Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática, 2018

Universidade do Porto Faculdade de Ciências, 2018

Universidade do Minho Departamento de Informática, 2018

VAHID MOKHTARI

RECOLHA E CONCEITUALIZAÇÃO DE EXPERIÊNCIAS DE ATIVIDADES ROBÓTICAS BASEADAS EM PLANOS PARA MELHORIA DE COMPETÊNCIAS NO LONGO PRAZO

GATHERING AND CONCEPTUALIZING PLAN-BASED ROBOT ACTIVITY EXPERIENCES FOR LONG-TERM COMPETENCE ENHANCEMENT

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática, 2018

Universidade do Porto Faculdade de Ciências, 2018

Universidade do Minho Departamento de Informática, 2018

VAHID MOKHTARI

RECOLHA E CONCEITUALIZAÇÃO DE EXPERIÊNCIAS DE ATIVIDADES ROBÓTICAS BASEADAS EM PLANOS PARA MELHORIA DE COMPETÊNCIAS NO LONGO PRAZO

GATHERING AND CONCEPTUALIZING PLAN-BASED ROBOT ACTIVITY EXPERIENCES FOR LONG-TERM COMPETENCE ENHANCEMENT

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Doutor em Engenharia Informática, realizada sob a orientação científica dos professores Luís Seabra Lopes e Armando Pinho, do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

O Júri / The Jury

Presidente / President	**Doutor Joaquim Manuel Vieira** Professor Catedrático, Universidade de Aveiro
Vogais / Examiners Committee	**Doutor Joachim Hertzberg** Professor Catedrático, Universidade Osnabrück, Alemanha
	Doutor Manuel Cabido Lopes Professor Associado, Instituto Superior Técnico, Universidade de Lisboa
	Doutor Paulo Martins de Carvalho Professor Associado, Universidade do Minho
	Doutor Luís Filipe de Seabra Lopes (Orientador/Supervisor) Professor Associado, Universidade de Aveiro
	Doutor José Nuno Panelas Lau Professor Auxiliar, Universidade de Aveiro

Acknowledgments / Agradecimentos

This work would not have been possible without the support and guidance of Prof. Luís Seabra Lopes, who encouraged me to explore many directions of this research during all the years of my PhD. I would like to express my deepest appreciation for his continual guidance and persistent help, without which this dissertation would have been incomplete.

I am grateful to Prof. Armando José Pinho for his enthusiasm and support for this direction of work. Armando's patience, encouragement and insightful advice were key in taking this entire direction of research from its inception to the results.

I am thankful to my colleagues from the EU RACE project who acquainted me with ontology-based robotics and provided the initial context and demonstration scenarios for this work. I thank the Foundation for Science and Technology Portugal – FCT for funding this work under grant SFRH/BD/94184/2013. I also express my profound gratitude to the Instituto de Engenharia Electrónica e Telemática de Aveiro – IEETA for hosting and supporting this work.

I would also like to thank Prof. Roman Manevich, University of Texas at Austin, who introduced me in the field of 3-valued logic and helped me to integrate and explore that topic in this work.

I am also thankful to Prof. Daniele Magazzeni, who kindly hosted me at King's College London and helped me to extend my work in new directions of the research within his research group.

My wife, Parinaz, has been a constant source of inspiration for my studies. Without her perspective and support, this work would not have reached fruition. I am also grateful to my parents, my brothers and my sisters for their support and encouragement.

I would also like to thank my friend, Hamidreza Kasaei, whose company allowed me to continue this work with enthusiasm.

Dedication / Dedicação

To my parents, brothers and sisters, for their endless love, support and encouragement.

To my beloved wife Parinaz, without whose encouragement this may never have started, and to my sweet daughter Niki.

Palavras-chave Domínios de Planeamento Baseados na Experiência; Aprendizagem Automática Planeamento Automático Representação do Conhecimento e Raciocínio;

Resumo Aprendizagem de robôs é uma direção de pesquisa proeminente em robótica inteligente. Em robótica, é necessário lidar com a questão da integração de várias tecnologias, como percepção, planeamento, atuação e aprendizagem. Na aprendizagem de robôs, o objetivo a longo prazo é desenvolver robôs que aprendem a executar tarefas e melhoram continuamente os seus conhecimentos e habilidades através da observação e exploração do ambiente e interação com os utilizadores. A investigação tem-se centrado na aprendizagem de comportamentos básicos, ao passo que a aprendizagem de representações de atividades de alto nível, que se decompõem em sequências de ações, e de classes de actividades, não tem sido suficientemente abordada. A aprendizagem ao nível da tarefa é fundamental para aumentar a autonomia e a flexibilidade dos robôs. O conhecimento de alto nível permite tornar o software dos robôs menos dependente da plataforma e facilita a troca de conhecimento entre robôs diferentes.

O objetivo desta tese é contribuir para o desenvolvimento de capacidades cognitivas para robôs, incluindo aquisição supervisionada de experiência através da interação humano-robô, aprendizagem de tarefas de alto nível com base nas experiências acumuladas e planeamento de tarefas usando o conhecimento adquirido. Propõe-se uma abordagem que integra diversas funcionalidades cognitivas para aprendizagem e reprodução de aspetos de alto nível detetados nas experiências acumuladas. Em particular, nós propomos e formalizamos a noção de Domínio de Planeamento Baseado na Experiência (Experience-Based Planning Domain, or EBPD) para aprendizagem e planeamento num âmbito temporal alargado. Uma interface para interação humano-robô é usada para fornecer ao robô instruções passo-a-passo sobre como realizar tarefas. Propõe-se uma abordagem para extrair experiências de atividades baseadas em planos, incluindo as percepções relevantes e as ações executadas pelo robô.

Uma metodologia de conceitualização é apresentada para a aquisição de conhecimento de tarefa na forma de schemata a partir de experiências. São utilizadas diferentes técnicas, incluindo generalização dedutiva, diferentes formas de abstracção e extração de características. A metodologia inclui detecção de ciclos, inferência de âmbito de aplicação e inferência de objetivos. A resolução de problemas em EBPDs é alcançada usando um sistema de planeamento com duas camadas, uma para planeamento abstrato, aplicando um schema aprendido, e outra para planeamento detalhado.

A arquitetura e os métodos de aprendizagem e planeamento são aplicados e avaliados em vários cenários reais e simulados. Finalmente, os métodos de aprendizagem desenvolvidos são comparados e as condições onde cada um deles tem melhor aplicabilidade são discutidos.

Keywords Experience-Based Planning Domains; Machine Learning; Automated Planning; Knowledge Representation and Reasoning;

Abstract Robot learning is a prominent research direction in intelligent robotics. Robotics involves dealing with the issue of integration of multiple technologies, such as sensing, planning, acting, and learning. In robot learning, the long term goal is to develop robots that learn to perform tasks and continuously improve their knowledge and skills through observation and exploration of the environment and interaction with users. While significant research has been performed in the area of learning motor behavior primitives, the topic of learning high-level representations of activities and classes of activities that, decompose into sequences of actions, has not been sufficiently addressed. Learning at the task level is key to increase the robots′ autonomy and flexibility. High-level task knowledge is essential for intelligent robotics since it makes robot programs less dependent on the platform and eases knowledge exchange between robots with different kinematics.

The goal of this thesis is to contribute to the development of cognitive robotic capabilities, including supervised experience acquisition through human-robot interaction, high-level task learning from the acquired experiences, and task planning using the acquired task knowledge. A framework containing the required cognitive functions for learning and reproduction of high-level aspects of experiences is proposed. In particular, we propose and formalize the notion of Experience-Based Planning Domains (EBPDs) for long-term learning and planning. A human-robot interaction interface is used to provide a robot with step-by-step instructions on how to perform tasks. Approaches to recording plan-based robot activity experiences including relevant perceptions of the environment and actions taken by the robot are presented. A conceptualization methodology is presented for acquiring task knowledge in the form of activity schemata from experiences. The conceptualization approach is a combination of different techniques including deductive generalization, different forms of abstraction and feature extraction. Conceptualization includes loop detection, scope inference and goal inference. Problem solving in EBPDs is achieved using a two-layer problem solver comprising an abstract planner, to derive an abstract solution for a given task problem by applying a learned activity schema, and a concrete planner, to refine the abstract solution towards a concrete solution.

The architecture and the learning and planning methods are applied and evaluated in several real and simulated world scenarios. Finally, the developed learning methods are compared, and conditions where each of them has better applicability are discussed.

Contents

List of Figures

List of Tables

List of Notations

Notation	*Meaning*
a, A	Action, set of actions
$\mathcal{A}$	Set of abstraction hierarchies
$\texttt{abs}(x)$	Return the absolute value of a real number x
$\texttt{args}(x)$	Set of arguments of a predicate or functional expression x
ASBP	Abstract Schema-Based Planner
B	Effective branching factor
β	Canonical abstraction function
C	2-valued logical structure/concrete structure
$\texttt{canon}(u)$	The canonical name of an object u
CNLCP	Contiguous Non-overlapping Longest Common Prefix array
$\mathcal{D}$	Panning domain
$\mathcal{D}_a$, $\mathcal{D}_c$	Abstract panning domain, concrete panning domain
Δ	Experience-Based Planning Domain
$\texttt{during}(p)$	Functional expression denoting that predicate p holds true during an experience
E	Effect of a planning operator
E^+, E^-	Positive effect, negative effect
$\mathcal{E}$	Ego nodes
e, $\mathcal{E}$	Experience, set of experiences
e_a	Abstracted experience
$\texttt{end}(p)$	Functional expression denoting that predicate p holds true in the final state of an experience
F	Set of features
g	Goal (set of predicates representing goal)
G	Unground goal

$\gamma(s, a)$	State-transition function (the state produced by applying a to s)
`goal(p)`	Functional expression denoting that the goal of an activity schema includes predicate p
h	Head (a functional expression of the form $n(x_1, \ldots, x_k)$)
h^+	Relaxation heuristic function
`head(`S`)`	Return the head (front) of a sequence S
ι	Interpretation function mapping predicates to their truth-values
`init(p)`	Functional expression denoting that predicate p holds true in the initial state of an experience
K, K_a	Key-properties, abstract key-properties
$\mathcal{L}$	First-order language
$\mathcal{L}_a$, $\mathcal{L}_c$	First-order language of the abstract domain, first-order language of the concrete domain
LCP	Longest Common Prefix array
`lcp(`A, B`)`	Return the longest common prefix of two strings A and B
`len(`S`)`	Return the length of a given string S
$\mathcal{N}$	Neighbors of egos
NLCP	Non-overlapping Longest Common Prefix array
`nlcp(`A, B`)`	Return the non-overlapping longest common prefix of two strings A and B
m, $\mathcal{M}$	Activity schema, set of activity schemata
`maybe(p)`	p represents an indefinite ($1/2$-valued) key-property in a 3-valued logical structure (p is a key-property that may exist in a 2-valued logical structure represented by the 3-valued logical structure)
`min(`L`)`	Return the minimum number in a given list L
o, $\mathcal{O}$	Planning operator, set of planning operators
$\mathcal{O}_a$, $\mathcal{O}_c$	Set of planning operators of the abstract domain, set of planning operators of the concrete domain
Ω, Ω_0	Enriched abstract plan, initial enriched abstract plan
ω	Enriched abstract operator
P	Precondition of a planning operator
P^+, P^-	Positive precondition, negative precondition
$\mathcal{P}$, $\mathcal{P}_a$	Task planning problem, abstracted task planning problem
$\mathcal{P}$	Penetrance ratio

