) (visited v3 v1)
 ))
)
```

Figure 1.3: Visit-All instance of a 3×3 grid

1.7 Thesis Outline

In this chapter we have introduced the classical planning model and different languages to represent a planning problem. We have chosen PDDL as the standard language in its version PDDL2.2 to use along this dissertation so we can use any off-the-shelf classical planner. We have formalized the use of literals and conditional effects in the planning model. The first one is used to simplify the formalization of different ideas that allow negative conditions. And the second one is required to trigger different action effects in different states, as happens in conformant planning where a plan must work for any initial state from the set of possible initial states.

In Chapter 2 we introduce the concept and formalization of a generalized planning problem and the solutions called generalized plans. Chapter 3 is the first contribution of this dissertation where a set of planning problems can be compiled into a single one, and solving this single problem corresponds to compute and validate a planning program that represents a generalized plan. Chapter 4 shows a way to compute planning programs simulating a stack with PDDL that allows nested procedures, procedures with parameters and recursive calls. It is followed by non-deterministic planning programs in Chapter 5. The first three correspond to control flow solutions, while the last one belongs to domain control knowledge. We report results for each kind of solution representation.

Another way to represent generalized plans from a different perspective are Finite State Controllers (FSCs), and there is a close connection between this and

```
(define (domain visit-all)
 (:requirements :typing)
 (:types value - object)
 (:predicates
  (xpos ?v - value)
  (ypos ?v - value)
  (consec ?v1 ?v2 - value)
  (visited ?v1 ?v2 - value)
 )
 (:action right
   :parameters (?v1 ?v2 - value)
   :precondition (and (xpos ?v1) (consec ?v1 ?v2) )
   :effect (and (xpos ?v2) (not (xpos ?v1)) )
 )
 (:action left
   :parameters (?v1 ?v2 - value)
   :precondition (and (xpos ?v2) (consec ?v1 ?v2) )
   :effect (and (xpos ?v1) (not (xpos ?v2)) )
 )
 (:action up
   :parameters (?v1 ?v2 - value)
   :precondition (and (ypos ?v1) (consec ?v1 ?v2) )
   :effect (and (ypos ?v2) (not (ypos ?v1)) )
 )
 (:action down
   :parameters (?v1 ?v2 - value)
   :precondition (and (ypos ?v2) (consec ?v1 ?v2) )
   :effect (and (ypos ?v1) (not (ypos ?v2)) )
 )
 (:action visit
   :parameters (?v1 ?v2 - value)
   :precondition (and (xpos ?v1) (ypos ?v2) )
   :effect (and (visit ?v1 ?v2) )
 )
)
```

Figure 1.4: Visit-All domain

```
(:action right
  :parameters ()
  :precondition ()
  :effect (and
    (forall (?v1 ?v2 - value)
        (when (and (xpos ?v1) (consec ?v1 ?v2))
              (and (not (xpos ?v1)) (xpos ?v2)))))
)
```

Figure 1.5: Parameter free action with conditional effects in Visit-All domain

planning programs. We talk about FSCs in Chapter 6 and how to use the PDDL stack to allow a FSC to call another FSC, so that solutions are now Hierarchical FSCs.

The Chapters 3 to 6 are the basic contributions of this thesis, and constitute the baseline to create new landscapes for the planning community as Chapter 7 shows for structured prediction of planning instances, where planning problems are classified into planning program clusters, or Chapter 8 where planning programs can be used to generate context-free grammars.

Finally, we talk in Chapter 9 about the related work to the different chapters, and in Chapter 10 we sum up this work and show ongoing and future work directions that build bridges between planning and learning communities through program synthesis.

CHAPTER 2

Generalized Planning

In Chapter 2 we review how Artificial Intelligence has solved problems with the intention to generalize over domains since the early beginning. We continue with relevant definitions in Generalized Planning (GP) field that affect to problem and solution representation. We also show strategies to compute GP plans, and how problems can be described with a general scheme and evaluated with a set of characteristics.

2.1 Introduction

Since the early days in AI, researchers have been interested in finding a tool that can imitate human *thinking process* by solving problems generally (Newell et al., 1959). The concept of *general solving* has two possible meanings, the most common refers to the mechanism used for solving problems from any domain independently, and the other one is about finding solutions that can be used to solve any problem from the same domain.

Classical Planning or the vanilla model of *Automated Planning* (AP), introduced in Chapter 1, uses a special case of general problem solvers that are planning solvers. Then, given a model description and a set of instances, the solver has to search for a valid plan that solves each particular planning instance. These solutions cannot be used to solve other planning instances.

Generalized Planning (GP) is an extension of AP that combines challenges from planning and knowledge representation communities. There are multiple languages for describing GP problems, and different ways to represent and compute solutions. We define a solution to a GP problem, the one that can solve a set

of problems. GP problems are defined either by properties shared by all instances (Srivastava et al., 2011a) or a set of representative instances (Hu and De Giacomo, 2011).

We use the formal definition of a GP problem from previous approaches (Hu and De Giacomo, 2011):

Definition 2.1 (Generalized Planning Problem)**.** *A Generalized Planning Problem $\mathcal{P} = \{P_1, P_2, \ldots\}$ is the problem of solving a set of planning instances that share some common structure (also known as sharing the agent), i.e. the action scheme and observations.*

For instance, our approaches to generalized planning are represented as a finite set with T planning problems. Thus, we formalize a generalized planning problem as $\mathcal{P} = \{P_1, \ldots, P_T\}$. We usually define GP problems with $T \geq 2$ but in some special cases we can generalize from a single instance.

This definition of a generalized planning problem is not as restrictive as it first may appear. We can define a large fluent set F and action set A, and use the initial state I_t of each planning problem P_t to *"switch on/off"* certain elements. This way, we can address planning problems of various sizes in $\mathcal{P}$, as long as F and A are sufficiently large to accommodate the largest planning problem in $\mathcal{P}$. For example, in a list traversal problem, we can define a set of list nodes $x_0, \ldots, x_{20}$, and use the fluent $\mathsf{end}(x_5)$ to indicate that the given list has length 5. Even though there are fluents in F and actions in A associated with the list nodes $x_6, \ldots, x_{20}$, these fluents and action are not used in this particular planning problem.

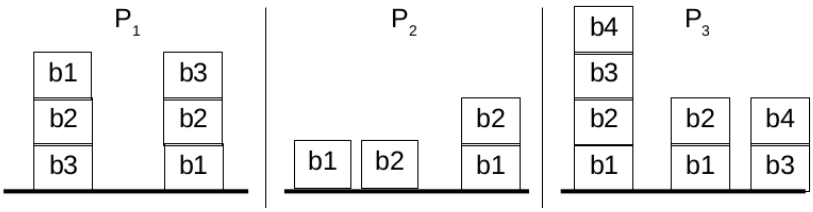


Figure 2.1: Three different example instances from *blocksworld*. Each instance shows the blocks configuration for the initial state (left side) and goal state (right side).

In Figure 2.1 there are several planning problems from *Blocksworld* domain where a configuration of blocks should be changed into another setting. Therefore, we can formulate a *Blocksworld* generalized planning problem from Figure 2.1 as $\mathcal{P} = \{P_1, P_2, P_3\}$.

Then, a generalized planner is required to compute a solution a generalized planning task $\mathcal{P}$. We also use *generalized plan* as a generalized planning task solution with the following formal definition:

Definition 2.2 (Generalized Plan)**.** *A generalized plan* Π *is an algorithm-like structure that solves a generalized planning task* $\mathcal{P}$ *composed of a set of planning instances.*

We consider a generalized plan Π *valid* if and only if it solves each one of the planning instances in $P_t \in \mathcal{P}$ s.t. $t \leq |\mathcal{P}|$. A generalized plan in Figure 2.1 can be any algorithmic-like solution that builds the correct block towers setting after unstacking and putting all blocks onto the table.

that put all blocks onto the table and then stack blocks in the correct order and tower.

We have two possible problems from a generalized planning task specification. The first one is the *generation* of a generalized plan Π given a generalized planning task $\mathcal{P}$. We also refer to the *generation* problem as *computation* or *synthesis*. The second one is the *validation* of a generalized plan Π in a planning instance $P = \langle F, A, I, G \rangle$.

Once a generalized plan is computed, we can talk about concepts that in other fields like learning have been explored for long time, e.g. *generalization*. In *Supervised Learning*, generalization is used to measure the out-of-sample error or the error of the computed solution with samples that have not been used in the training process. Thus we can define generalization in Generalized Planning as the following:

Definition 2.3 (Generalization)**.** *Generalization is the validation of a generalized plan* Π*, synthesized from GP problem* $\mathcal{P}$*, over a different set of planning instances* $\mathcal{P}'$ *such that* $\mathcal{P}' \cap \mathcal{P} = \emptyset$*. Even though, new planning instances from* $\mathcal{P}'$ *must share the same set of actions and observations as the ones in* $\mathcal{P}$*.*

There are general solvers that produce general plans with diverse forms like *DS-planners* (Winner and Veloso, 2003), *generalized policies*(Martín and Geffner, 2004) and *Finite State Machines* (Bonet et al., 2010) (FSMs). Each of them has different syntax and semantics but all include conditional transitions that when executed allow to solve multiple planning instances.

2.2 Abstract Generalized Planning Framework

In this section we introduce the concept of a common framework for generalized planning, where any GP problem and solution can be represented and computed whether they meet the definitions from Section 2.1. We will explain and formalize several representations and computing algorithms of generalized plans in the following chapters.

A contribution in the Generalized Planning community can be in any of these topics: i) problem representation; ii) generalized planner for computing a plan; or iii) plan representation.

We cotribute in this dissertation with a common generalized problem representation, where multiple planning instances are compiled into a single one, as in previous approaches (Hu and De Giacomo, 2011). This compilation produces a single PDDL task which is benefited from state-of-the-art planning techniques by using any off-the-shelf classical planner. Furthermore, we contribute significantly with several computations and representations of generalized plans, i.e. *programs*, *finite state controllers*, *grammars* as we will explain later.

The *representation of a generalized plan* depends on the level of specification (fully, partially or non-specified generalized plans), the algorithm-like execution of the generalized plan (Albore et al., 2009; Levesque, 2005; Bonet et al., 2010; Jiménez and Jonsson, 2015), and evaluation metrics like coverage or succinctness. The *computation of a generalized plan* describes the strategy to find solutions and how they can be reused as previous knowledge to find solutions to other problems.

Independently of the approach used, generalized planning follows the structure of an abstract framework where the input is a *generalized problem* $\mathcal{P}$ and the output is a *generalized plan* Π. This introduces the concept of *generalized planner*:

Definition 2.4 (Generalized Planner)**.** *A Generalized Planner is a framework that solves generalized problems whose solutions are generalized plans. The Generalized Planner is in charge of interpreting and if needed translating the input, as well as computing solutions and mapping those to the desired output structures that correspond to generalized plans.*

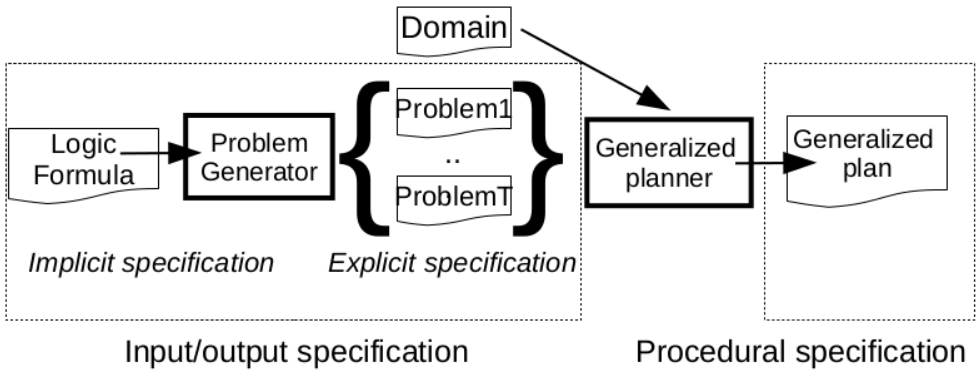


Figure 2.2: Abstract Generalized Planning Framework

The abstract generalized planning framework is graphically described in Figure 2.2. The input to a generalized planner is explicitly defined as a set of tasks or problems and a domain formalized as $\mathcal{P}$, and the set of problems are created from a *problem generator* that follows either *implicitly* a logic formula or property that all instances have to satisfy, or *explicitly* a set of planning instances, as in our approaches. Then the generalized planner has to compute a solution to the set of problems. The solution is the computation of the generalized plan that solve all input problems.

We have also contributed to other representations, that are not explained in the thesis, like high-level state features in the form of *conjuctive queries* that are automatically synthesized (Lotinac et al., 2016). As explained in Section 1.4, we use the representation of actions with conditional effects for all the compilations.

We constrain the state representation to *first order logic* that uses *quantified variables*. Each state contains a set of literals with the `and`, `or` and `not` logical connections. Then, *quantified variables* with transitive closures can represent unbounded sets of states. In Figure 2.3 we have an example of a derived predicate with a transitive closure, that describes when a block is above another block in a blocksworld setting.

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

Figure 2.3: PDDL derived predicate with one existentially quantified variable `?z` that leverages recursion to capture when a block `?x` is above another block `?y`.

2.3 Generalized Plans

Most of the contributions in the generalized planning community are novel algorithms for computing generalized plans. As we introduced, a generalized plan is an algorithm-like structure that solve a set of planning tasks and eventually can be used to solve new planning instances.

In this section we explain the categories for representing generalized plans, as well as their execution and validation, the synthesis strategies and the multiple metrics for evaluating generalized plans.

2.3.1 Representation

The representation of a generalized plan is the level of the specification of the solution that consist on the problem of selecting the action to apply next. Throughout this thesis we are going to introduce approaches with different generalized plan representations.

- **Fully specified** are the generalized plans that have only one applicable action every time, so the selection problem becomes trivial. We refer to fully specified solutions as *deterministic execution* because there is no search in the selection problem.
- **Partially specified** are the generalized plans that only consider a subset of actions from the whole set of applicable actions at each time step while searching for a plan. Thus these generalized plans are called *Domain Control Knowledge* (DCK), where the selection problem is a decision making problem that is constrained to a subset of actions.
- **Non-specified** is the case where all applicable actions are considered as the next action to apply. Therefore the selection problem is equivalent to a planning problem given a domain specification. A domain model can be seen as a form of generalized plan that covers any solvable instance representable with the domain, but the complexity is still PSPACE-complete when applied to a planning instance (see Section 1.5.

2.3.2 Execution and Validation

In contrast to a classical plan, a generalized plan must include the relevant information for a posterior execution and validation over new planning instances. As we explained above, a generalized plan is control flow when is fully specified, or domain control knowledge when it is partially specified. In both cases we need mechanisms that work for multiple problems like *branching* and *looping*.

Definition 2.5 (Branching)**.** *Branching in generalized planning is a control structure that when executed in a GP problem branches the plan after evaluating a condition in the current state.*

Definition 2.6 (Looping)**.** *Lopping in generalized planning is the repetition mechanism that allows to execute multiple times parts of an algorithm-like solution.*

These are the two main mechanisms to exploit the structure for multiple problems. Branching is useful whether input problems differ in their solutions in a specific condition, while looping is used for repetitive tasks that always follow

the same execution sequence. The problem in Figure 2.4 consist of unstacking and dropping blocks from a tower until a green block is held and collected. Thus, Figure 2.4 is a clear example that uses branching and looping techniques with a FSC. This FSC works as follows in $q0$: i) if it holds a block and there is no green in top of the tower, it collects the block and remains in $q0$; ii) if it does not hold a block but observes green block in the top of the tower, the block is unstacked and continues in $q0$; and iii) if it does not observe H or G, then a block from the top of the tower is unstacked and moves to controller state $q1$. Then, the FSC behaves in $q1$: i) unstacking a block from the top of the tower if it does not observe H or G; ii) dropping the current block if it observes H and G; and iii) dropping the block if it is holding one and it is not observing green from the top of the blocks tower. Independently of what $q1$ observes, it performs an action and goes back to $q0$ (looping); while $q0$ evaluates three possible expressions where only one can be true in the current state, and in two of these expressions it continues in $q0$ and in the other one it jumps to $q1$ (branching).

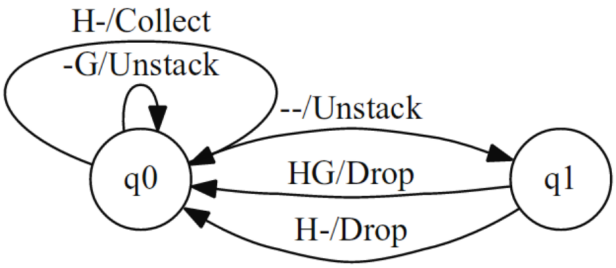


Figure 2.4: FSC for collecting a green block in a tower of blocks by observing whether a block is being held (H), and whether the top block is green (G).

In the literature, branching and looping in planning have been explored as multiple formalisms. Approaches like HTNs, AND/OR trees or policies that use branching, can be compiled to classical planning (Alford et al., 2009; Albore et al., 2009; Ivankovic and Haslum, 2015). Other approaches have shown the benefits of planning with loops as programs (Levesque, 2005), using policies that generalize (Bonet and Geffner, 2015) or automatically deriving Finite State Machines (FSMs) (Bonet et al., 2010). These mechanisms can be effectively modelled with planning actions extended with conditional effects.

The execution of any generalized plan formalism can be *sequential* or *parallel* for the sets of planning instances. In the sequential execution, instances in the generalized planning task $P_t \in \mathcal{P}$ are solved one after the other in the given order. In parallel execution, instances are solved in parallel but requires a more complex control structure for state progression. In this dissertation, we follow a sequential approach to compute generalized plans to sets of planning instances.

The *validation* of a generalized plan is obtained from the execution of the generalized plan in a given planning instance. The result from executing a plan in a planning instance is either a *success proof* when the generalized plan applied in the initial state reaches a goal state, or a *failure* otherwise. The two possible failures are:

1. The generalized plan is *unsound* when:
   - the execution is trapped into an *infinite loop*,
   - the action a that must be applied next is invalid because its preconditions does not hold in the current state $\texttt{pre}(a) \not\subseteq s$.
2. The generalized plan is *incomplete* when it is not in a terminal state, i.e. $G \not\subseteq s$, and it does not find an action to apply next (e.g. a policy with no applicable rule in the current state).

The validator tool VAL (Howey et al., 2004) was used for first time in the 3^{rd} International Planning Competition (Long and Fox, 2003). VAL can be used to validate generalized plans that are in PDDL and fully specified whose executions are deterministic. In these cases it is trivial to detect any failure of a generalized plan in a planning instance. Although for partially specified generalized plans, a planner is required to validate the execution in a given instance and more complex structures are needed to justify execution failures.

2.3.3 Computation

A previous step to the execution of a generalized plan is its computation or synthesis. Although the representation of the generalized plan has a significant impact in the synthesis of the solution.

There are two main strategies in the computation of generalized plans. The first one is the *bottom-up* approach that receives as an input planning instances, computes a solution to a single planning instance, generalizes and finally it tries to merge this solution to other solutions incrementally. The second one is the *top-down* approach where given a set of planning instances, the solution is searched in plan space to solve the whole set of instances. All our works is based in *top-down* strategies for different plan representations and features.

These strategies are tightly connected to Machine Learning techniques. The *bottom-up* strategy is an *on-line* algorithm whose generalize plan can be updated anytime, while the *top-down* strategy is an *off-line* algorithm that just covers the input set of planning instances.

Regarding with the implementation complexity, *top-down* approaches can be easily compiled into other forms of problem solving, while *bottom-up* requires a complex mechanism of merging solutions.

Throughout this thesis we are going to introduce compilations to PDDL that follow a *top-down* approach whose solutions correspond to programs or FSCs. We will also show how these generalized plan representations allow to reuse previous knowledge so we can compute much complex problems in a procedural execution of previous generalized plans.

2.3.4 Evaluation

The plans in classical planning are evaluated in time performance and quality as expressed in Equation 1.1. The set of planning problems are benchmarks, and the main evaluation of the solutions is the *coverage*.

Definition 2.7 (Coverage)**.** *The Coverage is the number of solved problems from the total amount of planning problems.*

In generalized planning, the generalized planning problem $\mathcal{P}$ is described as a set of planning tasks which is similar to a benchmark of planning problems. Thus, one possible metric to measure the quality of a generalized plan is the coverage of the solution over a set of planning instances. One of the requirements of a generalized plan Π is to solve all planning instances from $\mathcal{P}$, so the coverage should be measured using planning instances that do not belong to the generalized planning task $P \notin \mathcal{P}$ as we expressed in Definition 2.3 of generalization.

Generalized plans are represented as data structures or algorithms, so we can use metrics from algorithm analysis like *succinctness* and *complexity* in addition to *coverage*.

Definition 2.8 (Succinctness)**.** *The Succinctness of a generalized plan* Π *is the size of the solution* $|\Pi|$*. The size can be measured as the number of programmed lines in an algorithm, the number of controllers in a FSC, the amount of policy rules and so on.*

Definition 2.9 (Complexity)**.** *The Complexity of a generalized plan is the asymptotic analysis of time and space functions regarding with the input variables that are used to describe the planning instances. For instance, the big-O notation is used in computational complexity theory for function analysis and the study of complexity classes.*

The coverage and succinctness metrics can be empirically measured and formalized, but complexity must be analyzed by hand and only the memory and execution time of generalized plans over many planning instances can give a clue

of *space complexity* and *time complexity* of the solution. In this thesis we measure coverage in many experiments and succinctness is intrinsic to the problem representation.

2.4 Summary

In Chapter 2 we have introduced the concept of Generalized Planning and how it can be described as an abstract framework that is composed of a problem representation, a generalized planner that computes an algorithm, and a generalized plan representation.

We use the definition of a generalized planning problem from previous work (Hu and De Giacomo, 2011), where a generalized planning task is a set of T planning instances $\mathcal{P} = \{P_1, \cdots, P_T\}$. Then, our generalized planner is going to translate the set of planning tasks into a single planning problem that can be solved with an off-the-shelf classical planner. The output of a generalized planner is an algorithm-like solution called generalized plan Π that must solve each planning instance from the generalized planning task.

PART II

Planning Programs and Domain Control Knowledge

CHAPTER 3

Planning Programs

In this chapter we introduce the concept of planning programs, that is our representation and computation of generalized plans, and we prove some theoretical properties. Then we run experiments for generating and validating basic planning programs, and we show the performance of planning programs empirically when inputs and bounds are randomly generated, and a comparison of heuristics.

3.1 Introduction

The aim of a generalized planning problem $\mathcal{P}$ (see Definition 2.1) is to compute a solution called generalized plan Π (see Definition 2.2). In this dissertation we are going to focus on representing generalized plans as *planning programs*.

Definition 3.1 (Planning Program)**.** *A Planning Program is a generalized plan represented by a sequence of instructions* $\Pi = \langle w_0, ..., w_n \rangle$ *given a* STRIPS *frame* $\Phi = \langle F, A \rangle$*. A planning program* Π *is valid if and only if it solves each planning instance from the generalized planning task* $P \in \mathcal{P}$*.*

Planning programs can be synthesized and validated from a *generlized planning* perspective. In this case, GP is related to *automated programming* which is the computation of programs from abstract input information that have some structure or rules in common. These program solutions range from simple functions to algorithm-like formalisms. Major fields like *mathematical programming* (Vajda, 2009), *inductive logic programming* (Muggleton, 1999) or more specifically *agent programming* (De Giacomo et al., 2016) can be considered special cases of *automated programming*.

Our representation of planning programs requires a more complex execution than a sequence of planning actions to represent solutions to multiple planning instances. Thus, they are enhanced with *goto instructions* that allow *branching* and *looping*, i.e. conditional constructs for jumping forwards (branching) or backwards (looping) in the program.

Representing plans as programs have been previously explored (Lang and Zanuttini, 2012). They introduce program formalisms for planning similar to ours as Knowledge-Based Programs (KBP) that can be interpretable in execution time and they can be an exponential number of times more compact than classical plans. Thus, this work is about compact representation of solutions as programs. The programs are represented as logical formulae from an *epistemic logic* perspective that allow branching and looping. Then, they analyze that program verification complexity of a KBP is EXPSPACE-complete in the general case while we express planning programs as a set of instructions that can be synthesized and validated with classical planners whose complexity is analyzed in Section 1.5. In this chapter we constrain the programs to deterministic and fully observable environments.

3.2 Planning Program Representation

A basic planning program is a set of instructions where each instruction w_i, such that $0 \leq i \leq n$, is associated with a *program line* i and is drawn from a set of instructions $\mathcal{I}$. We define the set of instructions as $\mathcal{I} = A \cup \mathcal{I}_{\texttt{go}} \cup \{\texttt{end}\}$, where $\mathcal{I}_{\texttt{go}} = \{\texttt{goto}(i', !f) : 0 \leq i' \leq n, i' \notin \{i, i+1\}, f \in F\}$.

In other words, each instruction is either a planning action $a \in A$, a goto instruction $\texttt{goto}(i', !f)$ or a termination instruction `end`. A termination instruction acts as an explicit marker that program execution should end, similar to a `return` statement in programming. We explicitly require that the last instruction w_n should equal `end`, and since this instruction is fixed, we say that Π has $|\Pi| = n$ program lines, even though Π in fact contains $n + 1$ instructions.

The execution model for a planning program Π consists of a *program state* (s, i), i.e. a pair of a planning state $s \subseteq F$ and a program counter whose value is the current program line $i, 0 \leq i \leq n$. Given a program state (s, i), the execution of instruction w_i on line i is defined as follows:

- If $w_i \in A$, the new program state is $(s', i + 1)$, where $s' = \theta(s, w_i)$ is the result of applying action w_i in planning state s, and the program counter is simply incremented.
- If $w_i = \texttt{goto}(i', !f)$, the new program state is $(s, i + 1)$ if $f \in s$, and (s, i') otherwise. We adopt the convention of jumping to line i' whenever f is *false*

in s. Note that the planning state s remains unchanged.

- If $w_i = \texttt{end}$, execution terminates.

To execute a planning program Π on a planning problem $P = \langle F, A, I, G \rangle$, we set the initial program state to $(I, 0)$, i.e. the initial state of P and program line 0. We say that Π *solves* P if and only if the execution terminates and the goal condition holds in the resulting program state (s, i), i.e. $G \subseteq s \wedge w_i = \texttt{end}$. Because planning programs are generalied plans, they have the same failure reasons (see Subsection 2.3.2) when a planning program Π is applied to a classical planning instance P:

1. Execution terminates in program state (s, i) but the goal condition does not hold, i.e. $G \nsubseteq s \wedge w_i = \texttt{end}$.
2. When executing an action $w_i \in A$ in program state (s, i), the precondition of w_i does not hold, i.e. $\mathsf{pre}(w_i) \nsubseteq s$.
3. Execution enters an infinite loop that never reaches an `end` instruction.

This execution model is *deterministic* and hence a basic planning program can be viewed as a form of compact reactive plan for the subset of planning problems defined by the STRIPS frame $\Phi = \langle F, A \rangle$ that display a certain structure.

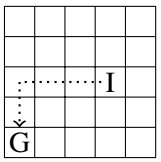
Figure 3.1(a) shows an example planning program Π for navigating to the $(1, 1)$ cell in a grid. Variables x and y represent the position of the current grid cell. Instructions `dec(x)` and `dec(y)` decrement by 1 the value of x and y. The goto instructions `goto(0,!(x=1))` and `goto(2,!(y=1))` jump to line 0 when $x \neq 1$ and to line 2 when $y \neq 1$, respectively. The execution of Π on the planning problem illustrated in Figure 3.1(b) produces the following sequence of planning actions: $\langle \texttt{dec(x)}, \texttt{dec(x)}, \texttt{dec(x)}, \texttt{dec(y)}, \texttt{dec(y)} \rangle$. In this example Π solves *any* planning problem of this type whose goal is to be at cell $(1, 1)$, no matter the grid size.

3.3 Computing Programs with Classical Planning

In this section we describe an approach to automatically compute basic planning programs. The idea is to define a compilation that takes as input a generalized planning problem $\mathcal{P} = \{P_1, ..., P_T\}$ and a number of program lines n and outputs a classical planning problem P_n. The intuition behind the compilation is to extend a given planning frame $\langle F, A \rangle$ with new fluents for encoding the instructions on the program lines of the planning program, as well as the program state (s, i). In Section 3.4 we show proofs of correctness for this compilation. With respect to the actions, the compilation replaces the actions in A with:

```
0. dec(x)
1. goto(0,!(x=1))
2. dec(y)
3. goto(2,!(y=1))
4. end
```

(a)



(b)

Figure 3.1: (a) Example planning program Π for navigating to cell $(1, 1)$; (b) An execution of Π starting at cell (4,3).

- *Programming actions* that program an instruction (*sequential, conditional goto or termination*) on a given program line. Only empty lines can be programmed and initially all the program lines are empty.

- *Execution actions* that implement the execution model described in the previous section, thereby updating the program state. To execute an instruction on a program line, the instruction has to be programmed first. However, it is not necessary to program all instructions before executing: rather, programming and executing are interleaved, where a programming action is performed if the program counter is on an empty program line, and an executing action requires a programmed instruction to update the state and program counter.

For simplicity, we first define the compilation for a single planning problem, and later extend it to generalized planning. Given a planning problem $P = \langle F, A, I, G\rangle$ and a number of program lines n, the output of the compilation is a classical planning problem $P_n = \langle F_n, A_n, I_n, G_n\rangle$. The idea is to define P_n such that any plan π that solves P_n induces a planning program Π that solves P.

To specify P_n we have to introduce prior notation. Let $F_{\mathsf{pc}} = \{\mathsf{pc}_i : 0 \leq i \leq n\}$ be the fluents encoding the program counter and let $F_{\mathsf{ins}} = \{\mathsf{ins}_{i,w} : 0 \leq i \leq n, w \in \mathcal{I} \cup \{\mathsf{nil}\}\}$ be the fluents encoding that instruction w was programmed on line i. Here, nil denotes an empty instruction, indicating that a line has not yet been programmed.

Let $w \in \mathcal{I}$ be an instruction in the *instruction set* $\mathcal{I} = A \cup \mathcal{I}_{\mathsf{go}} \cup \{\mathsf{end}\}$. For each program line i, $0 \leq i \leq n$, we define w_i as a classical planning action that executes instruction w on line i. In doing so, we disallow instructions other than end on the last line n, and we disallow end on the first line 0.

- For each *sequential* instruction $a \in A$, let a_i, $0 \leq i < n$, be a classical planning action with precondition $\mathsf{pre}(a_i) = \mathsf{pre}(a) \cup \{\mathsf{pc}_i\}$ and conditional effects $\mathsf{cond}(a_i) = \mathsf{cond}(a) \cup \{\emptyset \rhd \{\neg\mathsf{pc}_i, \mathsf{pc}_{i+1}\}\}$

- For each *conditional goto* instruction $\mathsf{goto}(i', !f) \in \mathcal{I}_{\mathsf{go}}$, let $\mathsf{go}_i^{i',f}$, $0 \leq i < n$, be a classical action with precondition $\mathsf{pre}(\mathsf{go}_i^{i',f}) = \{\mathsf{pc}_i\}$, and conditional effects $\mathsf{cond}(\mathsf{go}_i^{i',f}) = \{\emptyset \rhd \{\neg\mathsf{pc}_i\}, \{\neg f\} \rhd \{\mathsf{pc}_{i'}\}, \{f\} \rhd \{\mathsf{pc}_{i+1}\}\}$.

- Let the *termination* instruction end_i, $0 < i \leq n$, be a classical action defined as $\mathsf{pre}(\mathsf{end}_i) = G \cup \{\mathsf{pc}_i\}$ and $\mathsf{cond}(\mathsf{end}_i) = \{\emptyset \rhd \{\mathsf{done}\}\}$.

Since w may be executed multiple times, we define two versions: $\mathsf{P}(w_i)$, that is only applicable on an empty line i and programs w on that line, and $\mathsf{E}(w_i)$, that is only applicable when instruction w already appears on line i and repeats the execution of w:

$$\begin{aligned}\mathsf{pre}(\mathsf{P}(w_i)) &= \mathsf{pre}(w_i) \cup \{\mathsf{ins}_{i,\mathsf{nil}}\},\\ \mathsf{cond}(\mathsf{P}(w_i)) &= \{\emptyset \rhd \{\neg\mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,w}\}\},\end{aligned}$$

$$\begin{aligned}\mathsf{pre}(\mathsf{E}(w_i)) &= \mathsf{pre}(w_i) \cup \{\mathsf{ins}_{i,w}\},\\ \mathsf{cond}(\mathsf{E}(w_i)) &= \mathsf{cond}(w_i).\end{aligned}$$

In other words, $P(w_i)$ programs w_i on an empty line i, and $E(w_i)$ repeats the execution of w_i when it is already programmed on line i.

At this point we are ready to define $P_n = \langle F_n, A_n, I_n, G_n\rangle$:

- $F_n = F \cup F_{\mathsf{pc}} \cup F_{\mathsf{ins}} \cup \{\mathsf{done}\}$,

- $A_n = \{\mathsf{P}(a_i), \mathsf{E}(a_i) : a \in A, 0 \leq i < n\} \cup \{\mathsf{P}(\mathsf{go}_i^{i',f}), \mathsf{E}(\mathsf{go}_i^{i',f}) : \mathsf{goto}(i', !f) \in \mathcal{I}_{\mathsf{go}}, 0 \leq i < n\} \cup \{\mathsf{P}(\mathsf{end}_i), \mathsf{E}(\mathsf{end}_i) : 0 < i \leq n\}$,

- $I_n = I \cup \{\mathsf{ins}_{i,\mathsf{nil}} : 0 \leq i \leq n\} \cup \{\mathsf{pc}_0\}$,

- $G_n = \{\mathsf{done}\}$.

We next extend to address generalized planning problems $\mathcal{P} = \{P_1, \ldots, P_T\}$ defined over multiple planning instances. In this case the solution plan π is a sequence of actions that programs Π and simulates the execution of the induced program Π on each of the T classical planning instances, each with a different initial state and goal condition. Specifically, the process of executing the induced planning program Π in a set of planning instances works as follows:

1. we set $t = 1$, which means the current planning instance to solve is the first one in the GP task $\mathcal{P}$.

2. Then, we verify that Π solves the first planning problem P_t,

3. we move to the next instance with $t = t + 1$ and repeat previous step while $t \leq T$.

Because the execution is sequential, we say the GP problem is correctly validated if we have validated the last instance P_T from the set $\mathcal{P}$.

Given a generalized planning task $\mathcal{P} = \{P_1, \ldots, P_T\}$, the output of the compilation is a classical planning task $P'_n = \langle F'_n, A'_n, I'_n, G'_n \rangle$. Since P'_n is similar to the planning task P_n presented above, we only describe the differences:

- The set of fluents $F'_n = F_n \cup F_{\mathsf{test}}$ includes a fluent set $F_{\mathsf{test}} = \{\mathsf{test}_t : 1 \leq t \leq T\}$ that models the active individual planning problem. Initially test_1 is true and test_t are false for $2 \leq t \leq T$, and the initial state on fluents in F is I_1, i.e. $I'_n = I_1 \cup \{\mathsf{ins}_{i,\mathsf{nil}} : 0 \leq i \leq n\} \cup \{\mathsf{pc}_0\} \cup \{\mathsf{test}_1\}$, and the goal is $G'_n = \{\mathsf{done}\}$.

- The set of actions A'_n contains all actions in A_n, but redefines the actions corresponding to the termination instructions. Actions $\mathsf{end}_{t,i}$, $0 < i \leq n$ are now defined differently for each individual planning problem t, $1 \leq t \leq T$:

  - Nonending tests ($t < T$) preconditions: $\mathsf{pre}(\mathsf{end}_{t,i}) = G_t \cup \{\mathsf{pc}_i, \mathsf{test}_t\}$
  - Conditional effects of nonending tests: $\mathsf{cond}(\mathsf{end}_{t,i}) = \{\emptyset \rhd \{\neg\mathsf{pc}_i, \mathsf{pc}_0, \neg\mathsf{test}_t, \mathsf{test}_{t+1}\}\} \cup \{\{\neg f\} \rhd \{f\} : f \in I^{t+1}\} \cup \{\{f\} \rhd \{\neg f\} : f \notin I^{t+1}\}$
  - Ending tests ($t = T$) preconditions: $\mathsf{pre}(\mathsf{end}_{T,i}) = G_T \cup \{\mathsf{pc}_i, \mathsf{test}_T\}$
  - Conditional effects of ending tests: $\mathsf{cond}(\mathsf{end}_{T,i}) = \{\emptyset \rhd \{\mathsf{done}\}\}$.

  For $t < T$, action $\mathsf{end}_{t,i}$ is applicable when G_t and test_t hold, and the effect is resetting the program counter to pc_0, incrementing the current active test and setting fluents in F to their value in the initial state I^{t+1} of the next planning problem. Action $\mathsf{end}_{T,i}$ is defined as the previous action end_i, and is needed to achieve the goal fluent done. As before, we add actions $\mathsf{P}(\mathsf{end}_{t,i})$ and $\mathsf{E}(\mathsf{end}_{t,i})$ to the set A'_n for each t, $1 \leq t \leq T$, and i, $0 < i \leq n$.

3.4 Theoretical Properties

In this section we prove several theoretical properties regarding basic planning programs. First, we show that the compilation is sound and complete and provide a bound on its size. Then, we prove that plan validation and plan existence is PSPACE-complete. Since a planning program Π is defined in terms of the number of program lines $|\Pi|$, we focus on *bounded* plan existence; if the number of program lines is unbounded, a basic planning program can represent any sequential plan without the need for goto instructions.

3.4.1 Soundness, Completeness and Size

Theorem 3.1 (Soundness)**.** *Any plan π that solves P'_n induces a planning program $\Pi = \langle w_0, \ldots, w_n \rangle$ that solves $\mathcal{P}$.*

Proof. As already argued, the subsequence of programming actions of type $\mathsf{P}(w_i)$ that appear in the plan π induces a planning program Π. The only way to achieve the goal fluent done is to execute the termination instruction $\mathsf{end}_{T,i}$. Hence the last instruction programmed has to be a termination instruction, ensuring that the induced planning program Π is well-defined. (Note that Π might have less than n program lines since we could program the termination instruction on a line i satisfying $0 < i < n$.)

The remainder of the proof follows from observing that the execute actions $\mathsf{E}(w_i)$ precisely implement the execution model for basic planning programs. Executing an action instruction a in program state (s, i) has the effect of updating the planning state as $s' = \theta(s, a)$ and incrementing i. Executing a goto action $\mathsf{go}_i^{i',f}$ in (s, i) has the effect of jumping to line i' if f does not hold in s and else increment i. Finally, executing a termination action $\mathsf{end}_{t,i}$ is only possible if the goal condition G_t holds for the current planning problem P_t, $1 \leq t \leq T$. The effect of $\mathsf{end}_{t,i}$ is to reset the program state to $(I_{t+1}, 0)$, i.e. the initial state of the next planning problem P_{t+1} and program line 0.

As detailed above, the execution of a basic planning program Π is a deterministic process that fails to solve a generalized planning problem $\mathcal{P}$ only under three conditions. If the plan π is generated via our compilation, none of the three conditions hold: the precondition of each action instruction has to hold during execution, the goal condition is checked once execution terminates, and infinite loops would prevent the plan π from solving the compiled planning problem P'_n. Execution starts on the first individual classical planning problem $P_1 \in \mathcal{P}$ and finishes when the last problem P_T has been solved. The only way to achieve this condition is by programming the instructions of Π, which cannot be deleted once they are programmed, and successively validating that the program solves all the problems in $\mathcal{P}$, iteratively switching from one problem $P_t \in \mathcal{P}$ to the next. Switching is only possible when G_t, the goal condition of problem P_t, holds. Hence for π to solve P'_n, the simulated execution of the induced planning program Π has to solve each problem P_t, $1 \leq t \leq T$, i.e. Π solves $\mathcal{P}$. □

Theorem 3.2 (Completeness)**.** *If there exists a planning program Π that solves $\mathcal{P}$ such that $|\Pi| \leq n$, there exists a corresponding plan π that solves P'_n.*

Proof. We construct a plan π as follows. Whenever we are on an empty line i, we program the instruction specified by Π. Otherwise we repeat execution of the instruction already programmed on line i. The plan π constructed this way has

the effect of programming Π and simulating the execution of Π on each planning problem in $\mathcal{P}$. Since Π solves $\mathcal{P}$, Π solves each individual problem P_t, $1 \leq t \leq T$, and hence the goal condition G_t is satisfied after simulating the execution of Π on P_t. This implies that the plan π solves P'_n. □

Note that the compilation is not complete in the sense that the bound n on the number of program lines may be too small to accommodate a planning program Π that solves $\mathcal{P}$. In many domains a bounded program can only solve a generalized planning task if a high-level state representation is available that accurately discriminates among states. For instance, the program in Figure 3.1 cannot be computed if $n < 4$. Larger values of n do not formally affect the completeness of our approach but they do affect its practical performance since classical planners are sensitive to the input size.

Theorem 3.3 (Size)**.** *Given a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ on a planning frame $\Phi = \langle F, A \rangle$ and a bound n on the number of program lines, the size of the compiled problem $P'_n = \langle F'_n, A'_n, I'_n, G'_n \rangle$ is given by $|F'_n| = O(n|A| + n^2|F| + T)$ and $|A'_n| = O(n|A| + n^2|F| + nT)$.*

Proof. By inspection of the fluent set F'_n and the action set A'_n. The set F'_n is composed of F, F_{pc}, F_{ins}, F_{test} and the single fluent done. The size of F_{pc} is $n+1$, the size of F_{test} is T, while F_{ins} contains $n+1$ copies of each action in A, goto instruction in $\mathcal{I}_{go}$, empty marker nil and end instruction end. Hence the size of F_{ins} is $(n+1)(|A|+(n+1)|F|+2) = O(n|A|+n^2|F|)$, which dominates the sizes of F and F_{pc}. The action set A'_n defines one action per instruction in $\mathcal{I}$, including T copies of the end instruction, for a total of $|A|+(n+1)|F|+T$. There are n copies of each such instruction, one per program line, and two versions that program and repeat an instruction on a given line, for a total of $2n(|A| + (n+1)|F| + T) = O(n|A| + n^2|F| + nT)$. □

Note that the number of goto instructions is the dominant term, growing as $O(n^2|F|)$. We now introduce an optimization that reduces the number of goto instructions from $O(n^2|F|)$ to $O(n|F| + n^2)$. The idea is to split actions of type $\mathsf{goto}_i^{i',f}$ into two actions: eval_i^f, that evaluates condition f on line i, and $\mathsf{jmp}_i^{i'}$, that performs the conditional jump according to the evaluation outcome. This is inspired by assembly languages that separate comparison instructions that modify flags registers, e.g., `CMP` and `TEST` in the *x86 assembly* language, from jump instructions that update the program counter according to these flag registers, e.g., `JZ` and `JNZ` in *x86 assembly*.

To implement the split we introduce two new fluents acc and eval, initially false. Fluent acc records the outcome of the evaluation, while eval indicates that

the evaluation has been performed. Actions eval_i^f and $\mathsf{jmp}_i^{i'}$ are defined as

$$\begin{aligned}
\mathsf{pre}(\mathsf{eval}_i^f) &= \{\mathsf{pc}_i, \neg\mathsf{eval}\},\\
\mathsf{cond}(\mathsf{eval}_i^f) &= \{\{f\} \rhd \{\mathsf{acc}\}\} \cup \{\emptyset \rhd \{\mathsf{eval}\}\},\\
\mathsf{pre}(\mathsf{jmp}_i^{i'}) &= \{\mathsf{pc}_i, \mathsf{eval}\},\\
\mathsf{cond}(\mathsf{jmp}_i^{i'}) &= \{\emptyset \rhd \{\neg\mathsf{pc}_i, \neg\mathsf{eval}\}, \{\neg\mathsf{acc}\} \rhd \{\mathsf{pc}_{i'}\}, \{\mathsf{acc}\} \rhd \{\mathsf{pc}_{i+1}, \neg\mathsf{acc}\}\}.
\end{aligned}$$

Likewise, we can replace each fluent $\mathsf{ins}_{i,w} \in F_{ins}$ which indicates that a goto instruction $w = \mathsf{goto}(i', !f)$ has been programmed on line i by two fluents $\mathsf{ins}_{i,f}$ and $\mathsf{ins}_{i,i'}$, where the former indicates that f is the condition of the goto instruction on line i, and the latter indicates that we should jump to line i' if f is false. As a result of the optimization, the size of the planning problem P'_n becomes $|F'_n| = O(n(|A| + |F| + n) + T)$ and $|A'_n| = O(n(|A| + |F| + n + T))$.

3.4.2 Plan Validation and Bounded Plan Existence

We formally define two decision problems for the class of basic planning programs, which we call PP. The two decision problems correspond to plan validation and bounded plan existence of basic planning programs.

> VAL(PP) (plan validation for basic planning programs)
> INSTANCE: A planning problem $P = \langle F, A, I, G\rangle$ and a planning program Π.
> QUESTION: Does Π solve P?

> BPE(PP) (bounded plan existence for basic planning programs)
> INSTANCE: A planning problem $P = \langle F, A, I, G\rangle$ and an integer n.
> QUESTION: Does there exist a planning program Π with at most n program lines that solves P?

We proceed to prove that the complexity of both decision problems is PSPACE-complete.

Theorem 3.4. VAL(PP) *is* PSPACE*-complete.*

Proof. Membership: Simply use the execution model to check whether a given planning program Π solves a planning problem P. To store the program state (s, i) we need $|F| + \log n$ space. Processing an instruction and testing for failure conditions 1 and 2 can be easily done in polynomial time and space. To check whether execution enters into an infinite loop, we can maintain a count of the number of instructions processed. If this count exceeds $2^{|F|}n$ without reaching the `end` instruction, this means that at least one program state has been repeated, in which case we stop and report failure. To store the count we also need $|F| + \log n$ space, which is polynomial in P and Π.

Hardness: We adapt Bylander's reduction from polynomial-space deterministic Turing machine (DTM) acceptance to plan existence for classical planning (Bylander, 1994). Given a DTM M with a tape of fixed size K_M, the idea is to define a planning frame $\Phi_M = \langle F, A \rangle$ such that the set F contains fluents derived from the following predicates:

- $\mathsf{at}(j, q)$: Is M currently at tape position j and state q?
- $\mathsf{in}(j, \sigma_M)$: Is σ_M the symbol in tape position j of M?
- accept: Does M accept on a given input?

We define a single action `simulate` with empty precondition and one conditional effect per transition of M. In other words, $A = \{\texttt{simulate}\}$. Each conditional effect of `simulate` is on the following form:

$$\{\mathsf{at}(j, q), \mathsf{in}(j, \sigma_M)\} \rhd \{\neg\mathsf{at}(j, q), \neg\mathsf{in}(j, \sigma_M), \mathsf{at}(j', q'), \mathsf{in}(j, \sigma'_M)\}.$$

When M is at tape position j and state q and σ_M is the symbol in j, the transition replaces σ_M with σ'_M and moves to tape position $j' \in \{j - 1, j + 1\}$ and state q'. If, instead, M accepts the current configuration, the conditional effect becomes

$$\{\mathsf{at}(j, q), \mathsf{in}(j, \sigma_M)\} \rhd \{\mathsf{accept}\}.$$

Given the planning frame Φ_M and an input string x_M, we can construct a planning problem $P^x_M = \langle F, A, I, G \rangle$ such that the initial state I initializes the tape position to 1 and the state to q_0, and encodes the input string x_M on the tape:

$$I = \{\mathsf{at}(1, q_0), \mathsf{in}(0, \#), \mathsf{in}(1, x_{M_1}), \ldots, \mathsf{in}(k, x_{M_k}), \mathsf{in}(k{+}1, \#), \ldots, \mathsf{in}(K_M, \#)\},$$

where $\#$ is the blank symbol, k is the length of the input string x_M and K_M is the fixed size of the tape. The goal condition is always defined as $G = \{\mathsf{accept}\}$.

We now construct the following planning program Π_M with two program lines:

```
0. simulate
1. goto(0,!accept)
2. end
```

Because of the conditional effects of `simulate`, lines 0 and 1 constitute a loop that repeatedly simulates a transition of M starting from the initial state. This loop only terminates if M eventually accepts, in which case the goal state G is trivially satisfied when execution terminates on line 2. Hence, for any input string x_M, the planning program Π_M solves P^x_M if and only if M accepts on input x_M. Since the size of P^x_M and Π_M is polynomial in the size of M, we have produced a reduction from DTM acceptance to VAL(PP). Since the former is a PSPACE-complete decision problem, this proves that VAL(PP) is PSPACE-hard. □

Theorem 3.5. BPE(PP) *is* PSPACE*-complete for* $n \geq 2$.

Proof. Membership: Non-deterministically guess a planning program Π with n program lines. Due to Theorem 3.4, validating whether Π solves P is in PSPACE. Hence the overall procedure is in $\text{NPSPACE} = \text{PSPACE}$.

Hardness: Given a DTM M and an input string x_M, consider the planning problem P^x_M from the proof of Theorem 3.4. There exists a planning program with 2 program lines, namely Π_M, that solves P^x_M if and only if M accepts on input x_M. Hence we have reduced DTM acceptance to BPE(PP) for $n = 2$, implying that the latter is PSPACE-hard. □

3.5 Experiments

Along this dissertation we are going to use mainly, in all the experiments, the classical planner Fast Downward (Helmert, 2006b) (FD) in the LAMA-2011 setting (Richter and Westphal, 2010), and sometimes we explicitly introduce other planners from the Lightweight Automated Planning Toolkit (LAPKT) (Ramirez et al., 2015). All compilations introduced in this dissertation correspond to settings of the *Automated Programming Framework* (APF) [1]. The computer used for these experiments is an Intel Core i5 3.10 Ghz x 4 processor with a 4 GB memory bound.

For planning programs we perform two sets of experiments. In the first set we take as input a generalized planning problem $\mathcal{P}$, and use the compilation P'_n to automatically generate a planning program Π with at most n lines that solves $\mathcal{P}$. In the second set of experiments we take as input a planning problem $\mathcal{P}$ and a planning program Π, and determine whether Π solves $\mathcal{P}$. Thus the two sets of experiments roughly correspond to the two decision problems BPE(PP) and VAL(PP), although plan generation goes beyond plan existence in that we actually produce the planning program Π that solves the instance (or set of instances).

In order to compute the compilation P'_n we use the APF. For *plan generation* we use FD running in an Intel Core i5 3.10 Ghz x 4 processor with a time limit of 3600s and a memory bound of 4 GB. For *validation* we use FD in the same setting and a Breadth First Search (BrFS) planner from the LAPKT.

We evaluate this approach in the following domains: Find, Reverse, Select and Triangular. In **Find** we must count the number of occurrences of a specific element in a vector. In **Reverse** we have to reverse the content of a vector. In **Select**, given a vector of integers we have to search for the minimum element and

[1] The public APF repository is at the following URL: https://github.com/aig-upf/automated-programming-framework.

corresponding index. In **Triangular** the aim is to compute the triangular number $\sum_{i=1}^{N} i$ for a given integer N.

The domains above have been defined and fine-tuned independently so far. However, when a set of domains share many features, they can be defined on a common planning frame $\Phi = \langle F, A \rangle$. Therefore, we have defined a domain called Pointers that represents a vector of pointers pointing to certain elements, and actions to increment/decrement pointers and swapping the content of two pointers. From the common planning frame we can generate all planning instances of the three domains Find, Reverse and Select. The implementation of common planning frames is similar to programming, in which the set of possible statements is fixed for different problems.

In Table 3.1, for each domain we report the number of **lines** required to generate a planning program, and the number of **instances** of the generalized planning problem $\mathcal{P}$ provided as input, where each instance may test a corner case. Then, we solve the compiled planning instance P'_n using a classical planner, generating a number of **fluents** and **actions** in the preprocessing step. The resulting plan π induces a planning program Π that solves the generalized planning problem, and we report the **searching**, **preprocessing** and **total** time. The fluents and actions provide intuition of the size of the output planning problem P'_n.

	n	**Instances**	F	A	**Search(s)**	**Prepro(s)**	**Total(s)**
Find	4	3	671	1044	274.20	0.66	274.86
Reverse	4	2	666	1041	86.96	0.92	87.86
Select	4	4	1028	1688	178.94	25.26	204.20
Triangular	3	2	323	324	0.38	0.47	0.85

Table 3.1: Plan generation for Planning Programs. Program lines and number of used instances; fluents and actions; search, preprocess and total time (in seconds) elapsed while computing the solution.

When domains are created as common planning frames for multiple problems, it could affect to the performance of plan search as reported in Table 3.1 for Find, Reverse and Select. The reason is that defining more actions broadens the scope of problems that can be solved, yet possibly increasing the branching factor. However, the irrelevant actions (actions that will never be used but are considered in the search) to solve a specific planning instance can be removed in the preprocessing phase by including static fluents in the instance to select the action scheme from the domain, i.e. including fluents from predicate $(\mathsf{available} - \mathsf{action?a} - \mathsf{actionID})$ and adding each fluent in the precondition of the corresponding action.

Figure 3.2 shows the resulting planning programs in the four domains. In Find,

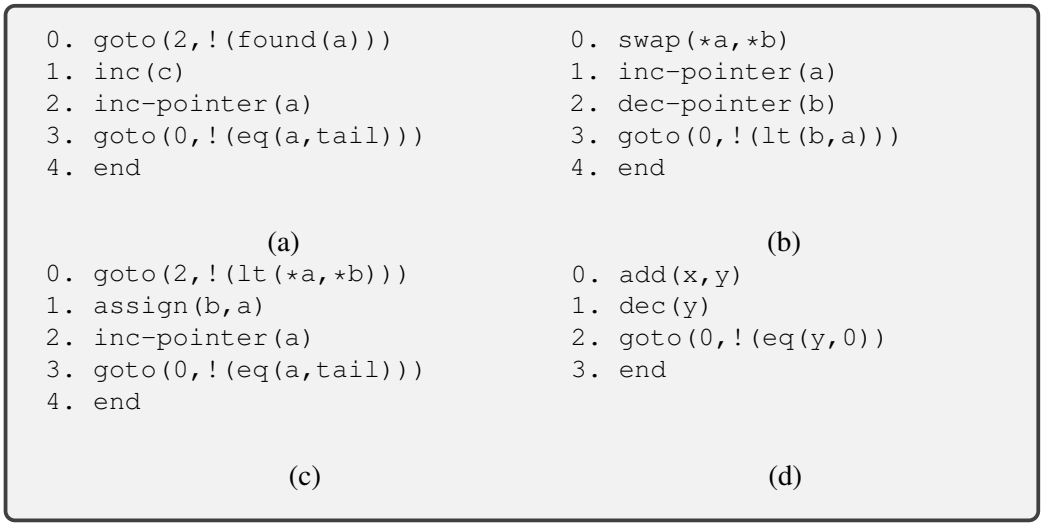


Figure 3.2: Illustration of the generated programs. (a) Π^{find} for counting the number of occurrences of an element in a vector; (b) $\Pi^{reverse}$ for reversing a vector; (c) Π^{select} for selecting the minimum element of a vector; (d) $\Pi^{triangular}$ for computing $\sum_{i=1}^{N} i$.

pointer a initially points to the head of the vector, while counter c equals 0. Lines 2-3 use a to iterate over all elements, while lines 0-1 increment c whenever the content of a equals the element we are looking for. In Reverse, a initially points to the head and b to the tail. The program repeatedly swaps the contents of a and b and move a and b towards the middle of the vector. In Select, a and b initially point to the head, and again, lines 2-3 use a to iterate over all elements. Whenever the content of a is less than that of b, line 1 assigns a to b, effectively storing the minimum element in b. In Triangular, y initially stores the integer N, and the program stores the result $\sum_{i=1}^{N} i$ in x.

In a second set of experiments we validate each planning program from Figure 3.2 on a larger instance, also defined as instances with more objects but with the same set of observations and action scheme. Because the planning programs are given as input, we directly add fluents of type $\mathsf{ins}_{i,w}$ to the initial state of P'_n. In *Find*, *Reverse* and *Select*, we tested the planning programs on vectors of size 30, significantly larger than those used as input for plan generation. In *Triangular*, the aim was to compute the sum of the first 6 natural numbers. Apart from validating each planning program using the two planners FD and BrFS (compiled tests), we also compare the time taken for each planner to compute the solution from scratch without using the planning program (classical tests).

The results of the second set of experiments are shown in Table 3.2. For Compiled Tests, BrFS performs always faster than FD and capable of solving

every compiled instance. In several cases, and mostly for Classical Tests, the planners were unable to compute a solution within the given time limit (TE = Time-Exceeded) showing how helpful DCK is.

	Compiled Tests				**Classical Tests**			
	F	A	**FD**	**BrFS**	F	A	**FD**	**BrFS**
Find	118	9	0.73	0.65	427	5	TE	TE
Reverse	404	7	TE	1.82	394	4	TE	TE
Select	119	9	4.74	0.64	264	4	TE	TE
Triangular Number	53	6	TE	0.33	44	5	TE	0.38

Table 3.2: Generalized plan validation. In Compiled Tests, we compute the fluents, actions and total time (in seconds) to obtain a plan for FD and BrFS. In Classical Tests, we compute the fluents, actions and time taken by FD and BrFS to solve the instance without using the planning program.

Even though validation only involves the deterministic execution of a program on a given instance, the preprocessing step of FD often struggles to generate the corresponding fluents and actions. In particular, this happens when the number of objects of the input planning instance is very large. For instance, the *Find* domain is validated on an instance with 31 objects corresponding to values and indices of a vector, and the total time of FD is dominated by preprocessing, while search is extremely fast. This is the reason that FD fails to validate many input instances. The BrFS setting of the LAPKT planner performs a simpler preprocessing step, so the total time is always smaller than that of FD.

3.6 Generalization Ability of the Compilation

In this section we use the same benchmarks as for basic planning programs: *Find*, *Reverse*, *Select* and *Triangular number*. We analyze the generalization ability of the compilation in two experiments. In the first experiment, we assign the same instances for program generation as in previous experiments and iteratively increment the bounds of the compilation. In the second experiment, we provide the correct bounds for which a generalized plan can be found but the input instances are randomly generated. Both experiments give an insight into how well the compilation generalizes to different parameters and inputs. We used the FD system in the LAMA-2011 setting for these experiments, and a time bound of 600 seconds for generating a planning program and 600 seconds more to validate every instance. Finally we analyze FD on different heuristics to check how informative they are for generating planning programs.

We describe the methodology that we use in order to generate and validate programs. Our framework requires a configuration file that specifies the compilation type (basic or extension), the domain, the set of instances for generation, the set of instances for validation and the required bounds (number of lines, size of the stack, maximum execution time, etc.). Often, for a planning program to generalize it is necessary to define appropriate derived predicates and provide just the right amount of informative input instances. For instance, a grid domain where the problem is to reach the rightmost cell can be captured in diverse settings. One option is to include a derived predicated that is true in the goal cell (similar to looping on observations), another option is to provide two instances of different row lengths to avoid converging on a constant row length, etc.

In Table 3.3 we show the results of the first experiment. Every row has a bounded number of lines and stack size that are used to check when FD finds a solution. In this case the only human input is the set of instances, while bounds are incremented automatically. For each combination of lines and stack size we report the total time that FD takes to solve the problem. NSF means no solution was found, and TE means that preprocessing did not finish within the alloted time bound. In case a program is generated, we validate it over a random set of instances as a metric of generalization. There are four subcolumns F (Find), R (Reverse), S (Select) and T (Triangular Sum) for each column of Instances, Total Time and Validation, indicating the different domains. Results in bold indicate the best solution obtained for each domain. We remark that for the best solutions found, failure to solve some instances was due to timeout in the preprocessing step, not due to an incorrect solution.

From Table 3.3 we can conclude that when the bounds are too small, it is impossible for the planner to find a solution. On the other hand, when the bounds are too large, the planner often does not find a solution, either because preprocessing takes too long, or because search cannot handle the larger state space. In addition, when the bounds are too large, the planner often finds a program with more lines that does not generalize well to other instances. It is clear that the compilation is relatively sensitive to the bounds, but the selection of bounds can still be automated by iterating over different combinations of the bounds.

In the second experiment, instances are randomly generated while the number of lines and stack size are fixed. In each row we specify the number of instances that are chosen from the full set of randomly generated instances. Then we run four independent experiments, each with a different set of instances. Table 3.4 reports the best program found among the four experiments for each row. We evaluate programs according to how many instances they solve in the validation phase, and as a tiebreaker we use the preprocessing and search time.

Since previous experiments established the best possible validation performance of generated programs, we can compare the obtained solutions to the best

Lines	Stack	Instances				Total Time(s)				Validation			
		F	**R**	**S**	**T**	**F**	**R**	**S**	**T**	**F**	**R**	**S**	**T**
2	1	3	2	4	2	NSF	NSF	NSF	NSF	0/40	0/40	0/40	0/6
2	5	3	2	4	2	NSF	NSF	NSF	NSF	0/40	0/40	0/40	0/6
2	10	3	2	4	2	NSF	NSF	NSF	NSF	0/40	0/40	0/40	0/6
2	20	3	2	4	2	NSF	NSF	NSF	NSF	0/40	0/40	0/40	0/6
3	1	3	2	4	2	NSF	NSF	NSF	**1.27**	0/40	0/40	0/40	**4/6**
3	5	3	2	4	2	NSF	NSF	NSF	4.39	0/40	0/40	0/40	**4/6**
3	10	3	2	4	2	NSF	NSF	NSF	8.86	0/40	0/40	0/40	**4/6**
3	20	3	2	4	2	NSF	NSF	TE	17.76	0/40	0/40	0/40	**4/6**
4	1	3	2	4	2	**245.96**	90.58	**199.57**	1.49	**27/40**	**20/40**	**35/40**	**4/6**
4	5	3	2	4	2	NSF	376.62	NSF	5.40	0/40	**20/40**	0/40	**4/6**
4	10	3	2	4	2	NSF	NSF	NSF	10.63	0/40	0/40	0/40	**4/6**
4	20	3	2	4	2	NSF	NSF	TE	20.94	0/40	0/40	0/40	**4/6**
5	1	3	2	4	2	570.36	**66.13**	NSF	1.97	2/40	**20/40**	0/40	**4/6**
5	5	3	2	4	2	NSF	290.10	NSF	7.15	0/40	**20/40**	0/40	**4/6**
5	10	3	2	4	2	NSF	30.20	NSF	13.38	0/40	5/40	0/40	**4/6**
5	20	3	2	4	2	NSF	71.55	TE	27.43	0/40	5/40	0/40	**4/6**
6	1	3	2	4	2	NSF	188.90	NSF	3	0/40	**20/40**	0/40	**4/6**
6	5	3	2	4	2	NSF	12.46	NSF	12.40	0/40	5/40	0/40	**4/6**
6	10	3	2	4	2	NSF	24.90	TE	26.53	0/40	5/40	0/40	**4/6**
6	20	3	2	4	2	NSF	49.22	TE	42.31	0/40	5/40	0/40	**4/6**
7	1	3	2	4	2	NSF	4.66	NSF	5.56	0/40	5/40	0/40	**4/6**
7	5	3	2	4	2	NSF	17.50	NSF	21.24	0/40	5/40	0/40	**4/6**
7	10	3	2	4	2	NSF	34.01	TE	41.06	0/40	5/40	0/40	**4/6**
7	20	3	2	4	2	NSF	73.74	TE	84.23	0/40	5/40	0/40	**4/6**
8	1	3	2	4	2	100.9	5.98	NSF	7.36	3/40	5/40	0/40	**4/6**
8	5	3	2	4	2	78.20	20.10	NSF	28.39	3/40	5/40	0/40	**4/6**
8	10	3	2	4	2	NSF	47.31	TE	55.61	0/40	5/40	0/40	**4/6**
8	20	3	2	4	2	NSF	97.97	TE	116.26	0/40	5/40	0/40	**4/6**
9	1	3	2	4	2	NSF	6.91	NSF	10.38	0/40	5/40	0/40	**4/6**
9	5	3	2	4	2	NSF	27.88	NSF	40.65	0/40	5/40	0/40	**4/6**
9	10	3	2	4	2	NSF	48.65	TE	81.49	0/40	5/40	0/40	**4/6**
9	20	3	2	4	2	NSF	100.40	TE	NSF	0/40	5/40	0/40	0/6
10	1	3	2	4	2	121.37	7.76	NSF	15.6	3/40	5/40	0/40	2/6
10	5	3	2	4	2	NSF	28.78	NSF	64.04	0/40	5/40	0/40	2/6
10	10	3	2	4	2	NSF	56.30	TE	320.81	0/40	5/40	0/40	2/6
10	20	3	2	4	2	NSF	116.31	TE	NSF	0/40	5/40	0/40	0/6

Table 3.3: Plan generation for different bounds in Find (**F**), Reverse(**R**), Select(**S**) and Triangular Sum(**T**) domains using FD in the LAMA-2011 setting. Program lines, stack size and number of instances used; total time (in seconds including preprocessing and search) elapsed while computing the solution for each domain. In case the planning program is correctly generated, the last column shows the validation over multiple instances. We report here No-Solution-Found (NSF) when the planner explores the state space without finding a solution or if there is no solution within the time bound. Also we use Time-Exceeded (TE) when the preprocessing did not finish within the time bound.

# Inst	Lines				Total Time(s)				Validation			
	F	**R**	**S**	**T**	**F**	**R**	**S**	**T**	**F**	**R**	**S**	**T**
1	4	4	4	3	0.99	**40.47**	3.96	0.19	3/40	**17/40**	2/40	1/6
2	4	4	4	3	0.61	48.4	NSF	**0.96**	4/40	15/40	0/40	**4/6**
3	4	4	4	3	416.27	48.03	TE	TE	6/40	15/40	0/40	0/6
4	4	4	4	3	**20.94**	TE	NSF	TE	**15/40**	0/40	0/40	0/6
5	4	4	4	3	NSF	49.23	TE	TE	0/40	15/40	0/40	0/6
6	4	4	4	3	NSF	TE	TE	TE	0/40	0/40	0/40	0/6
7	4	4	4	3	NSF	TE	**423.04**	TE	0/40	0/40	**35/40**	0/6
8	4	4	4	3	NSF	99.25	TE	TE	0/40	15/40	0/40	0/6
9	4	4	4	3	NSF	TE	TE	TE	0/40	0/40	0/40	0/6
10	4	4	4	3	NSF	TE	TE	TE	0/40	0/40	0/40	0/6

Table 3.4: Plan generation with random instances in Find (**F**), Reverse(**R**), Select(**S**) and Triangular Sum(**T**) using FD in the LAMA-2011 setting and only one stack level. For each setting we ran four random experiments with different input instances, reporting the program lines and total time (in seconds). For each setting we choose the best result among the four randomly generated inputs in terms of the validation showed in the last column. We report No-Solution-Found (NSF) when the planner explores the state space or exceeds the time bound, and Time-Exceeded (TE) when the preprocessing step exceeds the time bound.

results in Table 3.3. In the Find domain, the best validation outcome is produced with an input of 4 random instances, while the results in Table 3.3 were obtained using only three instances, and the resulting program does not generalize as well, solving 15 instances compared to 27. In Reverse, the best validation is with just one instance, but generalization is again worse, solving 17 instances compared to 20. In Select and Triangular Sum, the best solution achieves the same validation measure as in Table 3.3 (35 and 4, respectively), but Select needs 7 input instances, more than the 4 instances used in Table 3.3. Again, we can conclude that the compilation is relatively sensitive to the nature and number of input instances.

Finally, we ran experiments with different heuristics to find out whether some heuristics are better at searching solutions to our planning problems compilation. Since the structure of the output planning problems is similar for all domains, we believe that this is an indicative of how heuristics handle these problems. The outcome of this experiment appears in Table 3.5 and we use the same setting as in Table 3.1, but only one heuristic is allowed for each generation phase.

The heuristic evaluation shown in Table 3.5 is about the performance on generation of basic planning programs for different domains like Find, Reverse, Select and Triangular Sum. The results show that delete-free relaxation is only helpful for easy problems like Triangular Sum but becomes unfeasible when problems

have deeper plan lengths yielding in most of the cases either to Time-Exceeded or Memory-Exceeded. We have to remark that *Landmark count* has shown a great computational time performance even in complex domains like Select, but the main limitation using this heuristic is memory because of its fast generation rate. Thus, LAMA-2011 setting is not the one with best performance but has shown great adaptability to every domain balancing exploration and generation rates, being the only one capable of generating every single planning program.

3.7 Summary

In this chapter we have introduced the concept of *planning programs* as a formalism for generalized planning. We represent *planning programs* with a set of instructions that could be either *sequential*, *terminal* or *conditional gotos* that allow *branching* and *looping*. Then, we split the computation logic of a planning program into programming and executing actions. Whilst the first ones are used to synthesize a program, the second ones are used to apply actions and the program execution logic.

We also prove in this chapter *soundness*, *completeness* and *size* theorems for computing planning programs. In addition, we prove that decision problems for basic planning programs (PP) like *plan validation* (VAL(PP)) and *bounded plan existence* (BPE(PP)) are PSPACE-complete.

Finally, we report several experiments where the first ones are for testing VAL(PP) and BPE(PP) decision problems, and the second ones to measure performance on random inputs and bounds. We conclude from the last experiments that bounds can be found iteratively when the set of input instances is given, but planning programs are hard to generalize whether the set of instances is randomly chosen.

Domain	Heuristic	Expanded	Evaluated	Generated	Plan	Prepro(s)	Search(s)
Find	Additive	-	-	-	-	0.66	ME
	Blind	-	-	-	-	0.71	ME
	Causal graph	-	-	-	-	0.72	ME
	Context-enhanced	-	-	-	-	0.67	ME
	FF	**139703**	**526586**	**605448**	**51**	**0.71**	**30.96**
	Goal count	-	-	-	-	0.75	ME
	LAMA-2011	298514	2558305	5970288	51	0.66	274.20
	Landmark count	-	-	-	-	0.70	ME
Reverse	Additive	-	-	-	-	1.03	ME
	Blind	-	-	-	-	1.03	ME
	Causal graph	1846988	1846989	3108541	22	0.92	20.88
	Context-enhanced	-	-	-	-	1.01	TE
	FF	-	-	-	-	0.92	TE
	Goal count	-	-	-	-	0.98	ME
	LAMA-2011	226342	557364	7095513	22	0.92	86.96
	Landmark count	**2785250**	**2785251**	**4507445**	**22**	**1.05**	**19.52**
Select	Additive	-	-	-	-	31.36	TE
	Blind	-	-	-	-	28.94	ME
	Causal graph	3931314	3931315	7291302	73	31.58	235.86
	Context-enhanced	-	-	-	-	28.47	TE
	FF	185522	3282971	3435140	73	25.1	604.48
	Goal count	-	-	-	-	29.12	ME
	LAMA-2011	152195	434383	6279369	73	25.26	178.94
	Landmark count	**10643057**	**10643058**	**21145893**	**73**	**28.20**	**99.44**
Triangular Sum	**Additive**	**46**	**62**	**314**	**26**	**0.30**	**0.02**
	Blind	266183	266184	400632	26	0.28	1.60
	Causal graph	540	541	987	26	0.32	0.04
	Context-enhanced	**46**	**62**	**314**	**26**	**0.30**	**0.02**
	FF	46	62	314	26	0.40	0.12
	Goal count	266183	266184	400632	26	0.28	1.18
	LAMA-2011	2872	5744	8406	26	0.47	0.38
	Landmark count	75906	75907	139145	26	0.66	0.36

Table 3.5: Heuristics evaluations for planning program generation in Find, Reverse, Select and Triangular Sum domains. The columns indicate the number of expanded, evaluated and generated nodes during the search phase. We report the plan size, preprocessing time and search time (in seconds). There are cases where heuristics do not help to find the planning program reporting Time-Exceeded (TE) or Memory-Exceeded (ME) that corresponds to slow exploration and/or fast generation of nodes. The results with best computational total time for each domain are marked in bold.

CHAPTER 4

Generalized Planning with Procedural Domain Control Knowledge

In Chapter 2 we introduced the different kinds of generalized plan representations from which we are going to use two throughout this thesis, *control flow* and *domain control knowledge*. In this chapter, we extend planning programs defined in Chapter 3 with a *stack model* that allow execution of previous knowledge in the form of procedures. Then, we explore previous knowledge as procedures with parameters and non-deterministic executions including the *choice instructions* and *lifted instructions*. Finally, we report theoretical properties and experiments for each planning program representation.

4.1 Introduction

A generalized planning task $\mathcal{P}$ is a set of planning instances that share some common structure. We introduce an approach that applies when problems can be hierarchically decomposed. We make the assumption that a solution to a generalized planning task can be found using *branching* and *looping* control structures, and previous knowledge in the form of procedures. We follow a *divide-and-conquer* approach where first we compute a planning program from a generalized planning problem and then we use that solution as a *callable procedure* to solve the overall generalized planning task.

Definition 4.1 (Procedure)**.** *A Procedure* Π *is an executable set of program instructions* $\{w_0, \cdots, w_n\}$ *used to solve a subtask from a complex programming problem* $\mathcal{P}$*. In structured programming, complex problems are solved by decomposing them into procedures that can be used as previous knowledge by calling them.*

In Subsection 2.3.1 we introduced the possible generalized plan representations. Procedures are generalized plans that can be reused for solving more complex problems, but their execution depends on the representation. If procedures are fully specified, the execution is deterministic with only one possible action to apply next, while partially specified procedures are used as DCK that narrows the search to a subset of applicable actions but still requires a solver to compute a fully specified solution.

Previous work shows the improvements of procedural DCK in search. *Macro-actions* were among the first suggestions to speed up planning and their computation have been deeply studied (Fikes et al., 1972; Botea et al., 2005; Coles and Smith, 2007; Jonsson, 2009). They can help to solve planning problems faster, but they are not always applicable even if they are parameterized, e.g. a macro that moves 4 times to the right in a row can not solve a problem of moving 5 times right or even worse, it will fail trying to solve the problem of going in the opposite direction like going to the leftmost cell of the row.

Other approaches to represent DCK are expressed as *control rules* (Veloso et al., 1995), *temporal logic formulae* (Bacchus and Kabanza, 2000), *HTNs* (Erol et al., 1996), *reactive policies* (Yoon et al., 2008; De la Rosa et al., 2011), *procedural DCK* (Baier et al., 2007) or *finite state machines* (Bonet et al., 2010; Hu and De Giacomo, 2013).

We are going to focus on procedural DCK in the form of planning programs. Recent work implemented planning programs with one level callable procedure (Jiménez and Jonsson, 2015). In that case, the planning program is composed of a main program and a procedure where both are computed at the same time and the main program can use the procedure.

In this chapter we introduce one of the core contributions of this thesis compared to previous work. We have implemented a stack model in PDDL that allows callable procedures in the form of planning programs. Planning programs can be automatically synthesized, but they can be also partially or fully hand-coded and used as previous knowledge to solve complex tasks in planning. We have contributed four planning program representations, two that are deterministic and two that are non-deterministic. The first two are with or without paremetrized procedures, while the other two use a *choice instructions* and *lifted instructions* that require a solver to compute a plan.

4.2 Nested Procedure Calls

There are planning problems that can be decomposable into subtasks. For every subtask we can synthesize a planning program that is included as a procedure in a bag of knowledge. We define a *planning program with procedures* as a set of instructions w_i that can call any of previous m procedures and nest planning programs executions. Then, we need a special mechanism for nesting the execution of procedures. Thus, we have modelled a stack in PDDL where state-of-the-art planners can push and pop procedures in execution time.

In our approach, a procedure Π is a planning program that requires *call instructions* to interact and call other procedures. The set of call instructions is defined as $\mathcal{I}_{call} = \{\texttt{call}(j') : 0 \leq j' \leq m\}$. Then, the set of possible instructions for planning programs is extended with the set of call instructions, $\mathcal{I} = A \cup \mathcal{I}_{go} \cup \{\texttt{end}\} \cup \mathcal{I}_{call}$.

The set of computed planning programs or procedures can be used as previous knowledge and executed as procedural DCK. In previous work (Baier et al., 2007) procedural DCK is hand-coded by experts and tested in several planning domains like *rovers, storage* and *trucks*. Fritz et al. [2008] formalized callable procedures but they were not implemented in PDDL. The first implementation of callable procedures in PDDL (Jiménez and Jonsson, 2015) worked only for 1-level procedure calls.

Given a classical planning frame $\Phi = \langle F, A \rangle$, we formalize the extension of *planning programs with procedures* as $\Pi = \langle \Theta, F_L \rangle$ where:

- $\Theta = \{\Pi^0, \cdots, \Pi^m\}$ is a set of planning programs defined in Φ.
- $F_L \subseteq F$ is a subset of *local fluents*.

The *main procedure* is always Π_0 by convention and the planning programs from Θ is previous knowledge with the name of *auxiliary procedures* $\{\Pi_1, \cdots, \Pi_m\}$. We define the size of a planning program with procedures $|\Pi|$ as the total number of program lines of all procedures, $|\Pi| = |\Pi^0| + \cdots + |\Pi^m|$.

4.2.1 The stack model

In the execution model of planning programs with procedures, we distinguish between local and global states that are used later in the *call stack* implementation.

Procedures are defined on sets of global and local fluents. So, states are partitioned as $s = s_g \cup s_l$ where the first part correspond to the *global state* s_g and the second to the *local state* s_l. Given a subset of *local fluents* F_L, the global state is the projection (see Section 1.3) described as $s_g = s|_{F \setminus F_L}$ and the local state as

$s_l = s|_{F_L}$. Now that we have formalized the projection of the global and local states, we can define the *call stack*.

Definition 4.2 (Call Stack)**.** *A Call Stack Ξ is a tuple (j, i, s_l), where j is an index that refers to a procedure Π^j, $0 \leq j \leq m$, i is a program line, $0 \leq i \leq |\Pi^j|$, and s_l is a local state. This data structure processes the elements following a Last-In First-Out (LIFO) strategy. The possible actions to perform in a stack are adding (push) procedures, removing (pop) procedures and modifying the element on the top of the stack.*

In what follows we use $\Xi \oplus (j, i, s_l)$ to denote a call stack recursively defined by a call stack Ξ and a top element (j, i, s_l). The size of the call stack is defined by $|\Xi|$, and to ensure that the execution model remains bounded in the PDDL compilation, we impose an upper bound on the size of the call stack $|\Xi| = \ell$.

4.2.2 Executing Planning Programs with Nested Procedure Calls

The execution model for a planning program with procedures consists of a *program state* (s_g, Ξ), where s_g is a global state and Ξ is a call stack. Given a program state $(s_g, \Xi \oplus (j, i, s_l))$, the execution of instruction w_i^j on line i of procedure Π^j is defined as follows:

- If $w_i^j \in A$, the new program state is $(s'|_{F \setminus F_L}, \Xi \oplus (j, i+1, s'|_{F_L}))$, where $s' = \theta(s_g \cup s_l, w_i^j)$ is the state resulting from applying action w_i^j in state $s = s_g \cup s_l$ and $s'|_{F \setminus F_L}$ and $s'|_{F_L}$ are the corresponding global and local states. Just as in the execution model for basic programs, the program line i is incremented.

- If $w_i^j =$ `goto(i', !f)`, the new program state is $(s_g, \Xi \oplus (j, i+1, s_l))$ if $f \in s_g \cup s_l$ and $(s_g, \Sigma \oplus (j, i', s_l))$ otherwise. The only effect is changing the program line, and a jump only occurs if f is false, like in the execution model for basic programs.

- If $w_i^j =$ `call(j')`, the new program state is $(s_g, \Xi \oplus (j, i+1, s_l) \oplus (j', 0, \emptyset))$. In other words, calling a procedure $\Pi^{j'}$ has the effect of (1) incrementing the program line i at the top of the stack; and (2) pushing a new element onto the call stack to start the execution of the new procedure $\Pi^{j'}$ on line 0.

- If $w_i^j =$ `end`, the new program state is (s_g, Ξ), i.e. a termination instruction has the effect of terminating a procedure by popping element (j, i, s_l) from the top of the call stack. The execution of a planning program with procedures does not necessarily terminate when executing an end instruction.

Instead, execution terminates when the call stack becomes empty, i.e. in program state (s_g, Ξ) such that $|\Xi| = 0$.

To execute a planning program with procedures Π on a planning problem $P = \langle F, A, I, G \rangle$, we set the initial program state to $(I|_{F \setminus F_L}, (0, 0, I|_{F_L}))$, i.e. the initial state of P is partitioned into global and local states and execution is initially on program line 0 of the main program Π^0. We assume that the goal condition G is completely defined on the set of global fluents $F \setminus F_L$ and we say that Π *solves* P if and only if execution terminates and the goal condition holds in the resulting program state, i.e. $(s_g, \Xi) \wedge G \subseteq s_g \wedge |\Xi| = 0$. As a consequence of the bound ℓ on the size of the call stack, there is now a fourth reason why a planning program with procedures may fail to solve a generalized planning problem:

4. Execution does not terminate because, when executing a call instruction `call(j')` in program state (s_g, Ξ), the size of Ξ equals ℓ, i.e. $|\Xi| = \ell$.

Executing such a procedure call would result in a call stack whose size exceeds the upper bound ℓ, i.e. a *stack overflow*. The extended execution model is still *deterministic*, so a planning program with procedures can again be viewed as a form of compact reactive plan for the *class* of planning problems defined by the corresponding generalized planning task.

Figure 4.1(a) shows an example planning program with procedures $\langle \{\Pi^0, \ldots, \Pi^4\}, \emptyset \rangle$ for visiting the four corners of an $n \times n$ grid starting from any initial position in the grid. Variables x and y that represent the agent position in the grid are global, and there are no local fluents, i.e. $s|_{F \setminus F_L} = s$ and $s|_{F_L} = \emptyset$ for every state s. Procedure Π^1 refers to the basic planning program defined in Figure 3.1, Π^2 is a procedure for reaching the last column of an $n \times n$ grid, Π^3 is a procedure for reaching the last row of an $n \times n$ grid and Π^4 is a procedure for reaching the first column.

To allow for arbitrary grid sizes, we must define derived fluents like $x = n$ and $y = n$ whose values are true if the agent is currently in the last column or row, respectively. These derived fluents can be described with `is-max(x)` that is true precisely when the current assigment to x equals the maximum value which must be included as fluent in the instance, e.g. `max-value(x, v5)` to indicate taht maximum value in the domain of variable x is 5. The conditional effect of action `inc(x)` does not trigger if `is-max(x)` is true, i.e. if the value of x is already maximal. Figure 4.1(b) shows an example execution of the program on a planning problem whose initial state places the agent at position $(4, 3)$.

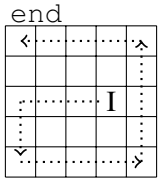
Π^0:
```
0. call(1)
1. call(2)
2. call(3)
3. call(4)
4. end
```

Π^2:
```
0. inc(x)
1. goto(0,!(x=n))
2. end
```

Π^3:
```
0. inc(y)
1. goto(0,!(y=n))
2. end
```

(a)

Π^4:
```
0. dec(x)
1. goto(0,!(x=1))
2. end
```



(b)

Figure 4.1: Planning program with procedures $\langle\{\Pi^0, \ldots, \Pi^4\}, \emptyset\rangle$ for visiting the four corners of an $n \times n$ grid (Π^1 is defined by the program in Figure 3.1) and an execution example of the program in a 5×5 grid starting from cell $(4, 3)$.

4.3 Parameterized Procedures

In this section we take advantage of variable representation to extend procedure calls with *arguments*. Procedural arguments make it possible to reduce the size of programs and to represent compact plans for tasks that demand recursive solutions.

Defining actions on variables is mostly a matter of problem representation. As already mentioned in Section 1.3, we assume that the fluents of a classical planning frame $\Phi = \langle F, A\rangle$ are instantiated from a set of predicates Ψ and a set of objects Ω. We now introduce the additional assumption that there exists a predicate $\mathsf{assign}(v, x) \in \Psi$ and that Ω is partitioned into two sets Ω_v (the *variable objects*) and Ω_x (the *value objects*). Intuitively, a fluent $\mathsf{assign}(v, x)$, $v \in \Omega_v$ and $x \in \Omega_x$, is true if and only if x is the value currently assigned to the variable v. A given variable represents exactly one value at a time, so for a given v, fluents $\mathsf{assign}(v, x)$, $x \in \Omega_x$, are *mutex invariants* (only one is true at any moment). All other predicates in Ψ are instantiated on value objects in Ω_x only.

Let $F_K = \{\mathsf{assign}(v, x) : v \in \Omega_v, x \in \Omega_x\}$ be the subset of fluents induced by the predicate assign. Given a planning program with procedures $\Pi = \langle\Theta, F_L\rangle$, we assume that $F_K \subseteq F_L$, i.e. that all fluents in F_K are local. Furthermore, to each procedure $\Pi^j \in \Theta$, $0 \leq j \leq m$, we associate an arity $ar(j)$ and a parameter list $\varpi(j) \in \Omega_v^{ar(j)}$ consisting of $ar(j)$ variable objects. This is similar to how action schema are defined using arguments in PDDL. We also redefine the set of procedure calls as

$$\mathcal{I}_{call} = \{\,\texttt{call(j',}\omega\texttt{)} : 0 \leq j' \leq m, \omega \in \Omega_v^{ar(j')}\},$$

where ω is the list of $ar(j')$ variable objects passed as arguments when calling procedure $\Pi^{j'}$.

The execution model for planning programs with procedural arguments (including the conditions for *termination*, *success* and *failure*) is inherited from the execution model previously defined for planning programs with procedures. The only term that has to be redefined is the execution of a procedure call instruction with arguments:

- If $w_i^j = \texttt{call(j',}\omega\texttt{)}$ and the current program state is $(s_g, \Xi \oplus (j, i, s_l))$, then the new program state is $(s_g, \Xi \oplus (j, i{+}1, s_l) \oplus (j', 0, s'_l))$ where the local state s'_l is obtained as follows. For each value object $x \in \Omega_x$ and each z, $1 \leq z \leq ar(j')$, we set $\mathsf{assign}(\varpi(j')_z, x)$ to true if and only if $\mathsf{assign}(\omega_z, x)$ is true in s_l. This has the effect of *copying* the value of variable ω_z onto its corresponding variable $\varpi(j')_z$ in the parameter list of procedure $\Pi^{j'}$. For that we need copies of local fluents F_L for each stack level as we will introduce in the following section.

4.4 Computing Planning Programs with Procedures

In this section we extend the compilation from Chapter 3 with procedures, implementing the procedure call mechanism with a finite-size *stack* that can be modeled in PDDL and that is inspired by the compilation of *fault tolerant planning* into classical planning (Domshlak, 2013). Our finite-size stack is a pair $\langle F_L, \ell \rangle$ where $F_L \subseteq F$ is the subset of *local fluents*, i.e., fluents that can be allocated in the stack, and ℓ is the maximum number of levels in the stack. Implicitly this stack model defines:

- A set of fluents $F_L^\ell = \{f^k : f \in F_L, 1 \leq k \leq \ell\}$ that contains replicas of the fluents in F_L parameterized with the stack level k. These fluents represent the ℓ partial states that can be stored in the stack.

- A set of fluents $F_{top}^\ell = \{\mathsf{top}^k\}_{0 \leq k \leq \ell}$ representing which is the top level of the stack at the current time.

- Actions push and pop that are the canonical stack operations, with push_{F_Q} for pushing a subset of stackable fluents $F_Q \subseteq F_L$ to the top level of the stack and pop for popping any fluent in F_L from the top level of the stack.

To compute programs with callable procedures we extend our compilation with new local fluents representing 1) the current procedure; 2) the current program line of the procedure; and 3) the local state of each call stack level. We also add new actions that implement programming and execution of *procedure call* instructions as well as *termination* instructions for the procedures. Intuitively the

execution of a *procedure call* instruction pushes onto the stack the current procedure, the program line and the local state. Likewise the execution of a *termination* instruction pops all this information from the stack.

As a first step we detail the compilation for computing programs with procedures without arguments, and then we explain the extension of the compilation to deal with procedural arguments. Formally the new compilation takes as input a generalized planning problem $\mathcal{P} = \{\langle F, A, I_1, G_1\rangle, \ldots, \langle F, A, I_T, G_T\rangle\}$ and three bounds n, m and ℓ and outputs a classical planning problem $P^{\ell}_{n,m} = \langle F^{\ell}_{n,m}, A^{\ell}_{n,m}, I^{\ell}_{n,m}, G^{\ell}_{n,m}\rangle$. Here, n bounds the number of lines for each procedure, m bounds the number of procedures and ℓ bounds the stack size.

Given the planning problem P_t, $1 \leq t \leq T$, let $I_{t,g} = I_t \cap (F \setminus F_L)$ be the initial global state of P_t, and let $I^1_{t,l} = \{f^1 : f \in I_t \cap F_L\}$ be the initial local state of P_t encoded on level $k = 1$ of the stack. The planning problem $P^{\ell}_{n,m}$ is defined as follows:

- $F^{\ell}_{n,m} = (F \setminus F_L) \cup F^{\ell}_L \cup F^{\ell}_{top} \cup F^{\ell}_{pc} \cup F^{m}_{ins} \cup F_{test} \cup \{\mathsf{done}\}$ where
  - F^{ℓ}_{pc} contains the local fluents for indicating the current line and procedure executed. Formally, $F^{\ell}_{pc} = \{\mathsf{pc}^k_i : 0 \leq i \leq n, 1 \leq k \leq \ell\} \cup \{\mathsf{proc}^k_j : 0 \leq j \leq m, 1 \leq k \leq \ell\}$.
  - F^{m}_{ins} encodes the instructions of the main and auxiliary procedures. In other words, the same fluents F_{ins} defined for the previous compilation but parametized with the procedure id, plus new fluents that encode *call instructions*: $F^{m}_{ins} = \{\mathsf{ins}_{i,j,w} : 0 \leq i \leq n, 0 \leq j \leq m, w \in A \cup \mathcal{I}_{go} \cup \mathcal{I}_{call} \cup \{\mathsf{nil}, \mathsf{end}\}\}$.
- The initial state sets all the program lines (`main` and auxiliary procedures) as empty and sets the procedure on stack level 1 to 0 (the `main` procedure) with the program counter pointing to the first line of that procedure. The initial state on fluents in F is I_1, hence $I^{\ell}_{n,m} = I_{1,g} \cup I^1_{1,l} \cup \{\mathsf{ins}_{i,j,\mathsf{nil}} : 0 \leq i \leq n, 0 \leq j \leq m\} \cup \{\mathsf{top}^1, \mathsf{pc}^1_0, \mathsf{proc}^1_0\}$. As before, the goal condition is defined as $G^{\ell}_{n,m} = \{\mathsf{done}\}$.
- The actions are defined as follows:
  - For each instruction $w \in A \cup \mathcal{I}_{go}$, an action $w^k_{i,j}$ parameterized not only on the program line i but also on the procedure j and stack level k. Let w^k_i be the corresponding action w_i defined in Section 3.3 with superscript k added to all program counters and local fluents. In other words, we have copies of the action w_i for each stack level. Then $w^k_{i,j}$

is defined as

$$\begin{aligned}\mathsf{pre}(w^k_{i,j}) &= \mathsf{pre}(w^k_i) \cup \{\mathsf{top}^k, \mathsf{proc}^k_j\},\\ \mathsf{cond}(w^k_{i,j}) &= \mathsf{cond}(w^k_i).\end{aligned}$$

Note that these actions only alter the top element of the call stack.

– For each call instruction `call(j')` $\in \mathcal{I}_{call}$, an action $\mathsf{call}^{j',k}_{i,j}$ also parameterized on i, j and k, $1 \leq k < \ell$:

$$\mathsf{pre}(\mathsf{call}^{j',k}_{i,j}) = \{\mathsf{pc}^k_i, \mathsf{ins}_{i,j,call(j')}, \mathsf{top}^k, \mathsf{proc}^k_j\}, \tag{4.1}$$

$$\mathsf{cond}(\mathsf{call}^{j',k}_{i,j}) = \{\emptyset \rhd \{\neg\mathsf{pc}^k_i, \mathsf{pc}^k_{i+1}, \neg\mathsf{top}^k, \mathsf{top}^{k+1}, \mathsf{pc}^{k+1}_0, \mathsf{proc}^{k+1}_{j'}\}\}. \tag{4.2}$$

Note that the effect is to push a new program line $(j', 0)$ onto the stack. Also note that $j = j'$ implies a recursive call.

– An action $\mathsf{end}^{k+1}_{i,j}$ that simulates the termination on line i of procedure j on stack level $k+1$, $0 \leq k < \ell$:

$$\begin{aligned}\mathsf{pre}(\mathsf{end}^{k+1}_{i,j}) &= \{\mathsf{pc}^{k+1}_i, \mathsf{ins}_{i,j,end}, \mathsf{top}^{k+1}, \mathsf{proc}^{k+1}_j\},\\ \mathsf{cond}(\mathsf{end}^{k+1}_{i,j}) &= \{\emptyset \rhd \{\neg\mathsf{pc}^{k+1}_i, \neg\mathsf{top}^{k+1}, \neg\mathsf{proc}^{k+1}_j, \mathsf{top}^k\},\\ &\qquad \emptyset \rhd \{\neg f^{k+1} : f \in F_L\}\}.\end{aligned}$$

Note that the effect is to pop the program line (j, i) from the stack, deleting all local fluents.

- As before, the action set $A^\ell_{n,m}$ is composed of the program action $P(w^k_{i,j})$ and execute action $E(w^k_{i,j})$ for each action $w^k_{i,j}$ defined above.

- For each planning problem P_t, $1 \leq t \leq T$, we also need a termination action term_t that simulates the successful termination of the planning program on P_t when the stack is empty:

$$\begin{aligned}\mathsf{pre}(\mathsf{term}_t) &= G_t \cup \{\mathsf{top}^0\}, t < T,\\ \mathsf{cond}(\mathsf{term}_t) &= \{\emptyset \rhd I^1_{t+1,l} \cup\\ &\qquad \{\neg\mathsf{top}^0, \mathsf{top}^1, \mathsf{pc}^1_0, \mathsf{proc}^1_0\}\} \cup\\ &\qquad \{\{\neg f\} \rhd \{f\} : f \in I_{t+1,g}\} \cup\\ &\qquad \{\{f\} \rhd \{\neg f\} : f \notin I_{t+1,g}\}, t < T,\\ \mathsf{pre}(\mathsf{term}_T) &= G_T \cup \{\mathsf{top}^0\},\\ \mathsf{cond}(\mathsf{term}_T) &= \{\emptyset \rhd \{\mathsf{done}\}\}.\end{aligned}$$

Note that the effect of term_t, $t < T$, is to reset the program state to the initial state of problem P_{t+1}.

Now we define the extension to our compilation for programming and executing *parameterized calls to procedures*. Apart from the program counter and the current procedure, a *procedure call* with arguments also pushes onto the stack the arguments of the call. Formally the classical planning task $P^{\ell}_{n,m}$ that results from the compilation is extended as follows:

- We assume that there exists a new set of local fluents $\{\mathsf{assign}(v, x) : v \in \Omega_v, x \in \Omega_x\} \subseteq F_L$ that encode assignments of type $v = x$.
- The actions for programming a call instruction are redefined to indicate not only the called procedure but also the specific values passed to that procedure. To define the actions that execute a call to procedure j' passing a list of parameters we use the actions $\mathsf{call}^{j',k}_{i,j}$ defined in Equations (4.1) and (4.2). For each variable combination $\Gamma(j') \in \Omega_v^{ar(j')}$, we introduce a new action $\mathsf{call}^{j',k}_{i,j}\Gamma(j')$ formulated as:

$$\begin{aligned}
\mathsf{pre}(\mathsf{call}^{j',k}_{i,j}\Gamma(j')) &= \mathsf{pre}(\mathsf{call}^{j',k}_{i,j}),\\
\mathsf{cond}(\mathsf{call}^{j',k}_{i,j}\Gamma(j')) &= \mathsf{cond}(\mathsf{call}^{j',k}_{i,j}) \cup \\
&\quad \{\{\mathsf{assign}^{k}(v_q, x)\} \rhd \{\mathsf{assign}^{k+1}(u_q, x)\} : v_q \in \Gamma(j'),\\
&\qquad x \in \Omega_x, u_q \in \varpi(j'), 1 \leq q \leq |\varpi(j')|\}.
\end{aligned}$$

  In other words, $\mathsf{call}^{j',k}_{i,j}\Gamma(j')$ has the effect of *copying* the value of each variable $v_q \in \Gamma(j')$ on level k of the stack to the corresponding local variable $u_q \in \varpi(j')$ on level $k+1$ of the stack. Recall that $\varpi(j') \in \Omega_v^{ar(j')}$ is the parameter list of procedure j consisting of $ar(j')$ variable objects.

4.4.1 Benefits of computing planning programs with parameterized procedures

Compared to the compilation for basic planning programs, the number of actions increases approximately by a factor of ℓm, which is to be expected since actions are parameterized on the procedure and stack level.

The benefit of extending the compilation with procedure calls comes from:

1. Representing solutions compactly using procedural arguments and recursion. An example is one of the programs we generate in Section 4.6 for DFS traversing binary trees whose depth does not exceed the size of the call stack. A classical plan for this task consists of an action sequence whose length is linear in the number of tree nodes, and hence exponential in the depth of the tree. In contrast, a recursive definition of Depth-First Search (DFS) only requires a 6-line program, as reported in the experiments below.

2. Reusing existing programs as auxiliary procedures. If auxiliary procedures are provided, the only instructions that a plan has to program are those of the main program Π^0. Reducing the number of empty program lines decreases the number of possible planning programs and hence the number of applicable actions. In the experiments below we computed the planning program of Figure 4.1 given the four auxiliary procedures Π^1, Π^2, Π^3 and Π^4 that have already been programmed.

The final benefit of including a given auxiliary procedure is contingent on how much the reused procedure contributes to solving the overall problem. A key issue for effectively reusing existing programs as auxiliary procedures is how to generate auxiliary procedures that are helpful to solve a given generalized planning task. One option is for a domain expert to hand-craft auxiliary procedures (Baier et al., 2007), and this might be the best choice if such knowledge is readily available. However, since each procedure is indeed a program of its own, we can use our compilation to compute each of these program from examples (albeit without auxiliary procedures). For instance, to compute the auxiliary procedure Π^1 for navigating to the $(0, 0)$ position (see Figure 3.1), we define a generalized planning problem $\mathcal{P}^1$ by generating individual planning problems with different initial states whose goal condition is to be at position $(0, 0)$. We then use the compilation from Chapter 3 to generate a basic planning program Π^1 that solves $\mathcal{P}^1$. Similarly, we generate programs Π^2, Π^3, Π^4 for reaching the other three corners of a grid. We then designate these programs as procedures $\Pi^1, \ldots, \Pi^4$ of a planning program with procedures.

To incrementally compute and reuse auxiliary procedures we need to assume the existence of a specific decomposition of the overall problem into a set of subtasks, and appropriately extend the definition of a generalized planning problem. A similar assumption is done in Hierarchical Goal Network (HGN) planning where a network of subgoals is specified to boost the search of a hierarchical planner (Shivashankar et al., 2012). An interesting open research direction is to automatically discover these decompositions of planning problems and previous work on the automatic generation of planning hierarchies (Hogg et al., 2008; Lotinac and Jonsson, 2016) is a good starting point to address this research question.

4.5 Theoretical Properties of Planning Programs with Procedures

We formally prove several properties of the extended compilation for planning programs with procedures. In particular, we start proving the soundness, completeness and a bound on the compilation size. Then, we continue with a complex-

ity analysis of plan validation and bounded plan existence for planning programs with procedures.

Theorem 4.1 (Soundness)**.** *Any plan π that solves $P^{\ell}_{n,m}$ induces a planning program with procedures Π that solves $\mathcal{P}$.*

Proof. The proof is very similar to the proof of Theorem 3.1. Whenever the current program line of a procedure is empty, π has to program an instruction on that line, else repeat execution of the instruction already programmed on that line. Hence the fluent set F^{m}_{ins} implicitly induces a planning program with procedures Π.

Although the execution model is more complicated than for basic planning programs, the repeat actions of $P^{\ell}_{n,m}$ precisely implement the execution model for planning programs with procedures. Hence the plan π has the effect of simulating the execution of Π on each planning problem in $\mathcal{P}$. To solve $P^{\ell}_{n,m}$, the goal condition G_t has to hold for each problem $P_t \in \mathcal{P}$, proving that Π solves $\mathcal{P}$. □

Theorem 4.2 (Completeness)**.** *If there exists a planning program with procedures Π that solves $\mathcal{P}$ such that 1) Π contains at most m auxiliary procedures; 2) each procedure of Π contains at most n program lines; and 3) executing Π on the planning problems in $\mathcal{P}$ does not require a call stack whose size exceeds ℓ, then there exists a plan π that solves $P^{\ell}_{n,m}$.*

Proof. Construct a plan π by always programming the instruction indicated by Π, and repeatedly simulate the execution of Π on the planning problems in $\mathcal{P}$, terminating when the stack becomes empty. Since Π solves $\mathcal{P}$ and fits within the given bounds, the plan π constructed this way is guaranteed to solve $P^{\ell}_{n,m}$. □

The extended compilation adds a new source of incompleteness. The bound ℓ on the stack size limits the depth of nested procedure calls which can also make $P^{\ell}_{n,m}$ unsolvable. For example, a program that implements the recursive version of DFS, needs at least $\ell \geq 3$ stack levels for solving the problem of visiting all the nodes of a tree with depth 3 without causing a stack overflow.

Theorem 4.3 (Size)**.** *Given a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ on a planning frame $\Phi = \langle F, A \rangle$, a subset $F_L \subseteq F$ of local fluents and bounds ℓ, m and n, the size of the compiled problem $P^{\ell}_{n,m} = \langle F^{\ell}_{n,m}, A^{\ell}_{n,m}, I^{\ell}_{n,m}, G^{\ell}_{n,m} \rangle$ is given by $|F^{\ell}_{n,m}| = O(\ell(|F_L| + m + n) + mn(|A| + |F| + m + n) + T)$ and $|A^{\ell}_{n,m}| = O(\ell mn(|A| + |F| + m + n + T))$.*

Proof. By inspection of the fluent set $F^{\ell}_{n,m}$ and the action set $A^{\ell}_{n,m}$. The set $F^{\ell}_{n,m}$ is defined as $(F \setminus F_L) \cup F^{\ell}_{L} \cup F^{\ell}_{top} \cup F^{\ell}_{pc} \cup F^{m}_{ins} \cup F_{test} \cup \{\mathsf{done}\}$. The collective size of $F^{\ell}_{L} \cup F^{\ell}_{top} \cup F^{\ell}_{pc}$ equals $(\ell+1)(|F_L|+1+n+1+m+1) = O(\ell(|F_L|+m+n))$.

The size of F^m_{ins} equals $O(mn(|A| + |F| + m + n))$, where the term m^2n is due to call instructions and we have used the optimization for goto instructions from Chapter 3. The action set $A^\ell_{n,m}$ defines the same instructions as before, plus call instructions, for a total of $|A| + |F| + n + 1 + T + m + 1$. There are ℓmn copies of each such instruction, one per stack level, procedure and program line, and two versions that program and repeat an instruction, for a total of $2\ell mn(|A| + |F| + n + 1 + T + m + 1) = O(\ell mn(|A| + |F| + m + n + T))$. □

We also prove that plan validation and bounded plan existence are PSPACE-complete for planning programs with procedures. Hence including procedures in planning programs does not increase the worst-case complexity of the related decision problems, as long as we bound the size of the call stack. We extend the two decision problems from Chapter 3 to the class PP-P of planning programs with procedures.

VAL(PP-P) (plan validation for planning programs with procedures)
INSTANCE: A planning problem $P = \langle F, A, I, G\rangle$, a planning program Π and an integer ℓ.
QUESTION: Does Π solve P using a call stack of size no more than ℓ?

BPE(PP-P) (bounded plan existence for planning programs with procedures)
INSTANCE: A planning problem $P = \langle F, A, I, G\rangle$ and integers ℓ, m and n.
QUESTION: Does there exist a planning program Π with at most m procedures and n program lines that solves P using a call stack of size no more than ℓ?

Theorem 4.4. VAL(PP-P) *is* PSPACE*-complete.*

Proof. Membership: Similar to the proof for basic planning programs, we can use the execution model to check whether a planning program with procedures Π solves a planning problem P using a call stack of size no more than ℓ. To store a program state (s_g, Ξ) we need $|F|-|F_L|+\ell(\log m+\log n+|F_L|)$ space: $|F|-|F_L|$ space to store the global state $s_g \in F \setminus F_L$, and $\log m + \log n + |F_L|$ space to store each element (j, i, s_l) of the call stack, with a maximum of ℓ such elements. Processing an instruction and testing preconditions and the goal condition can be done in polynomial time and space. Testing whether we exceed the size of the call stack is also trivial given the current program state and instruction. To check whether execution enters into an infinite loop, we can maintain a count of the number of instructions processed, and report failure if this count exceeds the total number $2^{|F|-|F_L|} \times (mn2^{|F_L|})^\ell$ of possible program states. Maintaining this count also requires $|F| - |F_L| + \ell(\log m + \log n + |F_L|)$ space, which is polynomial in P, Π and ℓ.

Hardness: We show that there is a polynomial-time reduction from VAL(PP) to VAL(PP-P). Given a planning problem $P = \langle F, A, I, G \rangle$ and a basic planning program Π, construct a planning program with procedures $\Pi' = \langle \{\Pi\}, F \rangle$, i.e. Π' has a single procedure that acts as the main program and is equal to Π, and the set of local fluents is $F_L = F$. Since Π is a basic planning program it contains no recursive calls to itself, a call stack of size $\ell = 1$ is sufficient to execute Π'. Since Π is the main program of Π', Π' solves P if and only if Π solves P. Since VAL(PP) is PSPACE-complete due to Theorem 3.4, this implies that VAL(PP-P) is PSPACE-hard. □

Theorem 4.5. BPE(PP-P) *is* PSPACE*-complete for* $\ell \geq 1$*,* $m \geq 1$ *and* $n \geq 2$*.*

Proof. Membership: Non-deterministically guess a planning program with procedures Π with m procedures and n program lines. Due to Theorem 4.4, validating whether Π solves P using a call stack of size at most ℓ is in PSPACE. Hence the overall procedure is in NPSPACE = PSPACE.

Hardness: By reduction from BPE(PP). Namely, there exists a basic planning program Π with n program lines that solves P if and only if there exists a planning program with procedures $\Pi' = \langle \{\Pi\}, F \rangle$ that solves P using a call stack of size $\ell = 1$, where Π' has $m = 1$ procedures. Since BPE(PP) is PSPACE-complete for $n \geq 2$, this implies that BPE(PP) is PSPACE-hard for $\ell \geq 1$, $m \geq 1$ and $n \geq 2$. □

We remark that although there is no complexity theoretic benefit of extending planning programs with procedures, in practice decomposing a planning program into procedures often results in more compact solutions easier to be computed by planners. Procedures also make it possible to decompose a problem into sub-problems, compute a separate planning program for each sub-problem and reuse them at different problems.

4.6 Nested Procedure Experiments

We perform two sets of experiments for *planning programs with procedures*, corresponding to plan generation and plan validation. All domains used in these experiments can be hierarchically decomposed into procedures, so they can be represented succinctly. In *GreenBlock*, the aim is to find and collect a green block in a tower of blocks. In *Fibonacci*, the aim is to compute the n-th number in the Fibonacci sequence. In *Gripper*, the aim is to move all balls from one room to the next. In *Hall-A*, the aim is to visit the four corners of a grid (cf. Figure 4.1). In *Sorting*, the aim is to sort the elements of a vector. In *Trees*, the aim is to visit all the nodes of a given binary tree. In *Visit-all*, the aim is to visit all cells of a

grid. Finally, in *Visual-M*, the aim is to find a marked block in a configuration of multiple towers of blocks.

We also introduce a novel domain called *Excel*, inspired by the *Flash Fill* feature of Microsoft Excel to automatically program macros from examples. There are five problems in this domain: 1) add a parenthesis at the end of a word; 2) extract the number of seconds from a timer in the MM:SS.HH format (minutes, seconds and hundredths of a second); 3) given name and surname strings, form a single string formatted as surname, space and name; 4) given name and surname strings, form a string with the name, space and the first letter of the surname; and 5) given name and surname strings, form a string with first letter of the name, a space and the first letter of the surname.

Again we can represent different generalized planning tasks using the same planning frame $\Phi = \langle F, A \rangle$. For example, instances of *Triangular* and *Fibonacci* are represented using the *Variables* PDDL domain that include operators to increment or decrement variables, assign and add the value of a variable to another one. Likewise, the instances of the *Hall-A* and *Visit-all* tasks are represented using a generic *Grid* domain that includes operators for $visiting$ cells and $moving$ one cell, in any cardinal direction. All *Excel* instances are also represented using the same PDDL domain.

In each domain, except for *Trees* whose solution comprises just one procedure, plan generation proceeds in several steps. We manually decompose the overall problem into two or more subproblems. For each subproblem we provide a separate generalized planning problem whose instances correspond to that subproblem. For example, in *Sorting*, that corresponds to the selection-sort algorithm, the subproblem is to select the minimum element from a vector, and for each instance there is a pointer that starts at the first position of the vector, and the goal condition is to bring the pointer to the minimum element. In *Trees*, the problem is not decomposed, but the resulting main program makes recursive calls to itself, so the call stack is still needed to solve the problem.

Let $\mathcal{P}^0, \ldots, \mathcal{P}^m$ be the sequence of generalized planning problems, where $\mathcal{P}^0$ corresponds to the overall problem we want to solve. For each problem $\mathcal{P}^j$ in decreasing order of j $(m, m-1, \ldots)$, we generate a planning program Π^j that solves $\mathcal{P}^j$. While doing so, the programs $\Pi^{j+1}, \ldots, \Pi^m$ are included as auxiliary procedures. In other words, only the lines of the main program are empty, and the remaining programs are encoded as part of the initial state. Hence generating the program Π^j amounts to programming the instructions of the main program, which can include calls to the auxiliary procedures, and then executing the program to ensure it solves $\mathcal{P}^j$. Also note that the number of procedures increases for each subproblem, such that Π^m has no auxiliary procedures, while Π^0 uses all the other programs $\Pi^2, \ldots, \Pi^m$ as auxiliary procedures.

Table 4.1 reports the plan generation results. Compared to Table 3.1, we added

	Proc	Lines	Inst	Stack	Fluents	Actions	Search(s)	Total(s)
GreenBlock	2	(4,3)	(6,5)	(2,2)	(306,250)	(932,559)	(0.90,0.04)	4.1
Excel-1	2	(3,2)	(2,1)	(2,2)	(4167,4265)	(11517,11838)	(3.16,0.53)	9.75
Excel-2	2	(3,2)	(2,1)	(2,2)	(4167,3282)	(11517,6122)	(3.23,0.24)	11.92
Excel-3	2	(3,3)	(2,1)	(2,2)	(4167,7301)	(11517,20538)	(3.09,5.29)	19.94
Excel-4	2	(3,3)	(2,1)	(2,2)	(4167,7301)	(11517,20538)	(3.13,328.41)	342.91
Excel-5	2	(3,3)	(2,1)	(2,2)	(4167,7301)	(11517,20538)	(3.09,0.64)	15.30
Fibonacci	2	(3,3)	(2,4)	(2,2)	(321,341)	(579,607)	(1.01,1.48)	4.94
Gripper	3	(3,3,3)	(2,2,2)	(2,2,2)	(305,307,403)	(651,665,859)	(0.05,0.04,1.06)	2.49
Hall-A	5	(5,5, 5,5,4)	(2,2, 2,2,2)	(2,2, 2,2,2)	(1029,1041, 1053,1065,888)	(3925,3951, 3977,4003,2624)	(86.77,69.18,100.97, 350.23,455.73)	1069.12
Sorting	2	(4,4)	(4,3)	(2,2)	(556,549)	(1988,1779)	28.91	
Trees	1	6	1	4	638	4164	154.76	155.85
Visit-all	3	(3,2,4)	(2,2,2)	(2,2,2)	(801,582,816)	(1911,879,2569)	(0.22,0.04,24.70)	26.67
Visual-M	3	(4,4,4)	(4,2,5)	(2,2,2)	(274,238,279)	(724,649,650)	(0.38,0.03,5.90)	12.25

Table 4.1: Plan generation for planning programs with procedures. Number of procedures; for each procedure: program lines, instances, stack size, fluents, actions, search time; total time including preprocessing (in seconds) elapsed while computing the overall solution.

bounds on the number of procedures and stack size. Since subproblems are solved separately, each procedure corresponds to a separate call to the classical planner. For each procedure, we therefore report the number of program lines, number of instances, stack size, number of fluents, number of actions, and search time. We also report the total time to solve all subproblems related to a domain. Times are reported in seconds.

As an illustration, we show four of the obtained programs where subproblems were hand-picked for each auxiliary procedure. The solution to **Gripper** appears in Figure 4.2. In Π^1 the agent picks up balls with both grippers and moves to the second room. In Π^2 the agent drops both balls and moves back to the first room. Π^0 makes repeated calls to Π^1 and Π^2 until there are no balls left in the first room. Note that we changed the representation of the Gripper domain in order to generalize: instead of representing individual balls, we represent the *number* of balls in each room, and the conditional effect of actions such as `pick-left` is to decrement the number of balls in the current room of the robot.