π, π_a	Plan, abstract plan
$\texttt{parent}(x)$	Return the parent of a concept x in an abstraction hierarchy
R	Average branching factor
S	Static precondition of a planning operator
s, $\mathcal{S}$	State, set of states
$\mathcal{S}_a$, $\mathcal{S}_c$	Set of states of the abstract domain, set of states of the concrete domain
s_0, s_g	Initial state, goal state
$\mathfrak{S}$	Scope of an activity schema/3-valued logical structure/abstract structure
SA	Suffix array
SBP	Schema-Based Planner
Σ	Static world information
Σ_a, Σ_c	Static world information of the abstract domain, static world information of the concrete domain
σ	Subset of static world information
$\texttt{summary}(o)$	Object o is a summary object in a 3-valued logical structure (an object that corresponds to two or more objects in a 2-valued logical structure represented by the 3-valued logical structure)
t	Task
τ	Temporal symbol of a predicate ($during$, $init$ and end)
$\texttt{tail}(S)$	Return the tail (remaining) of a sequence S
U	Universe (set of objects)
$\sqsubseteq$	The embedding function, e.g., $C \sqsubseteq \mathfrak{S}$ denotes C is embedded in $\mathfrak{S}$
$\preceq$	The information order of two Kleene's 3-valued propositions, e.g., $p \preceq q$ denotes p has more definite information than q
$\sqcup$	The join (least-upper-bound) operation of two Kleene's 3-valued propositions with respect to $\preceq$ (denoted by e.g., $p \sqcup q$)
$\lvert A \rvert$	Return the number of elements in a set A (the cardinality of A)
$(a_1, \ldots, a_k)$	k-tuple (k is fixed)
$\langle a_1, \ldots, a_n \rangle$	Sequence of length n (n may vary)
$\cdot$	Concatenation
$\varnothing$	nil
$\emptyset$	Empty set/sequence

Chapter 1

Introduction

Recent research into robotics shows growing trends towards employing robots in scenarios different from the conventional industrial applications. Early robots only worked in controlled environments of laboratories and factories. They had been preprogrammed to receive accurate information about the environment and perform certain tasks. Reprogramming such robots was often a costly process requiring an expert. In order for robots to become part of our everyday life, the ability to learn new knowledge is essential. Enabling robots to learn from a human user by demonstrating how to achieve tasks would simplify the robots' installation and reprogramming. In a longer time perspective, the vision is that robots will move out of factories into our homes and offices. Such robots should be able to learn how to perform real-world tasks such as set a table, or fill the dishwasher. They may require interaction with humans and ingenious programming approaches to adapt and operate in dynamic environments. Clearly, classical approaches do not meet all the new requirements that human-robot interaction and changing environments demand. This is the reason why robot learning is one of the key research areas in robotics. Constructing a robot that is able to learn what is shown is a challenging problem. While significant research has been performed in the area of learning motor behavior primitives from a teacher's demonstration, the topic of learning high-level task knowledge has not been sufficiently addressed. In learning of high-level task knowledge, the main challenge for the learner is to extract all the necessary information pertaining to the task, eliminate all the observations that are irrelevant and abstract and generalize the correct task representation in the case when single or few demonstrations are given. In this thesis, we are, in particular, interested in learning of task models from examples provided by a human expert. This thesis addresses this problem and proposes new methodologies that enable the acquisition of task knowledge from a single demonstration of a

task, and the exploitation of the acquired knowledge for problem solving. This work is demonstrated in different scenarios in real robot platforms and simulated domains.

This chapter elaborates on the motivations of this thesis in Section 1.1. The thesis objectives are addressed in Section 1.2. Section 1.3 presents the research context and the used cognitive robotic platform in this thesis. The thesis outline and its contributions are presented in Section 1.4. Finally, section 1.5 lists the publications of the thesis' novelties and results.

1.1 Motivation

A truly functional autonomous robot must have the ability to learn from its own experiences. It should not rely on a human programmer once it is started up, but rather, it must be trainable. Experiences provide a rich resource for learning, solving problems, avoiding difficulties, predicting the effects of activities, and obtaining commonsense insights. Current robots do not, in general, possess this ability, and this is a decisive reason for the often perceived "lack of intelligence" of current robotic systems: they repeat mistakes, do not learn to anticipate happenings in their environment, and need detailed instructions for each specific task.

> Consider an everyday task of a service robot, such as grasping a cup from a cupboard and bringing it to a guest sitting at a table. This task may occur in different variations. For example, guests may sit at different sides of a table, guests may be served with a restaurant tea service including a teapot, cups, and plates, etc. It is clearly infeasible to provide the robot with precise instructions for all contingencies at design time or to specify tasks with highly detailed instructions for each particular concrete situation which may arise. Hence without such knowledge, robot behavior is bound to lack robustness if the robot cannot autonomously adapt to new situations.

Intelligent robotics is concerned with the integration of different approaches to make robots learn from experiences and human teachers, and reason about how to deal with new environments. This involves to have robots with cognitive capabilities allowing them to understand, analyze and react according to the interaction with a human (Seabra Lopes and Connell, 2001). Learning from Demonstration (LfD) is one approach in which a robot can generalize and learn a specific task

from an experience, provided by a human demonstrator (Billard et al., 2008; Argall et al., 2009). LfD provides a powerful way to speed up learning new skills. LfD in robotics can be broadly divided into two levels of abstraction (Billard et al., 2008). A *low-level* representation of the skill, taking the form of a non-linear mapping of sensory-motor information that produces an action to be performed by actuators, also referred to as *trajectories encoding*. These mappings can produce the same trajectories as observed during demonstrations or might be adapted to the robot's morphology but still result in the same actions. Many studies have addressed the problem of low-level learning and reproduction of behaviors (Ekvall and Kragic, 2005; Skoglund et al., 2010; Ijspeert et al., 2013). Another aspect of learning is related to a *high-level* representation of the skill, where the skill is decomposed in a sequence of action-perception units, also referred to as *conceptualization* or *symbolic encoding*. Various techniques for learning high-level behaviors have been developed (Kuniyoshi et al., 1994; Friedrich et al., 1996; Nicolescu and Mataric, 2003; Ekvall and Kragic, 2006). While most works in the field have focused on the learning of low-level skills and kinematics of motions, little work concerns learning of high-level task knowledge and representation, i.e., symbolic learning of planning models and high-level task compositions, and make use of the acquired knowledge for problem solving. High-level task learning makes robot programs independent from the platform and eases their exchange between robots with different kinematics.

The central contribution of this thesis is towards the development of capabilities to acquire high-level task planning models, by conceptualizing past experiences, for solving any particular instances of same tasks. We present a framework for autonomous competence enhancement by teaching a robot how to perform a complex task and conceptualizing that information to generate new task knowledge. In this framework, basic manipulation skills or controlling techniques are not investigated and it is presumed that the robot is equipped with a set of basic skills, e.g., pickup and putdown. By contrast, we focus on a strategy that would help the robot to construct a high-level task representation of a complex task (e.g., serve a coffee to a guest) built from the existing behavior set. This framework is an artificial cognitive system that can be embodied in a robot to build a high-level understanding of robotic tasks by storing and exploiting appropriate memories of its experiences. The experiences are records of past happenings stored by the robot, as observed through its sensors, and interpreted according to the robot's conceptual framework, in a coherent subset of space-time. Recorded experiences

are the potential inputs for conceptualizing past robot activities and forming new concepts. The framework provides the robot with new learning capabilities that increase its ability to extend and utilize past solutions for different instances of the same problem or even for other problems that are to some extent similar. Thus, in this context, robot competence is obtained by generalizing and abstracting experiences, and broadening robot task knowledge.

1.2 Objectives

The core objective of this thesis is defined by the following longstanding problem statement:

> Given an "experience" of a previously achieved task, generate a "high-level task planning model" for "efficiently" solving a "class" of similar tasks.

This thesis heads towards developing novel techniques for experience-based learning, in order to improve concept formation, and methods of problem solving using the learned concepts. The developed methods are part of an architecture that is particularly tailored for learning high-level aspects of task demonstrations. The architecture employs techniques to sequentially learn and reproduce solutions in order to make a robot capable of achieving the tasks in the same way as demonstrated. The architecture uses different learning methods to identify the most important elements of an experience including loops of actions, scope of applicability and goal.

Considering the current state of research in robot learning from experience, and the gaps in learning of high-level task models from former solution plans and experience-based robot task planning, this thesis work aimed to undertake the following missions:

- Development and formalization of a well-defined notion of experience-based task learning and planning to support long-term learning and problem solving in robotics.
- Development of a representation which utilizes the expressiveness of first-order logic, while allowing a heuristic, directed-search based approach for task planning.

- Development of a cognitive framework for teaching tasks to robots and identifying and storing relevant occurrences during the execution of tasks, as part of experiences.
- Development of methods for generating task planning knowledge with loops by extracting useful information from robots' experiences.
- Development of a planning system for efficiently generating solution plans, based on the acquired knowledge, to given task problems.

1.3 Research Context

Our proposed learning and planning system was initially developed in the framework of the European project, RACE: Robustness by Autonomous Competence Enhancement (Rockel et al., 2013; Hertzberg et al., 2014) [1]. The overall aim of the RACE project was: *to develop an artificial cognitive system, embodied by a service robot, able to build a high-level understanding of the world it inhabits by storing and exploiting appropriate memories of its experiences*. In this context, robot competence is obtained by abstracting and generalizing from experiences about objects (Kasaei et al., 2015), scene layouts (Dubba et al., 2014) and activities (Mokhtari et al., 2016b), thus extending task planning and execution beyond preconceived situations. Activities successfully carried out by the robot for specific objects at specific locations may be generalized to activity concepts applicable to a larger variety of objects at variable locations.

In RACE an OWL ontology (Antoniou and Van Harmelen, 2004) provides a common representation for all knowledge possessed by the robot. However, the ontology does not explicitly represent all details of the robot's knowledge base. Some concepts in the ontology may contain more detailed knowledge, expressed in other formalisms at lower levels of abstraction. Overall, RACE adopts an ontology-based, multi-level knowledge-representation framework.

RACE was developed based on an artificial cognitive architecture, as shown in Figure 1.1. In this architecture, an RDF database is used as *Blackboard*, which serves as a *working memory* for computations and communications of all modules. Data stored in the blackboard is mostly of semantic nature, for which an ontology has been developed. It keeps track of the evolution of both the internal state of the robot and the events observed in the environment. The blackboard can be read

[1] http://project-race.eu/

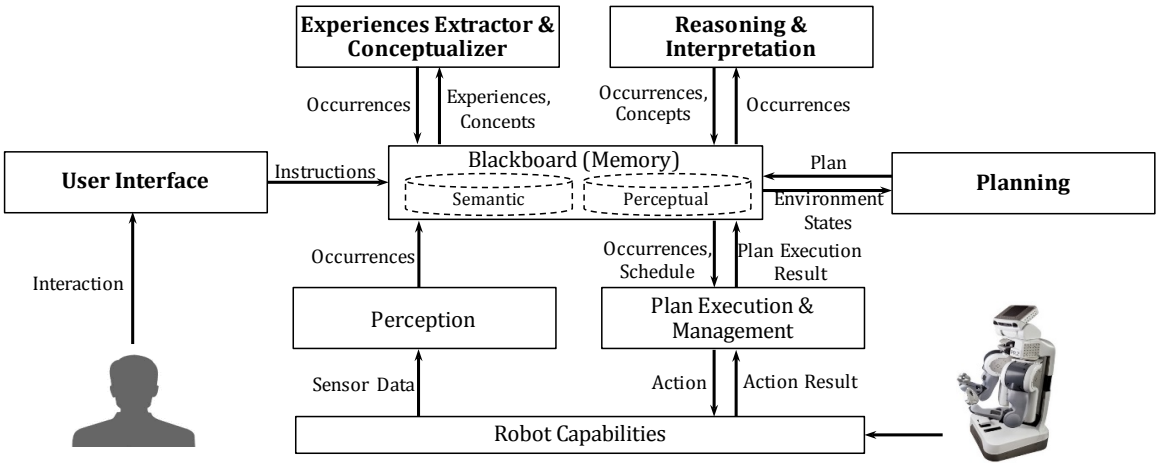


Figure 1.1: The RACE Architecture.

or written by different modules. Every unit of data written to the Blackboard is called *fluent*, which describes an instance of some concept in the ontology.

The *Reasoning and Interpretation* includes high-level interpreter, and temporal and spatial reasoners. *Perception* contains several modules for symbolic proprioception and exteroception which generate occurrences. New concepts are added to the *Memory* and stored in the OWL ontology format. *User Interface* sends instructions that generate a goal which is then relayed to *Planning*. Planning is carried out using JSHOP2, a *Hierarchical Task Network (HTN)* planner (Š. Konečný et al., 2014; Stock et al., 2014). The generated plan is dispatched to the *Plan Execution and Management*, and the robot actuators are finally demanded to perform actions. The history of occurrences in the memory is continuously observed by the *Experience Extractor* to detect and extract potentially relevant experiences, based on plan structure, user feedback and similarity with stored experiences. Experiences are subsequently fed to the *Conceptualizer* to generate and store new concepts or update existing concepts, resulting in more robust and flexible future behavior.

We developed our system within this architecture. We focus on high-level learning of complex behaviors, e.g., serving a coffee to a guest. A complex behavior consists of several sub-behaviors that are executed in sequence. This work involves a study on how to identify relevant occurrences during a robot performance as part of an experience; how to conceptualize acquired experiences to form new concepts; and how to make plans in new situations based on the learned concepts. In this thesis, we address the user interface and experience extractor in Chapter 4, conceptualization in Chapter 5, and planning in Chapter 6. The robot capabilities, plan execution (Š. Konečný et al., 2014) and perception (Kasaei et al., 2015;

Oliveira et al., 2016) are out of scope of this thesis.

1.4 Thesis Outline and Contributions

The development of this thesis may be split in several phases that are addressed and organized in the following chapters:

- **Chapter 2: A Review of Robot Task Learning and Planning**

  In this chapter, we study machine learning techniques which have been successfully applied in robotics applications. In particular, we study existing techniques for learning of high-level task planning knowledge in robotics and which techniques are needed for encoding, representing and reproducing a complex behavior successfully. We also address *Automated Planning*, a branch of Artificial Intelligence that focuses on the computational synthesis of ordered sets of actions to perform given tasks. Planning is a key ability for intelligent robots to increase their autonomy and flexibility. A review of machine learning techniques for planning, related to our thesis, is conducted in this chapter.

- **Chapter 3: Experience-Based Planning Domains**

  The basics of this thesis begin by formalizing the notion of *Experience-Based Planning Domains* (EBPDs). An EBPD is a planning domain definition language, supported by a long-term learning process. EBPDs are intended to endow intelligent robots with the capability of problem solving by learning from experience. In this chapter, we present the EBPD representation scheme which uses notation derived from first-order logic. This formalization captures many interesting aspects of the experience-based learning and planning, which can foster the reuse of this research.

- **Chapter 4: Human-Robot Interaction and the Extraction of Experiences**

  This chapter presents our approach and the infrastructure designed for *supervised experience extraction* of plan-based robot activities. For this purpose, ontological concept representations and a command-line interface are developed to support human-robot interaction, allowing an inexpert user to instruct a robot to perform tasks as well as teach new task concepts. We also present a graph simplification approach, based on the concept of *ego network* (Newman, 2003), to extract the most relevant information in a robot performance as part of an experience.

- **Chapter 5: Learning Planning Knowledge**

  In this chapter, we present a *conceptualization* methodology for abstracting and generalizing robot experiences to generate task planning concepts, also called *activity schemata*. The proposed conceptualization approach is a combination of different techniques including deductive generalization, different forms of abstraction, feature extraction, loop detection, scope inference and goal inference. Knowledge generated through conceptualization is then used for general problem solving and planning.

- **Chapter 6: Planning Using the Learned Knowledge**

  In order to utilize acquired knowledge for problem solving, we propose a two-layer problem solver which includes an *abstract planner* and a *concrete planner*. The abstract planner first derives an abstract solution to a given task problem by following a learned activity schema. Then, the concrete planner refines the abstract solution towards a concrete solution. This chapter presents a planning system for generating a plan solution to a given task problem using a learned activity schema.

- **Chapter 7: Implementation, Demonstration and Evaluation**

  This chapter presents the results of our experiments with different classes of problems in both real and simulated environments. The results involve different aspects of the system in the course of its development. We present two real robotic platforms to demonstrate the utility of our system. The first demonstration is performed in the RACE project on a real robot PR2, and the second demonstration is performed in a robotic arm platform. We also evaluate the performance of our system in simulated planning domains.

- **Chapter 8: Conclusions**

  This chapter discusses possible routes of research arising from this thesis, and sums up the contributions of this work and concluding remarks.

1.5 Publications

Most of the work presented in this thesis has already been presented in the following publications:

- **Journals**

1. **Mokhtari, V.**, Seabra Lopes, L., Pinho, A.: *Learning robot tasks with loops from experiences to enhance robot adaptability*. Pattern Recognition Letters, 99(Supplement C):57–66. (2017c)
2. **Mokhtari, V.**, Seabra Lopes, L., Pinho, A.: *Experience-based planning domains: An integrated learning and deliberation approach for intelligent robots*. Journal of Intelligent & Robotic Systems 83(3), pp. 463–483 (2016b)
3. Hertzberg, J., Zhang, J., Zhang, L., Rockel, S., Neumann, B., Lehmann, J., Dubba, K., Cohn, A., Saffiotti, A., Pecora, F., Mansouri, M., Konečný, Š., Günther, M., Stock, S., Seabra Lopes, L., Oliveira, M., Lim, G., Kasaei, S.H., **Mokhtari, V.**, Hotz, L., Bohlken, W.: *The RACE project*. KI - Künstliche Intelligenz 28(4), pp. 297–304 (2014)

- **Conferences**

1. **Mokhtari, V.**, Seabra Lopes, L., Pinho, A.J.: *An approach to robot tasks learning with loops from experiences*. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 6033–6038 (2017a)
2. **Mokhtari, V.**, Seabra Lopes, L., Pinho, A.J.: *Learning and planning of robot tasks with loops*. In: 2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC), pp. 296–301 (2017b)
3. **Mokhtari, V.**, Seabra Lopes, L., Pinho, A.J.: *Experience-based robot task learning and planning with goal inference*. In: Proceeding of the 26th International Conference on Automated Planning and Scheduling (ICAPS), pp. 509–517 (2016c)
4. **Mokhtari, V.**, Lim, G.H., Seabra Lopes, L., Pinho, A.J.: *Gathering and conceptualizing plan-based robot activity experiences*, In: Intelligent Autonomous Systems 13: Proceedings of the 13th International Conference IAS-13, chap. pp. 993–1005. Springer (2016a)
5. **Mokhtari, V.**, Seabra Lopes, L., Pinho, A.J., Lim, G.H.: *Planning with activity schemata: Closing the loop in experience-based planning*. In: Autonomous Robot Systems and Competitions (ICARSC), 2015 IEEE International Conference on, pp. 9–14 (2015b)
6. **Mokhtari, V.**, Seabra Lopes, L., Pinho, A.J., Lim, G.H.: *Experience-based planning domains: An approach to robot task learning*. In: Autonomous Proceedings of the 21st Portuguese Conference on Pattern Recognition(RecPad), Faro, Portugal, October 2015, pp. 30–31 (2015a)

7. Lim, G.H., Oliveira, M., **Mokhtari, V.**, Kasaei, S.H., Chauhan, A., Seabra Lopes, L., Tome, A.: *Interactive teaching and experience extraction for learning about objects and robot activities*. In: Robot and Human Interactive Communication, 2014 RO-MAN: The 23rd IEEE International Symposium on, pp. 153–160 (2014)

Chapter 2

A Review of Robot Task Learning and Planning

Widespread interest in robot learning goes back to three decades ago (Connell and Mahadevan, 1993). Robot learning poses a major challenge of building a robot that learns to perform a task through longterm training and feedback. This involves dealing with the issue of integration of multiple technologies, such as sensing, planning, acting, and learning. Although, over the past years, machine learning techniques have had great success in each of these fields, many difficulties have restrained the robots to only operate in lab environments. The level of complexity of real-world tasks is high, not only in terms of perception and manipulation capabilities, but also in the required degree of adaptation to the new environment. In order to sufficiently involve robots in real-world tasks, robot platforms and information processing algorithms must support the ability for the user to customize the robots' behaviors. The motivation for tackling this challenge centers on the belief that it is impossible to preprogram all the necessary knowledge into a robot operating in a diverse, dynamic and unstructured environment. Instead, end-users should be allowed to customize the functionality of these robotic systems. To that end, robots must have the abilities to interact with humans and learn and generalize from task demonstrations.