```
0. call(1)
1. call(2)
2. goto(0,!(no-balls-in-rooma))
3. end
```

(a) Π^0: pick up and drop balls until none left

```
0. pick-left
1. pick-right
2. move
3. end
```

(b) Π^1: pick up balls

```
0. drop-left
1. drop-right
2. move
3. end
```

(c) Π^2: drop balls

Figure 4.2: Program with procedures for the *Gripper*.

The solution to **Sorting** appears in Figure 4.3 and corresponds to the *selection sort* algorithm. There are four pointers: `itermax`, pointing to the tail of the vector; `outer`, indicating the start position in every loop; `inner`, iterating from `outer` to `itermax` in each loop; and `mark`, pointing to the minimum element found so far in each loop. Procedure Π^1 repeatedly increments `inner`, and assigns `inner` to `mark` if its content is the smallest found so far. Procedure Π^0 repeatedly calls Π^1 to select the minimum element (stored in `mark`), and then swaps the contents of `outer` and `mark`. We remark that the action `inc-pointer(outer)` on line 2 has the secondary effect of setting the pointer `inner` equal to `outer`; this is the reason why line 3 refers to `inner` instead of `outer`.

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

(a) Π^0: repeatedly select minimum value and swap contents

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

(b) Π^1: select minimum value from current position `outer`

Figure 4.3: Program with procedures for *Sorting*.

In **Trees**, there is a set of global fluents to mark nodes as visited and two variables `current` and `child` that are used locally to point to nodes of the tree. Figure 4.4 shows the resulting planning program. The program first visits the current node and copies the left child of the current node to `child`. In case the current node is a leaf (i.e. not internal) execution finishes, else the right child of the current node is copied to `current`. Then the program makes two recursive calls to itself for each of the two variables `child` and `current`. In this way it visits all the tree nodes in a depth-first search fashion.

The programs for **Excel** are shown in Figure 4.5. Each string has two indices `lo` and `hi`. Each of the five Excel tasks share the same procedure Π^1 which is parameterized on a string `str-var` and copies all characters of `str-var` between `lo` and `hi` to another string `res`. Program $\Pi^{(0,E_1)}$ calls Π^1 and then appends a right parenthesis. Program $\Pi^{(0,E_2)}$ selects the substring between ':' and '.' by setting the two indices appropriately before copying. Program $\Pi^{(0,E_3)}$ first copies the surname, then appends a space character, and then copies the name. Program $\Pi^{(0,E_4)}$ copies the name and then appends a space character and the first letter of the surname. Finally, program $\Pi^{(0,E_5)}$ appends the first character of the name and surname with a space character in between.

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

Figure 4.4: Recursive program for *Trees*.

```
0. append-str(res,str-var)
1. inc-loindex(str-var)
2. goto(0,!(empty(str-var)))
3. end
```

(a) Π^1: copy substring of `str-var` to `res`

```
0. call(1,str-var)
1. append-char(res,')')
2. end
```

(b) $\Pi^{(0,E_1)}$: append ')' to a string

```
0. get-substr(str-var,':','.')
1. call(1,str-var)
2. end
```

(c) $\Pi^{(0,E_2)}$: get the seconds from a timer

```
0. call(1,surname-var)
1. append-char(res,' ')
2. call(1,name-var)
3. end
```

(d) $\Pi^{(0,E_3)}$: copy surname, space and name

```
0. call(1,name-var)
1. append-char(res,' ')
2. append-str(res,surname-var)
3. end
```

(e) $\Pi^{(0,E_4)}$: copy name, space and initial

```
0. append-str(res,name-var)
1. append-char(res,' ')
2. append-str(res,surname-var)
3. end
```

(f) $\Pi^{(0,E_5)}$: copy space-separated initials

Figure 4.5: Planning programs for *Excel*, where Π^1 is a common procedure.

Like in Chapter 3, we ran a second set of experiments in which we validated the generated programs. In *GreenBlock*, we tested the planning program on a tower of 100 blocks where the green block was the third starting from the bottom. In *Excel* we used the name "MAXIMILIAN" and surname "FEATHERSTONE-HAUGH" for problems $[3, 5]$, and the same surname in problem 1. In Problem 2 the given timer was $01{:}59.23$. In *Fibonacci*, we tested the program on the 6^{th} Fibonacci number. In *Gripper*, the test consisted in moving 30 balls to the next room. In *Hall-A*, the agent had to visit the four corners of a 100×100 grid. In Sorting, the test was to sort a vector of 50 random elements. In *Trees*, we tested

the program on a binary tree with 20 nodes and maximum depth 8. In *Visit-all*, all cells of a 30×30 grid must be visited. Finally, in *Visual-M*, the agent had to process 10 towers with maximum height 10, with the marked block in the last tower.

Table 4.2 presents the plan validation results obtained with the FD and BrFS planners. The reported data include the number of fluents and actions, and the total time FD and BrFS needs to compute a solution. Results are obtained for two configurations: i) the instance that encodes the given planning program in the initial state (compiled tests); and ii) the classical domain and instance without using the planning program (classical tests). This way, we can compare how hard it is to solve a given classical planning instance compared to validating a planning program on the instance (i.e. using the program as control knowledge). As before, Time-Exceeded (TE) indicates that no solution was found within the given time limit.

	Compiled Tests				**Classical Tests**			
	Fluents	**Actions**	**FD-Total**	**BrFS-Total**	**Fluents**	**Actions**	**FD-Total**	**BrFS-Total**
GreenBlock	421	14	2.14	1.89	404	3	1.01	0.87
Excel-1	352	10	0.34	0.39	433	56	0.38	ME
Excel-2	106	10	0.67	0.21	117	56	0.08	0.13
Excel-3	640	11	1.33	0.94	798	106	2.18	ME
Excel-4	450	11	0.78	0.82	558	106	1.02	1.76
Excel-5	347	4	0.48	0.54	408	106	0.81	0.83
Fibonacci	71	11	TE	0.29	56	8	TE	0.36
Gripper	206	15	1.02	0.47	158	5	0.49	TE
Hall-A	10026	46	5.88	5.02	10206	5	1.94	TE
Sorting	2820	17	TE	26.12	2803	4	TE	TE
Trees	661	197	0.56	0.50	61	10	0.03	8.04
Visit-all	1045	19	7.58	1.93	1029	5	TE	TE
Visual-M	45	22	0.55	0.32	48	5	0.12	0.07

Table 4.2: Plan validation for planning programs with procedures. In Compiled Tests, we compute the fluents, actions and total time (in seconds) to obtain a plan for FD and BrFS. In Classical Tests, we compute the fluents, actions and time taken by FD and BrFS to solve the instance without using the planning program.

We see that in most domains, the planners are able to quickly compute a solution even in the absence of a planning program, sometimes even faster than when a planning program is provided. The reason of being faster without the compilation is because some classical instances require less steps than compiled instances, and no matter how many objects the instance has if it can always be solved with few actions, thus the complexity on those domains is on generating programs but not on validating them. A similar situation is found when the actions to solve an instance, like in GreenBlock domain, are repetitive becoming a straight forward search in the classical instance so the control flow only adds complexity. Also, FD spend all the alloted time in preprocessing for some domains

like *Fibonacci* and *Sorting* where predicates with 2 or 3 arity like `(sum v1 v2 v3)` or `(lessthan v1 v2)` require to ground all value combinations, while a simple blind search is enough to get a plan. Remarkably in *Sorting* and *Visit-all* domains, no planner is able to solve the planning instance without the given planning program showing that control flow becomes important for complex domains where heuristic search is not helpful at all.

4.7 Summary

Planning programs as generalized planning is a flexible formalism that permits many extensions to represent problems and solution structures.

In this work we define the concept of *procedures* similar to functions for programming languages. When executing code, functions are pushed and popped from a stack, so we implemented a *call stack* in PDDL that simulates that behavior. This extension allows nesting procedure calls but it may fail if the execution reaches the stack size which is described as *stack overflow*.

When procedures are parameterized, the state is divided into local and global states. This allows to synthesize algorithms like depth-first search (DFS) in a binary tree traversing problem. We also prove the same theorems as in the last chapter but applied to planning programs with procedures (PP-P) instead of basic planning programs (PP).

We conclude with experiments that compute and validate programs with different bounds in the number of procedures, program lines, input instances and stack size for tasks that are hierarchically decomposable like *Flash Fill* or *Selection-Sort*.

CHAPTER 5

Non-Deterministic Planning Programs

In Chapter 4 we have introduced planning programs with procedures that are computed and validated deterministically. The executions up to this moment correspond to a control flow where only one action is applicable at every time step. This is no longer valid for problems with choices like BLOCKSWORLD, where an unstack from a tower is not enough if blocks are not serialized. Also, instances in the generalized planning task sometimes do not share a clear common structure and need to be treated separately. An alternative approach for such scenarios is to represent and compute solutions with non-deterministic execution (Baier et al., 2007), i.e. solutions with open segments that are only determined when the solution is executed on a particular instance. This Chapter 5 introduces *choice instructions* and *lifted action instructions*, two different extensions of our *planning program* formalism to achieve generalization through non-deterministic execution. The chapter describes both how to represent and compute solutions of this kind.

5.1 Planning Programs with Choice Instructions

The first extension refers to *choice actions* that, inspired by the `let` instructions from functional programming, assign a value to a variable in a generalized plan. Our approach closely follows that of Srivastava et al. [2011b] who first introduced choice actions for generalized planning.

The extension is based on the formalism for *planning programs with variables*: for any planning frame $\Phi = \langle F, A \rangle$, fluents in F are instantiated from a set of

predicates Ψ and a set of objects Ω. In addition, Ω is partitioned into *variable objects* Ω_v and *value objects* Ω_x and there exists a predicate $\mathsf{assign}(v, x) \in \Psi$ such that $v \in \Omega_v$ and $x \in \Omega_x$. With this defined we can now extend the instruction set $\mathcal{I}$ with the following set of choice instructions:

$$\mathcal{I}_{choice} = \{\mathtt{choose}(\omega, p) : \omega \in \Omega_v^{ar(p)}, p \in \Psi\}.$$

A choice instruction $\mathtt{choose}(\omega, p)$ assigns a value to each of the variables in ω. These values result from a unification of the predicate p with the current state, which means that a fluent instantiated by assigning those values to p holds in the state where the choice instruction is executed. In general, the arguments of a choice action are determined by evaluating a given first order formula (Srivastava et al., 2011b), but we restrict this formula to a single predicate (the formula could however be extended to a *conjunctive query*).

The execution model for a choice instruction w_i is defined for basic planning programs as follows (we assume that the current program state is (s, i) and that w_i is the instruction on the current line i):

- If $w_i = \mathtt{choose}(\omega, p)$, the new program state is $(s', i + 1)$, where the new state s' is constructed in two steps. The first step non-deterministically chooses a unification of p with the current state. The second step assigns to each variable $v \in \omega$ the corresponding argument value from the chosen unification. The assignment is done making fluent $\mathsf{assign}(v, x)$ true and fluents $\mathsf{assign}(v, x')$ false for each value object $x' \neq x$ that is not in the chosen unification. Other than this assignment, s' is identical to s.

A basic planning program has three failure conditions (see Section 3.2). When we include choice instructions, we are introducing a fourth failure condition:

4. Execution does not terminate because, when executing a choice instruction $\mathtt{choose}(\omega, p)$, we cannot unify predicate p with the current state.

We are considering extensions from Chapter 4 and Chapter 5 independent, so failure conditions are independent too. Although both extensions could be combined, in which case they would have five failure conditions.

We illustrate the idea of a planning program with choice instructions using the well-known BLOCKSWORLD domain. The following planning program is able to solve any instance of BLOCKSWORLD for which the goal is to put all the blocks on the table. This generalized planning task is more complex than unstacking a single tower of blocks, a task commonly solved by FSCs (Bonet et al., 2010), because there can now be an arbitrary number of towers, each with different height. A compact generalized plan that solves this task can be defined using one choice instruction:

```
0. choose(v,clear)
1. unstack(v)
2. putdown(v)
3. goto(0,!(all-clear))
4. end
```

The choice instruction $\texttt{choose}(v, clear)$ assigns to the variable v a block object that is currently clear and captures the key knowledge of *the block to move next*. This block is then unstacked and put down on the table. The derived predicate `all-clear` tests whether all blocks are clear, which is only possible if they are all on the table. This program assumes that the actions `unstack(v)` and `putdown(v)` are defined in terms of a variable object v rather than a block object (the latter is usually how the BLOCKSWORLD domain is modeled).

To compute planning programs with choice actions we extend the compilation explained in the computation of basic planning programs (see Section 3.3). For each choice instruction $\texttt{choose}(\omega, p) \in \mathcal{I}_{choice}$, let $\mathsf{choose}_i^{\omega,p,\chi}$ be a classical action where $\chi \in \Omega_x^{ar(p)}$ is a list of value objects that can be used to unify predicate p with the current state:

$$
\begin{aligned}
\mathsf{pre}(\mathsf{choose}_i^{\omega,p,\chi}) =& \{\mathsf{pc}_i, p(\chi)\},\\
\mathsf{cond}(\mathsf{choose}_i^{\omega,p,\chi}) =& \{\emptyset \rhd \{\neg\mathsf{pc}_i, \mathsf{pc}_{i+1}\}, \cup\\
& \{\mathsf{assign}(v_q, x_q)\} : v_q \in \omega, x_q \in \chi, 1 \leq q \leq |ar(p)|, \cup\\
& \{\neg\mathsf{assign}(v_q, x)\} : v_q \in \omega, x \in \Omega_x, x \neq x_q, 1 \leq q \leq |ar(p)|\}.
\end{aligned}
$$

The compilation is extended with two versions of the previous classical planning action, $\mathsf{P}(\mathsf{choose}_i^{p,\omega,\chi})$, that is only applicable on an empty line i and programs the corresponding choice instruction on that line, and $\mathsf{E}(\mathsf{choose}_i^{p,\omega,\chi})$, that is only applicable when the choice instruction already appears on line i and executes the action.

5.2 Planning Programs with Lifted Action Instructions

Here we go one step further and increase the portion of a planning program that can be unspecified. A *planning program with lifted action instructions* is a planning program in which action instructions have unknown arguments until the instruction is executed on a particular planning instance.

A PDDL action scheme (also called *operators* or *lifted actions*) is parameterized on a set of arguments. Similar to how fluents are induced from predicates,

a set of actions is induced from a PDDL lifted action by assigning objects to its arguments. In the following we assume that the actions of a planning frame $\Phi = \langle F, A \rangle$ are induced from a set of lifted actions $\mathcal{A}$ and a set of objects Ω. Under this assumption, the instruction of a planning program can be a lifted action $\alpha \in \mathcal{A}$. Since the arguments of a lifted action are unspecified, a lifted action instruction effectively models a non-deterministic choice.

Given the current program state (s, i), the execution model for lifted action instructions is defined as:

- If $w_i \in \mathcal{A}$, the new program state is $(s', i+1)$, where $s' = \theta(s, w_i(x))$ is the result of applying an action $w_i(x) \in A$ induced from w_i, where every argument $x \in \Omega^{ar(w_i)}$ of w_i is non-deterministically chosen such that $\mathsf{pre}(w_i(x)) \subseteq s$, i.e. such that $w_i(x)$ is applicable in s.

Additionally, the execution of a *lifted action instruction* fails if there is no possible applicable instantiations for that instruction.

Now we show a *planning program with lifted action instructions* for solving the BLOCKSWORLD task of putting all the blocks on the table. Each execution of `unstack(?b1,?b2)` non-deterministically assigns concrete block objects to parameters `?b1` and `?b2` among the possible applicable instantiations. Likewise each `putdown(?b3)` execution assigns a concrete block object to parameter `?b3`.