In this chapter, we provide an introduction and overview of the field, and present the technical challenges associated with designing robots that learn from human instruction. In the first part of this chapter, we study Learning from Demonstration (LfD), as a powerful approach to speed up learning new skills in robotics, and present challenges in this field (Section 2.1). In the second part of this chapter, we study Artificial Intelligence (AI) planning as a key ability for intelligent robots to increase their autonomy and flexibility (Section 2.2). Finally, we

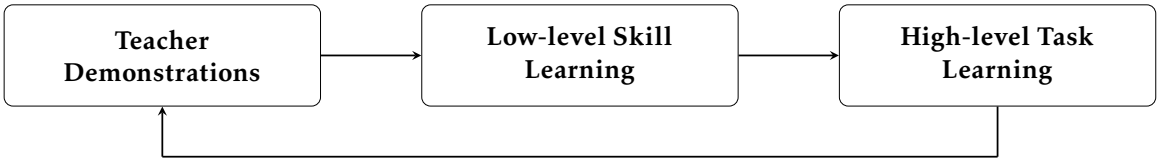


Figure 2.1: An abstract illustration of the Learning from Demonstration pipeline.

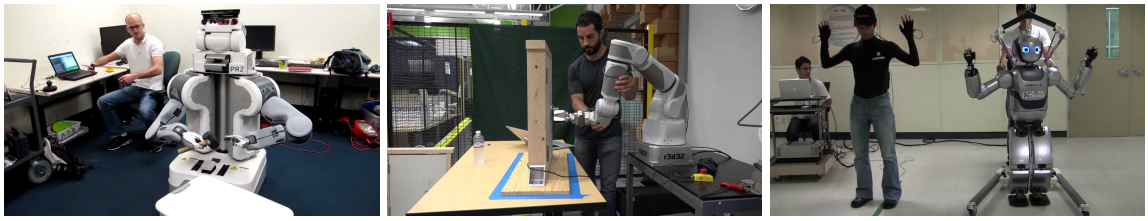
present a survey of machine learning techniques for learning planning knowledge from past solutions (Section 2.3).

2.1 Robot Learning from Demonstration

Learning from Demonstration (LfD) explores techniques for learning a task from examples provided by a human teacher (Billard et al., 2008; Argall et al., 2009; Chernova and Thomaz, 2014). The field of LfD has grown into an extensive body of literature over the past decades, with a wide variety of approaches for encoding human demonstrations, modeling skills and tasks, and reproducing the acquired skills in new contexts. Regardless of the algorithms used in existing LfD platforms, all LfD techniques share certain key properties. Figure 2.1 illustrates an abstraction of the LfD pipeline which frames the design process for building an LfD system. Here, we discuss the strategies for providing data to a robot learner, and present the algorithms for low-level skill learning and high-level task representation.

2.1.1 Teacher Demonstrations

The assumption in all LfD work is that there is a teacher who demonstrates a desired behavior. Most LfD work has made use of human demonstrators, although some techniques have also examined the use of robotic teachers, hand-written control policies and simulated planners. Demonstrations are typically composed of state-action pairs which are recorded during teacher executions of the desired behaviors. How the demonstrations are recorded, and what platform the teacher uses for the execution, may vary across approaches. Two approaches for providing demonstration data to the robot learner are commonly addressed in LfD techniques:



(a) Teleoperation (b) Kinesthetic teaching (c) Shadowing

Figure 2.2: Different techniques for directly transferring demonstration data to a robot learner. (a) A person teaching a pr2 how to hammer a plastic box using thumb joysticks; (b) A person teaching a robotic arm how to open a door through kinesthetic teaching; (c) A humanoid robot is controlled in real time by a human operator wearing a motion-capture system.

Direct Demonstration

A teacher performs a task by directly controlling the robot. The robot records sensory information during the demonstration and is then able to reproduce the task. One approach is *teleoperation* which provides the most direct method for information transfer within demonstration learning. During teleoperation, the robot is operated by the teacher while recording information from its own sensors. Teleoperation simplifies the learning process significantly as it does not require the mapping function to translate human capabilities to those of the robot. Some application of recording demonstrations through human teleoperation by a joystick includes: flying a robotic helicopter (Abbeel et al., 2007), soccer kicking motions (Browning et al., 2004), and robotic arm assembly tasks (Chen and Zelinsky, 2003).

Kinesthetic teaching also offers a variant of direct demonstration. In this method, the robot is not actively controlled, but rather its passive joints are moved through the desired motions while the robot records the trajectory (Calinon and Billard, 2007; Shafii et al., 2016). This technique has been extensively used in motion trajectory learning. A key benefit of teaching through this method of interaction is that it ensures that the demonstrations are constrained to actions that are within the robot's abilities.

Shadowing is another direct demonstration technique, in which the robot records the execution using its own sensors while attempting to mimic the teacher motion as the teacher executes the task. In comparison to teleoperation, shadowing requires an extra algorithmic component which enables the robot to track and actively shadow the teacher. Body sensors are often used to track the teacher's movement with a high degree of accuracy (Koenemann et al., 2014). Figure 2.2 shows some application of the direct demonstration techniques in robotics.

Indirect Demonstration

In some situations, the robot learns by observing a human performing the task. The teacher performs the task demonstration using its own body instead of controlling the robot directly. This method is much more difficult to realize, as it requires the robot to have sensing capabilities (vision), and reasoning about what it is observing. The trajectory of the demonstration cannot be mapped directly to the robot because of the different kinematics. The low-level sensory information must be transformed to high-level situation-action descriptors, and then mapped back to low-level motor controls dependent on the world state at run-time. Wearable sensors, and other forms of specialized recording devices, provide a high degree of accuracy in the observations. Visual markers are also often used to improve the quality of visual information. A number of approaches in this family have been proposed (Kuniyoshi et al., 1994; Billard and Matarić, 2001; Bentivegna et al., 2002; Yang et al., 2015). In recent years, the availability of low-cost depth sensors (e.g., Microsoft Kinect) and their associated body pose tracking methods make this a great source of input data for LfD methods that rely on external observations of the teacher (León et al., 2011; Henry et al., 2012).

The choice between direct or indirect robot interaction modalities may be restricted by environmental conditions or user factors of the target application. Interactions between humans and robots was firstly associated with teleoperation of robotic factory platforms (Sheridan, 1992). This topic has also been extensively studied within the field of Human-Robot Interaction (HRI) (Goodrich and Schultz, 2007).

2.1.2 Low-Level Skill Learning from Demonstration

Given an experience of a state-action example that has been acquired during the execution of a demonstration, learning methods can be applied for acquiring a *primitive action*. The goal of learning in this context is to build an accurate model of a demonstrated primitive action (e.g., `pickup` or `putdown`), such that it could be generally applied to a variety of domain specific tasks. We distinguish between approaches to learning primitive actions from *low-level function approximation* to *high-level skill representation* (Billard et al., 2008). The lowest level of learning a primitive action can involve simply learning an approximation to the state-action mapping. In the literature there are also several different names given to this class

of low-level action learning, such as learning of a, *policy*, *trajectory*, *mapping function*, and *motor skill*. Many learning approaches have been applied in the context of low-level function approximation such as Dynamic Movement Primitives (DMPs) (Ijspeert et al., 2002), probabilistic modeling methods such as Hidden Markov Models (HMMs) (Kulic et al., 2009) and Gaussian Mixture Regression (GMR) (Calinon et al., 2007), and Reinforcement Learning (RL) (Peters and Schaal, 2008). In order for the learned behaviors to be usable by a planning system, it is necessary to learn and represent the pre- and post-conditions at a symbolic level.

Some research into learning of primitive actions based on human demonstrations focuses on high-level symbolic representation of actions in the form of planning operators (Agostini et al., 2011). These operators, specifying pre- and post-conditions, are the basic pieces of knowledge required for the integration of a symbolic task planner into a robot allowing the robot to generate plans for concrete tasks or goals. Ahmadzadeh et al. (2015) present an approach to learn and reproduce trajectory-based primitive actions from a set of demonstrations through kinesthetic teaching. Using a set of recorded trajectories during the tutor demonstrations and applying DMP, a robot learns a set of behaviors by which it can reproduce an action starting from a different pose towards a target. Simultaneous with the tutor demonstrations, a visual perception system (Ahmadzadeh et al., 2013) captures one pre-action observation and one post-action observation for each operation executed by the tutor. The perception system extracts the position of detected objects in the world, and creates symbolic representations for the objects which are used to identify preconditions and effects of the executed actions. The learned primitive actions are finally represented in Planning Domain Definition Language (PDDL), which can be used by a standard symbolic task planner.

2.1.3 High-Level Task Learning from Demonstration

Another aspect of learning in LfD is how the low-level actions, that were derived from motion trajectories, can be used to learn higher level tasks. While the line between high-level and low-level learning is not concrete, the distinction we make here is that techniques in this section assume the existence of a discrete set of action primitives that can be combined to perform a more complex behavior. Action primitives are often parameterized, e.g., `pickup(obj)`, and can be hand-coded, executed by a planner or learned through one of the techniques in low-level skill learning, addressed in the previous section. In this context, the algorithms are

targeted at learning compositions of primitive actions, and the teacher's demonstrations are typically performed at a higher level consisting of action primitives from a library of actions executable by the robot. Some LfD techniques address learning a reactive task policy using a function approximation of direct mapping from input states to output actions (Saunders et al., 2006; Rao et al., 2007; Sullivan et al., 2010). In this representation, demonstration data typically consists of state-action pairs, or trajectories of state-action pairs, that are examples of completing the task. Given these demonstrations, the goal of the algorithm is to reproduce the underlying teacher policy by generalizing over the set of available training examples. The result of learning is a *task policy* model that outputs actions for the given states.

Other techniques represent a desired robot behavior as a plan—a sequence of actions that lead from an initial state to a final goal state. In this context, actions are often defined in terms of pre-conditions—the state that must be established before the action can be performed, and post-conditions—the state resulting from the action's execution. These techniques often rely on additional information in the form of annotations or intentions from the teacher. Van Lent and Laird (2001) present a method for learning non-deterministic plans based on demonstration traces annotated with goal transition data. Garland and Lesh (2003) introduce an algorithm for learning a domain-specific hierarchical task model from demonstration. Within this approach, the teacher is able to annotate the sequence of demonstrated actions and provide high level instructions, for example, the fact that some actions can occur in any order. Note that in both of the above examples, state-action demonstrations are supplemented with additional information from the teacher to aid in generalization. Some research has also focused on interaction capabilities for teaching high-level task knowledge (Rybski et al., 2007; Mohseni-Kabir et al., 2015). Here, the teacher uses a convenient communication mechanism to teach action compositions, i.e., to directly transfer high-level planning knowledge to the robot. However, the acquired models in these techniques lack flexibility to be adapted to slightly different variations.

In one of the earliest LfD work (Kuniyoshi et al., 1994), known as Learning by Watching, the task demonstration is initially segmented into meaningful unit operations. These operations are then classified based on motion types, target objects and effects on targets. Finally, through dependency analysis and bottom-up plan inference, a model of the demonstrated task is created. This approach was applied to a robotic object manipulation. This, and other early work in this area (Ikeuchi,

1995), enabled robots to replay actions observed during demonstration. However, the learned models had little ability to generalize beyond the demonstrated environment.

More recent work in this area has focused on generalization, as well as techniques for learning complex plan structures through various interaction modalities. For example, Veeraraghavan and Veloso (2008) present an algorithm for learning generalized plans that represent sequential tasks with repetitions. In this framework, simple instructions, like pointing to objects with a laser pointer, have been used to teach a humanoid robot the repetitive task of collecting colored balls into a box, based on only two demonstrations. Nicolescu and Mataric (2003) also contribute techniques for learning from multiple demonstrations, presenting a framework for learning behavior networks—a high-level task structure that models the interaction between abstract and primitive behaviors and their effects. In this work, a task representation is derived from each demonstration, based on correspondences between execution data and behavior models. The generalization from multiple demonstrations is represented into a Directed Acyclic Graph (DAG) based on identifying the longest common subsequence. The framework enables the robot to generalize across multiple demonstrations and to refine the learned model based on speech input from the teacher. Pardowitz et al. (2007) forms task precedence graphs by computing the similarity of accumulating demonstrations. Each task precedence graph is a DAG that explains the necessity of specific actions or sequential relationships between the actions.

In (Ekvall and Kragic, 2008) the robot learns an abstract task goal from multiple demonstrations and describes it in a first-order logic language. The robot converts sequential relationships between all two states as temporal constraints. Whenever a sequence is added, the constraints that contain contradictions with the constraints of the new sequence were eliminated in order to extract general state constraints. In new situations the constraints are used by a symbolic planner to choose the best strategy for generating a plan that reproduces the task goal. This representation is an alternative to the DAG-like representations proposed by others.

The above techniques construct plan representations from the discretization of task demonstrations into compositions of action primitives. A number of techniques have also focused on bridging the gap between low-level trajectory input and high-level task learning, providing a means for extracting abstract task structures from motion trajectories (Ehrenmann et al., 2002; Dillmann, 2004). For ex-

ample, Niekum et al. (2012, 2013) present algorithms based on Bayesian nonparametric models to discover repeated structure in the demonstration data, identifying subgoals and primitive motions that best explain the demonstrations and can be recognized across different demonstrations and tasks. This process converts continuous demonstrations into a coherent discrete representation for finding additional structure, such as task-level sequencing information. The demonstrations are then segmented and used to generate a Finite-State Automaton (FSA). The authors also use interactive corrections, provided by the user, at the time of failure as additional demonstrations to improve the structure of the FSA.

Overall, the learning techniques used so far in robot task learning from instructions and/or demonstrations are characterized by poor expressivity of the adopted representations, as well as by limitations of the generalization techniques. Moreover, the exchange of the learned knowledge between robot platforms with different kinematics, as well as the exploitation of the learned knowledge by a standard planner, has received little attention.

The application of classical AI techniques for knowledge representation, planning and learning are likely to significantly improve the capabilities of experience-based planning systems. In the AI community, much of the work has focused on the high-level planning and conceptual representations of skills and state changes using propositional or first-order logic. The integration of a task planner into a robot system increases the robot's level of intelligence and flexibility by altering the way the robot is controlled, moving from predefined sequences of detailed user instructions to a more sophisticated goal oriented approach. In the next sections, we review the literature on classical automated planning as well as recent machine learning techniques for automatic definition of planning action models and planning control knowledge.

2.2 Artificial Intelligence Planning

Automated Planning (AP) has been a core topic of research in artificial intelligence since the beginning of AI in the late 1950s (Fikes and Nilsson, 1972; Ghallab et al., 2004). AP is defined as the task of finding a *plan*, i.e., a solution of a planning problem consisting of a sequence of actions, which leads from an initial state to a desired goal state. In other words, given a description of the initial state, goals, and possible actions, an automated planner finds a plan that reaches the goals from the initial state. AP is essentially characterize by two elements:

- AP *domain* comprising a set of states of the environment and a set of actions which carry out transitions between states.
- AP *problem* comprising the initial state of the environment and a set of facts or conditions representing the goal to be achieved.

The *classical* model for planning is a common restriction of the more general problem of selecting actions to reach a desired objective. The actions are assumed to be deterministic and the information about the environment, complete. Methods for classical planning have had great success, and are in continual development and subject of research. The good results of classical planning techniques have cleared the path to approaches that simplify more complicated problems to classical planning problems, in order to solve them efficiently.

2.2.1 Classical Planning Framework

A classical planning problem can be translated as a directed graph whose nodes represent states, and whose edges represent actions. The change of a state is represented as a transition from a source node representing it along an edge and toward a target node representing the next state. A solution plan is then a path from the node in the graph representing the initial state to a goal node representing a state recognized as a goal state of the problem, i.e., a linearly ordered finite sequence of actions. Following a STRIPS-oriented representation (Fikes and Nilsson, 1972) a planning domain $\mathcal{D} = (S, O)$, on a first-order logic language containing constant and predicate symbols, is described by a finite set of states S and a set of planning operators (actions) O. Any state $s \in S$ is a set of ground atoms which describes a situation of the world in the domain.

The transition between states is specified through a set of planning operators. A planning operator $o \in O$ is described by a pair (P, E), where P is the precondition of o, a conjunction of atoms that must be true in a state in order to use the operator o, and E is the effect of o, a set of atoms that the operator o will make true in the state. An instantiated operator is called an *action* which transforms a state into a new state.

A planning problem $\mathcal{P} = (s_0, g)$ is described by an initial state $s_0 \in S$ and a set of ground atoms describing the goal g. The problem solving task is to find a *plan* (a sequence of actions) $\pi = \langle o_1, \ldots, o_n \rangle$ which transforms the initial state into a goal state where the goal is achieved.

The plan existence problem in the classical setting, i.e., the problem of deciding if there exists a valid plan for an arbitrary problem instance, in the propositional planning model is decidable and has been shown to be PSPACE–complete (Bylander, 1994).

2.2.2 Planning Domain Definition Language

The classical representation in automated planning stems from STRIPS (Fikes and Nilsson, 1972), an early automated planning system. STRIPS takes a symbolic description of the current or initial world state, a desired goal condition, and a set of action descriptions, which characterize the pre- and post-conditions of each action. This system allowed to contain arbitrary well-formed formulas in first-order logic. However, there were a number of problems with this formulation, such as ineffective reasoning on problems of even moderate complexity and the difficulty of providing a well-defined semantics for it (Lifschitz, 1987).

The Action Description Language (ADL) representation, introduced by Pednault (1989, 1994), proposes a trade-off between the expressiveness of general logical formalism and the computation complexity of reasoning with that representation, i.e., computing the transition function γ. Starting from UCPOP (Penberthy et al., 1992), several planners (Chapman, 1987; Dean and Boddy, 1988; Peot and Smith, 1992) were generalized to ADL or to representations close to ADL. These extensions have been integrated in the PDDL planning language used in the International Planning Competition (IPC) since 1998 (Mcdermott et al., 1998). PDDL is based on Boolean state variables representation. Nowadays the PDDL language is widely used in the planning community and includes many specific features, such as trajectory constraints for temporal reasoning or soft constraints to express preferences (Fox and Long, 2002, 2003; Edelkamp and Hoffmann, 2004). Actions in PDDL are expressed as schemata, as shown in the following example:

```
(:action pick
   :parameters   (?obj - object ?tbl - table)
   :precondition (and (table ?tbl)(object ?obj)(on ?obj ?tbl))
   :effect       (and (holding ?obj)(not(on ?obj ?tbl))))
```

Here the action `pick` takes two parameters, `?obj` and `?tbl`, that will be eventually instantiated with the possible values of the types `object` and `table`, declared in the problem description.

2.2.3 Algorithms for Classical Planning

Solving classical planning problems can be seen as path-finding in a directed graph whose nodes represent states, and whose edges represent state transitions resulting from the execution of actions. Classical planning problems can then be solved by graph search algorithms to find a path from the initial state to a goal state. This graph search approach is not trivial, because the size of the graph may be exponential in the size of the description of the planning problem in used predicates form. Thus, blind search algorithms, such as breadth-first/depth-first search, are practically unfeasible.

An approach that has proved to be effective relies on the use of heuristic search. Heuristic search uses heuristic functions to evaluate the cost from any given state to a goal state (Bonet and Geffner, 2001; Ghallab et al., 2004). This estimation of the distance in the search space is then used by the search algorithm to derive the state space search, preferring to visit nodes considered more promising based on their heuristic value. The best well-known algorithms in this family are, A^* (Hart et al., 1968) and IDA* (Korf, 1985). Many planners used heuristic search (McDermott, 1996; Bonet et al., 1997), which is now the most successful approach to classical planning.

Other heuristics are also obtained by solving a simpler version of the original problem relaxing its constraints (Pearl, 1984). Relaxations directly derived from the problem description are useful and efficient, such as the successful "delete relaxation", obtained by dropping the negative effects of the actions (Hoffmann and Nebel, 2001).

One of the first approaches making use of domain-independent heuristics is the HSP planner (Bonet and Geffner, 1999). HSP makes use of best-first search coupled with h_{add} heuristic, that approximates the distance between two states by summing the distances between the propositions in the states, ignoring the delete effects.

The Fast-Forward (FF) planner by Hoffmann and Nebel (2001) is based on the same delete relaxation as HSP, but uses an explicit solution of the relaxed problem to estimate its heuristic h_{FF} and the extraction of helpful actions applied first when searching for a plan. When the incomplete but effective greedy search of FF (called "enforced hill-climbing") based on helpful actions fails, the planner launches a best-first search. The helpful actions are defined in the FF planner as those operators applicable in the current state that add some precondition of an operator in the plan. This search control technique has proved to be quite successful and

effective, being the base of many developments (Hoffmann, 2002; Hoffmann and Brafman, 2005, 2006).

The LAMA planner (Richter and Westphal, 2010) makes use of a pseudo-heuristic derived from landmarks, i.e., propositions that must be true in every solution of a planning task (Hoffmann et al., 2004). LAMA is built on top of the Fast Downward Planning System (Helmert, 2006), using in particular a multi-heuristic search. The "landmark counting heuristic" (Richter et al., 2008) estimates the goal distance of a state by counting the number of landmarks that still remain to be achieved.

Other techniques also apply to the classical planning formulation, such as planning graphs (Blum and Furst, 1997; Kambhampati et al., 1997) and planning as satisfiability (Kautz et al., 1992; Rintanen, 2010). They involve the construction of planning graphs and the translation of the planning problem into propositional axioms, in order to consequently apply a satisfiability algorithm to find a model that then corresponds to a solution plan.

2.2.4 Hierarchical Task Network Planning

Although classical planning systems have become much more efficient over the years, they still suffer from combinatorial complexity. Decomposing complex problems into simpler sub-problems is a long-lasting approach to organize domain knowledge and problem solving. Hierarchical Task Network (HTN) planning is an AI planning technique that breaks with the tradition of classical planning (Ghallab et al., 2004). In HTN planning, the planner is provided with a set of tasks to be achieved. A plan is then derived through task-decomposition methods (known as HTN methods), which recursively decompose complex tasks into simpler subtasks to reach the leaves of the task hierarchy. The leaf nodes correspond to actions which can be performed directly using the planning operators. Figure 2.3 shows an example of a task-decomposition tree for a method *move-stack*. From left to right, the leaves of the tree produce a plan solution to the *move-stack* task.