```
0. unstack(?b1,?b2)
1. putdown(?b3)
2. goto(0,!(all-clear))
3. end
```

We remark that the execution semantics of lifted action instructions is *angelic* (Marthi et al., 2007): not every assignment of blocks to the arguments `?b1`, `?b2` and `?b3` leads to a valid plan. Rather, it is necessary to use a planner to determine valid assignments of objects to arguments.

To compute *planning programs with lifted action* a small modification in the compilation of Chapter 3 for basic planning programs is required. The modification is a redefinition of the classical planning actions for programming and executing action instructions. A lifted action $w_i \in \mathcal{A}$ induces one action $w_i(x)$ for each assignment $x \in \Omega^{ar(w_i)}$ to the arguments of w_i. The actions for programming

and executing a lifted action instruction are defined as follows:

$$\begin{aligned}
\mathsf{pre}(\mathsf{P}(w_i(x)) &= \mathsf{pre}(w_i(x)) \cup \{\mathsf{ins}_{i,\mathsf{nil}}\},\\
\mathsf{cond}(\mathsf{P}(w_i(x)) &= \{\emptyset \rhd \{\neg\mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,w_i}\}\},\\
\mathsf{pre}(\mathsf{E}(w_i(x)) &= \mathsf{pre}(w_i(x)) \cup \{\mathsf{ins}_{i,w_i}\},\\
\mathsf{cond}(\mathsf{E}(w_i(x)) &= \mathsf{cond}(w_i(x)).
\end{aligned}$$

5.3 Experiments

In these experiments we use several domains from previous sections, but with a slight modification. In the new version of the domains, action instructions have parameters, so the resulting planning programs include lifted instructions. In previous experiments, because we were using parameter-free actions, the execution of planning programs was deterministic. In the new domains with parameterized instructions, the execution of a planning program requires the planner to select the values of the action parameters each time an action instruction is executed. Similar to Hierarchical Task Networks (HTNs) (Alford et al., 2009), non-deterministic planning programs require the planner to compute a fully specified solution, constraining the form of the solution by pruning the actions that do not agree with the given program. Therefore planning program with *choice instructions* or *lifted action instructions*, can be viewed as Domain-specific Control Knowledge (Zimmerman and Kambhampati, 2003; Jiménez et al., 2012). However, unlike HTNs, planning program with *choice instructions* or *lifted action instructions* can be exploited straightforward with an off-the-shelf classical planner.

Apart from existing domains, we added the Blocksworld domain. Our approach is to manually decompose the problem into two subproblems, where the first subproblem is to put all blocks on the table, and the second subproblem is to stack the blocks to form towers. The goal condition for the first subproblem is that all blocks should be on the table, and the goal condition for the second subproblem is usually expressed using fluents on(A,B), on(B,C), etc. However, in our simpler version, the goal is instead to form several towers of a given height, regardless of the specific placement of blocks.

Table 5.1 shows the results of the experiments for computing non-deterministic planning programs. The table reports the number of procedures; for each procedure: number of program lines, number of instances, stack size, number of fluents and actions, search time and preprocessing time; the total time to solve the overall problem. Note that the planner fails to solve one subproblem for Hall-A using the same decomposition as for procedures in the previous chapter when actions are given as lifted instructions. The programs obtained for each domain are similar to those described in Chapter 4. However, since action instructions are lifted, the

Domain	Proc	Lines	Inst	Stack	Fluents	Actions	Total
Hall-A	5	(5,5, 5,5,4)	(2,2, 2,2,2)	2	(1029,-, -,-,-)	(4645,-, -,-,-)	TE
Fibonacci	2	(3,3)	(2,4)	2	(321,341)	(2043,2209)	1.44
Visit-all	3	(3,2,4)	2	2	(585,438,616)	(1875,1038,2375)	44.49
Blocks	2	(4,3)	(6,5)	2	(306,250)	(1124,713)	205.75
Sorting	2	(4,4)	(4,3)	2	(540,549)	(2484,8981)	33.25
Triangular	1	3	2	1	291	588	0.31
Visual-M	3	(4,2,4)	(4,2,5)	2	(274,135,279)	(964,284,782)	12.57
Blocksworld	2	(3,4)	(3,2)	2	(214,214)	(651,651)	0.86

Table 5.1: Plan generation for non-deterministic planning programs. Number of procedures; for each procedure: number of lines and instances, stack size, number of fluents and actions; and total time (in seconds) elapsed while computing the solution.

planner has to assign objects to action parameters and perform search to reach the goal. Figure 5.1 shows the resulting planning program for the **Blocksworld** domain. Procedure Π^1 repeatedly unstacks a block from another and puts it on the table until all blocks are clear. Procedure Π^0 first calls Π^1, then repeatedly picks up a block and stacks it on top of another block, until the number of current towers equals the number of target towers.

```
0. call(1)
1. pick-up(?b1)
2. stack(?b2,?b3 )
3. goto(1,!(eq(current-towers,target-towers)))
4. end
```

(a) Π^0: reaches the number of target towers by stacking blocks onto others