The theoretical framework of the HTN planning is composed of a planning language, operators, task networks, methods, planning problems and plan solutions. The definitions of planning language, operators, actions, and plans are the same as in classical planning framework (see Section 2.2.1). However, the HTN planning framework also includes tasks, task networks and methods for achieving tasks.

A *task* is an activity to be performed. Syntactically, a task consists of a task symbol followed by a list of arguments, i.e., in the form $t(x_1,\ldots,x_k)$, where t is the task name and $x_1,\ldots,x_k$ are the task arguments. A task may be either *primitive* or

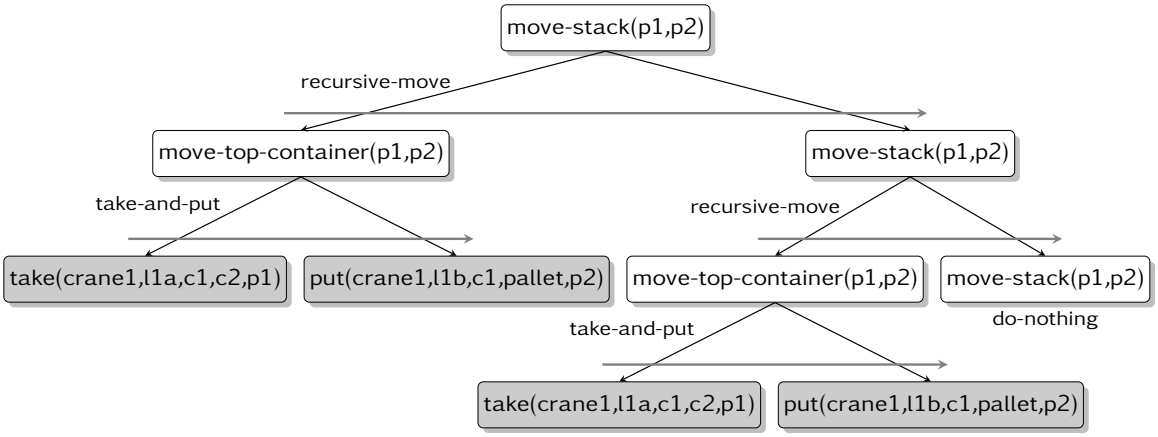


Figure 2.3: An example of the decomposition tree for the method *move-stack*.

compound. A primitive task is supposed to be accomplished by a planning operator: the task symbol is the name of the planning operator to use, and the task's arguments are the parameters for the operator. A compound task can be decomposed into smaller tasks using a method: any method whose head unifies with the task may potentially be applicable for decomposing the task.

Each method includes a task network which provides a way of decomposing a compound task into a partially ordered set of subtasks, each of which can be compound or primitive. A task network $w = (T, C)$ is described by a finite set of tasks T and a set of constraints C. The tasks in T can be primitive or (non-primitive) compound. Constraints in C specify restrictions over T that must be satisfied during the planning process and by the solution, e.g., a precedence (partial ordering) constraint of the form $t_u \prec t_v$ means that t_u must occur before t_v.

A method is a triple $m = (t, P, w)$ where t is the task to be performed by the method, P is the precondition that the current state must satisfy in order for the method to be applicable, and w is the task network that need to be accomplished in order to perform the task m.

For example, Listing 2.1 shows a set of methods for the Dock-Worker Robot domain (Ghallab et al., 2004) represented in SHOP2 (Nau et al., 2003). The method `move-top-container` gives a way to take a container `?c` from a pile `?p1` and put it on top of a pile `?p2`. The `:ordered` keyword specifies that the tasks in the task network (subtasks) are totally ordered: first take a container `?c` from the stack `?p1`, and then put the container on top of the stack `?p2`. The two methods for the `move-stack` task provide a strategy to move containers from a stack `?p` to a stack `?q` using the method `move-top-container`. The first method `move-stack` recursively

```
(:method
    (move-top-container ?p1 ?p2)  ; head
    (and                          ; precondition
      (top ?c ?p1)
      (on ?c ?x1)
      (attached ?p1 ?l1)
      (belong ?k ?l1)
      (attached ?p2 ?l2)
      (top ?x2 ?p2))
    (:ordered                     ;task network (subtasks)
      (take ?k ?l1 ?c ?x1 ?p1)
      (put ?k ?l2 ?c ?x2 ?p2)))

(:method                          ; a recursive method
    (move-stack ?p ?q)            ; head
    (and                          ; precondition
      (top ?c ?p)
      (on ?c ?x))
    (:ordered                     ; task network
      (move-top-container ?p ?q)
      (move-stack ?p ?q)))

(:method                          ; do-nothing
    (move-stack ?p ?q)            ; head
    (top ?pallet ?p)              ; precondition
    ())                           ; task network
```

Listing 2.1: A set of HTN methods represented in SHOP2. Every method has a name (head), a set of preconditions, and a task network (a set of subtasks).

calls itself while there is a container on top of the stack ?p. The second method `move-stack` terminates the task when there is no container on top of the stack ?p (i.e., the pallet is on top of the stack).

An HTN planning problem $\mathcal{P} = (s_0, w_0, O, M)$ is described by an initial state s_0, an initial task network w_0, a set of HTN planning operators O, and a set of HTN methods M. A solution for the HTN planning problem $\mathcal{P}$ is a plan π that performs the desired initial tasks w_0, when it is executed in the initial state s_0.

Hierarchical task networks are one of the best studied approaches for modeling planning knowledge about a problem domain (Nejati et al., 2006; Ilghami and Nau, 2006; Zhuo et al., 2009; Hogg et al., 2009; Georgievski and Aiello, 2015). According to Ghallab et al. (2004), "HTN planning has been more widely used for practical applications than any of the other AI planning techniques. This is partly because HTN methods provide a convenient way to write problem-solving *recipes* that correspond to how a human domain expert might think about solving a planning problem".

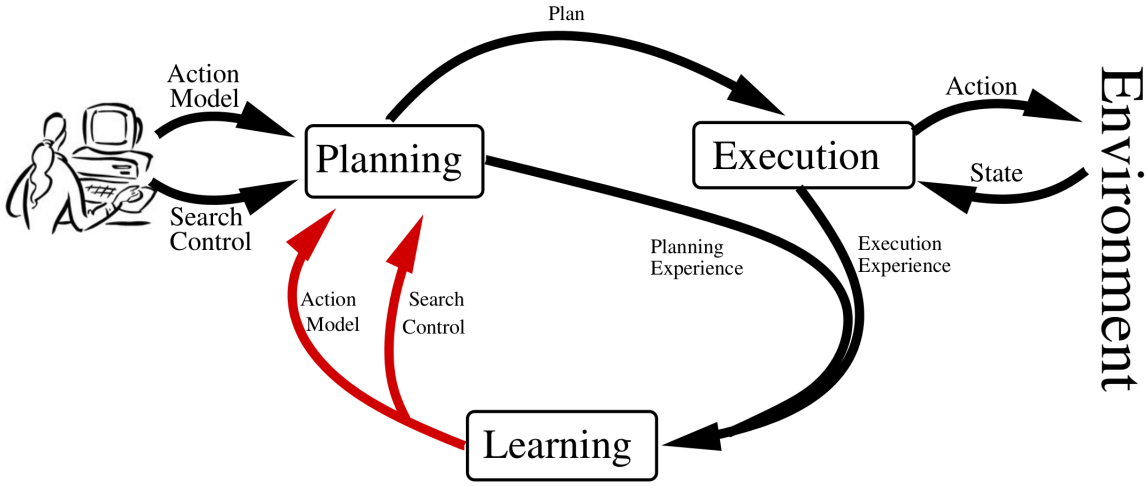


Figure 2.4: Improving the knowledge of a planner using machine learning (Jiménez et al., 2012).

2.3 Machine Learning for Automated Planning

Since the beginning of AI planning, many attempts have been made to represent planning tasks and to introduce efficient algorithms for solving them (Jiménez et al., 2012). In the last decades, AP systems have been successfully applied to real-world tasks (McGann et al., 2008; Reddy et al., 2011; Petrick and Foster, 2013), however, the application of AP to real-world problems is still complicated, because of mainly two knowledge definition problems:

- **Acquiring planning action model.** Accurate description of planning operators is essential for automated planners. This description involves modeling the actions to be executed in the environment, state of the environment and the goals to be achieved. Generating exact definitions of the planning tasks in advance is a challenging issue in most real-world domains.
- **Acquiring planning search control knowledge.** Finding a plan in AP is a PSPACE-complete problem. Standard classical planners fail to produce solutions to the AP tasks with large numbers of objects. The search process guided by heuristics is one of the dominant approaches to deal with this problem in state-of-the-art planners. Defining search control knowledge is difficult since it requires expertise in both the planning domain and the planning algorithm.

Machine Learning (ML) has been widely applied to deal with the above knowledge acquisition problems. Figure 2.4 shows the role of these two targets in ML.

Comprehensive studies of ML for AP have been conducted (Zimmerman and Kambhampati, 2003; Jiménez et al., 2012). Here, we mainly focus on learning high-level action compositions. Learning of planning action models is disregarded due to the scope of this thesis (however, see Section 2.1.2).

The efficiency of planning procedures can be significantly improved by exploiting knowledge about the structure of the AP tasks that is not explicitly encoded in the domain model. Planners that exploit domain-specific or problem-specific control knowledge can reason out faster than standard planners. Defining hand-coded search control knowledge requires expertise in the AP domain and planner's search algorithm. This review is organized into different ML approaches, namely *macro actions*, *hierarchical decomposition methods*, *generalized planning* and *explanation-based learning*.

2.3.1 Learning Macro Actions

Macro actions refer to the compound actions generated by assembling individual actions in sequence that are frequently used. Macro actions reduce the depth of the planner's search tree by directing the search more carefully, but they may also increase the search tree's branching factor, thus bringing on the utility problem (Minton, 1990; Holder, 1990). A trade-off between utility and branching factor is required to keep on working with macro actions. Learning macro actions was the first attempt to improve planning systems. The STRIPS PLANEX system (Fikes et al., 1972) first used macro actions as a tool for plan execution and analysis. In this work, the STRIPS planner has been extended with macros in the form of generalized plans as solutions to previous problems which are generalized by substituting constants with variables.

The general idea of creating the different components of a macro action involving parameters, preconditions and effects by assembling two actions a_1 and a_2 into a macro action a_{12} is as follows:

- The parameters: $par(a_{12}) = (par(a_1) \cup par(a_2))$.
- The preconditions: $pre(a_{12}) = (pre(a_1) \cup pre(a_2)) \setminus add(a_1)$.
- The positive effects: $add(a_{12}) = (add(a_1) \cup add(a_2)) \setminus del(a_2)$.
- The negative effects: $del(a_{12}) = (del(a_1) \cup del(a_2)) \setminus add(a_2)$.