```
0. unstack(?b1,?b2)
1. put-down(?b3)
2. goto(0,!(all-clear))
3. end
```

(b) Π^1: put all blocks on table

Figure 5.1: Non-deterministic planning program for Blocksworld.

Regarding performance, FD has to ground the lifted instructions by assigning objects to their parameters, which causes an increase in the number of operators because in previous chapters original actions were parameter-free with all logic in the conditional effects. As a result, both the preprocessing time and the search

time is generally larger than for deterministic planning programs. On the flip side, there are domains such as Blocksworld that can not be solved by deterministic planning programs, while lifted instructions make it possible.

For validation we use the same instances as in Chapter 4 and address the same experiment reported in Table 4.2, but with non-deterministic programs acting as DCK instead of determinstic planning program with procedures. For the *Blocksworld*, we used an instance with 100 blocks to provide an example of how hard is to solve large classical planning instances versus the corresponding compiled instance with a non-deterministic planning program acting as DCK. Table 5.2 summarizes the obtained results and reports the number of fluents and actions, and total time that FD and BRFS needed to compute a plan. Again, we tested both the compiled problem that encodes the planning program in the initial state (compiled tests), and the classical domain and instance without DCK (classical tests). Only in *Fibonacci* BRFS performs better than FD because of the required preprocessing burden. And the classical instance of Visual-M is No-Solution-Found (NSF) because the BrFS planner does not support axioms and is a requirement for the classical domain.

Domain	**Compiled Tests/Classical Tests**			
	Fluents	**Actions**	**FD-Total**	**BrFS-Total**
Hall-A	-/20200	-/10396	-/1.64	-/TE
Fibonacci	74/56	481/8	0.29/TE	0.008/0.02
Visit-all	2018/1984	1065/1081	9.79/3.91	26.43/158.02
Blocks	735/705	20209/399	7.63/0.94	54.01/0.83
Sorting	258/241	26073/26060	TE/186.32	33.58/TE
Triangular Number	151/242	1835/7740	0.31/1.35	0.012/0.02
Visual-M	62/26	48/41	0.33/0.02	0.37/NSF
Blocksworld	10540/10504	20009/20000	112.45/269.59	NSF/TE

Table 5.2: Plan validation of non-deterministic planning programs. In Compiled Tests, we compute the fluents, actions and total time (in seconds) to obtain a plan for FD and BrFS. In Classical Tests, we compute the fluents, actions and time taken by FD and BrFS to solve the instance without using the planning program. In every cell, the left value corresponds to a compiled test and the right value to a classical tests.

5.4 Summary

In this chapter we introduce extensions to basic planning programs (see Chapter 3) that use the concept of *choice actions* following Srivastava et al. [2011b] approach.

This extension is based on the formalism of *planning programs with variables* where we have a set of *variable objects* Ω_v and a set of *value objects* Ω_x that can be assigned to the first set. As before, every extension could add failure conditions which in this case is the execution of a choice instruction that cannot unify the corresponding predicate with the current state. This approach solves the problem of putting onto the table all blocks from a setting of blocks tower. In every execution, the planning program assigns a clear block from a tower and puts it onto the table until all blocks are clear.

Furthermore, planning programs can be unspecified with *lifted action instructions* as previous work on planning as DCK (Baier et al., 2007). This approach programs instructions but parameters are assigned online in the search phase of the planner. We show performance reports on synthesizing and validating *planning programs with lifted instructions*, including Blocksworld to previous domains.

PART III

Combining Generalized Planning and Hierarchies of Controllers

CHAPTER 6

Hierarchical Finite State Controllers for Generalized Planning

Finite State Controllers (FSCs) are mathematical models represented with states and links, that control the transitions between the states based on possible input events. There are two well-studied types of FSCs in the automata theory field, *deterministic* and *non-deterministic*.

In this chapter we are going to focus on *deterministic* FSCs, their adaptation to planning, and how they can be extended for generalized planning. Then we introduce a novel approach that correspond to a hierarchy of FSCs where transitions in the automaton can be calls to other automata or recursive calls. We also prove soundness and completeness of the approach as well as the relation between *Hierarchical FSCs* (HFSCs) and *Planning Programs*. Finally we report experiments of computing and validating HFSCs in different domains, and we analyze the performance of computing HFSCs for multiple permutations of the input tests.

6.1 Introduction

In Artificial Intelligence there are many solution representations as introduced in Chapter 2. One of the most effective and compact representations is Finite State Controllers (FSCs), also known as Finite State Machines (FSMs). They have been applied to different fields including robotics (Brooks, 1989) and videogames (Buckland, 2004).

Definition 6.1 (Finite State Controller)**.** *A Finite State Controller (FSC) is a graph structure where nodes are states and the edges are transitions, and each transition*

has a condition that causes the transition to fire. An FSC has an initial state where the execution starts and a finite number of states. It is optional to include a terminal controller state in an FSC, although executions could be infinite.

In planning, solutions can be exponentially large in the input size. Thus, expressing solutions as FSCs benefits planning in solution compactness (Bäckström et al., 2014) and generalization. Then, FSCs can be used as *generalized plan* representations that may solve multiple planning problems that share a common structure. These planning problems can be arbitrarily large, as well as described with partial observability and non-deterministic actions (Bonet et al., 2010; Hu and Levesque, 2011; Srivastava et al., 2011b; Hu and De Giacomo, 2013).

In Figure 6.1 we show a binary tree with max depth three and fifteen nodes. The problem consist of traversing the whole binary tree visiting all the nodes. In this example, a classical plan consist of a sequence of actions whose length is linear in the number of nodes, and hence exponential in the depth of the tree. In contrast, the recursive definition of a the Depth-First Search (DFS) algorithm only requires a few lines of code and it is able to traverse any binary tree, no matter its size. Basic FSCs are not capable of representing recursion, and the iterative definition of the DFS algorithm requires an external data structure, being much more complex to implement. Thus, the binary tree traversal is an example that shows the main limitations of FSCs and a motivation for defining *Hierarchical FSCs*.

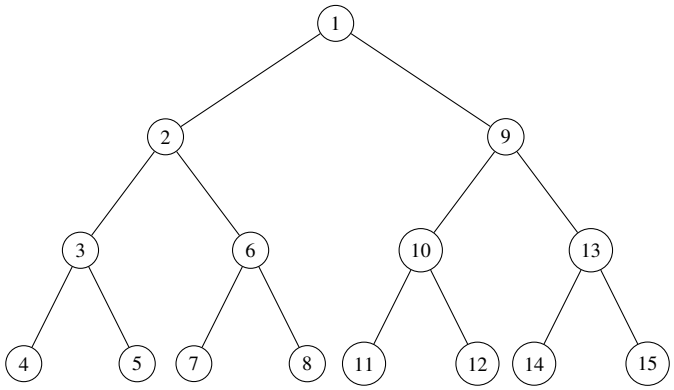


Figure 6.1: Binary tree with fifteen nodes. The nodes in the tree are labeled following a DFS order.

As we said above, FSCs can represent generalized plans, and by extension *Hierarchical FSCs* can be represented and computed as generalized planning solutions too. Our HFSCs approach is novel in three ways:

1. An HFSCs can involve multiple individual FSCs.

2. We have a *call stack* similar to previous chapters(see Definition 4.2), where FSCs are pushed every time they call other FSCs.

3. Each FSC has a paramater list. When an FSC call another FSC, it must specify the arguments assigned to the parameters. It is possible to implement recursion by allowing FSCs to call themselves assigning different arguments to the parameters.

The problem described in Figure 6.1 can be solved with the *recursive HFSC* shown in Figure 6.2 that is called $\mathcal{C}[n]$. The controller $\mathcal{C}[n]$ implements a recursive DFS that traverses a binary tree, and it is represented as a directed graph where controller states are connected to other controller states or themselves. The transitions of the controller states are named edges, and each one is tagged with a *condition/action* label, that denotes the condition under which the action is applied. Here the controller $\mathcal{C}[n]$ has a lone parameter that represents the current node of the binary tree. Condition $\mathsf{isNull}(n)$ tests whether n has no assigned value, $\mathsf{isVisited}(n)$ tests whether n is an already visited node, while a hyphen '$-$' indicates that the transition fires no matter what. Action $\mathsf{visit}(n)$ visits node n, while $\mathsf{copyL}(n, m)$ and $\mathsf{copyR}(n, m)$ assign the left and right child of node n to m, respectively. Action $\mathsf{call}(m)$ is a recursive call to the controller itself, assigning argument m to the only parameter of the controller and restarting execution from its initial controller state q_0. This is similar to how planning programs work, so we will show how they are connected later.

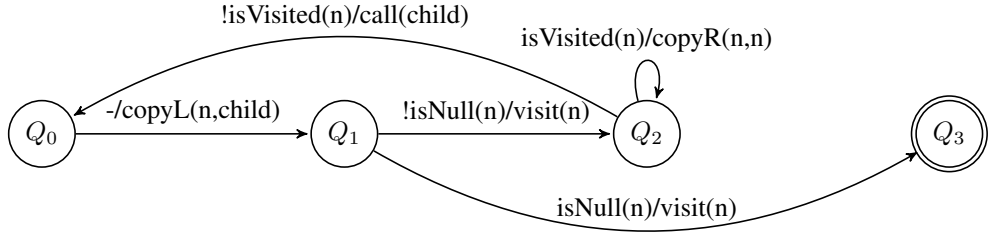


Figure 6.2: Hierarchical FSC $\mathcal{C}[n]$ that traverses a binary tree. The lone parameter $[n]$ of the controller represents the current node being visited of the binary tree.

Intuitively, by repeatedly assigning the right child of n to n itself (using the action $\mathsf{copyR}(n, n)$), the controller $\mathcal{C}[n]$ has the effect of visiting all nodes on the rightmost branch of the tree until the current variable has no value (or after visiting a leaf). Moreover, by assigning the left child of n to $child$ (using the action $\mathsf{copyL}(n, child)$) and making the recursive call $\mathsf{call}(child)$, the controller $\mathcal{C}[n]$ is recursively executed on all left sub-trees. The controller state q_3 is a *terminal state*,

and the action $\mathsf{visit}(n)$ on the transition to q_3 is in fact not needed and could be removed. However, the hierarchical FSC of Figure 6.2 is automatically generated by our approach, so we present conditions and actions exactly as they appear.

The main contributions of this chapter are in the representation, computation and theoretical properties of FSCs for planning. Firstly, we formalize *FSCs for planning* to allow the synthesis of the observation function, the transitions and the output functions of a FSC. Secondly, we propose a formal definition of *hierarchical FSCs for planning* where controllers can call other controllers, including recursion as a special case, using a call stack. Thirdly, we have implemented a novel approach to compute HFSCs for planning. The method consist of a compilation that takes as input a set of classical planning problems from a given domain, and the output is a single classical plan whose solution induces a HFSC. The HFSC is computed and validated at the same time on the input planning problems. Fourthly, we study the relation of *Mealy* machines and FSCs for classical and generalized planning, as well as the comparisson between FSCs and Planning Programs from Chapter 3. Lastly, we report exhaustive empirical experiments of HFSCs and they performance on different input orderings.

6.2 Finite State Controllers for Planning

FSCs for planning in the literature (Bonet et al., 2010; Hu and De Giacomo, 2013) are similar to *transducers* that, in addition to processing input, produce output. We first define transducers and then explain how FSCs for planning are derived from transducers.

Definition 6.2 (Mealy Machine)**.** *A finite state transducer or* Mealy machine *is a tuple* $\Delta = \langle Q, q_0, Q_\perp, \Sigma, \Lambda, \Upsilon, \Gamma \rangle$*:*

- Q *is a finite set of controller states where* $q_0 \in Q$ *is the* initial controller state *and* $Q_\perp \subseteq Q$ *is the subset of* terminal controller states*,*
- Σ *is a finite set of input symbols called the* input alphabet*,*
- Λ *is a finite set of output symbols called the* output alphabet*,*
- $\Upsilon : Q \times \Sigma \to Q$ *is a* transition function *mapping pairs of a controller state and an input symbol to the corresponding next state,*
- $\Gamma : Q \times \Sigma \to \Lambda$ *is an* output function *mapping pairs of a controller state and an input symbol to the corresponding output symbol,*

When in $q \in Q$, on receiving input $\sigma \in \Sigma$, the transducer Δ transitions to $q' = \Upsilon(q, \sigma)$ and outputs symbol $\lambda = \Gamma(q, \sigma)$. This process starts with $q = q_0$ and it is repeated for $q = q'$ until $q' \in Q_\perp$, is a terminal controller state, or until the sequence of input symbols is exhausted.

Given a classical planning problem with conditional effects $P = \langle F, A, I, G \rangle$, an FSC for P is a pair $\mathcal{C} = (\Delta, O)$ of a transducer $\Delta = \langle Q, q_0, Q_\perp, \Sigma, A, \Upsilon, \Gamma \rangle$ and an *observation function* $O : 2^F \to \Sigma$. The *observation function* maps the current planning state to an *observation* i.e., a symbol in the input alphabet of the transducer. The transitions of the transducer Δ do not rely then on an external input, but rather on the current planning state. On the other hand, the output alphabet of this particular transducer is given by the set of actions A in the planning problem P.

The *world state* of an FSC for P is a pair (q, s) that consists of a controller state $q \in Q$ and a planning state s, with the initial world state given by (q_0, I). Given $\mathcal{C}$, an FSC for P, and a world state (q, s), then $\mathcal{C}$ transitions to a new world state (q', s') that is computed as follows:

1. First we retrieve the observation $O(s) \in \Sigma$ symbol that is associated with the current planning state s.

2. Then we compute the new controller state $q' = \Upsilon(q, O(s))$ and the planning action $a \in A$ to apply next, $a = \Gamma(q, O(s))$.

3. Finally we compute the new planning state $s' = \theta(s, a)$ that results from applying action a in the planning state s.

We use $(q, s) \to_{\mathcal{C}} (q', s')$ to denote the transition from (q, s) to (q', s'), and we use $(q_0, s_0) \to^u_{\mathcal{C}} (q_u, s_u)$ to denote a sequence of u such transitions. For $(q_0, s_0) \to^u_{\mathcal{C}} (q_u, s_u)$ to be *well-defined*, all intermediate actions have to be applicable, i.e. $\mathsf{pre}(a_i) \subseteq s_{i-1}$ for each $a_i = \Gamma(q_{i-1}, O(s_{i-1}))$, $1 \leq i \leq u$. Each transition $\Upsilon(q, O(s))$ is associated with a single observation of the current planning state. However, FSCs for planning can represent expressive state queries including the no-op action in A.

Definition 6.3 (No-op action)**.** *The no-op action is an action that when applied does not modify the planning state and affects only the next controller state which allows to concatenate state queries, e.g.* decision trees*.*

.

In previous work (Bonet et al., 2010; Hu and De Giacomo, 2013), the set $Q_\perp$ of terminal controller states is empty, and termination is implicitly defined as $G \subseteq s$, i.e. the goal condition G has to hold in the planning state s of the current world state (q, s). Hence a given controller $\mathcal{C}$ solves P if and only if there exists

a well-defined transition sequence $(q_0, I) \rightarrow_{\mathcal{C}}^{u} (q_u, s_u)$, $u \geq 0$, such that $G \subseteq s_u$. This means that goal achievement has to be tested after executing every action for checking termination. Further, the authors assume that the *observation function* is given as input to synthesize a FSC that solves a given contingent planning problem.

Recall that a *generalized planning problem* $\mathcal{P} = \{P_1, \ldots, P_T\}$ is a finite set of planning instances, where each $P_t = \langle F_t, A, I_t, G_t \rangle$, $1 \leq t \leq T$, that share the same action schema A. An FSC for a generalized planning problem $\mathcal{P}$ is a pair $\mathcal{C} = (\Delta, \mathcal{O})$ of a transducer $\Delta = \langle Q, q_0, Q_\perp, \Sigma, A, \Upsilon, \Gamma \rangle$ and a set $\mathcal{O}$ of observation functions $O_t : 2^{F_t} \rightarrow \Sigma$, $1 \leq t \leq T$. The preconditions and conditional effects of actions in A could have different definitions for different planning problems in $\mathcal{P}$. An FSC $\mathcal{C} = (\Delta, \mathcal{O})$ solves $\mathcal{P}$ if and only if $\mathcal{C}_t = (\Delta, O_t)$ solves each P_t, $1 \leq t \leq T$. The authors (Hu and De Giacomo, 2011; Bonet et al., 2010) show how to synthesize the transition function Υ and output function Γ given all other elements of $\mathcal{C}$ (including the observation functions $O_1, \ldots, O_T$) such that $\mathcal{C}$ solves $\mathcal{P}$.

6.3 Computing Finite State Controllers

This section details our novel formalism of *FSCs for planning* and presents our compilation approach for computing FSCs using an off-the-shelf classical planner. The compilation takes as input a classical planning instance, and outputs another classical planning instance whose solution induces (1) an FSC plus (2), a proof that the FSC solves the input planning instance. The output of the compilation is formalized in the standard PDDL language so off-the-shelf classical planners can be used to compute the FSCs.

6.3.1 A Novel Definition of FSCs for Planning

We reformulate the previous definition of *FSCs for planning*. Given a classical planning problem with conditional effects $P = \langle F, A, I, G \rangle$, an FSC for P is a pair $\mathcal{C} = (\Delta, \varphi)$ of a transducer $\Delta = \langle Q, q_0, q_\perp, \{0, 1\}, A, \Upsilon, \Gamma \rangle$ and a function $\varphi : Q \rightarrow F$ that maps a controller state into a fluent from the given planning problem.

Compared to the previous definition of FSCs for planning, this novel formalism introduced the following modifications:

- There is a single *terminal controller state* $q_\perp$. The reason for including an explicit terminal controller state $q_\perp$ is that we will later extend our definition to hierarchies of FSCs in which the goal G is not necessarily satisfied when

the execution of an FSC terminates. In addition, including an explicit terminal state allows us to use controllers as acceptor automata for recognition tasks.

- The *input alphabet* is simply $\{0, 1\}$ and the mapping φ induces an observation function $O : Q \times 2^F \rightarrow \{0, 1\}$ that maps pairs of a controller state and a planning state into either 0 or 1. Formally, $O(q, s) = \varphi(q) \in s$, where $\varphi(q) \in s$ is interpreted as a test whose outcome equals 1 iff fluent $\varphi(q)$ is true in the planning state s, and 0 otherwise. Note that when $\varphi(q)$ is a *static fluent*, i.e. its value is not changed by any action in A, then the outcome of $O(q, s)$ is always the same (either 0 or 1).

- *The observation function O is defined on controller states in addition to planning states*. With this we aim synthesizing the observation function O in addition to Υ and Γ, in contrast to previous work in which O is given and shared among all the controller states. To keep the space of observation functions tractable, we restrict ourselves to the simplest observation set $\{0, 1\}$. Moreover, we only consider observation functions that use the mapping φ to test the truth value of a single fluent, so synthesizing O is equivalent to synthesizing φ. Note that our observation function is expressive since there is nothing that prevents a fluent $\varphi(q)$ from being a *derived fluent* that is, a fluent that holds when an arbitrary formula over primitive fluents holds.

The execution model of FSCs for planning is the same as before but now, we say that an FSC $\mathcal{C} = (\Delta, \varphi)$ solves P if and only if there exists a well-defined transition sequence $(q_0, I) \rightarrow_{\mathcal{C}}^{u} (q_\perp, s_u)$, $u \geq 0$, such that $G \subseteq s_u$, where $q_\perp$ is the terminal controller state. The execution of an FSC on a planning problem $P = \langle F, A, I, G\rangle$ can fail for any of the following three reasons similarly to basic planning programs failure conditions (see Section 3.2):

1. The execution terminates in a world state $(q_\perp, s_u)$ but the goal condition does not hold, i.e. $G \not\subseteq s_u$.

2. For some world state (q_i, s_i) with $0 \leq i \leq u$, the action $a_i = \Gamma(q_i, O(q_i, s_i))$ cannot be applied because the precondition of a_i does not hold in s_i, i.e. $\mathsf{pre}(a_i) \not\subseteq s_i$.

3. The execution enters an infinite loop that never reaches the terminal state $q_\perp$.

To illustrate our new definition of FSCs for planning, we use an example of an FSC of this kind for the traversal of a *linked list*. We model this task as a classical planning problem $P = \langle F, A, I, G\rangle$, where F contains the following fluents:

- For each list node x, fluents $\mathsf{visited}(x)$ denoting that x has been visited, $\mathsf{end}(x)$ denoting that node x is the end of the list, and $\mathsf{assign}(n, x)$ denoting that variable n points to the list node x.

- For each pair of list nodes x, y, a fluent $\mathsf{succ}(x, y)$ identifying y as the successor node of x in the linked list.

- A fluent $\mathsf{isEnd}(n)$ denoting that variable n points to the end of the linked list. This fluent is implemented with the derived predicate $\mathsf{isEnd}(n) \equiv \exists x\ \mathsf{assign}(n, x) \wedge \mathsf{end}(x)$.

The action set A contains a no-op action, named $a_\perp$, such that $\mathsf{pre}(a_\perp) = \mathsf{cond}(a_\perp) = \emptyset$ as well as the following two kinds of actions:

- $\mathsf{visit}(n)$, that mark the list node assigned to n as visited:

$$\begin{aligned}\mathsf{pre}(\mathsf{visit}(n)) &= \emptyset,\\ \mathsf{cond}(\mathsf{visit}(n)) &= \{\{\mathsf{assign}(n, x)\} \rhd \{\mathsf{visited}(x)\} : \forall x\}.\end{aligned}$$

- $\mathsf{step}(n)$, that move n to the next node in the linked list:

$$\begin{aligned}\mathsf{pre}(\mathsf{step}(n)) &= \emptyset,\\ \mathsf{cond}(\mathsf{step}(n)) &= \{\{\mathsf{assign}(n, x), \mathsf{succ}(x, y)\} \rhd\\ &\qquad \{\neg\mathsf{assign}(n, x), \mathsf{assign}(n, y)\} : \forall x, y\}.\end{aligned}$$

For a linked list of length k, the initial state I and goal condition G are defined as follows:

$$\begin{aligned}I &= \{\mathsf{assign}(n, x_0), \mathsf{succ}(x_0, x_1), \ldots, \mathsf{succ}(x_{k-1}, x_k), \mathsf{end}(x_k)\},\\ G &= \{\mathsf{visited}(x_0), \ldots, \mathsf{visited}(x_{k-1})\}.\end{aligned} \tag{6.1}$$

Figure 6.3 shows a three-state FSC that solves a traversing list planning problem P. The edge (q_0, q_2) with label $\mathsf{isEnd}(n)/a_\perp$ encodes that $\varphi(q_0) = \mathsf{isEnd}(n)$, $\Upsilon(q_0, 1) = q_2$, $\Gamma(q_0, 1) = a_\perp$, i.e. encodes the transition and associated action when $\mathsf{isEnd}(n)$ holds in the current planning state. The edge (q_0, q_1) with label $!\mathsf{isEnd}(n)/\mathsf{visit}(n)$ encodes the transition and action when $\mathsf{isEnd}(n)$ does not hold, i.e. $\Upsilon(q_0, 0) = q_1$ and $\Gamma(q_0, 0) = \mathsf{visit}(n)$. The edge (q_1, q_0) with label $-/\mathsf{step}(n)$ denotes that, when in q_1, the transition and action are always the same no matter the current planning state. We remind that $a_\perp$ is the no-op action (see Definition 6.3.

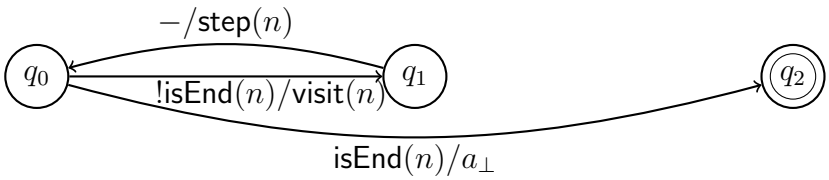


Figure 6.3: FSC for the task of traversing a linked list.

6.3.2 Computing FSCs for Classical Planning

This section describes our compilation for computing an FSC that solves a given classical planning problem. The idea behind the compilation is to include the current controller state, as part of the planning state, and to define two types of actions: *program actions* that program the three functions φ, Υ and Γ for each controller state, and *execute actions* that simulate the execution of the controller by evaluating the programmed functions φ, Υ and Γ in the current planning state.

Formally, the compilation takes as input a classical planning problem $P = \langle F, A, I, G\rangle$ and a bound n on the number of controller states, and outputs another classical planning problem $P_n = \{F_n, A_n, I_n, G_n\}$. Any plan that solves P_n generates an FSC $\mathcal{C} = (\langle Q, q_0, q_\perp, \{0,1\}, A, \Upsilon, \Gamma\rangle, \varphi)$ and validates that $\mathcal{C}$ solves P.

We first set $Q = \{q_0, \ldots, q_n\}$ and $q_\perp \equiv q_n$. The functions φ, Υ and Γ are not defined on $q_\perp$, so we say that $\mathcal{C}$ has n controller states (even though $|Q| = n+1$). Now we proceed to define the compilation.

The set of fluents $F_n = F \cup F_{fun} \cup F_{aux}$, where F_{fun} contains the fluents needed to encode the functions φ, Υ and Γ:

- For each $q \in Q$ and $f \in F$, a fluent cond_q^f that holds iff f is the condition associated with q, i.e. if $\varphi(q) = f$.
- For each $q, q' \in Q$ and $b \in \{0,1\}$, a fluent $\mathsf{succ}_{q,q'}^b$ that holds iff $\Upsilon(q,b) = q'$.
- For each $q \in Q$, $b \in \{0,1\}$ and $a \in A$, a fluent $\mathsf{act}_{q,a}^b$ that holds iff $\Gamma(q,b) = a$.
- For each $q \in Q$ and $b \in \{0,1\}$, fluents nocond_q, nosucc_q^b and noact_q^b that hold iff we have yet to program the functions φ, Υ and Γ, respectively, for q and b.

Moreover, F_{aux} contains the following fluents:

- For each $q \in Q$, a fluent cs_q that holds iff q is the current controller state.

- Fluents evl and app that hold iff we are done evaluating the condition or applying the action corresponding to the current controller state, and fluents o^0 and o^1 representing the outcome of the evaluation (0 or 1).

The initial state and goal condition equal $I_n = I \cup \{\mathsf{cs}_{q_0}\} \cup \{\mathsf{nocond}_q, \mathsf{noact}^b_q, \mathsf{nosucc}^b_q : q \in Q, b \in \{0,1\}\}$ and $G_n = G \cup \{\mathsf{cs}_{q_n}\}$. Finally, the set of actions A_n replaces the actions in A with the following actions:

- For each $q \in Q$ and $f \in F$, an action pcond^f_q for programming $\varphi(q) = f$:

$$\begin{aligned}\mathsf{pre}(\mathsf{pcond}^f_q) &= \{\mathsf{cs}_q, \mathsf{nocond}_q\},\\ \mathsf{cond}(\mathsf{pcond}^f_q) &= \{\emptyset \rhd \{\neg\mathsf{nocond}_q, \mathsf{cond}^f_q\}\}.\end{aligned}$$

- For each $q \in Q$ and $f \in F$, an action econd^f_q that evaluates the condition of the current controller state:

$$\begin{aligned}\mathsf{pre}(\mathsf{econd}^f_q) &= \{\mathsf{cs}_q, \mathsf{cond}^f_q, \neg\mathsf{evl}\},\\ \mathsf{cond}(\mathsf{econd}^f_q) &= \{\emptyset \rhd \{\mathsf{evl}\}, \{\neg f\} \rhd \{\mathsf{o}^0\}, \{f\} \rhd \{\mathsf{o}^1\}\}.\end{aligned}$$

- For each $q \in Q$, $b \in \{0,1\}$ and $a \in A$, an action $\mathsf{pact}^b_{q,a}$ for programming $\Gamma(q,b) = a$:

$$\begin{aligned}\mathsf{pre}(\mathsf{pact}^b_{q,a}) &= \mathsf{pre}(a) \cup \{\mathsf{cs}_q, \mathsf{evl}, \mathsf{o}^b, \mathsf{noact}^b_q\},\\ \mathsf{cond}(\mathsf{pact}^b_{q,a}) &= \{\emptyset \rhd \{\neg\mathsf{noact}^b_q, \mathsf{act}^b_{q,a}\}\}.\end{aligned}$$

- For each $q \in Q$, $b \in \{0,1\}$ and $a \in A$, an action $\mathsf{eact}^b_{q,a}$ that applies the action of the current controller state:

$$\begin{aligned}\mathsf{pre}(\mathsf{eact}^b_{q,a}) &= \mathsf{pre}(a) \cup \{\mathsf{cs}_q, \mathsf{evl}, \mathsf{o}^b, \mathsf{act}^b_{q,a}, \neg\mathsf{app}\},\\ \mathsf{cond}(\mathsf{eact}^b_{q,a}) &= \mathsf{cond}(a) \cup \{\emptyset \rhd \{\mathsf{app}\}\}.\end{aligned}$$

- For each $q, q' \in Q$ and $b \in \{0,1\}$, an action $\mathsf{psucc}^b_{q,q'}$ for programming $T(q,b) = q'$:

$$\begin{aligned}\mathsf{pre}(\mathsf{psucc}^b_{q,q'}) &= \{\mathsf{cs}_q, \mathsf{evl}, \mathsf{o}^b, \mathsf{app}, \mathsf{nosucc}^b_q\},\\ \mathsf{cond}(\mathsf{psucc}^b_{q,q'}) &= \{\emptyset \rhd \{\neg\mathsf{nosucc}^b_q, \mathsf{succ}^b_{q,q'}\}\}.\end{aligned}$$

- For each $q, q' \in Q$ and $b \in \{0,1\}$, an action $\mathsf{esucc}^b_{q,q'}$ that transitions to the next controller state:

$$\begin{aligned}\mathsf{pre}(\mathsf{esucc}^b_{q,q'}) &= \{\mathsf{cs}_q, \mathsf{evl}, \mathsf{o}^b, \mathsf{app}, \mathsf{succ}^b_{q,q'}\},\\ \mathsf{cond}(\mathsf{esucc}^b_{q,q'}) &= \{\emptyset \rhd \{\neg\mathsf{cs}_q, \neg\mathsf{evl}, \neg\mathsf{o}^b, \neg\mathsf{app}, \mathsf{cs}_{q'}\}\}.\end{aligned}$$

Actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$ program the three functions φ, Υ and Γ, respectively that encode the possible transitions of the controller, while econd^f_q, $\mathsf{eact}^b_{q,a}$ and $\mathsf{esucc}^b_{q,q'}$ execute the corresponding function. Fluents evl and app control the order of the execution such that φ is always executed first, then Γ, and finally Υ.

We remark that the functions φ, Υ and Γ are programmed *online*: actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$ are only applicable when the current controller state is q and the current observation is b, respectively. As a consequence, a plan that solves P_n may not always program φ, Υ and Γ for all the controller states in Q, in which case the resulting FSC ignores the unprogrammed controller states. One benefit of this *online* approach is that we can immediately check whether the precondition of a given action $a \in A$ holds (note that $\mathsf{pre}(a)$ is part of the precondition of the action $\mathsf{pact}^b_{q,a}$ for programming a).

Breaking symmetries

The compilation P_n also includes a mechanism for symmetry breaking, which we proceed to describe. For simplicity we excluded this mechanism from the above definition of P_n. To program a transition to a controller state q' using an action $\mathsf{psucc}^b_{q,q'}$, q' has to be *available*. Initially, only q_1 and q_n are available. When we visit q_1 for the first time, q_2 becomes available, etc. It is straightforward to implement this mechanism using fluents $\mathsf{available}_q$ and conditional effects of $\mathsf{esucc}^b_{q,q'}$. With this mechanism in place, we avoid generating multiple permutations of the same FSC that only rename the controller states.

6.3.3 Example

We show how to generate the three-state controller of Figure 6.3 with our P_n compilation. Recall that the initial state output by our compilation is $I_n = I \cup \{\mathsf{cs}_{q_0}\} \cup \{\mathsf{nocond}_q, \mathsf{noact}^b_q, \mathsf{nosucc}^b_q : q \in Q, b \in \{0,1\}\}$. In this example, where I is given by the expression (6.1), the only applicable actions at I_n are $\mathsf{pcond}^f_{q_0}$ for programming the value of $\varphi(q_0)$. To produce the FSC in Figure 6.3, the planner chooses $f = \mathsf{isEnd}(\mathsf{n})$. The effect of action $\mathsf{pcond}^f_{q_0}$ is $\{\neg\mathsf{nocond}_{q_0}, \mathsf{cond}^f_{q_0}\}$.

In the resulting state, the only applicable action is $\mathsf{econd}^f_{q_0}$. Since $\mathsf{isEnd}(n)$ is not true in the current state, the effect of $\mathsf{econd}^f_{q_0}$ is $\{\mathsf{evl}, \mathsf{o}^0\}$. At this state, the only applicable actions are $\mathsf{pact}^0_{q_0,a}$, $a \in A$, for programming the value of $\Gamma(q_0, 0)$. To produce the FSC in Figure 6.3, the planner chooses $a = \mathsf{visit}(n)$. The effect of $\mathsf{pact}^0_{q_0,a}$ is to add fluent $\mathsf{act}^0_{q_0,a}$, causing $\mathsf{eact}^0_{q_0,a}$ to be the only applicable action. In turn, the effect of $\mathsf{eact}^0_{q_0,a}$ is $\{\mathsf{visited}(x_0), \mathsf{app}\}$, where $\mathsf{visited}(x_0)$ is the effect of a and app indicates that we have applied action $a = \Gamma(q_0, 0)$.

Now the only applicable actions are $\mathsf{psucc}^0_{q_0,q}$, $q \in Q$, for programming the value of $\Upsilon(q_0, 0)$. In the case of computing the FSC of Figure 6.3, the planner chooses $q = q_1$. The effect of $\mathsf{psucc}^0_{q_0,q}$ is to add fluent $\mathsf{succ}^0_{q_0,q}$, causing $\mathsf{esucc}^0_{q_0,q}$ to be the only applicable action. The effect of $\mathsf{esucc}^0_{q_0,q}$ is $\{\neg\mathsf{cs}_{q_0}, \neg\mathsf{evl}, \neg\mathsf{o}^0, \neg\mathsf{app}, \mathsf{cs}_{q_1}\}$, representing a transition from q_0 to q_1 and making actions $\mathsf{pcond}^f_{q_1}$ applicable. So far, the applied action sequence is:

$$\langle \mathsf{pcond}^{\mathsf{isEnd}(n)}_{q_0}, \mathsf{econd}^{\mathsf{isEnd}(n)}_{q_0}, \mathsf{pact}^0_{q_0,\mathsf{visit}(n)}, \mathsf{eact}^0_{q_0,\mathsf{visit}(n)}, \mathsf{psucc}^0_{q_0,q_1}, \mathsf{esucc}^0_{q_0,q_1} \rangle.$$

Next, to program and simulate the transition from q_1 to q_0, the planner chooses a similar action sequence. Here, any static fluent can be used to produce the FSC in Figure 6.3, e.g. $\varphi(q_1) = \mathsf{succ}(x_0, x_2)$, since this transition fires no matter what.

$$\langle \mathsf{pcond}^{\mathsf{succ}(x_0,x_2)}_{q_1}, \mathsf{econd}^{\mathsf{succ}(x_0,x_2)}_{q_1}, \mathsf{pact}^0_{q_1,\mathsf{step}(n)}, \mathsf{eact}^0_{q_1,\mathsf{step}(n)}, \mathsf{psucc}^0_{q_1,q_0}, \mathsf{esucc}^0_{q_1,q_0} \rangle.$$

Since $\Phi(q)$, $\Gamma(q, 0)$ and $T(q, 0)$ are already programmed for $q \in \{q_0, q_1\}$, to simulate now the transition from q_0 to q_1 and back to q_0 we only need *execute* actions (no *programming* actions are necessary here):

$$\langle \mathsf{econd}^{\mathsf{isEnd}(n)}_{q_0}, \mathsf{eact}^0_{q_0,\mathsf{visit}(n)}, \mathsf{esucc}^0_{q_0,q_1}, \mathsf{econd}^{\mathsf{succ}(x_0,x_2)}_{q_1}, \mathsf{eact}^0_{q_1,\mathsf{step}(n)}, \mathsf{esucc}^0_{q_1,q_0} \rangle.$$

The result is to visit x_1 and move n from pointing to x_1 to pointing to x_2. This cycle repeats until the effect of $\mathsf{econd}^{\mathsf{isEnd}(n)}_{q_0}$ is $\{\mathsf{evl}, \mathsf{o}^1\}$, indicating that fluent $\mathsf{isEnd}(n)$ is true. When this happens we can program and simulate the transition from q_0 to q_2 using the following action sequence:

$$\langle \mathsf{pact}^1_{q_0,a_\perp}, \mathsf{eact}^1_{q_0,a_\perp}, \mathsf{psucc}^1_{q_0,q_2}, \mathsf{esucc}^1_{q_0,q_2} \rangle.$$

Since q_2 is the terminal controller state and all list nodes have been visited, the goal condition G_n is satisfied. Note that G_n is only satisfied after the execution of the programmed FSC guarantees to solve the input planning problem P.

6.3.4 Properties

Here we analyze the theoretical properties of our P_n compilation for synthesizing an FSC that solves a classical planning task P.

Theorem 6.1 (Soundness)**.** *Any plan π that solves P_n induces an FSC that solves P.*

Proof. Once π programs the functions φ, Υ and Γ of the FSC they cannot be altered. Programming actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$ delete fluents nocond_q, noact^b_q and nosucc^b_q respectively, and there are no actions in A_n for adding these

fluents, which bans the later application of actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ or $\mathsf{psucc}^b_{q,q'}$ for the same values of q and b.

The plan π deterministically executes the FSC (the programmed φ, Υ and Γ functions) on the input planning problem P. The only way to change the current controller state from q to q' is to apply a partial action sequence $\langle \mathsf{econd}^f_q, \mathsf{eact}^b_{q,a}, \mathsf{esucc}^b_{q,q'} \rangle$ for some $f \in F$, $a \in A$, $b \in \{0,1\}$. Because of the corresponding preconditions cond^f_q, $\mathsf{act}^b_{q,a}$ and $\mathsf{succ}^b_{q,q'}$ of these actions, this means that programming actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$ have to be applied in advance. Further, since cond^f_q, $\mathsf{act}^b_{q,a}$ and $\mathsf{succ}^b_{q,q'}$ are true for at most a single combination of values of f, a and q', this uniquely determines the value of the functions φ, Υ and Γ for q and b.

Eventually π executes the programmed FSC on P until it solves P. The subset of fluents $\{\mathsf{cs}_q : q \in Q\} \cup F \subseteq F_n$ represents the current world state (q, s) of an FSC $\mathcal{C}$ with controller states $Q = \{q_0, \ldots, q_n\}$. The definition of I_n sets the initial world state is (q_0, I). To satisfy the goal condition G_n, π has to simulate a *well-defined* transition sequence $(q_0, I) \rightarrow^u_{\mathcal{C}} (q_n, s)$, $0 \leq u$, such that G holds in s, i.e. $G \subseteq s$.

The partial action sequence $\langle \mathsf{econd}^f_q, \mathsf{eact}^b_{q,a}, \mathsf{esucc}^b_{q,q'} \rangle$ precisely simulates a well-defined transition $(q, s) \rightarrow_{\mathcal{C}} (q', s')$ of $\mathcal{C}$. Action econd^f_q adds o^b where $b \in \{0,1\}$ is the truth value of f in s. Action $\mathsf{eact}^b_{q,a}$ applies the action a in s to obtain a new state $s' = \theta(s, a)$. Finally, action $\mathsf{esucc}^b_{q,q'}$ transitions to controller state q'. This deterministic execution continues until we reach a terminal state (q_n, s) or revisit a world state. If π solves P_n, execution finishes in (q_n, s) and the goal condition G holds in s, which is the definition of the FSC solving P. □

Theorem 6.2 (Completeness)**.** *If there exists an FSC* $\mathcal{C} = (\langle Q, q_0, q_\perp, \{0,1\}, A, \Upsilon, \Gamma \rangle, \varphi)$ *that solves* P*, there exists a corresponding plan* π *that solves* P_n *for each* $n \geq |Q| - 1$.

Proof. By definition, fluents cond^f_q, $\mathsf{act}^b_{q,a}$ and $\mathsf{succ}^b_{q,q'}$ can altogether determine the φ, Υ and Γ functions of any FSC (provided that there is an enough amount n of controller states). Therefore, a plan π can be built that programs the functions φ, Γ and Υ of any FSC making the appropriate fluents true among cond^f_q, $\mathsf{act}^b_{q,a}$ and $\mathsf{succ}^b_{q,q'}$ using the corresponding actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$.

The fact that $\mathcal{C}$ solves P implies that there exists a sequence of well-defined world transitions $(q_0, I) \rightarrow^u_{\mathcal{C}} (q_n, s)$, $0 \leq u$, such that $G \subseteq s$. The execution of the sequence of world transitions of any FSC can be simulated using action sequences of type $\langle \mathsf{econd}^f_q, \mathsf{eact}^b_{q,a}, \mathsf{esucc}^b_{q,q'} \rangle$. Therefore, the plan π can be extended simulating the execution of the functions φ, Γ and Υ, starting from (q_0, I) and until reaching a world state (q_n, s) s.t. $G \subseteq s$. This is the definition of π satisfying the goal condition G_n to solve P_n.

□

Our compilation is not complete in the sense that the bound n on the number of controller states may be too small to accommodate an FSC that solves P. For instance, the FSC in Figure 6.2 cannot be computed if $n < 3$. Larger values of n do not formally affect the completeness of our approach but they do affect its practical performance since the sets F_n and A_n grow with the parameter n and classical planners are sensitive to the input size.

6.3.5 Computing FSCs for Generalized Planning

This section presents an extension of our compilation to compute *FSCs for generalized planning*. The input to the compilation is no longer a single classical planning problem, but a finite set of classical planning problems that define the generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$. The output of the extended compilation is a classical planning problem whose solution induces an FSC $\mathcal{C}$ and validates that $\mathcal{C}$ solves every classical planning problem P_t, $1 \leq t \leq T$.

Let us modify the definition of a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ such that now the planning problems $P_t = \langle F, A, I_t, G_t \rangle$, $1 \leq t \leq T$, share the fluent set F in addition to the action set A. Even though the action set is shared, the precise effects of an action is determined by the state where the action is applied due to conditional effects. An FSC for $\mathcal{P}$ is a pair $\mathcal{C} = (\Delta, \varphi)$ of a transducer Δ and a mapping φ, inducing an observation function O which is shared among the planning problems in $\mathcal{P}$. As before, the FSC $\mathcal{C}$ solves $\mathcal{P}$ iff $\mathcal{C}$ solves each P_t, $1 \leq t \leq T$.

Since the extension of the compilation, $P'_n = \langle F'_n, A'_n, I'_n, G'_n \rangle$, is similar to the original compilation $P_n = \langle F_n, A_n, I_n, G_n \rangle$, we define P'_n in terms of P_n:

- The set of fluents is $F'_n = F_n \cup F_{test}$, where $F_{test} = \{\mathsf{test}_t : 1 \leq t \leq T\}$ models the current classical planning problem in $\mathcal{P}$ that is being solved.
- The set of actions is $A'_n = A_n \cup A_{end}$, where A_{end} includes actions end_t, $1 \leq t < T$, for ending the execution on the current classical planning problem in $\mathcal{P}$ and enabling the next one:

$$\begin{aligned} \mathsf{pre}(\mathsf{end}_t) &= G_t \cup \{\mathsf{cs}_{q_n}, \mathsf{test}_t\}, \\ \mathsf{cond}(\mathsf{end}_t) &= \{\emptyset \rhd \{\neg\mathsf{cs}_{q_n}, \mathsf{cs}_{q_0}, \neg\mathsf{test}_t, \mathsf{test}_{t+1}\}\} \\ &\quad \cup \{\{\neg f\} \rhd \{f\} : f \in I_{t+1}\} \cup \{\{f\} \rhd \{\neg f\} : \neg f \in I_{t+1}\}. \end{aligned}$$

  The precondition tests that we have reached the goals G_t of the current problem P_t in the terminal controller state q_n, while the effect resets the world state to (q_0, I_{t+1}), i.e. the initial state of the next problem P_{t+1}, which becomes the current problem.

- The initial state is $I'_n = I_1 \cup \{\mathsf{cs}_{q_0}, \mathsf{test}_1\} \cup \{\mathsf{nocond}_q, \mathsf{noact}^b_q, \mathsf{nosucc}^b_q : q \in Q, b \in \{0,1\}\}$, while the goal condition is $G'_n = G_T \cup \{\mathsf{cs}_{q_n}, \mathsf{test}_T\}$.

A plan solving P'_n induces an FSC $\mathcal{C}$ and iterates over the classical planning problems $P_t \in \mathcal{P}$, $1 \leq t \leq T$ using actions of type end_t and hence, validating that $\mathcal{C}$ solves every $P_1, \ldots, P_T$.

6.4 Hierarchical Finite State Controllers

This section extends our FSCs formalism to *hierarchical FSCs*. The extension allows a controller to call other controllers, forming hierarchies of FSCs and enabling the reuse of existing controllers. In addition, a *hierarchical FSC* have a list of parameters. This makes it possible to implement recursive solutions by allowing an given controller to call itself with different arguments.

6.4.1 Parameter passing

To explain the intuition behind hierarchical FSCs, we borrow several concepts from programming. An FSC is similar to a *procedure* in programming, i.e. an independent program unit with an associated set of program instructions. A procedure can be *called* an arbitrary number of times, which consists in executing the associated program instructions. Procedure calls are organized in a *call stack* that keeps track of where execution should continue once the execution of a given procedure ends.

A procedure may contain *local variables* whose values are different for each call to the procedure. Some of these local variables may be designated as *parameters* of the procedure. When a procedure is called, it is necessary to specify the values of its parameters. Since local variables have different values for different procedure calls, each level of the call stack maintains a *copy* of each local variable, storing its value for the current procedure call.

Similar to planning programs with procedures (see Chapter 4), the first step necessary for defining *hierarchical FSCs* is to introduce the concept of *local variable*. Given a classical planning problem $P = \langle F, A, I, G \rangle$ we assume that the set of fluents F of a planning problem are instantiated from a set of predicates Ψ and a set of objects Ω. Our approach is to designate a subset of objects, $\Omega_v = \{v_1, \ldots, v_q\} \subseteq \Omega$, as *local variables*. To represent the assignment of values to local variables we use the same formulation as in Section 4.3.

In programming, a procedure call can either assign a concrete value to a parameter, or pass a variable as argument such that the current value of the variable is assigned to the parameter. In this work we are assuming that *parameter passing* is always of the second type, i.e. the arguments passed to the parameters of a

controller are also local variable objects in Ω_v. With this regard, the same local variables can be reused for all the controllers in a *hierarchical FSC*, even if these local variables have different uses in the different controllers. The reason is that each level of the stack maintains a *copy* of each local variable.

6.4.2 Hierarchical FSCs for planning

We are now ready to define a hierarchical FSC for a classical planning problem P as a pair $\mathcal{H} = (\mathfrak{C}, \mathcal{C}_1)$, where $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ is a set of FSCs for P and $\mathcal{C}_1 \in \mathfrak{C}$ is the root FSC. Each FSC $\mathcal{C}_i = (\langle Q, q_0, q_\perp, \{0,1\}, \Lambda, \Upsilon_i, \Gamma_i \rangle, \varphi_i)$, $1 \leq i \leq m$, shares the set of controller states Q and output set Λ, and differs only in the three functions φ_i, Υ_i and Γ_i that govern the transitions of $\mathcal{C}_i$. In addition, each FSC $\mathcal{C}_i$ has an arity $r_i \leq |\Omega_v|$. Because of symmetry, WLOG we define the parameter list of $\mathcal{C}_i$ as $[v_1, \ldots, v_{k_i}]$, i.e. the r_i first local variable objects in Ω_v.

The shared output set $\Lambda = A \cup \mathcal{Z}$ extends the set of primitive planning actions A to include also the set of possible *calls* $\mathcal{Z} = \{\mathcal{C}_i[p] : \mathcal{C}_i \in \mathfrak{C}, p \in \Omega_v^{r_i}\}$. That is, each transition of a controller $\mathcal{C}_i$ either (1), applies an action in A or (2), calls an FSC $\mathcal{C}_j[p]$ with the specific arguments $p \in \Omega_v^{r_i}$. We remark that a call action in $\mathcal{Z}$ can be used to implement recursion making a controller to call itself.

The execution model of hierarchical FSCs for planning

To model the execution of a *hierarchical FSC* $\mathcal{H}$ on a classical planning problem P, we introduce the concept of a *call stack* for FSCs. Because of the local fluents $\mathsf{assign}(v, x)$, a planning state $s = s_l \cup s_g$ can be decomposed into a local state s_l and a global state s_g. Each level of the call stack maintains its own copy of the local state s_l, while the global state s_g is shared among all levels of the call stack. The execution of a hierarchical FSC on P starts at the root controller $\mathcal{C}_1$ in state (q_0, I) and on level 0 of the call stack.

For a given controller $\mathcal{C}_i$ and world state (q, s) with $s = s_l \cup s_g$, if $\Gamma_i(q, O(q, s)) = a$ returns an action $a \in A$, the execution semantics is the same as for single FSCs. However, when $\Gamma_i(q, O(q, s)) = \mathcal{C}_j[p]$ returns an FSC call in $\mathcal{Z}$, we push call $\mathcal{C}_j[p]$ onto the call stack, setting the world state to $(q_0, s'_l \cup s_g)$, i.e. the initial controller state q_0 and a new local state s'_l obtained from s_l by copying the value of each local variable in p to the corresponding parameter in the parameter list $[v_1, \ldots, v_{k_i}]$ of $\mathcal{C}_j$. Execution then proceeds on the next stack level following the definition of $\mathcal{C}_j$.

When we reach a terminal state $(q_\perp, s')$ of $\mathcal{C}_j$ with $s' = s'_l \cup s'_g$, control is returned to the parent controller $\mathcal{C}_i$ by popping the procedure call $\mathcal{C}_j[p]$ from the call stack. Specifically, the state of $\mathcal{C}_i$ becomes $(q', s_l \cup s'_g)$ where $q' = \Upsilon_i(q, O(q, s))$ is the next controller state according to the transition function Υ_i, and s_l is the

assignment to the local variables already stored in the call stack. The execution of $\mathcal{H}$ on P terminates when it reaches a terminal state $(q_\perp, s)$ on stack level 0, and $\mathcal{H}$ solves P iff $G \subseteq s$.

To ensure that the execution model remains finite, we define an upper bound ℓ on the size of the call stack. As a consequence, the execution of a *hierarchical FSC* $\mathcal{H}$ on a classical planning problem P has a fourth failure condition:

4. Execution does not terminate because, when executing an FSC call $\mathcal{C}_j[p] \in \mathcal{Z}$ the size of the stack equals ℓ. Executing such a call would result in a call stack whose size exceeds the upper bound ℓ, i.e. a *stack overflow*.

An example of hierarchical FSC for planning

To illustrate our definition of *hierarchical FSCs* for planning, we use binary tree traversal as an example (Figure 6.1). In addition to the tree nodes, we introduce two local variable objects $\Omega_v = \{n, child\}$, and define the actions on variable objects. We can model this task as a planning problem $P = \langle F, A, I, G \rangle$, where F contains the following fluents:

- For each tree node x, a fluent $\mathsf{visited}(x)$ denoting that x has been visited.
- For each pair of tree nodes x, y, two fluents $\mathsf{left}(x, y)$ and $\mathsf{right}(x, y)$ denoting that y is the left or right child of x, respectively.
- For each variable $v \in \{n, child\}$ and tree node x, a fluent $\mathsf{assign}(v, x)$ denoting that x is assigned to the variable v, and a fluent $\mathsf{isNull}(v)$ indicating that v is empty.
- For each $v \in \{n, child\}$, a fluent $\mathsf{isVisited}(v)$ denoting that the node assigned to v has been visited, modelled as a derived predicate $\mathsf{isVisited}(v) \equiv \exists x\ \mathsf{assign}(v, x) \wedge \mathsf{visited}(x)$.

The action set A contains the following actions:

- For each $v \in \{n, child\}$, an action $\mathsf{visit}(v)$ that marks the node assigned to v as visited:

$$\mathsf{pre}(\mathsf{visit}(v)) = \emptyset,$$
$$\mathsf{cond}(\mathsf{visit}(v)) = \{\{\mathsf{assign}(v, x)\} \rhd \{\mathsf{visited}(x)\} : \forall x\}.$$

- For each $u, v \in \{n, child\}$, an action $\mathsf{copyL}(u, v)$ that copies the left child of u onto v:

$$\begin{aligned}\mathsf{pre}(\mathsf{copyL}(u,v)) &= \emptyset,\\ \mathsf{cond}(\mathsf{copyL}(u,v)) &= \{\emptyset \rhd \{\mathsf{isNull}(v), \neg\mathsf{assign}(v,x) : \forall x\}\}\\ &\cup\ \{\{\mathsf{assign}(u,x), \mathsf{left}(x,y)\} \rhd\\ &\qquad \{\neg\mathsf{isNull}(v), \mathsf{assign}(v,y)\} : \forall x, y\}.\end{aligned}$$

  By default, the effect is $\mathsf{isNull}(v)$, but if u has a left child that node is assigned to v.

- For each $u, v \in \{n, child\}$, an action $\mathsf{copyR}(u, v)$ that copies the right child of u onto v, with a definition analogous to $\mathsf{copyL}(u, v)$.

For the binary tree in Figure 6.1, the initial state I and goal condition G are defined as

$$\begin{aligned}I &= \{\mathsf{assign}(n, x_0), \mathsf{left}(x_0, x_1), \mathsf{right}(x_0, x_9), \ldots, \mathsf{right}(x_{13}, x_{15})\},\\ G &= \{\mathsf{visited}(x_1), \ldots, \mathsf{visited}(x_{15})\}.\end{aligned} \tag{6.2}$$

Figure 6.2 shows a hierarchical FSC $\mathcal{H} = (\{\mathcal{C}\}, \mathcal{C})$ that solves P. Even though $\mathcal{H}$ contains a single FSC $\mathcal{C}$, it is still hierarchical in the sense that $\mathcal{C}$ includes a call to itself, represented by the edge (q_2, q_0) with label $!\mathsf{isVisited}(n)/\mathsf{call}(child)$. Note that $\mathcal{C}$ has a single parameter, which we define as the first variable object in Ω_v, namely n.

6.4.3 Computing Hierarchical Finite State Controllers

We now describe a compilation from a classical planning problem $P = \langle F, A, I, G\rangle$ into another classical planning problem $P^{\ell}_{n,m} = \langle F^{\ell}_{n,m}, A^{\ell}_{n,m}, I^{\ell}_{n,m}, G^{\ell}_{n,m}\rangle$, such that solving $P^{\ell}_{n,m}$ amounts to programming a *hierarchical FSC* $\mathcal{H} = \langle \mathfrak{C}, \mathcal{C}_1\rangle$ and simulating its execution on P. The parameters of the compilation are n, a bound on the number of controller states, m that is a bound on the number of FSCs and ℓ, a bound on the stack size. For each $\mathcal{C}_i \in \mathfrak{C}$, the compilation also needs to specify a bound r_i on the number of parameters of the corresponding controller.

The set of fluents is given by $F^{\ell}_{n,m} = F_r \cup F^{\ell}_a \cup F^{m}_{fun} \cup F^{\ell}_{aux} \cup F_H$ where

- F_r is the set of fluents instantiated from predicates different from assign.
- $F^{\ell}_a = \{f^k : 0 \le k \le \ell, f \in F_a\}$, where $F_a = \{\mathsf{assign}_{v,x} : v \in \Omega_v, x \in \Omega \setminus \Omega_v\}$ is the set of fluents instantiated from assign. By parameterizing on the stack level k, all fluents in F_a are evaluated with respect to the current stack level.

- $F^m_{fun} = \{f^i : f \in F_{fun}, 1 \leq i \leq m\}$ where, as before, F_{fun} is the set of fluents modelling the functions φ, Υ and Γ, which we parameterize on the FSC $\mathcal{C}_i$, $1 \leq i \leq m$.

- $F^\ell_{aux} = \{f^k : f \in F_{aux}, 0 \leq k \leq \ell\}$ where, as before, F_{aux} is the set of fluents representing the execution model, which we parameterize now on the stack level k.

Moreover, F_H contains the following additional fluents:

- For each k, $0 \leq k \leq \ell$, a fluent lvl^k denoting that k is the current level of the call stack.

- For each $\mathcal{C}_i \in \mathfrak{C}$ and k, $0 \leq k \leq \ell$, a fluent $\mathsf{fsc}^{i,k}$ denoting that $\mathcal{C}_i$ is the FSC being executed on stack level k.

- For each $q \in Q$, $b \in \{0,1\}$, $\mathcal{C}_i, \mathcal{C}_j \in \mathfrak{C}$ and $p \in \Omega^{r_j}_v$, a fluent $\mathsf{call}^{b,i}_{q,j}(p)$ denoting that $\Gamma_i(q,b) = \mathcal{C}_j[p]$.

The initial state and goal condition are defined as $I^\ell_{n,m} = (I \cap F_r) \cup \{f^0 : f \in I \cap F_a\} \cup \{\mathsf{cs}^0_{q_0}, \mathsf{lvl}^0, \mathsf{fsc}^{1,0}\} \cup \{\mathsf{nocond}^i_q, \mathsf{noact}^{b,i}_q, \mathsf{nosucc}^{b,i}_q : q \in Q, b \in \{0,1\}, \mathcal{C}_i \in \mathfrak{C}\}$ and $G^\ell_{n,m} = G \cup \{\mathsf{cs}^0_{q_n}\}$. In other words, fluents of type $\mathsf{assign}_{v,x} \in F_a$ are initially marked with stack level 0, the controller state on level 0 is q_0, the current stack level is 0, the FSC on level 0 is $\mathcal{C}_1$, and the functions φ_i, Υ_i and Γ_i are yet to be programmed for any FSC $\mathcal{C}_i \in \mathfrak{C}$. To satisfy the goal we have to reach the terminal state q_n on level 0 of the stack.

To establish the actions in the set $A^\ell_{n,m}$, we first adapt all actions in A_n by parameterizing on the FSC $\mathcal{C}_i \in \mathfrak{C}$ and stack level k, $0 \leq k \leq \ell$, adding preconditions lvl^k and $\mathsf{fsc}^{i,k}$, and modifying the remaining preconditions and effects accordingly. As an illustration we provide the definition of the resulting action $\mathsf{pcond}^{f,i,k}_q$:

$$\mathsf{pre}(\mathsf{pcond}^{f,i,k}_q) = \{\mathsf{lvl}^k, \mathsf{fsc}^{i,k}, \mathsf{cs}^k_q, \mathsf{nocond}^i_q\},$$
$$\mathsf{cond}(\mathsf{pcond}^{f,i,k}_q) = \{\emptyset \rhd \{\neg\mathsf{nocond}^i_q, \mathsf{cond}^{f,i}_q\}\}.$$

Compared to the old version of pcond^f_q, the current controller state $\mathsf{cs}^k_q \in F^\ell_{aux}$ refers to the stack level k, and fluents nocond^i_q and $\mathsf{cond}^{f,i}_q$ in F^m_{fun} refer to the FSC $\mathcal{C}_i$. The precondition models the fact that we can only program the function φ_i of $\mathcal{C}_i$ in controller state q on stack level k when k is the current stack level, $\mathcal{C}_i$ is being executed on level k, the current controller state on level k is q, and φ_i has not been previously programmed in q.

In addition to the actions adapted from A_n, the set $A^\ell_{n,m}$ also contains the following new actions:

- For each $q \in Q$, $b \in \{0,1\}$, $\mathcal{C}_i, \mathcal{C}_j \in \mathfrak{C}$, $p \in \Omega_v^{r_j}$ and k, $0 \leq k < \ell$, an action $\mathsf{pcall}_{q,j}^{b,i,k}(p)$ to program a call from the current FSC $\mathcal{C}_i$ to FSC $\mathcal{C}_j$:

$$\begin{aligned}\mathsf{pre}(\mathsf{pcall}_{q,j}^{b,i,k}(p)) &= \{\mathsf{lvl}^k, \mathsf{fsc}^{i,k}, \mathsf{cs}_q^k, \mathsf{evl}^k, \mathsf{o}^{b,k}, \mathsf{noact}_q^{b,i}\},\\ \mathsf{cond}(\mathsf{pcall}_{q,j}^{b,i,k}(p)) &= \{\emptyset \rhd \{\neg\mathsf{noact}_q^{b,i}, \mathsf{call}_{q,j}^{b,i}(p)\}\}.\end{aligned}$$

- For each $q \in Q$, $b \in \{0,1\}$, $\mathcal{C}_i, \mathcal{C}_j \in \mathfrak{C}$, $p \in \Omega_v^{r_j}$ and l, $0 \leq k < \ell$, an action $\mathsf{ecall}_{q,j}^{b,i,k}(p)$ that executes an FSC call:

$$\begin{aligned}\mathsf{pre}(\mathsf{ecall}_{q,j}^{b,i,k}(p)) &= \{\mathsf{lvl}^k, \mathsf{fsc}^{i,k}, \mathsf{cs}_q^k, \mathsf{evl}^k, \mathsf{o}^{b,k}, \mathsf{call}_{q,j}^{b,i}(p), \neg\mathsf{app}^k\},\\ \mathsf{cond}(\mathsf{ecall}_{q,j}^{b,i,k}(p)) &= \{\emptyset \rhd \{\neg\mathsf{lvl}^k, \mathsf{lvl}^{k+1}, \mathsf{fsc}^{j,k+1}, \mathsf{cs}_{q_0}^{k+1}, \mathsf{app}^k\}\}\\ &\quad \cup\ \{\{\mathsf{assign}_{p_r,x}^k\} \rhd \{\mathsf{assign}_{v_r,x}^{k+1}\} : 1 \leq r \leq r_j, x \in \Omega_x\}.\end{aligned}$$

- For each $\mathcal{C}_i \in \mathfrak{C}$ and k, $0 < k \leq \ell$, an action $\mathsf{term}^{i,k}$:

$$\begin{aligned}\mathsf{pre}(\mathsf{term}^{i,k}) &= \{\mathsf{lvl}^k, \mathsf{fsc}^{i,k}, \mathsf{cs}_{q_n}^k\},\\ \mathsf{cond}(\mathsf{term}^{i,k}) &= \{\emptyset \rhd \{\neg\mathsf{lvl}^k, \neg\mathsf{fsc}^{i,k}, \neg\mathsf{cs}_{q_n}^k, \mathsf{lvl}^{k-1}\}\} \cup\\ &\qquad \{\emptyset \rhd \{\neg\mathsf{assign}_{v,x}^k : v \in \Omega_v, x \in \Omega_x\}\}.\end{aligned}$$

As an alternative to $\mathsf{pact}_{q,a}^{b,i,k}$, the action $\mathsf{pcall}_{q,j}^{b,i,k}(p)$ programs an FSC call $\mathcal{C}_j[p]$, i.e. defines the output function as $\Gamma_i(q,b) = \mathcal{C}_j[p]$. Action $\mathsf{ecall}_{q,j}^{b,i,k}(p)$ executes this FSC call by incrementing the current stack level to $k+1$ and setting the controller state on level $k+1$ to q_0. The conditional effect $\{\mathsf{assign}_{p^r,x}^k\} \rhd \{\mathsf{assign}_{\mathcal{L}_j^r,x}^{k+1}\}$ effectively copies the value of the argument p^r on level k to the corresponding parameter $\mathcal{L}_j^r$ of $\mathcal{C}_j$ on level $k+1$. When in the terminal state q_n, the termination action $\mathsf{term}^{i,k}$ decrements the stack level to $k-1$ and deletes all temporary information about stack level k.

Besides computing a *hierarchical FSC* starting from scratch, the $P_{n,m}^\ell$ compilation is flexible to reuse existing solutions. In this regard, our compilation can also partially specify the functions Γ_i, Λ_i and φ_i of an FSC $\mathcal{C}_i$ by setting to `True` the corresponding fluents of type $\mathsf{cond}_q^{f,i}$, $\mathsf{act}_{q,a}^{b,i}$, $\mathsf{succ}_{q,q'}^{b,i}$ and $\mathsf{call}_{q,j}^{b,i}(p)$ in the initial state $I_{n,m}^\ell$. This way, we can incorporate prior knowledge regarding the configuration of some previously existing FSCs in $\mathfrak{C}$. Interestingly, this idea can be exploited to determine whether a given string e belongs to the regular language defined by a given FSC by ignoring the actions for programming the transition function. In this case, a solution plan π represents the transitions in the given FSC proving that the string e is accepted.

The $P_{n,m}^\ell$ compilation can also be extended to address a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ in a way analogous to P_n. Specifically, each action

end_t, $1 \leq t < T$, should have precondition $G_t \cup \{\mathsf{cs}^0_{q_n}\}$ and reset the state to $I_{t+1} \cup \{\mathsf{cs}^0_{q_0}\}$, i.e. the system should reach the terminal state q_n on stack level 0 and satisfy the goal condition G_t of P_t before execution proceeds on the next problem $P_{t+1} \in \mathcal{P}$. To solve $P^\ell_{n,m}$, a plan hence has to simulate the execution of $\mathcal{H}$ on all planning problems in $\mathcal{P}$.

6.4.4 Example

We show how the compilation $P^\ell_{n,m}$ computes the hierarchical FSC in Figure 6.2 for binary tree traversal (with $m = 1$). Recall that the initial state is given by $I^\ell_{n,m} = (I \cap F_r) \cup \{f^0 : f \in I \cap F_a\} \cup \{\mathsf{cs}^0_{q_0}, \mathsf{lvl}^0, \mathsf{fsc}^{1,0}\} \cup \{\mathsf{nocond}^i_q, \mathsf{noact}^{b,i}_q, \mathsf{nosucc}^{b,i}_q : q \in Q, b \in \{0,1\}, \mathcal{C}_i \in \mathfrak{C}\}$, with I given by the expression (6.2).

At $I^\ell_{n,m}$ the only applicable actions are $\mathsf{pcond}^{f,1,0}_{q_0}$, $f \in F$, for programming the value of $\varphi_1(q_0)$ on stack level 0. In Figure 6.2 this transition fires no matter what so any static fluent can be programmmed here, e.g. $f = \mathsf{left}(x_1, x_3)$. Similar to the example in Section 6.3.3, the following action sequence programs and simulates the complete transition from q_0 to q_1 on level 0 of the stack:

$$\langle \mathsf{pcond}^{\mathsf{left}(x_1,x_3),1,0}_{q_0}, \mathsf{econd}^{\mathsf{left}(x_1,x_3),1,0}_{q_0}, \mathsf{pact}^{0,1,0}_{q_0,\mathsf{copyL}(n,child)},$$
$$\mathsf{eact}^{0,1,0}_{q_0,\mathsf{copyL}(n,child)}, \mathsf{psucc}^{0,1,0}_{q_0,q_1}, \mathsf{esucc}^{0,1,0}_{q_0,q_1} \rangle.$$

The action sequences for programming the transitions from q_1 to q_2 and from q_2 to itself are analogous. The resulting action sequence on A is $\langle \mathsf{copyL}(n, child), \mathsf{visit}(n), \mathsf{copyR}(n, n) \rangle$, and the corresponding effect on F is $\{\mathsf{assign}^0_{child,x_2}, \mathsf{visited}(x_1), \neg\mathsf{assign}^0_{n,x_1}, \mathsf{assign}^0_{n,x_9}\}$.

The programming and simulation of the recursive call of the controller in Figure 6.2 occurs in the partial state $\{\mathsf{cs}_{q_2}, \mathsf{o}^0\}$, i.e. when $\mathsf{isVisited}(n)$ is false in controller state q_2. To replicate the FSC in Figure 6.2, the planner should program and simulate the recursive call using actions $\mathsf{pcall}^{0,1,0}_{q_2,1}(child)$ and $\mathsf{ecall}^{0,1,0}_{q_2,1}(child)$. The effect of $\mathsf{ecall}^{0,1,0}_{q_2,1}(child)$ is $\{\neg lvl^0, \mathsf{lvl}^1, \mathsf{fsc}^{1,1}, \mathsf{cs}^1_{q_0}, \mathsf{app}^0, \mathsf{assign}^1_{n,x_2}\}$, where the assignment $\mathsf{assign}^1_{n,x_2}$ is copied from $\mathsf{assign}^0_{child,x_2}$ due to the argument $child$ being passed to the lone parameter n of the FSC C_1 being called. As a result, execution of the controller on P continues on level 1 of the call stack.

On stack level 1, execution is deterministic, resulting in the same transition sequence $q_0 \to q_1 \to q_2 \to q_2 \to q_0$ and causing another recursive call but using now action $\mathsf{ecall}^{0,1,1}_{q_2,1}(child)$, that assigns node x_3 to n on level 2 of the stack. Recursion continues until $\mathsf{isNull}(n)$ becomes true, in which case we can program and simulate the transition from q_1 to the terminal controller state q_3. In turn, this allows us to pop an FSC call from the stack. At this point, since app^k was added by $\mathsf{ecall}^{0,1,k}_{q_2,1}(child)$ at the previous stack level k, we can finally program and simulate the transition from q_2 to q_0 using actions $\mathsf{psucc}^{0,1,k}_{q_2,q_0}$ and $\mathsf{esucc}^{0,1,k}_{q_2,q_0}$. Note

that the bound ℓ on the stack size has to be sufficiently large to accomodate all recursive calls otherwise $P^{\ell}_{n,m}$ will not have a solution. In the particular case of binary tree traversal, ℓ has to be larger than the tree depth.

6.4.5 Properties

Here we analyze the theoretical properties of our compilation $P^{\ell}_{n,m}$ for synthesizing hierarchical FSCs.

Theorem 6.3 (Soundness)**.** *Any plan π that solves $P^{\ell}_{n,m}$ induces a hierarchical FSC $\mathcal{H} = (\mathfrak{C}, \mathcal{C}_1)$ that solves P.*

Proof. First we show that at any moment, a single fluent of type lvl^k is true, denoting the top level of the stack, and that all fluents for levels $k+1$ and higher are set to `False`. In the initial state $I^{\ell}_{n,m}$, the top level is lvl^0 and all fluents for level 1 and higher are false. The only actions for changing the top level are $\mathsf{ecall}^{b,i,k}_{q,j}(p)$ for pushing an FSC call onto level $k+1$ of the stack, and $\mathsf{term}^{i,k}$ for popping an FSC call from level k of the stack. Note that $\mathsf{ecall}^{b,i,k}_{q,j}(p)$ deletes lvl^k and adds lvl^{k+1}, and only adds fluents for level $k+1$. Likewise, $\mathsf{term}^{i,k}$ deletes lvl^k and adds lvl^{k-1}, and deletes all fluents for level k proving that the claim holds.

Next we show that for each stack level k, at or below the top, fluents $\mathsf{fsc}^{i,k}$ and cs^k_q uniquely determine the current FSC $\mathcal{C}_i \in \mathfrak{C}$ and the current controller state $q \in Q$. In the initial state, only $\mathsf{fsc}^{1,0}$ and $\mathsf{cs}^0_{q_0}$ are true, i.e. the current FSC is $\mathcal{C}_1$ in controller state q_0 on stack level 0. Action $\mathsf{ecall}^{b,i,k}_{q,j}(p)$ adds $\{\mathsf{fsc}^{j,k+1}, \mathsf{cs}^{k+1}_{q_0}\}$, indicating that $\mathcal{C}_j$ is the FSC pushed onto level $k+1$ of the stack in controller state q_0. Action $\mathsf{term}^{i,k}$ deletes $\{\mathsf{fsc}^{i,k}, \mathsf{cs}^k_{q_n}\}$, indicating that the FSC call to $\mathcal{C}_i$ is popped from level k of the stack. As happens with simple FSCs, the only action for changing the controller state is $\mathsf{esucc}^{b,i,k}_{q,q'}$, which transitions from q to q' when b is the outcome of the observation function in q. The only difference here is that $\mathsf{esucc}^{b,i,k}_{q,q'}$ is parameterized by i and k. Because of precondition $\{\mathsf{lvl}^k, \mathsf{fsc}^{i,k}\}$ of $\mathsf{esucc}^{b,i,k}_{q,q'}$, k has to be the top level of the stack and $\mathcal{C}_i$ has to be the FSC that is executing on level k of the stack.

Now we show that actions $\mathsf{pcall}^{b,i,k}_{q,j}(p)$ and $\mathsf{ecall}^{b,i,k}_{q,j}(p)$ for programming and executing an FSC call $\mathcal{C}_j[p]$ and action $\mathsf{term}^{i,k}$ for terminating an FSC call correctly simulate the execution model for *hierarhical FSCs*. Since $\mathsf{pcall}^{b,i,k}_{q,j}(p)$ has the same precondition $\{\mathsf{cs}^k_q, \mathsf{evl}^k, \mathsf{o}^{b,k}, \mathsf{noact}^{b,i}_q\}$ as $\mathsf{pact}^{b,i,k}_{q,a}$, and since both delete $\mathsf{noact}^{b,i}_q$, programming an FSC call for q and b in FSC $\mathcal{C}_i$ is an alternative to programming an action for q and b, effectively establishing the value of the function $\Gamma_i(q,b) \in A \cup \mathcal{Z}$. The effect of $\mathsf{ecall}^{b,i,k}_{q,j}(p)$ is $\{\mathsf{lvl}^{k+1}, \mathsf{fsc}^{j,k+1}, \mathsf{cs}^{k+1}_{q_0}\}$, pushing the FSC call $\mathcal{C}_j[p]$ onto the call stack and making $\mathcal{C}_j$ the active FSC on the top level

$k+1$. Moreover, the conditional effect $\{\{\mathsf{assign}^{k}_{p_r,x}\} \rhd \{\mathsf{assign}^{k+1}_{v_r,x}\} : 1 \leq r \leq r_j, x \in \Omega_x\}$ copies the values of the variable objects in p onto the variable objects $v_1, \ldots, v_{r_j}$ that constitute the parameters of $\mathcal{C}_j$. Finally, action $\mathsf{term}^{i,k}$ pops the FSC call involving $\mathcal{C}_i$ from the stack when the terminal controller state q_n has been reached.

It only remains to reuse the argument from the proof of Theorem 6.1 to show that the actions adapted from A_n, program and simulate the execution of an FSC $\mathcal{C}_i$ on a single stack level. The actions adapted from A_n, i.e. those for programming or executing a function among φ_i, Υ_i and Γ_i, $1 \leq i \leq m$, include the extra precondition $\{\mathsf{lvl}_k, \mathsf{fsc}^{i,k}\}$. In other words, the corresponding FSC $\mathcal{C}_i$ has to be active on the top of the stack. Other than that, these actions behave just as for single FSCs.

□

Theorem 6.4 (Completeness)**.** *If there exists a hierarchical FSC $\mathcal{H} = (\mathfrak{C}, \mathcal{C}_1)$ that solves P then there exists a plan π that solves $P^{\ell}_{n,m}$ given that the n, m, and ℓ bounds are large enough.*

Proof. The plan π is built as follows. Whenever the execution of $\mathcal{H}$ on P (starting with the initial planning state I, the first state of the root controller $\mathcal{C}_1$, and an empty call stack) reaches a controller state $q \in Q$ of an FSC $\mathcal{C}_i$ for the first time, then π programs the three functions φ_i, Υ_i and Γ_i of the FSC $\mathcal{C}_i$ as specified by $\mathcal{H}$, and using the corresponding programming actions $\mathsf{pcond}^{f,i,k}_{q}$, $\mathsf{pact}^{b,i,k}_{q,a}$ or $\mathsf{psucc}^{b,i,k}_{q,q'}$. As an alternative to $\mathsf{pact}^{b,i,k}_{q,a}$, the FSC calls of $\mathcal{H}$ are programmed with $\mathsf{pcall}^{b,i,k}_{q,j}(p)$.

Once this execution reaches a controller state whose transition functions encoded by φ_i, Υ_i and Γ_i are already programmed, an action sequence of type $\langle \mathsf{econd}^{f,i,k}_{q}, \mathsf{eact}^{b,i,k}_{q,a}, \mathsf{esucc}^{b,i,k}_{q,q'} \rangle$ that simulates the execution of the corresponding transition is added to the plan π. Note that in this case an action $\mathsf{ecall}^{b,i,k}_{q,j}(p)$ is executed as an alternative to $\mathsf{eact}^{b,i,k}_{q,a}$, when the transition to simulate represents a controller call, and a $\mathsf{term}^{i,k}$ action is used to simulate the termination of the execution of a controller $\mathcal{C}_i \in \mathfrak{C}$.

A plan π built this way has the effect of programming $\mathcal{H}$ and simulating the execution of $\mathcal{H}$ on P. The fact that $\mathcal{H} = (\mathfrak{C}, \mathcal{C}_1)$ solves P implies that the simulation of the execution of $\mathcal{H}$ on P ends achieving G while leaving the root controller $\mathcal{C}_1$ at its terminal state and with the call stack empty, which is also the definition of a plan π solving $P^{\ell}_{n,m}$. □

6.5 Comparing FSCs and Planning Programs

Planning programs represent compact and generalized plans assigning planning actions to an enumeration of program lines (see Chapter 3). Apart from planning actions, the lines of a planning program can also contain *goto instructions* to implement control flow. Figure 6.4(a) shows a three-line planning program for decreasing the value of variable n until achieving the goal condition $n = 0$. This program assumes that n initially has a positive value, and that the effect of action `dec(n)` is to decrement the current value of n. Instruction `goto(0,!(n=0))`, in line one, indicates a conditional jump to line 0 if the value of variable n is not 0.

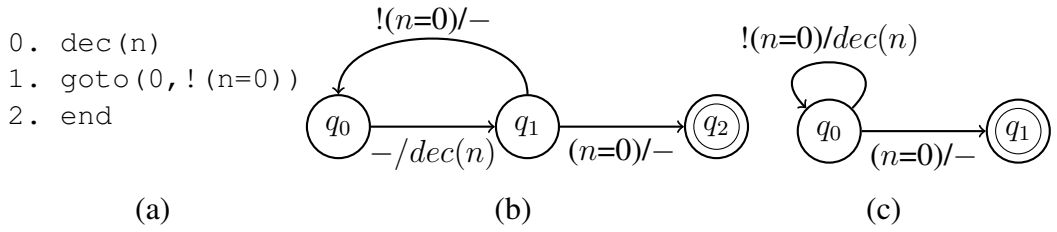


Figure 6.4: a) Three-line *planning program* for decreasing variable n until achieving $n = 0$; b) an equivalent three-state FSC; c) a more compact FSC representing the same generalized plan.

Essentially, *planning programs* can be understood as syntactic sugar for FSCs that separate the control flow structures from the primitive planning actions to become more human readable. On the other hand, FSCs can be more compact. Both programs and FSCs can represent hierarchical and recursive solutions as well as reuse existing solutions. Moreover, both programs and FSCs can be computed following a *top-down* approach that searches in a bounded space of solutions, e.g. exploiting compilations that *programs* an automaton and *validates* it on the input instances.

Theorem 6.5. *For any planning program Π with n program lines, there exists an equivalent FSC $\mathcal{C}$ with n controller states.*

Proof. Given a planning problem $P = \langle F, A, I, G\rangle$ and a planning program $\Pi = \langle w_0, \ldots, w_n\rangle$, we prove the theorem by constructing an equivalent FSC $\mathcal{C} = (\langle Q, q_0, q_n, \{0, 1\}, A, \Upsilon, \Gamma\rangle, \varphi)$, where $Q = \{q_0, \ldots, q_n\}$ has as many controller states as Π has program lines. For each controller state $q_i \in Q$, the functions φ, Υ and Γ are defined as follows:

- If the instruction w_i on line i of program Π is a primitive planning action, i.e. an action in A, then $\Upsilon(q_i, 0) = \Upsilon(q_i, 1) = q_{i+1}$ always transitions to the next controller state q_{i+1}, and $\Gamma(q_i, 0) = \Gamma(q_i, 1) = w_i$ always returns action w_i ($\varphi(q_i)$ can be arbitrarily defined since the transition is always to q_{i+1} no matter what fluent we associate with q_i).

- If the instruction w_i is a $goto(j, !f)$ instruction, then $\varphi(q_i) = f$ associates fluent f with q_i, $\Upsilon(q_i, 0) = q_j$ and $\Upsilon(q_i, 1) = q_{i+1}$ cause a transition to q_j if f is false, else to q_{i+1}, and $\Gamma(q_i, 0) = \Gamma(q_i, 1) = a_\perp$ returns the no-op action.

The world state (s, i) of a planning program Π comprises a planning state s and a program line i. We prove by induction that executing Π from (s, i) is equivalent to executing $\mathcal{C}$ from (q_i, s), where q_i is the controller state that corresponds to line i.

The base case is given by (s, n), in which case the execution of both Π and $\mathcal{C}$ terminates in the same state s (just as q_n is always a terminal controller state, the last instruction w_n of Π is always a termination instruction). The inductive case is given by world state (s, i) such that $i < n$:

- If $w_i \in A$, the resulting world state is $(s', i + 1)$ for Π, and (q_{i+1}, s') for $\mathcal{C}$, where $s' = \theta(s, w_i)$ is the result of applying the action w_i in the planning state s.

- If w_i is a goto instruction $goto(j, !f)$, the resulting world state is (s, j) for Π and (q_j, s) for $\mathcal{C}$ if f is false, and $(s, i + 1)$ for Π and (q_{i+1}, s) for $\mathcal{C}$ if f is true.

In each case, the resulting pair of world states (s', i') and $(q_{i'}, s')$ are identical, so by hypothesis of induction executing Π from (s', i') is equivalent to executing $\mathcal{C}$ from $(q_{i'}, s')$. In particular, this means that executing Π from the initial world state $(I, 0)$ is equivalent to executing $\mathcal{C}$ from (q_0, I), proving that Π and C are equivalent generalized plans. □

Figure 6.4(b) shows the equivalent FSC that we construct following the proof of Theorem 6.5 for the planning program in Figure 6.4(a). Now we prove also the other direction of Theorem 6.5 and prove hence, that planning programs are as expressive as FSCs (but not necessarily as compact):

Theorem 6.6. *For any FSC $\mathcal{C}$ with n controller states, there exists an equivalent planning program Π with $5 \times n$ program lines.*

Proof. Given a planning problem $P = \langle F, A, I, G \rangle$ and an FSC $\mathcal{C} = (\Delta, \varphi)$ with $\Delta = \langle Q, q_0, q_n, \{0, 1\}, A, \Upsilon, \Gamma \rangle$, we prove the theorem by constructing an

equivalent planning program $\Pi = \langle w_0, \ldots, w_{5n} \rangle$ with five times as many lines as $\mathcal{C}$ has controller states.

For a given controller state $q_i \in Q$, let $\varphi(q_i) = f$, $\Upsilon(q_i, 0) = q_j$, $\Upsilon(q_i, 1) = q_k$, $\Gamma(q_i, 0) = a$, $\Gamma(q_i, 1) = b$ be the transitions from q_i. We can exactly represent these same transitions from q_i using the following partial program:

```
  5i: goto(5i+3,!f)
5i+1: b
5i+2: goto(5k,!false)
5i+3: a
5i+4: goto(5j,!false)
```

Here, `false` is a dummy fluent whose value is always false, causing the corresponding goto condition to trigger every time. Executing the above partial program replicates the transition from q_i, so Π and $\mathcal{C}$ are equivalent generalized plans. □

FSCs are at least as compact as planning programs, and often strictly more compact (as shown in Figure 6.4(c)). In practice, this gives FSCs an advantage over planning programs, since a smaller bound on the number of controller states typically results in faster generation of FSCs.

6.6 Evaluation

This section evaluates the performance of our approach for the computation of FSCs in a selection of generalized planning benchmarks and programming tasks taken from Bonet et al. 2010, Srivastava et al. 2011b and some from Chapter 4.

6.6.1 Experimental setup and benchmarks

All experiments are run on a processor *Intel Core* i7 2.60GHz x 4 with a 4GB memory bound and time limit of 3600s. We compute the corresponding FSCs solving the classical planning problem that results from our compilation. The classical planning instances output by our compilation are solved running the following two classical planners:

1. FAST DOWNWARD (Helmert, 2006b) with the LAMA-2011 setting (Richter and Westphal, 2010).

2. BEST-FIRST WIDTH SEARCH with the Dual-BFWS setting (Lipovetzky and Geffner, 2017).

We briefly describe here each of the evaluation benchmarks. In the A^nB^n domain the goal is to compute a general controller that parses strings consisting of n A's followed by n B's. In *Blocks*, the goal is to compute a controller that unstack blocks from a single tower until a green block is found. In *Fibonacci* we calculate the n^{th} Fibonacci number. In *Gripper*, the goal is to transport a set of balls from one room to another using a two-gripper robot. In *List*, the goal is to visit all the nodes of a linked list (as in the example of Section 6.3) while in *Reverse*, the goal is to reverse the elements of a linked list. In the *Serial Binary Adder* (SBA) domain, we compute a controller that implements the algorithm for the addition of two binary numbers of unbounded size. In *Triangular Sum*, the goal is to compute $\sum_{i=1}^{n} i$ for a given n. In *Tree/DFS*, the goal is to visit all nodes of binary trees whose the nodes may have one child, two children or none. Finally in *Visitall*, the goal is to visit all the cells of a rectangular grid.

6.6.2 Computing FSCs and Hierarchical FSCs with classical planning

We show the results of our compilation in the introduced benchmarks for computing hierarchical controllers in Table 6.1.

					FD		BFWS	
Domain	$\lvert\mathfrak{C}\rvert$	**Kind**	$\lvert Q\rvert$	$\lvert\mathcal{P}\rvert$	**Time(s)**	$\lvert\pi\rvert$	**Time(s)**	$\lvert\pi\rvert$
A^nB^n	1	R	2	1	0.52	46	0.58	39
Blocks	1	OC	3	5	2.73	65	1.08	65
Fibonacci	2	HC	3,2	2,4	2.81,8.28	30,173	3.54,4.81	34,183
Gripper	1	OC	3	2	5.34	140	8.65	135
Hall-A	5	HC	2,2,2, 2,2	2,2,2, 2,2	13.27,5.60,19.09, 32.93,203.03	58,46,46, 46,195	29.64,204.88,284.91, 82.06,20.31	46,46,43, 46,189
List	1	OC	2	6	0.14	159	0.14	159
Reverse	1	OC	3	2	43.56	62	13.85	49
SBA	7	HC	1,1,1,1 1,1,2	2,2,2,2 4,4,8	0.20,0.31,0.44,0.55, 0.79,0.91,271.92	14,14,14,14, 38,38,267	0.07,0.13,0.01,0.01, 0.62,1.79,ME	14,14,14,14, 38,38,ME
T. Sum	1	OC	2	4	7.06	61	14.40	66
Tree/DFS	1	RP	3	4	921.08	422	ME	ME
Visitall	2	HC	2,2	3,2	0.40,25.23	84,314	7.01,ME	84,ME

Table 6.1: Number of controllers used, solution kind (OC=One Controller, HC=Hierarchical Controller, R=Recursivity, RP=Recursivity with Parameters), solution size and instances in $\mathcal{P}$. For each controller: planning time and plan length required for computing the controller.

In many domains our compilation computes a single FSC (OC = One Controller) that solves the input planning instances; there are two domains where the computed solution is a recursive controller (R = Recursivity; RP = Recursivity with Parameters) that are single FSCs that call themselves. For the rest of domains the solutions are hierarchical FSCs (HC = Hierarchical Controller) where

controllers call to other controllers. All solutions with $|\mathfrak{C}| > 1$ fall into this last category.

In addition to the *kind* and *size* of the computed FSCs we also report $|\mathcal{P}|$, the number of classical planning instances given as input to the compilation. Last but not least we report, for each domain, the planning *time* and *plan length* required by the classical planners FAST DOWNWARD and BEST-FIRST WIDTH SEARCH to compute the controllers. We report *Memory Exceeded* (ME) if the planner overflows the memory limit. We have no outcomes reporting timebound errors, because memory overflows before exceeding the time limit.

Overall the BEST-FIRST WIDTH SEARCH configuration does not require as much preprocessing time as FAST DOWNWARD. However, even though the novelty exploration implemented by BFWS expands and evaluates nodes very fast, in certain benchmarks requires a larger amount of memory than FD. Next we discuss the obtained solutions for each domain.

The $A^n B^n$ domain is solved with a recursive FSC without parameters. String parsing problems can be formalized as a generalized planning problem. Actions in this domain parse the current letter in the input string and progress the string iterator. If the current letter is still an a, the FSC makes a recursive call to itself else, it parses a b before terminating. Thus, it will process each letter a using n recursive calls, and a letter b before returning from each recursion, processing a total of n b's.

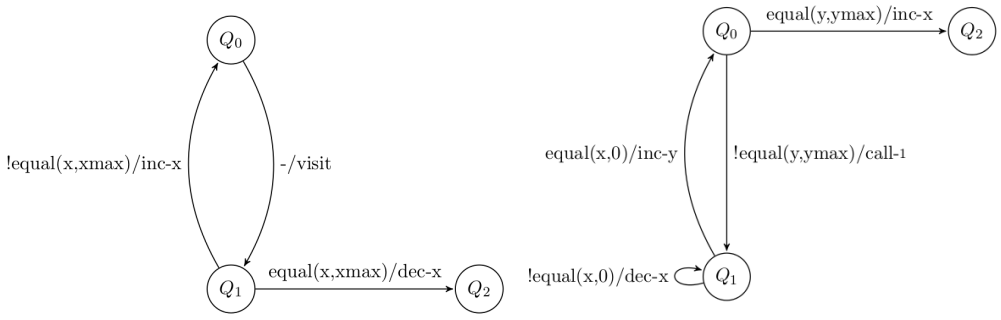
Compared to planning programs obtained in Chapter 4, the computed FSCs comprise a smaller number of controller states, reducing the time required to compute a generalized plan for the same tasks.

Results on the *Serial-Binary Adder* domain are reported to show how single FSCs, that solve a very specific problem like computing the bit sum and carry from two inputs and a carry, can be combined into a *hierarchical FSC* that iteratively calls previous FSCs to simulate the addition operation of two unbounded binary numbers.

The solution computed for the *Tree/DFS* domain corresponds to the FSC of Figure 6.2 encoding the condition $\mathsf{isNull}(n)$ as $\mathsf{equals}(n, n)$, where equals is a derived predicate that tests whether two variables have the same value. When applied to a leaf node n, the action $\mathsf{copyR}(n, n)$ deletes the current value of n without adding another value. Hence, evaluating $\mathsf{equals}(n, n)$ returns `false` when no node is assigned to variable n, i.e. $\mathsf{copyR}(n, n)$ or $\mathsf{copyL}(n, n)$ when n has no right or left child correspondingly.

In *Visitall* both, BEST-FIRST WIDTH SEARCH and FAST DOWNWARD, fail to generate a single FSC within the given time bound. Even if we attempt to generate a hierarchical FSC from scratch, these planners cannot find a solution within the given time bound. Instead, our approach is to generate a hierarchical FSC incrementally. We first generate a single FSC, Figure 6.5(a), that solves the

subproblem of visiting all cells in a single row. We then use our compilation to generate a second FSC, Figure 6.5(b), that iterates for each row and in every iteration, calls the first controller to visit the current row and then goes back to the first column until there are no more rows to visit.



(a) FSC-1, visits every cell in a row.

(b) FSC-2, for every row, calls FSC-1 and goes back to the first column.

Figure 6.5: Hierarchical FSC that visits all cells of a grid.

6.6.3 Assessing the influence of the input instances

Our compilation *programs* a controller and *validates* it on the set $\mathcal{P}$ of classical planning instances that is given as input. The next experiment evidences that the performance of this process is affected by the order in which the planning instances in $\mathcal{P}$ are given as input. This ordering has a positive impact in the performance when the first input instances forces the compilation to program an FSC that generalizes to the remaining instances. On the other hand, the ordering has a negative impact when the programmed FSCs overfit to the first input instances so the planner requires expensive backtracks to validate the remaining input instances.

Table 6.2 shows the performance of our compilation approach for all the possible orderings of the given input instances. For this analysis we considered only the domains that can be solved with a single FSC. For each domain, the Table 6.2 reports the number of instances and their possible orderings, factorial in the number of instances. Then for both classical planners, BFWS and FD, the table reports the minimum, maximum and average planning time (in seconds) computed for the given orderings. The last column reports the planning time given the input ordering reported in Table 6.1 to serve as a reference.

From these results we can conclude that there is a significant difference in the planning times (in some cases by orders of magnitude from *best* to *worst* case,

Domain	$\|\mathcal{P}\|$	Orderings	FD				BFWS			
			Min	Max	Avg	Ref	Min	Max	Avg	Ref
Gripper	2	2	3.12	4.88	4.00	4.88	8.12	34.86	21.49	8.12
Blocks	5	120	2.18	568.04	87.22	2.36	0.41	520.22	43.25	0.78
List	6	720	0.02	0.04	0.04	0.04	0.02	0.15	0.03	0.02
T. Sum	4	24	2.90	9.78	4.85	7.94	0.26	18.69	6.11	14.46
Reverse	2	2	42.0	51.34	46.67	42.00	16.16	86.59	51.38	16.16

Table 6.2: For each domain we report the number of instances and possible orderings. For each planner, the minimum, maximum and average times (in secs) for the orderings and the planning time given the ordering from Table 6.1.

like in the *Blocks* domain) that depends on the particular ordering of the input instances and the used planner. When a human specifies the input order (e.g. the results reported in Table 6.1) performance usually is below the average running time, but also depends on the used planning system. What can be a good ordering for one planner can result bad for the other planner, like in the *Gripper* domain.

6.7 Summary

Finite State Controllers (FSCs) for planning are similar to transducers or a *Mealy Machine*. In this chapter we show a model to compute *deterministic FSCs* with a novel definition that is related to planning programs (see Chapter 3). In contrast to previous approaches that synthesize FSCs (Bonet et al., 2010; Hu and De Giacomo, 2013), we include a technique for *breaking symmetries* that increase the synthesis performance.

Furthermore, we have proved that our approach to compute FSCs for planning are *sound* and *complete*. This compilation can be extended to generalized planning to solve many planning tasks with a single FSC. In addition, we can implement a *call stack* (see Section 4.2) which allows FSCs to call other FSCs or themselves in case of recursive solutions. These are the techniques required to describe one of the most important contributions in the thesis, *Hierarchical Finite State Controllers* (HFSCs). We also prove that HFSCs are sound and complete.

Then we compare FSCs and planning programs showing that for every planning program Π there is an equivalent FSC $\mathcal{C}$ and vice versa, but FSCs are at least as compact as planning programs, and often strictly more compact (see Section 6.5). In the evaluation we compute similar tasks with HFSCs as in planning programs (see Chapter 4) like DFS. Also we show that HFSCs solve recursively grammar parsing tasks like A^nB^n. Finally, we report time performance experiments on different ordering permutations in the set of input instances, concluding that the time impact when solving a set of instances sequentially is of orders of magnitude, leaving the automatic ordering of inputs as an open issue.

PART IV

New Landscapes for Planning

CHAPTER 7

Unsupervised Classification of Planning Instances

This Chapter 7 creates a new landscape for planning, where instances can be classified into labels such that we can predict their behavior. We start describing the meaning of an unsupervised classification task, how clusters can be computed and how to determine class labels to new input instances. After that, we formalize an unsupervised classification task for planning and show some results in the two different tasks of computing clusters and determining class labels. The aim of this chapter is to build bridges between Machine Learning and Planning through the connection of topics like unsupervised classification, behavior prediction and task generalization from a planning perspective.

7.1 Introduction

In Machine Learning (ML) there are two main classification approaches based on the input data and the way output is represented. The first one is *supervised learning*, where the input data has been classified into groups described by a set of features that corresponds to a desired output. Then, a cost function should converge to a minimum value such that the learnt model produce the best expected outcome over the input set. And the second one is *unsupervised learning*, where the input data is raw, thus there are no classes or features, and is the machine purpose to organize this information into different groups, even finding the features that describe them. For the feasibility of such a hard task, some kind of information must be provided to learn the model, like how many different classes we

can classify our input (clustering) and/or a distance metric among the input that should be minimized (kernel).

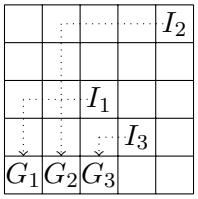
The outputs in ML are either integer values in the case of classification tasks or scalar values for regression tasks. Supervised learning can deal with both while unsupervised learning is used only for classification. Nevertheless, there is a new task called *structured prediction* (Bakir et al., 2007) where outputs represent more complex structural information about inputs. Classification and structured prediction are tightly related in terms of the approach to get the output, so one method to get solutions in raw inputs could be unsupervised learning.

This Chapter 7 is inspired in recent approaches like Lake et al. (2015) and Ellis et al. (2015) that synthesize *programs* in order to output structured information about inputs. The first one uses a Bayesian program learning framework that can generalize even from one example, and solutions are represented with probabilistic programs used for structured prediction of new inputs. Besides classification, they can generate examples from these programs which in many cases pass the "visual Turing tests" for world alphabets characters. The second one, provides an unsupervised program synthesis algorithm that given some hard constraints for the symbolic search and a specific grammar, it can enconde input images into a probabilistic program and generate images from this program. Programs are a natural knowledge representation for many domains, are easy to interpret by humans, and can compactly express relatively complex operations on inputs.

In ML we require some features and functions to find a structure or pattern to map inputs into outputs, but how is it possible to classify planning instances without a learning model?. Classical planning maps an initial state (input) into a goal state by applying a plan (output) that is composed of a sequence of actions, so it makes sense to classify planning instances according to *behaviors* rather than feature values. Using previous approaches based on structured prediction where solutions are expressed as programs, we can get the idea to automatically synthesize programs taking advantage of *planning programs* formalism introduced in Chapter 3.

Then, we used unsupervised classification of planning instances as a clustering problem where given a set of instances and a number of clusters, we can group instances into clusters that correspond to planning programs. Planning instances are assigned to a cluster if and only if the associated planning program maps their input states into a goal state for all of them. Thus, we can understand an *unsupervised classification task* as predicting the structured behavior of a planning instance.

As an illustrating example, consider the problem of navigating through a grid from an initial position to a goal position. Figure 7.1(a) represents three different planning instances of this type. These instances can all be represented using the same variables and actions: variables x and y that represent the cur-



(a)

```
0. dec(x)
1. goto(0,!(x = x_G))
2. dec(y)
3. goto(2,!(y = y_G))
4. end
```

(b)

Figure 7.1: (a) Three different instances of grid navigation; (b) a planning program Π_1 that solves all of them.

rent position, variables x_G and y_G that represent the goal position, and actions $\{\texttt{dec(x)}, \texttt{dec(y)}, \texttt{inc(x)}, \texttt{inc(y)}\}$ that decrement/increment the value of x or y. Although these three instances have different initial states and goals, they are solved by the planning program Π_1 in Figure 7.1(b): independently of the grid size x is decremented until reaching the goal column, and y is decremented until reaching the goal row. Note that we could solve these instances using three different planning programs, in which case they would no longer be considered to display the same behavior. This illustrates that we must have an idea of how many behaviors we want to capture in a domain. We understand behaviors as programs, and each instance from the set of input instances must be solved at least by one program. Whether we use a large bound of programs, they could overfit some tasks while using a low bound could make impossible finding a solution to the whole set of input instances.

The main contribution of this chapter is to show that classical planning is a good alternative for program synthesis, structured prediction and unsupervised methods. This alternative is represented using planning programs that are compiled into classical planning problems so they can be computed with an off-the-shelf classical planner. On the one hand, planning programs are more expressive than previous approaches. Ellis, Solar-Lezama, and Tenenbaum (2015), for example, only consider programs generated by an acyclic grammar, a restriction not shared by planning programs that allow conditional gotos and recursivity. On the other hand, planning programs cannot handle noisy inputs because they are *deterministic* while previous approaches are *probabilistic* and can deal with more realistic problems. Even though, there are many options to allow deterministic problems to deal with noisy inputs that we will comment later.

7.2 Classification of Planning Instances

The term classification means to assign a class, and if applied to planning instances is to assign a class to each instance. Thus we have an unlabelled set of planning problems (or instances without class) $\mathcal{E} = \{P_1, \ldots, P_T\}$. This set is equivalent to a generalized planning tasks from Chapter 2, where each planning problem $P_t \in \mathcal{E}$ with $1 \leq t \leq T$ is instantiated from a planning frame $\Phi = \langle F, A \rangle$ which fluents and actions are shared for all planning instances. We consider $\Gamma(\Phi)$ the function over the planning frame that yields to the set of all possible planning instances, so that each $P_t \in \Gamma(\Phi)$.

As we said, the set of unlabelled planning instances $\mathcal{E}$ is described in the same way of a generalized planning task $\mathcal{P}$, but in spite of getting a *single* generalized plan to solve all the instances, we cover all instances with *multiple* generalized plans such that each instance of $\mathcal{E}$ can be classified and solved by at least one of the generalized plans. Since we have multiple unlabelled instances that can be grouped according to some criterion into different generalized plans, we look upon this is an unsupervised classification task. In contrast, whether we know a priori the class label for each planning instance, the task would be transformed into a *supervised classification task* where $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2 \cup \ldots \cup \mathcal{E}_{T'}$ such that $T' \leq T, \forall(\mathcal{E}_i \subseteq \mathcal{E}) : (1 \leq i \leq T')$ and $\forall(\mathcal{E}_i \cap \mathcal{E}_j = \emptyset) : (1 \leq i, j \leq T'), (i \neq j)$, so $\mathcal{E}_i$ with $1 \leq i \leq T'$ represents the set of planning instances for i-th class that must be solved with a single generalized plan.

Because class labels for each one of the input instances are not given, we need an unsupervised method to solve the problem. A common approach for unsupervised classification is *clustering*, where instances are grouped into the same class (cluster) if they share some common *behavior* while the rest are classified in other classes (clusters).

This approach for unsupervised classification considers planning instances to be *similar* if their solution share some common structure, i.e. both can be solved using the same generalized plan. In other words, a generalized plan acts as a class prototype (Liu et al., 2009) if it solves all instances that belong to that class. For instance, Figure 7.1(b) shows the program Π_1 that solves the three planning instances illustrated in Figure 7.1(a) and represents the prototype for this cluster of different planning instances.

Then, we can formally describe an unsupervised classification task as a 3-tupple $\langle \Phi, \mathcal{E}, m \rangle$ where:

- $\Phi = \langle F, A \rangle$ is the classical planning frame.
- $\mathcal{E} = \{P_1, \ldots, P_T\}$ is the finite set of unlabelled planning instances drawn from Φ, i.e. $P_t \in \Gamma(\Phi)$, $1 \leq t \leq T$.

- And m is the number of clusters, that implicitly defines a set of class labels $C_L = \{c_1, \ldots, c_m\}$ that correspond to generalized plans $\{\Pi_1, \ldots, \Pi_m\}$ respectively.

After that, we introduce a *model* for the unsupervised classification task $\langle \Phi, \mathcal{E}, m \rangle$ defined by the 2-tupple $\langle \mathcal{G}, cf \rangle$ where:

- $\mathcal{G} = \{\Pi_1, \ldots, \Pi_m\}$ is the set of generalized plans, one for each class label, where all unlabelled instances can be classified and solved.
- $cf : \mathcal{E} \rightarrow C_L$ is a non-deterministic choice function that assigns a class label $cf(P_t) \in C_L$ to each input instance $P_t \in \mathcal{E}$ such that the corresponding generalized pla solves P_t.

The model $\langle \mathcal{G}, cf \rangle$ is *valid* if and only if each input instance $P_t \in \mathcal{E}$ can be classified into a class label $c_j \in C_L$, with the non-deterministic choice function $cf(P_t) = c_j$, whose corresponding generalized plan Π_j solves the instance P_t. Another option that is not used in this approach but can describe when a model is valid, is to check there are no misclassifications for the input instances in $\mathcal{E}$. This does not apply in our case, because we want to get models that can classify all instances, independently whether they are in the expected class or not, as long as the generalized plans can solve them correctly.

Regarding with cf function, it implicitly defines a *partition* of the input instances into each one of the clusters, in such a way that each planning instance is mapped into a cluster, and the group of instances that are in the same cluster are equivalent to a generalized planning problem $\mathcal{P}_j$, $1 \leq j \leq m$ whose solution is given by the generalized plan Π_j.

Once the model for unsupervised classification is introduced, we can define the two tasks for a set of planning instances:

1. Given an unsupervised classification task $\langle \Phi, \mathcal{E}, m \rangle$, compute a valid model $\langle \mathcal{G}, cf \rangle$ where all planning instances in E are classified and solved.
2. Given an unsupervised classification task $\langle \Phi, \mathcal{E}, m \rangle$ and a valid model $\langle \mathcal{G}, cf \rangle$, classify a planning instance $P \in \Gamma(\Phi) \setminus \mathcal{E}$ (i.e. instantiated from Φ but not included among the inputs) by determining its class label $c_j \in C_L$ that corresponds to a generalized plan Π_j.

This second task can also be understood as plan recognition (Ramírez and Geffner, 2010) using plan libraries in the form of generalized plans and where inputs are not sequences of observations but pairs of initial state and goal condition. Following this section we are going to explain with more detail how to carry out both tasks.

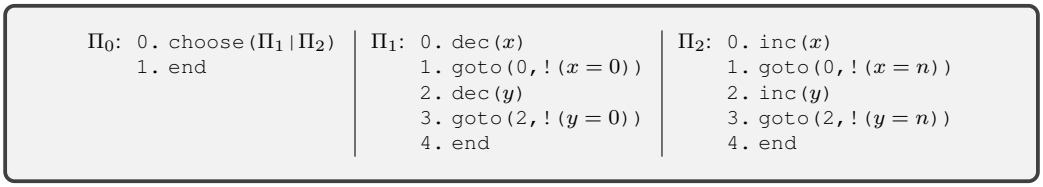


Figure 7.2: An unsupervised classification model with two clusters of planning instances represented as a planning program with a choice instruction $choose(\Pi_1|\Pi_2)$.

7.2.1 Computing Classification Models

In this section we explain how to compute a valid model $\langle \mathcal{G}, cf \rangle$ given an unsupervised classification task $\langle \Phi, \mathcal{E}, m \rangle$.

The new model is an extension of previous planning programs from Chapter 3. There are two main extensions, one is related to the set of planning programs $\mathcal{G}$, while the other is related to the non-deterministic choice function cf that maps input instances to class labels.

The first extension is to allow the execution of the cf function adding a *choice instruction* to the original set of instructions $\mathcal{I}$. The *choice instruction* is defined as $choose(\Pi_1, \ldots, \Pi_m)$ where each Π_j with $1 \leq j \leq m$ is a planning program in $\mathcal{G}$.

The second extension is a planning program that only contains the *choice instruction* called Π_0 (cf. Figure 7.2). From Π_0, with the non-deterministic function encoded in the choice instruction, we can choose which planning program among $\Pi_1, \ldots, \Pi_m$ is going to be executed and programmed in case of empty lines. Thus, *choosing* planning programs is akin to *call instructions* where planning programs can call to other planning programs introduced in Chapter 4, but in this case only Π_0 can call other programs in the execution of the choice instruction.

In Figure 7.2 there is an unsupervised classification model with two clusters or class labels that correspond to planning programs Π_1 and Π_2. The planning problem is represented in a $n \times n$ grid where an agent can move to any arbitrary position, but the planning programs act as prototypes for moving bottom-left corner and top-right corner respectively, so given an instance of the planning problem, a classical planner must decide to which cluster the instance belongs and validate the assignment. In order to generalize we use to include fluents to express the minimum and maximum values (e.g. x_{max}, y_{max}), and sometimes operators like equals, less than, greater than and so on.

The **execution model** of a planning program behaves in the same way as before for all instructions and conditions like *termination, success* and *failure*, the only difference is the extension of the choice instruction *choose*$(\Pi_1| \ldots |\Pi_m)$ that

must define how affects to the program state (s, i, j) where planning state s is on line i of planning program j.

Once the program state is in the choice instruction $w_i = \textit{choose}(\Pi_1|\ldots|\Pi_m)$ it must *choose* which planning program among $\Pi_1, \ldots, \Pi_m$ must be programmed and executed, setting the new program state to $(s, 0, j)$ that means the planning state remains the same while the program counter moves from $(s, 0, 0)$, first line of Π_0, to the first line of Π_j, the rest is equivalent to a planning program execution.

The new **compilation** requires as input an unsupervised classification task $\langle \Phi, \mathcal{E}, m \rangle$ and a number of lines n, and outputs a classical planning instance with conditional effects $P_{n,m} = \langle F_{n,m}, A_{n,m}, I_{n,m}, G_{n,m} \rangle$.

- $F_{n,m}$ is the fluent set that extends F_n with $m+1$ planning programs $\Pi_0, \Pi_1, \ldots, \Pi_m$ with n lines. So we make a *copy* of f_j of each fluent $f \in F_{pc} \cup F_{ins}$ for each planning program Π_j where $0 \leq j \leq m$.

- $A_{n,m}$ is the action set that extends A_n in different ways:

  - Every $a_i \in A_n$ that simulates an instruction in a specific line, is copied as $a_{i,j}$ for each planning program Π_j where $0 \leq j \leq m$. Then, for every $a_{i,j}$ we include two actions $\mathsf{P}(a_{i,j})$ and $\mathsf{E}(a_{i,j})$ for programming an instruction and executing a planning action $a_{i,j}$ respectively.
  - Also terminal actions are modified to adapt its execution for multiple planning programs. So, action $\mathsf{end}_{t,i,j}$ where $t < T$ resets the program state to $(I_{t+1}, 0, 0)$ that represents the initial state of instance P_{t+1} starting from line 0 in the planning program Π_0.
  - The last extension are the m choose actions $choose_j$ with $1 \leq j \leq m$. It selects one of the planning programs among $\Pi_1, \ldots, \Pi_m$ and is defined as:

$$\mathsf{pre}(\mathsf{choose}_j) = \{\mathsf{pc}_{0,0}\},$$
$$\mathsf{cond}(\mathsf{choose}_j) = \{\neg\mathsf{pc}_{0,0}, \mathsf{pc}_{0,j}\}.$$

- $I_{n,m}$ is the initial state and it is defined as I_n but including all empty lines for planning programs $\Pi_1, \ldots, \Pi_m$ and a choice instruction $choose(\Pi_1|\ldots|\Pi_m)$ in the first line of Π_0. The program state is initialized to $(I_1, 0, 0)$, i.e. the initial state of the input instance P_1 and line 0 of the planning program Π_0.

- $G_{n,m}$ is the goal condition and is defined with the fluent $\{\mathsf{done}\}$ as before.

Lemma 7.1. *Any plan π that solves $P_{n,m}$ corresponds to a valid model $\langle \mathcal{G}, cf \rangle$ for the unsupervised classification task $\langle \Phi, \mathcal{E}, m \rangle$.*

Proof. To solve $P_{n,m}$ the plan π has to simulate the choice instruction for each input instance P_t, $1 \leq t \leq T$, actively choosing a planning program among $\Pi_1, \ldots, \Pi_m$. This active choice corresponds exactly to the mapping cf from input instances to class labels. In addition, for each input P_t, π has to program the instructions (if not yet programmed) of the chosen planning program Π_j, $1 \leq j \leq m$, and then simulate the execution of Π_j on P_t. For π to solve $P_{n,m}$, this simulated execution has to successfully validate that Π_j solves P_t. Hence $\mathcal{G} = \{\Pi_1, \ldots, \Pi_m\}$ satisfies the property required for the model $\langle \mathcal{G}, cf \rangle$ to be valid. □

7.2.2 Determining Class Labels

A second task is, given a valid model $\langle \mathcal{G}, cf \rangle$ for the unsupervised classification task $\langle \Phi, \mathcal{E}, m \rangle$, to determine the class label of a classical planning instance $P = \langle F, A, I, G \rangle \in \Gamma(\Phi) \setminus \mathcal{E}$ choosing the planning program Π_j with $1 \leq j \leq m$ whose execution solves P. This instance is instantiated from the planning frame Φ and does not belong to the original set of unlabelled planning instances $\mathcal{E}$.

Once the model is given, planning programs are like policies where planners do not need to make decisions, they just check the preconditions and apply the effects of the actions. Nevertheless, Π_0 is the only that has a non-deterministic choice function cf, so given the model, this is the only decision for the planners that is to choose the planning program to execute. Once a planing program $\Pi_j \in \mathcal{G}$ is chosen, Π_j is executed so that we verify if it solves P. In case we do not solve P, the planner tries the next best option, otherwise we classify P with the class label $c_j \in C_L$.

In Figure 7.3 there is an example of an unsupervised classification task in a grid domain of size 5×5. The tasks to perform is moving bottom-left or top-right corners, so the model is the same as in Figure 7.2. Now consider $\mathcal{E} = \{P_1, P_2, P_3, P_4\}$ so they were used to compute the model, and we must determine the class $c_j \in C_L$ with $1 \leq j \leq m$ of a new instance P_5. In case the goal is to move bottom-left corner, $cf(P_5) = c_1$, otherwise $cf(P_5) = c_2$.

There are two main unsolved problems regarding with P classification:

1. There are multiple planning programs $\Pi_j \in \mathcal{G}$ that solve P.
2. There is no planning program $\Pi_j \in \mathcal{G}$ that applied on P reaches a goal condition G.

For the first case, the choosen planning program could be the one with the lowest cost and in case of a tie the planner would decide arbitrary any of the optimal planning programs. We can predict the structure of planning problem P, so any decision of a planning program that solves P can be considered correct but is still an open question. In the second case is more complex, there is nothing

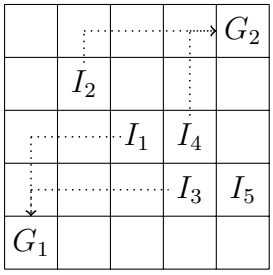


Figure 7.3: Unsupervised Classification Task in a Grid domain of 5×5 with an unlabelled planning instance P_5.

specified for those cases so one possible solution can be to choose the planning program that has the *closest* expanded state s to a goal condition G. One option for this measure is considering the planning program Π_j in the planning program state (s, i, j) with the largest set of fluents in the intersection $s \cap G$. In other words, consider $\mathcal{X}(\Pi(P))$ the set of expanded states of planning program Π executed on a planning instance P, and $\delta(s)$ a function that counts the fluents assigned to true in s. Then we can choose the planning program that minimizes the distance to a goal condition like $argmin_{\Pi_j} \delta(s \cap G) : s \in \mathcal{X}(\Pi_j(P)) \wedge \Pi_j \in \mathcal{G}$.

The field of Machine Learning has a variety of approaches for error minimization and improving classification accuracy of input data. In our case, determining class labels can be a first step in planning to connect with machine learning approaches for error detection and to improve the quality of planning programs.

7.3 Unsupervised Classification as Planning

Supervised classification can be modeled as a generalized planning task (Lotinac et al., 2016), where the set of noise-free input instances are described with features and every instance has been assigned to a label. Then, a solution to this generalized planning task would be a structured program that correctly classifies every instance into the corresponding label. These programs must synthesize the conditions with form of *conjunctive queries* over the set of features, that after evaluated to true or false determine to which class an instance belongs. The nesting of conditions in programs can be seen as decision trees for noise-free data that is close to contingent plans (Albore et al., 2009). In this work, they produce a decision tree as a contingent plan, where internal nodes are sensing actions and leafs are the classical actions to perform in the world. In other words, internal nodes can be considered as conditions while leafs are the actions to determine a label or class.

In this chapter we show that outputs from unsupervised classification as plan-

ning can have the form of a decision tree. The main difference is that instances are not labeled and it is the job of a planner to find the mapping of inputs into structured outputs. These outputs are clusters that correspond to planning programs. The aim of our approach is not to be competitive with state-of-the-art ML techniques when solving unsupervised classification tasks but to go deep in the connection of automated planning and ML models.

Definition 7.1 (Unsupervised classification Task)**.** *An Unsupervised Classification Task is the task of assigning a class label to each unlabeled input example from* $\mathcal{E} = \{e_1, \cdots, e_T\}$*, given a set of clusters* $C_L = \{c_1, \cdots, c_m\}$.

Then, each input e_t, $1 \leq t \leq T$ is an assignment of values to variables, where variables are in a finite set $X = \{x_1, \cdots, x_\iota\}$, and each variable x_i with $1 \leq i \leq \iota$ has an associated finite domain $D(x_i)$. Only one possible value can be assigned to a variable, considering the domain of the variable as an invariant of the task. Thus, the unsupervised classification task can be modeled as a generalized planning task $\mathcal{P} = \{P_1, \cdots, P_T\}$ such that each individual planning task $P_t = \langle F, A, I_t, G_t \rangle$ with $1 \leq t \leq T$, models the classification of the t^{th} example:

- F is the set of fluents to represent the classical planning instances, and some others to control if an instance has been already classified by assigning a class label.
- A contains the set of actions necessary to label a given example with a class label in C_L. For instance, in a two-class unsupervised classification, $C_L = \{c_1, c_2\}$ correspond to fluents $\mathsf{class_1}$ and $\mathsf{class_2}$, while $A = \{\mathsf{setClass_1}, \mathsf{setClass_2}\}$ is the set of actions that set these fluents to true and another one called classified to prevent classifying the same instances in multiple classes. Formally, action $\mathsf{setClass_c}$ is defined as follows:

$$\begin{aligned} \mathsf{pre}(\mathsf{setClass_c}) &= \{\neg\mathsf{classified}\}, \\ \mathsf{cond}(\mathsf{setClass_c}) &= \{\emptyset \rhd \{\mathsf{classified}, \mathsf{class_c}\}\}. \end{aligned}$$

- I_t contains the fluents that describe the t^{th} example and $\{\neg\mathsf{classified}\}$ while $G_t = \{\mathsf{classified}\}$ requires that the example has been labeled.

The generalized planning problem $\mathcal{P} = \{P_1, \cdots, P_T\}$ models an unsupervised classification task of T instances with a solution Π that maps every input instance into a class from cluster $C_L = \{c_1, \cdots, c_m\}$. The solution has a constraint that every class must be used at least once so $m \leq T$. As shown in previous chapters, this generalized planning task can be compiled into classical planning where Π has the form of a planning program that has to correctly assign classes to