The learned macro actions are represented in predicate logic and added as new actions to the planning domain. Listing 2.2 shows an overall example of a learned

```
;;; primitive actions
(:action unstack
    :parameters  (?x - block ?y - block)
    :precondition(and (on ?x ?y)(clear ?x)(handempty))
    :effect      (and (holding ?x)
                      (clear ?y)
                      (not (clear ?x))
                      (not (handempty))
                      (not (on ?x ?y))))
(:action putdown
    :parameters  (?x - block)
    :precondition(holding ?x)
    :effect      (and (not (holding ?x))
                      (clear ?x)
                      (handempty)
                      (ontable ?x)))

;;; macro actions
(:action unstack-putdown
    :parameters  (?x - block ?y - block)
    :precondition(and (on ?x ?y)(clear ?x)(handempty))
    :effect      (and (clear ?y)
                      (not (on ?x ?y))
                      (ontable ?x)))
```

Listing 2.2: Macro action unstack-putdown for the Blocks world domain.

unstack-putdown macro action from actions unstack and putdown for the Blocks world domain (Gupta and Nau, 1992).

Botea et al. (2005, 2007) introduce Macro-FF, an automated method that learns control knowledge from experiences and uses it to solve new problem instances. Authors present a four-step procedure: (1) Analysis, extract structural information about the domain; (2) Generation, build macro actions based on the discovered information; (3) Filtering, rank and filter macro actions to select the most useful; (4) Planning, exploit the macro actions to improve the planning for the future problems. They proposed the Component Abstraction Enhanced Domain (CAED) architecture which generates a set of macro actions from the PDDL formulations of a domain and several training problems. Macro actions are added to the initial domain with the same description language, causing an enhanced domain. The CAED architecture suffers from two limitations. First, it can be only applied to domains with static facts, and second, adding macros is only limited to STRIPS domains. To overcome the limitations of CAED, the second abstraction method, Solution Enhanced Planner (SOLEP) is presented. SOLEP extends the CAED macro representation from STRIPS to ADL by representing macros as sequences of operators and mappings of the operators' variables instead of building single operators in CAED.

Gerevini et al. (2011) study the learning of two types of knowledge, macro

actions and planning horizon, in the planning as satisfiability framework known as SATPLAN system (Kautz et al., 1992). The planning horizon is the length of an optimal plan for solving a problem. They propose a new variant of MiniSAT (Eén and Sörensson, 2003) which uses a given set of macro actions to improve the performance of SAT planners. To better exploit macro actions, an independent learning technique is presented to predict the optimal planning horizon of a given problem. The authors demonstrated that using macro actions can speed up the SAT planner when the Conjunctive Normal Form (CNF) encoding of the problem is satisfied.

Other techniques also successfully integrate macros with state-of-the-art heuristic search planners. Newton et al. (2007) present an offline macro action learning technique that works with arbitrary planners and domains. They employ a genetic algorithm to guide the macro actions generator while searching for macros in macro space. The macros are generated from randomly generated plans of smaller problems, and evaluated against other larger problems. Algorithms for n-grams analysis have also been applied to learning macro actions for the heuristic planner FF (Muise et al., 2009). Chrpa (2010) computes a matrix of candidates from all the training plans, and then selects a candidate for creating macro operators by looking for operators whose instances appear successively.

2.3.2 Learning Hierarchical Decomposition Models

HTN planning is one of the most successful and practical approaches to AI planning. However, designing an HTN planning domain requires considerable effort by a domain expert or a knowledge engineer. Here we review some ML techniques for automatically learning HTN methods.

Zhuo et al. (2014) present a learning algorithm, called HTNLearn, to automatically acquire HTN methods and primitive actions. HTNLearn receives a set of plan traces with partially annotated intermediate state information, and a set of annotated tasks that specify the conditions before and after the tasks' accomplishment. HTNLearn models the problem of learning HTN methods as a maximum satisfiability problem (MAX-SAT problem). It constructs constraints based on inputs, specifically, method constraints to encode the methods' preconditions and structures; state constraints to encode action models; decomposition and task constraints to improve the learned HTN models; and hard constraints to verify consistency between the learned HTN and the inputs. The corresponding weights to the constraints are calculated and fed to a weighted MAX-SAT solver. The solution

to this MAX-SAT problem is the HTN model including the set of actions and HTN methods that explain the given inputs.

HTN-MAKER is other work, presented by Hogg et al. (2008), which learns incrementally HTN planning knowledge. HTN-MAKER is fed by a set of planning operators, a set of initial states of classical planning problems and solutions to these problems, and a set of semantically-annotated tasks to be achieved. An annotated task is defined as a triple $t = (n, pre, eff)$ where n is a task name, and pre and eff are respectively sets of preconditions and effects in the form of atoms. HTN-MAKER generates a sequence of states by applying the actions in a given plan, starting from the initial state. Then it traverses the states in which the effects of an annotated task become true, as well as it looks for the preceding states, from which these effects are accomplished, and then finds annotated tasks whose effects might be true in this interval. To identify a sequence of subtasks that fulfills this task and the preconditions of the subtasks, if there is an annotated task whose effects and preconditions match correspondingly the states during the above interval, HTN-MAKER regresses the effects of the annotated task through the plan elements (actions or previously learned methods) that produce those effects. The learned HTN method is eventually added to the domain and also an instantiation of the method with its initial and final states is stored as a sub-plan within the plan for using as a subtask in further learning of new methods.

Ilghami et al. (2002) present a framework for supervised learning of HTN methods, called CaMel (Candidate Elimination Method Learner). CaMel learns incrementally conditions for HTN methods under expert supervision. It takes as input a set of adapted plan traces containing the solutions for HTN planning problems and information about inferences derived and decisions made during the plans, as well as an incomplete version of the domain, including instances of methods that are applicable to the planning problems. The authors adopt the candidate elimination machine learning approach (Mitchell, 1977) to yield a sound, complete and incremental algorithm. The output of the proposed method is a complete HTN domain induced from a given incomplete domain as input.

2.3.3 Generalized Planning

Generalized planning is the problem of inferring an algorithmic plan which works over all instances of the same class of problems by mapping states and goals into actions. This problem has been studied since the earliest work on STRIPS (Fikes

and Nilsson, 1972). Generalized planning is also referred in the literature as Generalized policies (Martín and Geffner, 2004; Srivastava et al., 2008; Hu and De Giacomo, 2011; Srivastava et al., 2011). A generalized plan is similar to an object with a method of instantiation, which creates a sequence of actions given an instance of the target class of problems. These are different approaches to efficiently generate plans for wide classes of problems. Generalized planning aims to decrease the cost of instantiation while extending the generality of plans, by including loops of actions to deal with various problem instances. An accurate generalized plan is particularly able to solve any problem from a given domain without any search, by repeatedly applying the preferred actions for each planning context.

Consider the simple problem of picking up blocks in a stack and putting them down on a table. Suppose a problem instance with three blocks a, b, c standing on top of each other, where block c is the topmost. The possible solution plan would be: pickup(c), putdown(c), pickup(b), putdown(b), pickup(a) and putdown(a). Classical planning has the objective of finding such plans for specific problem instances. By increasing the number of blocks in this problem, the complexity of solving it grows accordingly, though the solutions are similar to each other. Plan generalization extracts and utilizes these common solutions and problem structures. For instance, a fine generalized (algorithmic) plan to solve this kind of problem could be:

$$\texttt{unstack} \approx \mathit{while}(\exists \texttt{b}, (\texttt{ontop}(\texttt{b}) \wedge \texttt{ontable}(\texttt{b}))) : \texttt{pickup}(\texttt{b}), \texttt{putdown}(\texttt{b}).$$

Srivastava et al. (2011) address the problem of finding generalized plans for the situations in which the number of objects are unknown during planning. The authors propose a planning system, called Aranda, which takes an empty generalized plan and incrementally increases the applicability of the plan. Using the *canonical abstraction*, an abstract representation developed for Three-Valued Logic Analysis (TVLA) (Sagiv et al., 2002), and back-propagation, Aranda finds an abstract state space from a set of concrete states of problem instances with varying numbers of objects that guarantees completeness, i.e., the plan works for all inputs that map onto the abstract state. However, these strong guarantees come at a cost: (i) restrictions on the language of actions; and (ii) high running times.

Hu and Levesque (2011) propose to use the *situation calculus* (McCarthy, 1963; Levesque et al., 1998) for representing a planning problem which results in a new

representation with an infinite number of objects. The authors introduce a generalized plan representation in the form of a Finite State Automaton (FSA) and the FSA-PLANNER (Hu and Levesque, 2009) to generate FSA plans for dealing with planning problems represented in the situation calculus. An FSA plan is a directed graph, whose nodes are plan states labeled with their associated actions, and edges are the sensing results which identify next plan states. Based on this specification, Hu and Levesque identify a class of infinite planning problems, called one-dimensional, and prove that the execution of an FSA plan will terminate in a situation where the goal condition is satisfied.

LoopDISTILL (Winner and Veloso, 2007; Winner, 2008) can also be considered as a generalized planning technique. The LoopDISTILL algorithm automatically acquires looping domain-specific planners from example plans. LoopDISTILL identifies the largest matching sub-plan in a given example and converts the repeating occurrences of the sub-plans into a loop. The result is a domain-specific planning program (dsPlanner), i.e., a plan with if-statements and while-loops that can solve similar problems of the same class.

2.3.4 Explanation Based Learning

Explanation-Based Learning (EBL) has been frequently used in AI planning to generalize a proof or an explanation of a solution and using this explanation to guide the planning (DeJong and Mooney, 1986). Generalization of objects to variables in plans using deductive or analytical generalization goes back to the STRIPS PLANEX system (Fikes et al., 1972). Although, in the early 1980's, many research was carried out into knowledge-based learning from a single example, Mitchell et al. (1986) provided, for the first time, a unifying view of EBL, called Explanation-Based Generalization (EBG). An EBG method based on a generalization of Mitchell's EBG (Mitchell et al., 1986) is described in Figure 2.5.

This unifying framework combines the tradition, initiated by PLANEX, of producing generalizations of single examples based on some domain knowledge, with the emphasis on analysis and explanation, which characterized later work. Concepts are viewed as predicates over instances, and therefore denote sets of instances. The operationality criterion defines what it means for a concept to be useful. An explanation is a proof (resulting from a logical proof procedure or from some other reasoning scheme with well defined steps) that an object is an instance of a given concept, i.e., a record of all problem-solving steps that are required to prove such assertion. Generalization is the computation of the weakest conditions

Given:

1. *Target concept definition*: a definition describing the concept to be learned. (It is assumed that this concept definition fails to satisfy the operationality criterion).
2. *Training example*: an example of the target concept
3. *Domain theory*: a set of rules and facts to be used in explaining how the training example is an example of the target concept.
4. *Operationality criterion*: a predicate over concept definitions, specifying the form in which the learned concept definition must be expressed.

Determine:

5. A *generalization of the training example* that is a sufficient concept description for the target concept and that satisfies the operationality criterion.

Figure 2.5: The Explanation-Based Generalization method.

for which the proof structure still holds. Generalization, therefore, relies on the ability of the reasoner to explain why the instance is a member of the concept.