```
Π0: 0. goto(3,!(y = ¬x1 ∧ x2))
    1. setClass1
    2. end
    3. setClass2
    4. end
```

Figure 7.4: Example planning program for the two-cluster classification task of learning examples that correspond to the logic formulae $y = x_1 \rightarrow x_2$ and $y = x_2 \rightarrow x_1$.

unlabeled input instances. For instance, in a two-class unsupervised classification task, fluents class_1 and class_2 are also goals of the planning instance resulting from the compilation to classical planning.

Modelling a ML task as classical planning is straightforward if it is described using *logic*, like *Michalski's train* classification task (Michalski et al., 2013). Although, the features that describe each example may have finite domains, so fluents F in the classical planning task are instantiated from the set of predicates Ψ and objects Ω. The objects set is splitted in two parts, one for variable objects Ω_v, and the other one for value objects Ω_x. Then, given a variable $v \in \Omega_v$ and a value $x \in \Omega_x$, the predicate $\mathsf{assign}(v, x)$ is true iff variable v has value x. A given variable represents exactly one value at a time, so for a given v, fluents $\mathsf{assign}(v, x)$, such that $x \in \Omega_x$, are *invariants*.

We show here a simple unsupervised classification task that can be modeled in this way. The task is a two-cluster classification of examples that correspond to the different value combinations of three Boolean variables, x_1, x_2 and y. More precisely, there is a total of eight examples, four corresponding to the variable values of the formula $y = x_1 \rightarrow x_2$ (the first cluster) and the other four corresponding to the formula $y = x_2 \rightarrow x_1$ (the second cluster). Figure 7.4 shows a program Π that solves this unsupervised classification task.

Even though there are different ways of clustering the examples, a classical planner would prefer compact cluster models, like the one shown in Figure 7.4, i.e. that requires a small number of program lines. The fluent $y = \neg x_1 \wedge x_2$ that appears in the goto instruction (line 0) represents a *derived fluent* that holds in all the states for which the encoded formula is satisfied.

To synthesize the planning program in Figure 7.4 starting from scratch we need to automatically discover the *condition* $y = \neg x_1 \wedge x_2$ that determines how the input examples should be classified. In this case the unsupervised learning task is to identify the derived fluents that allow a compact classification model. This task can be addressed by extending the compilation for solving generalized planning

tasks with a unification mechanism (Lotinac et al., 2016). Note that this is a more traditional ML task than the structured prediction addressed in previous sections, which classifies examples according to *behavior* rather than their features.

7.4 Evaluation

The setting to run the experiments are the same explained in Chapter 3 but using an extended version of the original compilation. In this section we show two kinds of experiments, ones to solve an unsupervised planning task computing a valid model, and others to determining a planning program that solves an unlabelled planning instance given a valid model.

1. Compute a valid classification model $\langle \mathcal{G}, cf \rangle$ given an unsupervised classification task $\langle \Phi, \mathcal{E}, m \rangle$.

2. Classify a given planning instance P given an existing model $\langle \mathcal{G}, cf \rangle$ by determining the planning program in $\mathcal{G}$ that solves P.

		Insts	n	m	F	A	**Search**	**Prep**	**Total**
Grid	H-V	4	2	4	284	292	0.04	1.23	1.27
	Quadrant	4	4	2	356	634	5.77	1.04	6.81
List	Visit	4	4	2	362	762	0.20	0.51	0.71
Bool	Assign	8	2	2	220	194	0.17	0.38	0.55
	Nor-Nand	8	3	2	326	330	0.79	0.56	1.35

Table 7.1: Number of learning instances; bounds on the number of lines per cluster and clusters; number of fluents and actions in the compiled classical planning task; search, preprocessing and total time (in seconds) elapsed while computing the solution.

7.4.1 Benchmarks

The instances used in experiments come from three generic domains (i.e. planning frames). In these generic domains, all planning instances share the planning frame $\Phi = \langle F, A \rangle$. This means that we can first create clusters for the planning instances in $\mathcal{E}$, compute the planning program that provides the prototype behavior of each cluster, and then test multiple planning instances for each task to see how they are classified.

In the first domain, **Grid**, the goal is to navigate to a goal position by incrementing or decrementing the x or y values. The domain also includes fluents that represent the goal position (cf. the example in Figure 7.1). The second domain, **List**, models lists of integers, in which actions iterate over list elements or apply an operation to the current list element. In the third domain, **Boolean Circuits**, actions consist of applying Boolean functions such as *and*, *not*, and *or* with conditional effects, using Boolean variables $\{x_1, x_2, y\}$ to create the circuits.

For the **Grid** domain we created two different classification tasks: **H-V** that comprise horizontal and vertical navigation tasks and **Quadrant** where navigation is done towards the top-right or bottom-left quadrant (cf. Figure 7.3).

In the **List** domain we tested a **Visit** task whose classes are to perform operations (i.e. visit) on all the list elements or only on every second element (odd positions in the list). For the **Boolean Circuits** domain we implemented two tasks: **Assign**, in which the aim is to perform either of these two operations $x_1 \leftarrow x_2$ and $x_2 \leftarrow x_1$; and **NOR-NAND**, in which the aim is to correctly create and classify *nor* and *nand* circuits.

7.4.2 Computing Classification Models

Table 7.1 summarizes the results of the first kind of experiments. In this table we provide the number of input instances, the bounded number of lines for each planning program, and the number of clusters (each corresponding to a planning program). Also we report some data of the planning compilation like the number of fluents and actions the planner has to handle, and the times required for preprocessing and search.

The solutions obtained in **H-V** were four clusters, of two program lines each, corresponding to decreasing or increasing the x or y variables in order to reach the target position along a horizontal or vertical line.

$$\Pi_1 : 0.inc(x), 1.goto(0, !(x = x_G)),$$
$$\Pi_2 : 0.dec(y), 1.goto(0, !(y = y_G)),$$
$$\Pi_3 : 0.dec(x), 1.goto(0, !(x = x_G)),$$
$$\Pi_4 : 0.inc(y), 1.goto(0, !(y = y_G)).$$

Regarding the **Quadrant** task, the two planning programs have to navigate to a specific target position in one of the two quadrants (top-right or bottom-left). Both planning programs have four lines to solve the planning instances.

$$\Pi_1 : 0.inc(x), 1.goto(0, !(x = x_G)), 2.inc(y), 3.goto(0, !(y = y_G)),$$
$$\Pi_2 : 0.dec(x), 1.goto(0, !(x = x_G)), 2.dec(y), 3.goto(0, !(y = y_G)).$$

In the task for the **List** domain, the first program Π_1 visits all elements of the list, while the second program Π_2 only visits every second element by applying the `next` action twice in each iteration.

$$\Pi_1 : 0.visit(n), 1.next(n), 2.goto(0, !(n = nil)),$$
$$\Pi_2 : 0.visit(n), 1.next(n), 2.next(n), 3.goto(0, !(n = nil)).$$

In the **Boolean Circuits** domain, the input is always the whole set of Boolean variable assignments corresponding to a given Boolean circuit. In this case, instead of obtaining the expected Boolean functions, the planner finds sequences of actions that set the outcome to true or false. Thus it classifies input instances depending on the value of the variable in the goal condition (true or false). The prototype planning programs for clusters in **Assign** are

$$\Pi_1 : 0.not(x_1), 1.and(x_1, x_2),$$
$$\Pi_2 : 0.not(x_2), 1.or(x_2, x_1),$$

while in **NOR-NAND** they are

$$\Pi_1 : 0.not(y), 1.or(x_1, y),$$
$$\Pi_2 : 0.or(x_1, x_2), 1.not(x_1).$$

The *not(var)* function directly modifies the var value while *or(var_1,var_2)* and *and(var_1,var_2)* assign the result of the boolean operation to the left variable var_1. The Boolean functions computed for the **Assign** task appear in Table 7.2, and those for **NOR-NAND** in Table 7.3. Their input instances are the four possible combinations of two boolean variables x_1 and x_2 for two program classifiers (eight instances in total), and the goals are to assign specific values to x_1, x_2 and/or y. The planner classifications are in bold in the tables (e.g. **Assign** input $x_1 = 0$ and $x_2 = 1$ for task $x_2 \leftarrow x_1$, is classified into Π_2 such that x_2 becomes 0). In order to avoid planners to just assign a value to a variable, like **NOR-NAND** domain that requires a more complex strategy, the resulting values have to be assigned to x_1 and y.

Assign												
Input		**Output** Π_1		**Output** Π_2		**Goal** $x_1 \rightarrow x_2$			**Goal** $x_2 \rightarrow x_1$		
x_1	x_2	x_1	x_2	x_1	x_2	x_1	x_2	c	x_1	x_2	c
0	0	0	0	0	1	0	0	**1**	0	0	**1**
0	1	1	1	0	0	0	0	**1**	1	1	**2**
1	0	0	0	1	1	1	1	**2**	0	0	**1**
1	1	0	1	1	1	1	1	**2**	1	1	**2**

Table 7.2: Boolean results for **Assign** task. The columns are the initial state; the output of Π_1 and Π_2 from each initial state; the goals for both tasks and c is the class label assigned by the planner that corresponds to planning program Π_c that solves the planning problem.

NOR-NAND																
Input			**Output** Π_1			**Output** Π_2			**Goal NOR**				**Goal NAND**			
x_1	x_2	y	x_1	x_2	y	x_1	x_2	y	x_1	x_2	y	c	x_1	x_2	y	c
0	0	0	1	0	1	1	0	0	1	0	1	**1**	1	0	1	**1**
0	1	0	1	1	1	0	1	0	0	1	0	**2**	1	1	1	**1**
1	0	0	1	0	1	0	0	0	0	0	0	**2**	1	1	1	**1**
1	1	0	1	1	1	0	1	0	0	1	0	**2**	0	1	0	**2**

Table 7.3: Boolean results for **NOR-NAND** task. The columns are the initial state; the output of Π_1 and Π_2 from each initial state; the goals for both tasks and c is the class label assigned by the planner that corresponds to planning program Π_c that solves the planning problem.

7.4.3 Determining Class Labels

For the second kind of experiments we create additional instances for the tasks **H-V**, **Quadrant** and **Visit**. We were unable to test the **Boolean Circuits** domain because the previous inputs already included all possible variable assignments corresponding to a given Boolean circuit. In addition, we test a new generic domain called **Pointers** in which the three possible clusters have to perform find, select and reverse tasks on lists. The *find* task counts the number of occurrences of a given element in a list, the *select* task searches for the minimum element in the list, and the *reverse* task reverses the order of the elements in the list.

Table 7.4 displays the same features as in Table 7.1, but the planning programs are already programmed for each cluster, so the instructions are included as fluents in the initial state of the compiled classical planning instance and the set of instances are included as tests in the domain (correctly classified / total number of instances). The idea is to check the outcome of noise-free classification for more complex problems, instead of dedicating resources to the search of the planning

Domain	Task	Tests	n	m	Facts	Oper	Search	Prep	Total
Grid	H-V	8/8	2	4	180	52	0.06	10.31	10.37
	Quadrant	6/6	4	2	102	30	0.02	0.98	1.00
List	Visit	8/8	4	2	122	30	0.04	1.60	1.64
Pointers	FRS	7/7	4	3	175	46	0.43	23.42	23.85

Table 7.4: Number of tests (Correctly Classified/Total); bounds on lines per cluster and clusters; number of facts and operators; search, preprocessing and total time (in seconds) elapsed while computing the solution.

programs themselves, a costly operation due to the exponential complexity in the bound on the number of lines.

All the tests in the second table have been classified correctly using the provided knowledge in the form of existing planning programs. Nevertheless, we can find some extreme cases where planning programs with different structures can come up to the same result given an instance. In these situations, the classical planner can classify the instance in an arbitrary way, using by default the plan length, since its behavior is consistent with any of the clusters.

7.5 Summary

In this chapter we have formalized the concept of unsupervised classification tasks using classical planning. These kind of tasks follow a prototype-based clustering where different planning instances are grouped into clusters that share some common structure. To do so we have extended previous compilations for generalized planning tasks (see Chapter 3) to perform unsupervised classification of planning instances using an off-the-shelf classical planner.

As previous approaches, we assume some bounds are given like the maximum number of lines n for a planning program and the number of clusters m in the unsupervised planning task. In case these bounds are low, the planner would not be able to find a solution either because it needs more number of lines to find the program for a specific cluster or it detects more patterns over the input instances than the current number of clusters. In the other cases where bounds are high, it could affect in two different ways, either in search perfomance where the program space is huge or overfitting over input instances. We understand overfitting for planning programs as finding wrong patterns over conditions that do not generalize, or overfitting for unsupervised classification tasks as finding more clusters than expected with the same set of instances, being easier to fail on classifying a new instance.

Generalized planning in this chapter is strongly connected to unsupervised

learning. We can understand all the set of reachable states from a planning search as examples, and the action or actions we can apply in each state as the labeling process. In other words, it is the relation of assigning actions to reachable states. There are bottom-up approaches where the generalized plan is induced from classical plans, in these cases we consider classical plans as labels and generalized planning would correspond to a supervised learning approach (De la Rosa et al., 2011).

This work is capable to compute models for noise-free unsupervised classification tasks and for that reason can not compete with state-of-art ML approaches that can deal with noisy and continuous data (e.g. 2D clustering). So, we do not find a solution until all examples are classified into a cluster and solved, even though it does not mean we can misclassify some examples due to an ambiguous structure. However, automated planning and ML can take profit of each other with this work because ML can bring new benchmarks to planning while planning can push the scope of ML techniques in order to find more structured outputs.

There are different ways where this work can be pushed but one of the biggest issues is the complexity of planning in the number of lines. One possible direction is to find programs based on some criterion. For that we can use previous approaches like program by sketching (Solar-Lezama, 2008) or generate programs from acyclic grammars (Ellis et al., 2015) that are used to synthesize providing partial information or restrictions for structure prediction. A similar work on restricting the search space that can be used in the compilation as Domain Control Knowledge for programs (Baier et al., 2007).

CHAPTER 8

Generating Context-Free Grammars using Classical Planning

In this Chapter 8 we study the syntactic representation of languages in the field of formal languages. Specifically we focus in context-free grammars that are composed of symbols and production rules. This set of rules can be expressed as an action scheme in a planning frame. Thus, context-free grammars can be parsed with a common planning frame when production rules are shared. Once context-free grammars share a common structure they can be computed as a generalized planning problem like in Chapter 3 with the requirement of a choice instruction similar to the one introduced in Chapter 7.

We formalize a context-free grammar as a generalized planning problem. Then we can solve parsing and producing tasks, where the first one correctly validates an input string in the set of rules of a context-free grammar, and the second one generates the set of grammar rules given the most basic producing rules, e.g. producing characters a, b and so on.

8.1 Introduction

A *formal language* $\mathcal{F}$ is often defined as a *formal grammar* which is composed of an **alphabet** and a **set of rules**. This set of rules describe the structure of the language and the words that can be generated.

Definition 8.1 (Alphabet)**.** *An Alphabet $\mathcal{T}$ is a set of terminal symbols or letters in the context of a formal grammar. For instance, any binary number can be represented with an alphabet $\mathcal{T} = \{0, 1\}$.*

Definition 8.2 (Expressions)**.** *An Expression is a (possibly infinite) set of words that can be generated with the alphabet $\mathcal{T}$, and the syntactic rules $\mathcal{R}$ of a formal grammar. Non-terminal symbols V are expressions in themselves that can contain terminal and non-terminal symbols.*

Thus, formal grammars are defined by an alphabet, a set of non-terminal symbols and expressions that represent the grammar rules from which we derive two tasks:

- *Production*: Given a formal grammar $\mathcal{F}$, generate strings that belong to the language represented by the grammar.
- *Parsing* (also known as *recognition*): Given a formal grammar $\mathcal{F}$ and a string, determine whether the string belongs to the language represented by the grammar.

Chomsky defined four classes of formal grammars that differ in the form and generative capacity of their rules (Chomsky, 2002). In this chapter we focus on *Context-Free Grammars* (CFGs), where the left-hand side of a grammar rule is always a single non-terminal symbol, and the right-hand side is a set of expressions. The set of words that can be generated with a CFG never contain non-terminal symbols, so all strings that corresponds to the language are described with terminal symbols from the alphabet $\mathcal{T}$.

To illustrate this Figure 8.1(a) shows an example CFG that contains a single non-terminal symbol, S, and three terminal symbols (a, b and ϵ, where ϵ denotes the empty string). This CFG defines three production rules that can generate, for instance, the string $aabbaa$ by applying the first rule twice, then the second rule once and finally, the third rule once again. The *parse tree* in Figure 8.1(b) exemplifies the previous rule application and proves that the string $aabbaa$ belongs to the language defined by the grammar.

Learning the entire class of CFGs using only positive examples, i.e. strings that belong to the corresponding formal language, is not identifiable in the limit (Gold, 1967). In this chapter we address the generation of CFGs from positive examples but: (1) for a bounded maximum number of non-terminal symbols in the grammar and (2), a bounded maximum size of the rules in the grammar (i.e. a maximum number of expressions in the right-hand side of the grammar rules).

Our approach is compiling this inductive learning task into a classical planning task whose solutions are sequences of actions that build and validate a CFG compliant with the input strings. The reported empirical results show that our compilation can generate CFGs from small amounts of input data (even a single input string in some cases) using an off-the-shelf classical planner. In addition, we show that the compilation is also suitable for implementing the two canonical tasks of CFGs, *string production* and *string recognition*.

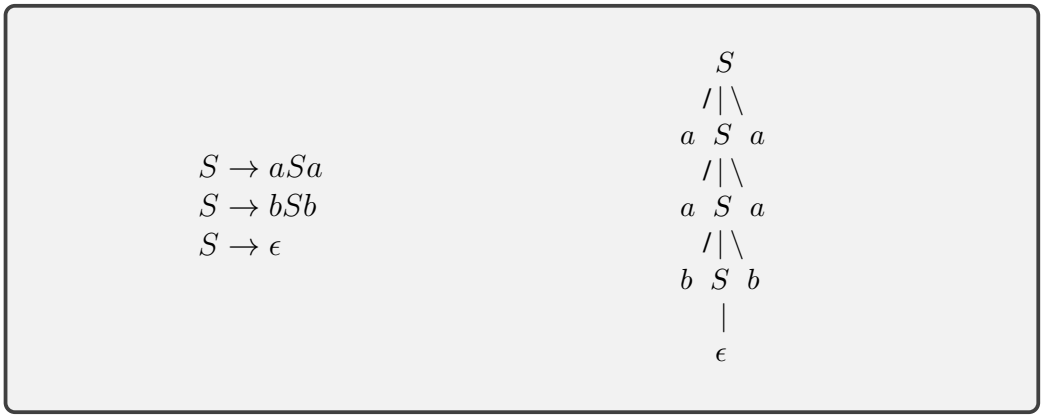


Figure 8.1: (a) Example of a context-free grammar; (b) the corresponding *parse tree* for the string $aabbaa$.

8.2 Context-Free Gramars

A context-free grammar can be defined as a formal grammar with a tuple $\mathcal{F} = \langle V, v_0, \mathcal{T}, \mathcal{R} \rangle$ where:

- V is the finite set of non-terminal symbols, also called variables. Each $v \in V$ represents a sub-language of the language defined by the grammar.
- $v_0 \in V$ is the initial non-terminal symbol that represents the whole grammar.
- $\mathcal{T}$ is the finite set of terminal symbols, which are disjoint from the set of non-terminal symbols, i.e. $V \cap \mathcal{T} \neq \emptyset$. The set of terminal symbols is the alphabet of the language defined by $\mathcal{F}$ and can contain the empty string, which we denote by $\epsilon \in \mathcal{T}$.
- $\mathcal{R} : V \to (V \cup \mathcal{T})^*$ is the finite set of production rules in the grammar. By definition rules $\zeta \in \mathcal{R}$ always contain a single non-terminal symbol on the left-hand side.

The example shown in Figure 8.1(a) is a 1-variable CFG since it defines a single non-terminal symbol. Therefore, the formal language described in this example has only one non-terminal symbol $V = \{S\}$ that is also the initial non-terminal symbol of the CFG $v_0 = S$. The alphabet is $\mathcal{T} = \{a, b, \epsilon\}$ which means that contains the empty string ϵ, and the two letters a and b, and the set of rules is $\mathcal{R} = \{S \to aSa, S \to bSb, S \to \epsilon\}$.

For any two strings $e_1, e_2 \in (V \cup \mathcal{A})^*$ we say that e_1 *directly yields* e_2, denoted by $e_1 \Rightarrow e_2$, if and only if there exists a rule $\eta \to \mu \in \mathcal{R}$ such that $e_1 = u_1 \eta u_2$ and

$e_2 = u_1 \mu u_2$ with $u_1, u_2 \in (V \cup \mathcal{A})^*$. Furthermore we say e_1 *yields* e_2, denoted by $e_1 \Rightarrow^* e_2$, iff $\exists i \geq 0$ and $\exists u_1, \ldots, u_i$ such that $e_1 = u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_i = e_2$. For instance, Figure 8.1(b) shows how the string S yields the string $aabbaa$. The language of a CFG, $\mathcal{L}(\mathcal{F}) = \{e \in \mathcal{A}^* : v_0 \Rightarrow^* e\}$, is the set of strings that contain only terminal symbols and that can be yielded from the string that contains only the initial non-terminal symbol v_0. We remark that in this example the language can generate an infinite set of words.

Given a CFG $\mathcal{F}$ and a string $e \in \mathcal{L}(\mathcal{F})$ that belongs to its language, we define a *parse tree* $\mathcal{K}_{\mathcal{F},e}$ as an ordered, rooted tree that determines a concrete syntactic structure of e according to the rules in $\mathcal{F}$:

- Each node in a parse tree $\mathcal{K}_{\mathcal{F},e}$ is either:
  - An *internal node* that corresponds to the application of a rule $\zeta \in R$.
  - A *leaf node* that corresponds to a terminal symbol $\tau \in \mathcal{T}$ and has no outgoing branches.
- Edges in a parse tree $\mathcal{K}_{\mathcal{F},e}$ connect non-terminal symbols to terminal or non-terminal symbols following the rules $\mathcal{R}$ in $\mathcal{F}$.

8.3 Representing CFGs as Generalized Planning

We model the generation of CFGs from positive examples as a generalized planning problem that is eventually solved with an off-the-shelf classical planner using the compilation proposed in Section 4.2. We formalize the task of generating CFGs as a tuple $\langle \mathcal{T}, \mathcal{E}, m \rangle$, where:

- $\mathcal{T}$ is the finite set of terminal symbols.
- $\mathcal{E} = \{e_1, \ldots, e_T\}$ is the finite set of input strings containing only terminal symbols: $e_t \in \mathcal{T}^*$, $1 \leq t \leq T$.
- m is a bound on the number of non-terminal symbols $V_m = \{v_0, \ldots, v_m\}$. As a consequence, m implicitly defines the space of possible rules, $V_m \rightarrow (V_m \cup \mathcal{T})^*$.

A solution to this inductive learning task is a CFG $\mathcal{F} = \langle V_m, v_0, \mathcal{T}, \mathcal{R}_m \rangle$ such that, for every $e \in \mathcal{E}$, there exists a parse tree $\mathcal{K}_{\mathcal{F},e}$.

Our approach for solving $\langle \mathcal{T}, \mathcal{E}, m \rangle$ is modeling this task as a generalized planning task $\mathcal{P} = \{P_1, \ldots, P_T\}$ where each input string $e_t \in \mathcal{E}$ corresponds to an individual classical planning task $P_t \in \mathcal{P}$ such that $1 \leq t \leq T$ and $P_t = \langle F, A, I_t, G_t \rangle$ is defined as follows:

- F comprises the fluents for modeling input strings as lists of symbols. These fluents are $\mathsf{string}(\mathsf{id}_1, \tau)$ and $\mathsf{next}(\mathsf{id}_1, \mathsf{id}_2)$, where $0 \leq id_1, id_2 \leq z$, $\tau \in \mathcal{T}$ and z is a bound on the string length. For instance, the string $abba$ is encoded as:

$$\mathsf{string}(\mathsf{i0}, \mathsf{a}), \mathsf{string}(\mathsf{i1}, \mathsf{b}), \mathsf{string}(\mathsf{i2}, \mathsf{b}), \mathsf{string}(\mathsf{i3}, \mathsf{a}),$$
$$\mathsf{next}(\mathsf{i0}, \mathsf{i1}), \mathsf{next}(\mathsf{i1}, \mathsf{i2}), \mathsf{next}(\mathsf{i2}, \mathsf{i3}), \mathsf{next}(\mathsf{i3}, \mathsf{i4}).$$

  In addition, F includes fluents $\mathsf{pos}(\mathsf{id})$, $0 \leq id \leq z$, to indicate the current string position, and $\mathsf{symb}(\tau)$, $\tau \in \mathcal{T}$, to indicate the symbol at the current string position.

- A contains the actions for parsing the current symbol of an input string. There is a parse_τ action for each symbol $\tau \in \mathcal{T}$, e.g. $A = \{\mathsf{parse}_\mathsf{a}, \mathsf{parse}_\mathsf{b}\}$ for the CFG of Figure 8.1(a). Action parse_τ recognizes that τ is at the current position of the string and advances the position.

$$\mathsf{pre}(\mathsf{parse}_\tau) = \{\mathsf{symb}(\tau)\},$$
$$\mathsf{cond}(\mathsf{parse}_\tau) = \{\{\mathsf{pos}(\mathsf{i}_1), \mathsf{next}(\mathsf{i}_1, \mathsf{i}_2), \mathsf{string}(\mathsf{i}_2, \tau')\} \triangleright$$
$$\{\neg\mathsf{pos}(\mathsf{i}_1), \mathsf{pos}(\mathsf{i}_2), \neg\mathsf{symb}(\tau), \mathsf{symb}(\tau')\} : \forall \mathsf{i}_1, \mathsf{i}_2, \tau'\}.$$

- I_t contains the fluents encoding the t-th string, $e_t \in \mathcal{E}$, and its initial position $\mathsf{pos}(\mathsf{0})$.

- G_t requires that e_t is parsed, i.e. $G_t = \{\mathsf{pos}(|\mathsf{e_t}|)\}$.

According to this definition, a solution Π to a generalized planning problem $\mathcal{P}$ that models a CFG generation task $\langle \mathcal{T}, \mathcal{E}, m \rangle$ parses every $e_t \in \mathcal{E}$. We assume that a solution to $\mathcal{P}$ is in the form of *planning programs* enhanced with *procedures*. However, there is no need of *conditional gotos* to represent CFGs in planning.

Then, we extend planning programs formalism with *choice instructions*. Choice instructions are intended to jump to a target line of a planning program and are defined as $\mathcal{I}_{choice} = \{\mathsf{choose}(\mathsf{Target})\}$, where $\mathsf{Target} \subseteq \{1, \ldots, n\}$ is a subset of possible target program lines.

Figure 8.2 shows a planning program with a *choice instruction* that encodes the CFG in Figure 8.1(a). In this example instruction `choose(1|5|8)` represents a jump to one of these three possible targets, line 1, line 5 or line 8. We also show in this example how to execute expressions that contain non-terminal symbols. In this case we use the call instructions that can call other non-terminal symbols and themselves recursively. We define the set of call instructions as $\mathcal{I}_{call} = \{\mathsf{call}(j') : 0 \leq j' \leq m\}$ where m is the number of non-terminal symbols. Therefore, after removing conditional gotos and including choice and call instructions, the set of instructions for a CFG is $\mathcal{I} = A \cup \mathcal{I}_{choice} \cup \mathcal{I}_{call} \cup \{\mathsf{end}\}$.

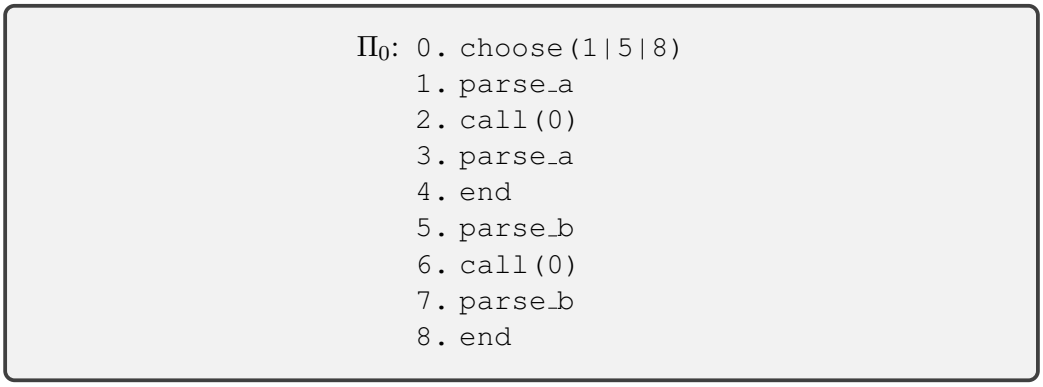


Figure 8.2: Planning program that represents the CFG in Figure 8.1(a).

The representation of CFGs with planning programs is done associating each non-terminal symbol $v_j \in V_m$ with a planning program Π_j. *Choice instructions* always appear on the first line of Π_j and represent possible jumps to the lines coding the grammar rules, $v_j \rightarrow (v_j \cup \mathcal{T})^*$, associated to the corresponding non-terminal symbol. Initially, the subset of target program lines only includes 1 and n, i.e. $choose(1|8)$ for the example in Figure 8.2. Whenever we program an *end* instruction on a line i, we add $i + 1$ to the subset of target lines, leading to the choice instruction $choose(1|5|8)$ in Figure 8.2.

The execution model for a planning program Π that represents a CFG consist of a program state (s, Ξ) where s is the planning state and Ξ is a call stack. Given a program state $(s, \Xi \oplus (j, i))$, the execution of instruction $w_i^j \in \mathcal{I}$ on line i of procedure Π_j is defined as follows:

- If $w_i^j \in A$, the new program state is $(s', \Xi \oplus (j, i+1))$, where $s' = \theta(s, w_i^j)$ is the resulting state from applying a sequential action w_i^j in state s.
- If $w_i^j = \mathsf{choose(Target)}$, it actively *chooses* to jump to a new line $i' \in Target$, changing the program state from $(s, \Xi \oplus (j, i))$ to $(s, \Xi \oplus (j, i'))$.
- If $w_i^j = \mathsf{call(j')}$, the new program state is $(s, \Xi \oplus (j, i+1) \oplus (j', 0)$. In other words, calling a procedure $\Pi_{j'}$ from Π_j has the effect of (1) incrementing the program line at the top of the stack; and (2) pushing a new element onto the stack to start the execution of $\Pi_{j'}$ on line 0.
- If $w_i^j = \mathsf{end}$, the new program state is (s, Ξ), i.e. a termination instruction has the effect of terminating a procedure by popping element (j, i) from the top of the call stack.

To execute a CFG as a planning program with procedures Π on a planning problem $P = \langle F, A, I, G \rangle$, we set the initial planning state on program line 0 of

the main program Π_0. To ensure that the execution remains bounded we impose an upper bound ℓ on the size of the call stack.

The execution terminates successfully when the goal condition holds in the planning state $G \subseteq s$ and the call stack is empty, i.e. in program state (s, Ξ) with $|\Xi| = 0$. Otherwise, executing Π on P can fail for the same reasons as in Section 3.2 but without entering into infinite loops because there are no goto conditions:

1. Execution terminates in program state (s, Ξ) with $|\Xi| = 0$ but the goal condition does not hold, i.e. $G \not\subseteq s$.
2. When executing action $w_i \in A$ in program state (s, Ξ), the precondition of w_i does not hold, i.e. $\mathsf{pre}(w_i) \not\subseteq s$.
3. Executing a call instruction $\mathsf{call(j')}$ in program state (s, Ξ) exceeds the stacks size $|\Xi| = \ell$, i.e. causes a *stack overflow*.

8.4 Computing CFGs with Classical Planning

To compute CFGs with classical planning we need as input a CFG generation task $\langle \mathcal{T}, \mathcal{E}, m \rangle$ such that $|e_t| \leq z$ for each $e_t \in \mathcal{E}$, a number of program lines n and a stack size ℓ, and outputs a classical planning instance $P^{\ell,z}_{n,m} = \langle F^{\ell,z}_{n,m}, A^{\ell,z}_{n,m}, I^{\ell,z}_{n,m}, G^{\ell,z}_{n,m} \rangle$. The compilation is almost identical to the compilation described in Section 4.4; the only relevant differences are that procedures do not have parameters, there are not conditional gotos, the fluents added in $F^{\ell,z}_{n,m}$ are the choice instructions $\mathsf{ins}_{0,j,choose}$, such that $0 \leq j \leq m$, and $A^{\ell,z}_{n,m}$ includes actions for simulating the execution of choice instructions $\mathsf{choose}(\mathsf{Target})^k_{i',j}$, where $i' \in \mathsf{Target}$:

$$\mathsf{pre}(\mathsf{choose}(\mathsf{Target})^k_{i',j}) = \{\mathsf{pc}^k_0, \mathsf{top}^k, \mathsf{proc}^k_j, \mathsf{ins}_{0,j,choose}\},$$
$$\mathsf{cond}(\mathsf{choose}(\mathsf{Target})^k_{i',j}) = \{\emptyset \rhd \{\neg\mathsf{pc}^k_0, \mathsf{pc}^k_{i'}\}\}.$$

Lemma 8.1. *Any classical plan π that solves $P^{\ell,z}_{n,m}$ induces a valid model $\mathcal{F} = \langle V_m, v_0, \mathcal{T}, \mathcal{R}_m \rangle$ for the CFG generation task $\langle \mathcal{T}, \mathcal{E}, m \rangle$.*

Proof. Once the instructions of a planning program $\Pi = \{\Pi_0, \ldots, \Pi_m\}$ are programmed they can only be executed. The classical plan π has to program the instructions (if not yet programmed) of Π and simulate its execution, actively choosing the jumps defined by the choice instructions and their corresponding subsets of *target* lines. This simulation is done for the planning task P_t encoding the t-th string in $\mathcal{E}$ and the *active choice* corresponds exactly to the construction of the parse tree for the t-th string. If this is done for every $1 \leq t \leq T$, the CFG induced by π satisfies the solution condition for the solutions of the CFG generation task $\langle \mathcal{T}, \mathcal{E}, m \rangle$. □

8.4.1 Parsing and Production with Classical Planning

String production and string parsing for arbitrary CFGs can also be addressed using our compilation and an off-the-shelf classical planner.

Parsing

Let $e \notin E$ be a new string, and let $P_e = \langle F^{\ell,z}_{n,m}, A^{\ell,z}_{n,m}, I_e, G_e \rangle$ be the classical planning instance for parsing the string e, i.e. instantiated on the planning frame as the problem $P^{\ell,z}_{n,m}$. In this case we can use a classical planner to determine whether $e \in \mathcal{L}(\mathcal{F})$.

Our approach is to specify $\mathcal{F}$ in the initial state of P_e, making initially true the fluents $\mathsf{ins}_{i,j,w}$ that correspond to the planning program that encodes $\mathcal{F}$, allowing only the executing actions $\mathsf{E}(w^k_{i,j})$ and ignoring the actions $\mathsf{P}(w^k_{i,j})$ for programming instructions. A solution plan π_e to P_e is then constrained to the actions that execute the instructions specified by $\mathcal{F}$ and represents a parse tree $\mathcal{K}_{\mathcal{F},e}$. Essentially, parsing consists in correctly choosing the target program line i' each time a choice instruction is executed.

Interestingly *parsing* can also be understood as activity recognition using plan libraries that are in the form of CFGs (Ramirez and Geffner, 2016).

Production

We can produce a string $e \in \mathcal{L}(\mathcal{F})$ using our compilation and an off-the-shelf classical planner.

Again we use a classical planning instance P_e that represents the parsing of example e with $\mathcal{F}$ also specified in the initial state of P_e and ignoring the actions for programming instructions. The difference with the previous task is that here the linked list that encodes the string e is initially empty (the corresponding fluents are false at I_e). This list is filled up, symbol by symbol, by the actions that execute $\mathcal{F}$ until reaching the end of the string. To do so actions parse_τ are replaced with actions $\mathsf{produce}_\tau$ that add symbol $\tau \in \mathcal{T}$ at the current position of the string. Formally, $\mathsf{produce}_\tau$ is defined as:

$$
\begin{aligned}
&\mathsf{pre}(\mathsf{produce}_\tau) = \{\mathsf{active}\},\\
&\mathsf{cond}(\mathsf{produce}_\tau) = \{\{\mathsf{pos}(\mathsf{i}_1)\} \rhd \{\mathsf{string}(\mathsf{i}_1, \tau)\} : \forall \mathsf{i}_1\}\\
&\cup\ \{\{\mathsf{pos}(\mathsf{i}_1), \mathsf{next}(\mathsf{i}_1, \mathsf{i}_2)\} \rhd \{\neg\mathsf{pos}(\mathsf{i}_1), \mathsf{pos}(\mathsf{i}_2)\} : \forall \mathsf{i}_1, \mathsf{i}_2\}\\
&\cup\ \{\{\mathsf{pos}(\mathsf{i}_1), \mathsf{next}(\mathsf{i}_1, z)\} \rhd \{\neg\mathsf{active}\} : \forall \mathsf{i}_1, \mathsf{i}_2\},
\end{aligned}
$$

where active is a fluent that keeps track of whether we have reached the end of the string to be generated.

Text production using grammars and classical planning is a well studied task (Schwenger et al., 2016; Koller and Hoffmann, 2010). This task is also related to the compilation of domain-specific control knowledge for planning (Baier et al., 2007; Alford et al., 2009).

8.5 Evaluation

We designed two types of experiments: (1) *Generation*, for computing CFGs compliant with a set of input strings and (2), *Recognition* for parsing unseen input strings given a CFG. All the experiments are run on an Intel Core i5 3.10 GHz x 4 with 4 GB of RAM, using the classical planner Fast Downward Helmert (2006b), with the `LAMA-2011` configuration, and a planning time limit of 600 seconds.

We created six domains that correspond to CFGs with different structure and alphabet. The domain **AnBn** corresponds to the $A^n B^n$ language. The **Parenthesis** domain corresponds to the strings that can be formed following one of two well-balanced parenthesis distributions, *sequential* $()() \ldots$ or *enclosing* $((\ldots))$. **Parenthesis Multiple** corresponds to the enclosing well-balanced parenthesis distribution but using a larger alphabet, $\mathcal{T} = \{(, \{, [,], \},)\}$. In the **Binary Arithmetics** domain the alphabet contains two possible binary values $\{0, 1\}$ and two operators $\{+, -\}$, and corresponds to the language of the arithmetic expressions composed by an arbitrary binary number, an operator and another binary number. For **Arithmetics** the alphabet includes the values $\{0, \ldots, 9\}$ and the operators $\{+, -, *, /\}$, and the language is the set of expressions formed as decimal number, operator and decimal number. Finally, a simplified **English Grammar** that includes *Sentence*, *Noun Phrase* and *Verb Phrase* as non-terminal symbols while *adjective*, *adverb*, *noun* and *verb* are terminal symbols.

Table 8.1 shows the results of the *Generation* tasks. For each domain we report the number m of non-terminal symbols, the size of the stack, the procedure lines (per non-terminal symbol), the number of input strings (per non-terminal symbol) and the planning time for computing each procedure corresponding to a non-terminal symbol.

The *Generation* results show that non-terminal symbols are used with two aims: i) abstracting a set of terminal symbols, e.g. the first procedure of the **Arithmetics** domain (with 20 lines, learned from 10 strings in 10.3 seconds) processes any digit in the set $\{0, \ldots, 9\}$; ii) grouping multiple rules, e.g. in **English Grammar** one procedure represents a Noun Phrase (NP) that is composed of one or more adjectives (a) and a noun (n), so it computes the rules $NP \rightarrow an|aNP$.

Table 8.2 shows the results for the *Recognition* tasks. In these experiments the CFGs grammars are given so we explore the performance of our approach using larger stack sizes. For each domain we report the size of the stack (which limits

	m	Stack	Lines	Strings	Time(s)
AnBn	1	5	5	1	0.3
Parenthesis	1	5	5	2	0.4
P. Multiple	1	5	12	3	53.1
Binary A.	2	4	(6,8)	(4,2)	(0.6,1.8)
Arithmetics	4	8	(20,8,3,4)	(10,4,1,4)	(10.3,3.7,3.5,14.6)
E. Grammar	3	10	(6,3,3)	(2,1,1)	(1.4,0.3,1.9)

Table 8.1: Generation task results.

the max depth of the possible parse trees), the number of strings, and the total planning time required for parsing the strings.

	Stack	Strings	Time(s)
AnBn	51	1	70.67
Parenthesis	52	1	19.29
P. Multiple	52	1	129.19
Binary A.	15	2	62.87
Arithmetics	25	4	137.76
E. Grammar	92	1	325.44

Table 8.2: Recognition task results.

8.6 Summary

In this Chapter 8 we have introduced the concept of a formal language $\mathcal{F}$, i.e. a context-free grammar, which is composed of an alphabet $\mathcal{T}$ and a set of syntactic rules $\mathcal{R}$. From a formal language we can perform two tasks. The first one is *producing* a (possibly infinite) set of words $\mathcal{L}(\mathcal{F})$ and the second one is *parsing* a string that belongs to the language. The formal language is formalized as a CFG, and we propose a compilation to classical planning problem whose solution corresponds to a CFG in the form of a *planning program*.

The generation of CFGs has been exhaustively studied in previous work. For instance, learning CFGs given a corpus of a correctly parsed input string (Sakakibara, 1992; Langley and Stromsten, 2000). Another example, is inferring CFGs using positive and negative examples (De la Higuera, 2010; Muggleton et al., 2014). In our approach, we generate CFGs using only a small set of positive examples (in some cases even one single string that belongs to the language).

Furthermore we have implemented a compilation to PDDL that benefits straightforwardly from research advances in classical planning and that is also suitable for *production* and *recognition* tasks with arbitrary CFGs.

The size of the compilation output depends direclty in the chosen bounds. We have four bounds m, n, ℓ and z that correspond to the number of non-terminal symbols, the maximum number of lines of a planning program (or CFG), the size of the stack and the length of the largest input string. Whether these bounds are too small, the classical planner used to solve the output planning task will not be able to find a solution. Larger values for these bounds affects practically to our approach in two ways: i) the performance of classical planners is sensitive to the size of the input, ii) and a stack size that is large enough to parse the string can overfit the problem by programming a choice to a terminal symbol and a recursive call.

The number of input strings also affects to the size of the compilation. Empirical results show that our approach is able to generate non-trivil CFGs from very small data sets. Interestingly our approach can also follow an incremental strategy where we generate the CFG for a given sub-language and then encode this *sub-grammar* as an auxiliary procedure for generating more challenging CFGs like in Chapter 4 and Chapter 6.

This work could be extended adding negative input strings, which would require a mechanism for validating that a given CFG does *not* generate a given string. Another interesting direction is to accept incomplete strings such that they can be completed by combining the parsing and production mechanisms.

PART V

Discussion

CHAPTER 9

Related Work

In Chapter 9 we are going to review the most relevant publications in topics connected to generalized planning with planning programs and finite state controllers, to program synthesis, activity recognition and generating grammars. We are aware that it is possible to find missing citations, but we have connected our work to those that inspired us to publish.

9.1 Contingent and Conformant 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* and *belief propagation*. On the one hand, the conformant planning problem is compiled into classical planning to compute a *prefix* plan that removes any relevant uncertainty (Palacios and Geffner, 2009), and a *postfix* plan 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. On the other hand, belief propagation consist of searching in the belief state space where the *root* belief state represents the set of possible initial states, and the *goals* are those belief states such that all possible states in the belief state satisfy a goal condition (Hoffmann and Brafman,

2006; Cimatti et al., 2004). 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.

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, 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 in a finite number of steps, and that conditions for branching and looping are constrained to 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, where: (1) the instances to be solved share the same goals, (2) the initial state of that instances is a state in the set of possible initial states of the contingent/POMDP planning task and (3), there is full observability so the conditions for branching and looping can refer to the value of any state variable.

9.2 Planning with Control Knowledge

In this section we review related work from multiple fields that are connected to generalized planning. We must mention that generalized planning (see Chapter 2) has been studied from planning and learning perspectives.

9.2.1 Macro-Actions

Macro-actions (i.e. action sub-sequences) were among the first suggestions to compute general knowledge for planning and there are several approaches in the literature for computing them (Botea et al., 2005; Coles and Smith, 2007; Jonsson,

2009). However, the sequential execution flow fixed by macro-actions is usually too rigid and, even when macros are parameterized, they have to be combined with control-flow structures in order to generalize to more planning instances.

9.2.2 Generalized Policies

Generalized policies are a more flexible formalism than macros. A generalized policy is a set of rules mapping states and goals into actions; hence generalized policies are *reactive* and do not explicitly represent action sequences. Computing good generalized policies is however complex. Early algorithms for computing generalized policies (Khardon, 1999; Martín and Geffner, 2004) first compute sequential plans, and then attempt to generalize the policy rules from these plans, a difficult task because of the high number of symmetries and transpositions that commonly appear in sequential plans. Moreover, a *generalized policy* cannot be added in a straightforward way to a domain theory. However, we have introduced several formalisms in previous chapters that are related to *generalized policies*. For instance, planning programs (see Chapter 3) and Finite State Controllers (see Chapter 6) guide classical planners effectively with *Domain Control Knowledge* (DCK), similar to previous approaches for generating generalized policies (Yoon et al., 2008; De la Rosa et al., 2011).

9.2.3 Procedural Domain Control Knowledge

Domain Control Knowledge refers to knowledge about the structure of planning solutions. *Planning with DCK* can also be seen as a form of generalized planning, that constrains the space of possible solutions, but requires a planner to produce a fully specified solution for a particular classical planning instance. This approach is connected to planning with control rules (Bacchus and Kabanza, 2000; Veloso et al., 1995) and hierarchical planning (Shivashankar et al., 2012; Nau et al., 2003).

Procedural DCK (see Chapter 4) in the form of Golog-like programs (Baier et al., 2007; Fritz et al., 2008) include conditionals and loops as well as non-deterministic choice actions that constrain the search for a solution plan. Nevertheless they are not proper generalized plans since it is still necessary to apply a planner to solve each individual planning problem. In contrast to our approach in Chapter 5, they are hand-crafted and do not implement PDDL mechanisms for procedure calling.

9.2.4 Finite State Machines

Previous work on computing *Finite State Machines* (FSMs) (Bonet et al., 2010) also uses a compilation that interleaves programming a FSM with verifying that

it satisfies a set of test cases. The generation of a FSM is however different since it starts from a partially observable planning model and uses a compilation from conformant to classical planning (Palacios and Geffner, 2009). FSMs can be understood as a way of representing and computing procedural DCK that (1) does not implement procedure calls and recursion; (2) does not reuse FSMs for similar tasks. Another difference is that our conditional goto instructions can branch on any fluent without the need to predefine a subset of observations, although we would also benefit from restricting the number of branch conditions.

9.2.5 Hierarchical Planning

Hierarchical planning is the problem of finding an ordered sequence of actions given a high-level goal specification. It is formalized as a Hierarchical Task Network (HTN) that is a powerful mechanism to represent libraries of plans (Nau et al., 2003; Shivashankar et al., 2012). HTNs are defined as a set of primitive tasks, a set of compound tasks that are decomposed into simpler tasks, the decomposition methods and a set of goal tasks. The level of specifications to reach a goal task has a strong relation with the production rules of CFGs (see Chapter 8).

The previous work in generating HTNs (Hogg et al., 2008; Lotinac and Jonsson, 2016) show that HTNs can be computed from problems to encode domain knowledge that can be applied to any instance of the domain. Thus, generating an HTN is similar to a generalize planning task which opens an interesting research direction to extend our approach for computing HTNs from flat sequences of actions. This is also related to Inductive Logic Programming (ILP) (Muggleton, 1999) that learns logic programs from examples. Unlike logic programs or HTNs, the CFGs we generate are propositional and do not include variables but they allow recursive solutions.

9.2.6 Case-Based Planning

There are two main strategies called *top-down* or *bottom-up* to compute a solution to a generalized planning task (see Subsection 2.3.3). In our approaches we use the *top-down* strategy, searching in the space of programs to cover the whole set of planning instances. While in *Case-Based Planning* (CBP) (Borrajo et al., 2015) they use the second approach. In order to solve a generalized planning task, they look at a single planning instance, compute a solution that solves it, generalizes it, and then applies to the current solution the merging method with the solutions found previously. The CBP techniques incrementally increase the coverage of a generalized plan but require complex merging methods. This approach is related to works on *plan repair* (Fox et al., 2006) since it demands identifying why a solution does not cover a given instance and adapting it to the uncovered instance.

The Distill system (Winner and Veloso, 2003) lies in this category and, as our work in Chapter 3, uses programs to represent generalized plans. However, its representation is different and its performance was not tested over a wide range of diverse generalized planning tasks as we did in Chapters 3, 4 and 5.

9.3 Program Synthesis

Program synthesis (Gulwani et al., 2017) is the task of automatically generating a program that satisfies a given high-level input 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 the most successful approaches for program synthesis:

9.3.1 Programming by Example

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 (see Section 4.6) 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 (DSL) 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.

9.3.2 Programming by Sketching

In *Programming by Sketching* (PbS), programmers provide a partially specified program, i.e. 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. Even though in the worst case, the synthesis problem to a quantified boolean satisfiability (QBF) is PSPACE-complete, and can be solved in time exponential in

the number of quantified variables, this counterexample-driven search terminates on many real problems after solving only a few SAT instances (Lake et al., 2015).

9.3.3 Bounded Synthesis

Bounded synthesis addresses the synthesis of finite-state transition systems that satisfy a given Linear Temporal Logic (LTL) formula (Schewe and Finkbeiner, 2007; De Giacomo and Vardi, 2015). The mainstream approach for bounded synthesis is compiling this task into a satisfiability modulo theories (SMT) (Barrett and Tinelli, 2018) problem. The compilation requires fixing a bound on the system parameters, such as the number of rejecting states, with which it proposes theories that are solved using SAT-based techniques. In our case we have a set of planning tasks to solve that must be covered in a specific order by a generalized plan (see Chapter 2). This is inspired in Machine Learning (ML) and unit tests in Test-Driven Development (TDD).

9.4 GOLOG

The GOLOG family of action languages has proven useful for programming autonomous behavior that is able to generalize (Levesque et al., 1997). Apart from conditionals, loops and recursive procedures, GOLOG programs can contain nondeterministic 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. The GOLOG programmer can determine the right balance between predefined behavior and leaving certain parts to be solved by the system by means of search. The basic GOLOG interpreter uses the PROLOG backtracking mechanism to resolve the search. This mechanism basically amounts to 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. We defined our own programming language that defines the space of possible generalized plans. In this language branching and loops are implemented with the same construct (conditional gotos) to keep the solution space as reduced as possible (see Chapter 3).

9.5 Hierarchical Finite State Controllers

We dedicate a section to compare work related to Hierarchical Finite State Controllers (see Chapter 6) for the relevance in this dissertation and the novelty of the approach. The main difference from our approach with previous work on automatically generating FSCs (Bonet et al., 2010; Hu and De Giacomo, 2013) rely on a partially observable planning model in which the *observation* function is given as input, while in our classical planning compilation, the *observation* function is synthesized in addition to the *transition* and *output* functions of FSCs. Thus, we generate hierarchical FSCs that branch on any fluent since all fluents, are considered observable. Further, our approach provides a *call mechanism* that makes it possible to generate recursive solutions and to incorporate prior knowledge as existing FSCs.

Hierarchical FSCs are related to the *planning programs* formalism for the representation and computation of compact and generalized plans (see Chapter 4). These programs are a special case of FSCs, and in general, FSCs can represent plans more compactly than planning programs. Another related formalism is *automaton plans* (Bäckström et al., 2014), which also store sequential plans compactly using hierarchies of finite state automata. However, automaton plans are *Mealy* machines whose transitions depend on the symbols of an explicit input string. Hence automaton plans are not suitable for representing generalized plans, and their focus is instead on the compression of sequential plans.

Apart from solution plans, finite state automata can represent other objects in planning. For instance, the *domain transition graph* is an automaton for representing the possible values of planning state variables (Chen et al., 2008). Toropila and Barták [2010] also used finite state machines to represent the domains of the state variables of a given planning instance. Another examples are the LOCM system, that uses finite state machines to represent planning domain models (Cresswell et al., 2013) or Hickmott et al. [2007] that used *Petri nets* to represent an entire planning instance.

9.6 Hierarchical Reinforcement Learning

Reinforcement Learning (RL) (Sutton and Barto, 1998) is the problem of learning a policy by interacting with the world while obtaining positives and negatives rewards. The literature explores different approaches like Shavlik [1990], Parr and Russell [1998], Dietterich [2000] and Chentanez et al. [2005] that leverages hierarchical decompositions of complex sequential problems and learn when every controller should be applied. In some cases this knowledge can be iteratively included in the bag of controllers, in other cases this set is closed but can be reused

to solve more complex tasks, options or to learn more complex concepts. This is related to Chapter 4 where the hierarchical decomposition is in form of planning programs with procedures or to Chapter 6 where we have hierarchical finite state controllers that allow recursive calls.

Although in the best case, learning converges to a solution that minimizes a cost function (or maximizes a reward); in domains with huge state spaces, full observability and a distant horizon, it may become unfeasible for RL approaches to reach a goal and even harder to converge, so our approach of generalized planning that benefits from planning techniques can be applied to explore only the promising state space sections.

9.7 Activity Recognition

On the one hand, the set of planning instances that belong to a generalized planning task can be understood as an unsupervised learning method (see Chapter 7) where input examples and class labels correspond to reachable states and a generalized plan. Thus, a generalize plan is the learned cluster model that applies to every reachable state its corresponding action. We consider a set of classical plans as labels if we can induce from this set a generalized plan. This would correspond to supervised learning (De la Rosa et al., 2011).

On the other hand, the generation of CFGs can also be understood in terms of activity recognition (see Chapter 8), such that the library of activities is formalized as a CFG, the library is initially unknown, and the input strings encode observations of the activities to recognize. Activity recognition is traditionally considered independent of the research done on automated planning, using handcrafted libraries of activities and specific algorithms (Ravi et al., 2005). An exception is the work by Ramírez and Geffner [2009; 2010] where goal recognition is formulated and solved with planning. As far as we know our compilation of CFGs to generalized planning (see Chapter 8) is the first that integrates the tasks of *grammar generation*, *string parsing* and *string production* using a common planning model and an off-the-shelf classical planner.

CHAPTER 10

Summary

In this Chapter 10 we show a summary of the most relevant contributions of this dissertation followed by a discussion of possible future directions where this work could be extended and new challenges.

10.1 Contributions

We have contributed to different fields of Artificial Intelligence from a planning perspective approach. Our intuition is that often, control strategies can be expressed using relatively simple pieces of code that interact with each other. All our models explain how several problems with different origins are connected when represented as programs, benefiting from state-of-the-art tools in classical planning. This is the list of most relevant contributions:

1. We have proposed several formalisms to *compute procedural DCK* (see Chapter 4 to Chapter 6) over a wide range of domains and exploit them with off-the-shelf classical planners. These are the first implementations of *procedural DCK* to PDDL while allowing nested and recursive calls. This allows us to compute generalized plans to non-trivial tasks such as *Selection-Sort* or *DFS* for traversing binary trees of variable size.

2. We set down the theoretical foundations of the *planning program* formalism and show that plan validation and plan existence for planning programs are PSPACE-complete. In addition we provide formal characterizations of the solutions that can be represented with our formalism and prove that the expressiveness of our different forms of planning programs is equivalent (see Chapter 3 and Chapter 4).

3. We formalized the concepts of *deterministic* and *non-deterministic* planning programs for generalized planning. Regarding *non-deterministic* planning programs execution, we introduced two new formalisms: *choice instructions* and *lifted action instructions* (see Chapter 5).

4. We show how to implement the validation mechanism for planning programs within our compilation. We use this mechanism to show that reusing generalized plans and using generalized plans as control knowledge allow us to solve planning instances that are difficult to solve using current classical planners, like blocksworld instances with 100 blocks (see Chapter 5).

5. We reformulated the transition function of FSCs for planning (see Chapter 6) to allow binary branching to reduce the space of possible controllers. Furthermore, we formalized the definition of *hierarchical FSCs* for planning that allows FSCs to call other FSCs, including recursion as special case using a *call stack*.

6. Work in Chapter 6 presents a study of the relation of *Mealy* machines and FSCs for classical and generalized planning, as well as a formal and empirical comparison of *hierarchical FSCs* with the *planning program* formalism for generalized planning (see Chapter 4).

7. In terms of performance, the computation of *planning programs* is enhanced by reducing the number of actions needed to execute *conditional goto instructions* (see Chapter 4), and the computation of *hierarchical FSCs* with a symmetry breaking control (see Chapter 6).

8. We connected classical planning with structured prediction with the synthesis of programs (see Chapter 7). The planning programs we propose (see Chapter 3) are more expressive than previous approaches (Ellis et al., 2015) that only consider programs generated by an acyclic grammar. However, our programs are *deterministic* rather than *probabilistic*, which limits their applicability when inputs are noisy.

9. Context-free grammars (CFGs) can be represented using planning programs (see Chapter 8). We can generate CFGs from small amounts of input data (even a single input string in some cases). A CFG is an extension of planning programs with *choice instructions* that decide which rule to apply for the two canonical tasks of CFGs, *string parsing* and *string production*.

10. Finally, we put in context all recent works of generalized planning. These works can be described with an abstract framework for generalized planning (see Chapter 2).

10.2 Future Work and Open Challenges

In the last section of this dissertation we explore different topics that can extend planning program or FSC representations. Generalized planning is a broad field which can benefit from planning and learning communities, where each contribution could be in long-term relevant to both.

Planning Program Heuristics

FAST DOWNWARD, the planner used in the evaluation, is the most widely used in the planning community and is a good touchstone to evaluate the capacity of classical planners to address generalized planning tasks. On the other hand, Fast Downward is a heuristic-search based planner whose heuristics are based in the delete relaxation of the planning problem and typically has difficulties with problems that include dead-ends, such as those in our compilations. The domains resulting from our compilation encode key information in the delete effects of actions: when an instruction is programmed on a line i, the fluent $\mathsf{ins}_{i,\mathsf{nil}}$ is deleted, preventing us from programming another instruction on the same line. In a delete-free relaxation, $\mathsf{ins}_{i,\mathsf{nil}}$ remains true, making it possible to program any number of instructions on the same line. As a consequence, a solution to the delete-free relaxation can effectively use different programs to solve the individual instances of a generalized planning problem, which results in a poor approximation of how difficult the original problem is. In addition, a program suitable for a particular individual instance might not be suitable for solving the next instance, causing dead-ends and deep backtracking. In the future it would be interesting to explore novel planning techniques that deal better with these issues like a proper heuristic for synthesizing planning programs.

Hierarchical FSCs for LTL representations

A different application of finite state machines for planning is to compile LTL representations of *temporally extended goals*, i.e. conditions that must hold over the intermediate states of a plan, into a non-deterministic automaton Baier and McIlraith [2006]. Related to this, the techniques for *bound synthesis* show how to address the computation of finite-state transition systems that satisfy a given LTL formula (Finkbeiner and Schewe, 2013). An interesting research direction is how to adapt our approach for the computation of automata of these kinds.

Bounds and Problem decomposition

Automatic generation of planning programs and FSCs take as input a bound on the number of lines, controller states, stack size and so on. An iterative deepening

approach could be implemented to automatically derive these bounds. Another issue is the specification of representative subproblems to generate procedures or hierarchical FSCs in an incremental fashion. Inspired by "Test Driven Development" (Beck et al., 2001), we believe that defining subproblems is a step towards automation.

So far, the utility of our compilations are limited to tasks solvable with small planning programs. Part of the reason is that the number of possible programs is exponential in the number of program lines. In addition, some domains require complex state queries to compute a compact generalized plan. For instance, a compact while general solution for *sokoban* requires defining complex connectivity properties, such as *reachable*, that involve recursion (Ivankovic and Haslum, 2015). For these reasons our current approaches cannot compute effective DCK for arbitrary classical planning domains. We have shown that we can address more challenging tasks if a subtask decomposition is available. In this case we can separately compute auxiliary procedures for each of the subtasks and incrementally reuse them. An interesting open research direction is then to automatically discover these subtask decompositions.

Covering strategy

The evaluation in Chapter 6 evidenced the impact of the order of the input tasks in the experimental performance of our compilation. Even though the planner could also be used to determine this order, a better approach is to compute generalized plans considering the parallel execution of the programs or controllers over all the instances. An interesting research direction is then the use of techniques for progressing belief states, like in *conformant* or *contingent* planning (Palacios and Geffner, 2009; Albore et al., 2009).

Generating relevant examples

The selection of relevant examples that can produce a solution that generalize is related to the topic of *quality data* versus *big data*. A key issue is to determine which instances generalize most efficiently. Currently this selection of informative instances is done by hand, and an interesting research direction is to develop techniques for automatic instance selection. In this case implicit representations of the planning instances (Srivastava et al., 2011b) seem more useful since they can act as generative models. Since planning problems are highly structured there is no guarantee that randomly sampled problems are relevant for solving a given generalized planning task. An interesting research direction would be to study how to generate examples of this kind. A good starting point could be previous

work on generating random walks (Fern et al., 2004) and active learning (Fuentetaja and Borrajo, 2006) in planning domains.

Generalizing metrics

We follow an inductive approach to generalization, and hence we can only guarantee that the solution generalizes over the instances of the generalized planning problem, much as in previous work on computing programs or FSCs. This is an open issue in planning that can be solved following a machine learning approach, where the validation of a generalized solution is traditionally done by means of statistics and validation sets.

More expressive planning programs

Finally, we are only able to generate high-level state features in the form of conjunctive queries, and hence we cannot produce from scratch programs that contain features with unbounded transitive or recursive closures. This kind of features are known to be useful for some planning domains, e.g. the *above* feature (the transitive closure of *on*) for the Blocksworld domain. Also functional STRIPS (Geffner, 2000; Frances and Geffner, 2016) would enhace our compilations avoiding to ground objects like mathematical expressions. In the near future we would like to extend our approach to generating more expressive features.

We hope to solve all these challenges in the future as well as detecting new ones that could be relevant to the scientific community.

PART VI

Appendix

APPENDIX **A**

Notation

Classical Planning

- $\mathcal{S}$: state model
- S: set of states
- s: a state
- s_0: initial state (state model)
- S_G: set of goal states
- A: set of actions
- a: an action
- $a(s)$: applicable action
- θ: transition function
- c: cost function
- π: classical plan
- $|\pi|$: size of a classical plan
- P: classical planning problem
- F: set of fluents

- f: a fluent
- I: initial state(STRIPS model)
- G: goal condition
- L: set of literals
- l: a literal
- $\texttt{pre}(a)$: preconditions of an action
- $\texttt{del}(a)$: delete effects of an action
- $\texttt{add}(a)$: adding effects of an action
- Φ: classical planning frame
- $\texttt{cond}(a)$: set of conditional effects of an action
- C: condition of an action (conditional effects)
- E: effect of an action (conditional effects)
- $C \rhd E \in \texttt{cond}(a)$: conditional effect of an action
- $\texttt{eff}(s, a)$: triggered effects in a state-action pair
- $\texttt{PE}(\mathcal{S})$: plan existence problem
- $\texttt{PC}(\mathcal{S}, k)$: bounded plan existence
- π^*: optimal plan

Generalized Planning

- $\mathcal{P}$: generalized planning problem
- Π: generalized plan
- $\Pi(P)$: application of a generalized plan to a planning problem P
- $|\Pi|$: size of a generalized plan

Basic Planning Programs

- Π: planning program
- w: program instruction

- $\mathcal{I}$: set of instructions
- $\mathcal{I}_{\mathsf{go}}$: set of goto instructions
- n: bounded number of lines
- P_n: compilation of a planning problem bounded with a planning program of n lines
- F_{pc}: set of fluents that encode the program counters
- F_{ins}: set of fluents that encode the instructions
- $\mathsf{P}(w)$: programming action of instruction w
- $\mathsf{E}(w)$: executing action of instruction w
- T: number of planning instances from a generalized planning task
- P'_n: compilation of a generalized planning task bounded with a planning program of n lines
- F_{test}: set of test fluents that models the active individual planning problems
- M: Deterministic Turing Machine (DTM)
- K_M: tape size of a DTM
- Φ_M: planning frame of a DTM
- σ_M: symbol in a tape position of a DTM
- x_M: input string of a DTM
- P^x_M: DTM planning problem
- Π_M: DTM planning program

Planning Programs with Procedures

- $\mathcal{I}_{call}$: set of call instructions
- m: number of procedures
- Θ: set of planning programs defined by Φ
- F_L: subset of local fluents

- Π: planning program with procedures defined by $\langle \Theta, F_L \rangle$
- Π^0: main procedure
- Π^m: m-th procedure
- $s|_{F'}$: projection of s onto subset of fluents F'
- s_g: global state
- s_l: local state
- Ξ: call stack
- $\Xi \oplus (i, j, s_l)$: call stack with top element (i, j, s_l)
- ℓ: call stack size
- (s_g, Ξ): program state
- Ψ: set of predicates
- Ω_v: variable objects (a variable object is v)
- Ω_x: value objects(a value objects is x)
- F_K: subset of local fluents induced by the predicate assign
- $ar(j)$: arity of procedure Π^j
- $\varpi(j)$: parameter list with $ar(j)$ variable objects
- k: current stack level
- F_L^ℓ: set of local fluents for each stack level
- f^k: local fluent in the stack level k
- F_Q: subset of stackable fluents
- $P_{n,m}^\ell$: a planning problem bounded with n lines, m procedures and a stack size of ℓ
- F_{top}^ℓ: set of fluents that represent the top level of the stack
- $I_{t,g}$: initial global state of P_t
- $I_{t,l}^1$: initial local state of P_t in the stack level 1

- F_{pc}^{ℓ}: set of program counters and procedures fluents for each stack level
- F_{ins}^{m}: set of fluents that represent the instructions (including call instructions) of the main and auxiliary procedures.
- $\Gamma(j')$: variable from a parameter list
- $call_{i,j}^{j',k}\Gamma(j')$: action that copies each variable in the next stack level

Non-deterministic Planning Programs

- $\mathcal{I}_{choice}$: set of choice instructions
- χ: list of value objects that can be used to unify a predicate with the current state

Hierarchical Finite State Controllers

- Δ: *Mealy machine* or finite state transducer
- Q: finite set of controller states
- q_0: initial controller state
- $Q_\perp$: subset of terminal controller states
- Σ: is a finite set of input symbols or *input alphabet*
- Λ: is a finite set of output symbols or *output alphabet*
- Υ: transition function that maps a controller state and input symbol to the corresponding next controller state
- Γ: output function that maps a controller state and input symbol to the corresponding output symbol
- σ: input symbol
- λ: output symbol
- $\mathcal{C}$: FSC composed of a Mealy machine and an observation function.
- O: observation function that maps a planning state to an input symbol.
- (q, s): world state (FSCs)
- $\mathcal{O}$: set of observation functions that maps a classical planning state of a test to an input symbol

- φ: maps a controller state into a fluent from the given planning problem
- $a_\perp$: no-op action
- F_{fun}: set of fluents that encode the conditions, transitions and actions between the controller states. It also encodes the empty φ, Υ and Γ (to be programmed).
- F_{aux}: set of fluents corresponding to current state, evaluation of conditions and if an action has been applied.
- b: boolean result of evaluating a condition
- pcond_q^f: program condition as fluent f in controller state q
- $\mathsf{pact}_{q,a}^b$: program action a in controller state q when evaluated to b
- $\mathsf{psucc}_{q,q'}^b$: program successor controller state q' in q when evaluated to b
- econd_q^f: execute condition (evaluate)
- $\mathsf{eact}_{q,a}^b$: execute action (apply sequential action)
- $\mathsf{esucc}_{q,q'}^b$: execute successor (move to new controller state q')
- $\mathcal{H}$: hierarchical finite state controller (HFSC)
- $\mathfrak{C}$: set of FSCs
- $\mathcal{C}_1$: root FSC
- $\mathcal{Z}$: set of FSC calls
- $P_{n,m}^\ell$: planning problem compilation (HFSC) with stack size ℓ, n controller states per FSC, and m FSCs
- F_r: set of fluents instantiated from predicates that are different from assign.
- F_a^ℓ: set of fluents instantiated from assign predicate for each stack level
- F_{fun}^m: copy of each F_{fun} fluent for each FSC
- F_{aux}^ℓ: copy of each F_{aux} fluent for each stack level
- F_H: set of fluents to indicate the level in the call stack, the FSC executing in each stack level, and call fluents

- $\mathsf{pcall}_{q,j}^{b,i,k}$: program call action in controller state q, stack level k, when evaluated as b. The current FSC is $\mathcal{C}_i$ that calls FSC $\mathcal{C}_j$.
- $\mathsf{ecall}_{q,j}^{b,i,k}$: execute call action
- $\mathsf{term}^{i,k}$: popping $\mathcal{C}_i$ from stack level k
- $\mathcal{C}_j[p]$: j-th FSC with argument p
- p^r: value of FSC argument
- $\mathcal{L}_j^r$: parameter of FSC $\mathcal{C}_j$ with an assigned copy of p^r

Unsupervised Classification of Planning Instances

- $\mathcal{E}$: unlabeled set of planning instances
- $\Gamma(\Phi)$: function over the planning frame that yields to the set of all possible planning instances
- m: number of clusters (unsupervised classification)
- $\langle \Phi, \mathcal{E}, m \rangle$: unsupervised classification task
- $\mathcal{G}$: set of generalized plans (set of classes where each class correspond to a planning program)
- C_L: class labels
- $cf : \mathcal{E} \rightarrow C_L$: non-deterministic choice function
- $\langle \mathcal{G}, cf \rangle$: model for the unsupervised classification task
- $\mathcal{X}(\Pi(P))$: set of expanded states when planning program $\Pi(P)$ is executed
- $\delta(s)$: counting function of true assignments of fluents in state s
- X: finite set of variables
- $D(x_i)$: finite domain of values for variable x_i

Generating Context-Free Grammars using Classical Planning

- $\mathcal{F}$: formal language, i.e. a context-free grammar
- $\mathcal{T}$: an alphabet or set of terminal symbols
- $\mathcal{R}$: set of syntactic rules of a formal grammar

- V: set of non-terminal symbols
- ϵ: empty string
- v_0: initial non-terminal symbol that represents the whole grammar
- ζ: a definition rule that contains a non-terminal symbol in the left-hand side and an expression in the right-hand side.
- $\mathcal{L}(\mathcal{F})$: the language of a context-free grammar. It is the whole set of possible strings generated from an initial non-terminal symbol.
- e: string from a CFG language $\mathcal{L}(\mathcal{F})$
- $\mathcal{K}_{\mathcal{F},e}$: parse tree of string e using a formal language $\mathcal{F}$
- τ: a terminal symbol
- $\mathcal{E}$: finite set of input strings
- m: bound in the number of non-terminal symbols (similar to the number of procedures)
- $\langle \mathcal{T}, \mathcal{E}, m \rangle$: context-free grammar generation task
- parse_τ: action that recognizes the current terminal symbol τ in the string
- $\mathcal{I}_{choice}$: set of choice instructions
- $\mathsf{Target} \subseteq \{1, \ldots, n\}$: subset of possible target program lines
- $P^{\ell,z}_{n,m}$: classical planning problem compilation that represents a CFG generation task
- $\mathsf{ins}_{0,j,choose}$: choose instruction programmed in line 0 of a planning program procedure Π_j
- $\mathsf{choose}(\mathsf{Target})^k_{i',j}$: action to choose a Target program line i', in a planning program procedure Π_j and stack level k
- π_e: classical plan that solves the parsing problem of an input string e
- $\mathsf{produce}_\tau$: action that generates a terminal symbol τ in the current position of a string

Bibliography

Albore, A., Palacios, H., and Geffner, H. (2009). A translation-based approach to contingent planning. In *International Joint Conference on Artificial Intelligence*.

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

Alford, R., Kuter, U., and Nau, D. S. (2009). Translating htns to PDDL: A small amount of domain knowledge can go a long way. In *International Joint Conference on Artificial Intelligence*, pages 1629–1634.

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

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

Bäckström, C. and Nebel, B. (1995). Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655.

Baier, J. and McIlraith, S. (2006). Planning with Temporally Extended Goals Using Heuristic Search. In *International Conference on Automated Planning and Scheduling*.

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

Bakir, G., Taskar, B., Hofmann, T., Schölkopf, B., Smola, A., and Vishwanathan, S. (2007). *Predicting Structured Data*. MIT Press.

Barrett, C. and Tinelli, C. (2018). Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). The agile manifesto.

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

Bonet, B. and Geffner, H. (2015). Policies that generalize: Solving many planning problems with the same policy. *Proc IJCAI 2015*.

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

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

Botea, A., Enzenberger, M., MÃ$\frac{1}{4}$ller, M., and Schaeffer, J. (2005). Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621.

Brenner, M. (2003). A multiagent planning language. In *Proc. of the Workshop on PDDL, ICAPS*, volume 3, pages 33–38.

Brooks, R. (1989). A robot that walks; emergent behaviours from a carefully evolved network. *Neural Computation*, 1:253–262.

Buckland, M. (2004). *Programming Game AI by Example*. Wordware Publishing, Inc.

Bylander, T. (1994). The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204.

Chen, Y., Huang, R., and Zhang, W. (2008). Fast planning by search in domain transition graph. In *AAAI Conference on Artificial Intelligence*, pages 886–891.

Chentanez, N., Barto, A. G., and Singh, S. P. (2005). Intrinsically motivated reinforcement learning. In *Advances in neural information processing systems*, pages 1281–1288.

Chomsky, N. (2002). *Syntactic structures*. Walter de Gruyter.

Cimatti, A., Roveri, M., and 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., and Röger, G. (2008). Integrating golog and planning: An empirical evaluation. In *Non-Monotonic Reasoning Workshop. Sydney, Australia*. Citeseer.

Coles, A. and Smith, A. (2007). Marvin: A heuristic search planner with online macro-action learning. *J. Artif. Intell. Res.(JAIR)*, 28:119–156.

Cresswell, S. and Coddington, A. M. (2004). Compilation of ltl goal formulas into pddl. In *ECAI*, pages 985–986.

Cresswell, S. N., McCluskey, T. L., and West, M. M. (2013). Acquiring planning domain models using locm. *The Knowledge Engineering Review*, 28(2):195–213.

De Giacomo, G., Gerevini, A. E., Patrizi, F., Saetti, A., and Sardina, S. (2016). Agent planning programs. *Artificial Intelligence*, 231:64–106.

De Giacomo, G. and Vardi, M. Y. (2015). Synthesis for ltl and ldl on finite traces. In *IJCAI*, volume 15, pages 1558–1564.

De la Higuera, C. (2010). *Grammatical inference: learning automata and grammars*. Cambridge University Press.

De la Rosa, T., Jiménez, S., Fuentetaja, R., and Borrajo, D. (2011). Scaling up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research*, 40:767–813.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Domshlak, C. (2013). Fault tolerant planning: Complexity and compilation. In *ICAPS*.

Edelkamp, S. and Hoffmann, J. (2004). Pddl2. 2: The language for the classical part of the 4th international planning competition. *4th International Planning Competition (IPC'04), at ICAPS'04*.

Ellis, K., Solar-Lezama, A., and Tenenbaum, J. (2015). Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems 28*.

Erol, K., Hendler, J., and Nau, D. S. (1996). Complexity results for htn planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93.

Fern, A., Yoon, S. W., and Givan, R. (2004). Learning domain-specific control knowledge from random walks. In *ICAPS*, pages 191–199.

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

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208.

Finkbeiner, B. and Schewe, S. (2013). Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539.

Fox, M., Gerevini, A., Long, D., and Serina, I. (2006). Plan stability: Replanning versus plan repair. In *ICAPS*, volume 6, pages 212–221.

Fox, M. and Long, D. (2003). Pddl2. 1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)*, 20:61–124.

Frances, G. and Geffner, H. (2016). $\exists$-strips: Existential quantification in planning and constraint satisfaction. In *IJCAI*, pages 3082–3088.

Fritz, C., Baier, J. A., and McIlraith, S. A. (2008). Congolog, sin trans: Compiling congolog into basic action theories for planning and beyond. In *KR*, pages 600–610.

Fuentetaja, R. and Borrajo, D. (2006). Improving control-knowledge acquisition for planning by active learning. In *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings*, pages 138–149.

Geffner, H. (2000). Functional strips: a more flexible language for planning and problem solving. In *Logic-based artificial intelligence*, pages 187–209. Springer.

Geffner, H. and 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. and Long, D. (2006). Preferences and soft constraints in pddl3. In *ICAPS workshop on planning with preferences and soft constraints*, pages 46–53.

Gold, E. M. (1967). Language identification in the limit. *Information and control*, 10(5):447–474.

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

Gulwani, S., Polozov, O., Singh, R., et al. (2017). Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

Helmert, M. (2006a). The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, 26:191–246.

Helmert, M. (2006b). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246.

Hickmott, S., Rintanen, J., Thiébaux, S., and White, L. (2007). Planning via Petri Net Unfolding. In *International joint conference on Artificial intelligence*, pages 1904–1911.

Hoffmann, J. (2001). Ff: The fast-forward planning system. *AI magazine*, 22(3):57.

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

Hogg, C., Munoz-Avila, H., and Kuter, U. (2008). Htn-maker: Learning htns with minimal additional knowledge engineering required. In *AAAI*, pages 950–956.

Howey, R., Long, D., and Fox, M. (2004). Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 294–301. IEEE.

Hu, Y. and De Giacomo, G. (2011). Generalized planning: Synthesizing plans that work for multiple environments. In *International Joint Conference on Artificial Intelligence*, pages 918–923.

Hu, Y. and De Giacomo, G. (2013). A generic technique for synthesizing bounded finite-state controllers. In *International Conference on Automated Planning and Scheduling*.

Hu, Y. and Levesque, H. J. (2011). A correctness result for reasoning about one-dimensional planning problems. In *International Joint Conference on Artificial Intelligence*, pages 2638–2643.

Ivankovic, F. and Haslum, P. (2015). Optimal planning with axioms. In *International Joint Conference on Artificial Intelligence*, pages 1580–1586. AAAI Press.

Jiménez, S., De La Rosa, T., Fernández, S., Fernández, F., and Borrajo, D. (2012). A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(04):433–467.

Jiménez, S. and Jonsson, A. (2015). Computing Plans with Control Flow and Procedures Using a Classical Planner. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS-15*, pages 62–69.

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

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

Koller, A. and Hoffmann, J. (2010). Waking up a sleeping rabbit: On natural-language sentence generation with ff. In *ICAPS*, pages 238–241.

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

Lang, J. and Zanuttini, B. (2012). Knowledge-based programs as plans-the complexity of plan verification. In *Proc. 20th Conference on European Conference on Artificial Intelligence (ECAI 2012)*, pages 6–p.

Langley, P. and Stromsten, S. (2000). Learning context-free grammars with a simplicity bias. In *European Conference on Machine Learning*, pages 220–228. Springer.

Levesque, H. J. (2005). Planning with loops. In *International Joint Conference on Artificial Intelligence*, pages 509–515.

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

Lipovetzky, N. and Geffner, H. (2012). Width and serialization of classical planning problems. *HSDIP 2012*, page 9.

Lipovetzky, N. and Geffner, H. (2017). Best-first width search: Exploration and exploitation in classical planning. In *AAAI*, pages 3590–3596.

Liu, M., Jiang, X., and Kot, A. C. (2009). A multi-prototype clustering algorithm. *Pattern Recognition*, 42(5):689–698.

Long, D. and Fox, M. (2003). The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res.(JAIR)*, 20:1–59.

Long, D., Kautz, H., Selman, B., Bonet, B., Geffner, H., Koehler, J., Brenner, M., Hoffmann, J., Rittinger, F., Anderson, C. R., et al. (2000). The aips-98 planning competition. *AI magazine*, 21(2):13.

Lotinac, D. and Jonsson, A. (2016). Constructing hierarchical task models using invariance analysis. In *European Conference on Artificial Intelligence*.

Lotinac, D., Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2016). Automatic generation of high-level state features for generalized planning. In *International Joint Conference on Artificial Intelligence*.

Marthi, B., Russell, S., and Wolfe, J. (2007). Angelic Semantics for High-Level Actions.

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

McCarthy, J. (1963). Situations, actions, and causal laws. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

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

Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (2013). *Machine learning: An artificial intelligence approach*. Springer Science & Business Media.

Mitchell, T. M. (1982). Generalization as search. *Artificial intelligence*, 18:203–226.

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

Muggleton, S. H., Lin, D., Pahlavi, N., and Tamaddoni-Nezhad, A. (2014). Meta-interpretive learning: application to grammatical inference. *Machine learning*, 94(1):25–49.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *J. Artif. Intell. Res.(JAIR)*, 20:379–404.

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

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

Parr, R. and Russell, S. J. (1998). Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*, pages 1043–1049.

Patrizi, F., Lipoveztky, N., De Giacomo, G., and Geffner, H. (2011). Computing infinite plans for ltl goals using a classical planner. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

Pednault, E. P. (1994). Adl and the state-transition model of action. *Journal of logic and computation*, 4(5):467–512.

Ramírez, M. and Geffner, H. (2009). Plan recognition as planning. In *Proceedings of the 21st international joint conference on Artifical intelligence. Morgan Kaufmann Publishers Inc*, pages 1778–1783.

Ramírez, M. and Geffner, H. (2010). Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the Conference of the Association for the Advancement of Artificial Intelligence (AAAI 2010)*, pages 1121–1126.

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

Ramirez, M., Lipovetzky, N., and Muise, C. (2015). Lightweight Automated Planning ToolKiT. http://lapkt.org/. Accessed: 2017-05-15.

Ravi, N., Dandekar, N., Mysore, P., and Littman, M. L. (2005). Activity recognition from accelerometer data. In *Aaai*, volume 5, pages 1541–1546.

Richter, S. and Westphal, M. (2010). The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39:127–177.

Röger, G., Helmert, M., and Nebel, B. (2008). On the relative expressiveness of adl and golog: The last piece in the puzzle. In *KR*, pages 544–550.

Russell, S., Norvig, P., and Intelligence, A. (1995). A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27.

Sakakibara, Y. (1992). Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60.

Sardina, S., De Giacomo, G., Lespérance, Y., and 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.

Schewe, S. and Finkbeiner, B. (2007). Bounded synthesis. In *International Symposium on Automated Technology for Verification and Analysis*, pages 474–488. Springer.

Schwenger, M., Torralba, A., Hoffmann, J., Howcroft, D. M., and Demberg, V. (2016). From openccg to ai planning: Detecting infeasible edges in sentence generation. In *International Conference on Computational Linguistics*.

Shavlik, I. W. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5(1):39–70.

Shivashankar, V., Kuter, U., Nau, D., and Alford, R. (2012). A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 981–988. International Foundation for Autonomous Agents and Multiagent Systems.

Solar-Lezama, A. (2008). *Program Synthesis By Sketching*. PhD thesis.

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

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

Srivastava, S., Immerman, N., Zilberstein, S., and Zhang, T. (2011b). Directed search for generalized plans using classical planners. In *International Conference on Automated Planning and Scheduling*, pages 226–233.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.

Toropila, D. and Barták, R. (2010). Using Finite-State Automata to Model and Solve Planning Problems. In *Proceedings of the 11th Italian AI Symposium on Artificial Intelligence (AI*IA)*, pages 183–189.

Vajda, S. (2009). *Mathematical programming*. Courier Corporation.

Vallati, M., Chrpa, L., Grzes, M., McCluskey, T. L., Roberts, M., and 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., and Blythe, J. (1995). Integrating planning and learning: The prodigy architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):81–120.

Winner, E. and Veloso, M. (2003). Distill: Learning domain-specific planners by example. In *International Conference on Machine Learning*, pages 800–807.

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

Younes, H. L. and Littman, M. L. (2004). Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*.

Zimmerman, T. and Kambhampati, S. (2003). Learning-assisted automated planning: looking back, taking stock, going forward. *AI Magazine*, 24(2):73.