EBL techniques have been used in planning both to improve search and to reduce domain modeling burden. In the former case, EBL is used to learn "control knowledge" to speedup the search process (Minton et al., 1989; Kambhampati et al., 1996), or to improve the quality of the solutions found by the search process (Estlin and Mooney, 1997). In the latter case, EBL is used to develop domain models (e.g., action models). One old example is BAGGER2 (Shavlik, 1990), which uses example solutions and domain knowledge to learn an algorithm for problem solving in the form of recursive structures, but relies on hand-coded background domain knowledge. In a more recent work, Levine and DeJong (2006) used EBL for learning low-level operators (i.e., for learning domain knowledge rather than control knowledge) as encapsulated control loops that are specialized to best fit a particular distribution of observed learning problems. With the benefit of prior knowledge, an operator design module searches for the simplest causal explanation of the world dynamics and generates a qualitative relationship graph, then associates a numerical function to it, and finally assesses the capabilities of the function over planning problems, and produces one or more operator definitions for use by the planner.

2.3.5 Other Techniques for Acquiring Planning Knowledge

Heuristic functions are other alternative to improve the complexity of planning. They compute an estimate of the path cost from a given node to a goal node. Numerous approaches to AP have exploited admissible heuristic functions to guide the search of a solution plan through pruning unpromising nodes of the search space (De la Rosa et al., 2013; Yoon et al., 2008; Lelis et al., 2012; Bonet, 2013). A typical way to obtain a heuristic function is to find the actual cost of the simplified form of the problem and use it as the heuristic function of the original problem.

Abstraction has also long been recognized as an important technique for improving problem solving performance (Saitta and Zucker, 2013). A few works have actually combined abstraction with EBG to learn more generic concepts called *abstract plans* or *schemata* (Knoblock et al., 1991; Seabra Lopes, 1997). The idea of representation change for the purpose abstraction (rather than simply dropping sentences) was introduced in the PARIS system (Bergmann and Wilke, 1995), which accumulates generalized and abstracted plans in a case base with indexing and retrieval mechanisms. In this system, it is necessary to specify an abstraction theory that defines how representation change takes place.

Similar ideas were independently developed for acquiring failure recovery schemata in a robotics application (robotized assembly) (Seabra Lopes, 1997, 1999, 2007). In this case, schemata are learned from successful failure recovery experiences, and abstraction involves representation change of states and operators. Moreover, certain properties or features are associated to the generalized and abstracted operators in the learned schemata based on a hand-coded feature extraction strategy. When applying a schema to a new situation, a planner chooses the plan closest to the schema based on the ratio of schema features present in the plan.

Several works have adopted case-based approaches to reduce planning search (Hammond, 1986; Borrajo et al., 2015). In case-based reasoning systems like Priar (Kambhampati and Hendler, 1992) or Prodigy/Analogy (Veloso, 1992, 1993) cases are usually not explicitly generalized in advance. They are kept fully instantiated in a case library, annotated with the created explanations. During problem solving, those cases are retrieved which contain explanations applicable to the current problem (Carbonell and Veloso, 1988; Kambhampati and Hendler, 1992). The detailed decisions recorded in these cases are then replayed or modified to become a solution to the current problem. All these methods tend to suffer from the utility problem, in which learning more information can be counterproductive due to the

difficulty with storage and management of the information and with determining which information should be used to solve a particular problem.

2.4 Summary

Autonomous intelligent robots, such as domestic and personal robots, are desirable in applications where the environment is open-ended and largely unknown. These robots cannot be preprogrammed by foreseeing, at the design stage, all possible courses of actions they may require. They should be rather able to develop their own cognitive abilities by themselves. For that, they must possess a symbolic and internal model of their local environment, and the sufficient logical reasoning capacity to make decisions and to execute the necessary tasks to reach their objectives.

Robot learning is an active research field at the intersection of Robotics and Artificial Intelligence. It studies techniques allowing a robot to acquire novel skills through learning algorithms and to achieve tasks on its own deliberation. Although several cognitive functions are required to endow intelligent robots to operate in dynamic environments (Ingrand and Ghallab, 2014, 2015), in this thesis, we address particularly two deliberative functions:

- Learning: allows a robot to acquire, adapt and improve through experience the models needed to achieve real-world tasks.
- Planning: combines prediction and search to synthesize a trajectory in some abstract action space based on predictive models of the environment and feasible actions in order to achieve some goals.

In this chapter, we presented Learning from Demonstration (LfD) as a particular approach to robot learning from examples. We addressed challenges and learning techniques in this field. Over the past decades, the application of LfD technique to speed up learning and adaptation upon experiences has proven very successful. Although LfD is useful in learning primitive action-control policies (such as for object manipulation), it is unsuitable for learning complex tasks. LfD usually requires many examples in order to induce the intended control structure (Allen et al., 2007). Moreover, the representations are task-specific and are not likely to transfer to structurally similar tasks (Chao et al., 2011).

Planning is a key ability for intelligent robots, increasing their autonomy and flexibility. Numerous methods have been proposed in AI Planning for acquiring and improving action models at the task levels. We described learning techniques in task planning to speed-up planners with control knowledge and macro actions, generalized planning, specific heuristics and hierarchical task decompositions. These efforts are usually concerned with planning speed, however, they have seldom been used in robotics. These approaches use more expressive representations and more advanced algorithms than those used so far in robot learning from demonstration; therefore, we see high application potential.

Considering the gaps in the development of advanced learning techniques and expressiveness of task representation in LfD, we aim to develop (*i*) a unified representation of experience-based robot task planning which utilizes the expressiveness of first-order logic; (*ii*) machine learning techniques for acquiring task planning knowledge from robot experiences; and (*iii*) experience-based problem solving algorithms for efficiently generating solutions to task problems. In the following chapter, we formalize and present the notion of Experience-Based Planning Domains (EBPDs) to support long-term learning and planning. The EBPDs representation scheme uses notation derived from first-order logic. This formalization captures many interesting aspects of the experience-based learning and planning, which can foster the reuse of this research.

Chapter 3

Experience-Based Planning Domains

A necessary input to any planning algorithm is a description of a planning domain and a problem to be solved. The planning domain representation specifies which transitions are possible in each state. Fox and Long (2002) state:

> The adoption of a common formalism for describing planning domains fosters far greater reuse of research and allows more direct comparison of systems and approaches, and therefore supports faster progress in the field. A common formalism is a compromise between expressive power (in which development is strongly driven by potential applications) and the progress of basic research (which encourages development from well-understood foundations). The role of a common formalism as a communication medium for exchange demands that it is provided with a clear semantics.

We propose the notion of *Experience-Based Planning Domain* (EBPD), an extension of the standard notion of *planning domain*, which in addition to planning operators, includes experiences and methods (called *activity schemata*) for solving classes of problems. EBPDs are planning domains that are supported by a long-term learning process. They are intended to endow intelligent robots with the capability of problem solving by learning from experience. Like standard planning domains, EBPDs use a representation scheme derived from first-order logic. Syntactically, the notation used for representing EBPDs is an adaptation and extension of the Planning Domain Definition Language (PDDL) (Mcdermott et al., 1998). EBPDs offer several significant extensions, e.g., abstract operators/actions, experiences, activity schemata, abstraction hierarchies which link concrete predicates/actions with abstract predicates/actions, etc. In this chapter, we first present

a basic learning and planning framework underlying EBPDs. Then the formal terminology used for describing EBPDs is introduced.

3.1 Running Examples

As running examples throughout this thesis, we develop and use two planning domains, `RACE` and `Stacking-Blocks`, for illustrating different aspects of our approach. A full representation and implementation of the `Stacking-Blocks` domain in EBPDs is given in Appendix A.

3.1.1 The RACE Domain

We developed a domain for a restaurant environment. This domain is based on the EU RACE (Robustness by Autonomous Competence Enhancement) project, funded by the EC Seventh Framework Program theme FP7-ICT-2011-7 (Rockel et al., 2013; Hertzberg et al., 2014). The RACE project aimed to develop an artificial cognitive system, embodied by a service robot, able to build a high-level understanding of the world it inhabits by storing and exploiting appropriate memories of its experiences.

Assume a restaurant environment and a guest sitting at a table in the restaurant. A service robot serves a coffee to the guest at the table by picking up a coffee cup from a counter and setting it down on the table in front of the guest. The state of the world in the `RACE` domain is described by the following predicates:

- `table(x)`, `counter(x)`, `robot(x)`, `guest(x)`, `mug(x)`, `leftarm(x)`, `rightarm(x)`, `torso(x)`, `gripper(x)`, `manipulationareasouth(x)`, `manipulationareanorth(x)`, `manipulationareaeast(x)`, `manipulationareawest(x)`, `manipulationarea(x)`, `premanipulationareasouth(x)`, `premanipulationareanorth(x)`, `premanipulationareaeast(x)`, `premanipulationareawest(x)`, `premanipulationarea(x)`, `placingarealeft(x)`, `placingarearight(x)`, `torsoupposture(x)`, `torsodownposture(x)`, `torsomiddleposture(x)`, `armtuckedposture(x)`, `armcarryposture(x)`, `armtosideposture(x)`, `armunnamedposture(x)`: `x` is respectively a table, counter, robot, guest, mug, arm, torso, gripper, manipulation area, premanipulation area, placing area, torso posture, or arm posture.
- `hassittingarea(x,y)`, `hasmanipulationarea(x,y)`, `haspremanipulationarea(x,y)`, `hasplacingarea(x,y)`: table `x` has respectively a sitting area, manipulation area, premanipulation are, or placing area `y`.

- at(x,y): guest x is at sitting area y.
- on(x,y), objectobserved(x,y): object x is on or observed at placing area y.
- robotat(x,y): robot x is at area y.
- hastorso(x,y), hasarm(x,y): robot x has torso or arm y.
- hasarmposture(x,y), hasgripper(x,y): arm x has arm posture or gripper y.
- hastorsoposture(x,y): torso x has torso posture y.

The RACE domain has the following actions:

- tuck_arms(l,r,u,v,x,y), move_arms_to_carryposture(l,r,u,v,x,y): move the postures of arms l and r from u and v to x and y respectively.
- move_arm_to_side(a,x,y): move the postures of arm a from x to y.
- move_torso(t,x,y): move the posture of torso t from x to y.
- move_base(r,x,y), move_base_blind(r,x,y): robot r moves from x to y. The posture of the robot makes a distinction between these two actions, i.e., in move-base action, the robot moves when the robot has a certain posture, while in move-base-blind action, the robot moves blindly with no condition.
- pick_up_object(r,a,o,p,h,p1,t,p2),place_object(r,a,o,p,h,p1,t,p2): robot r at area a with arm h, arm posture p1, torso t, and torso posture p2, picks up/puts down object o from/at placing area p.
- observe_object_on_area(r,a,o,p): robot r at area a observes object o at placing area p.

3.1.2 The Stacking Blocks Domain

We also developed a Stacking-Blocks domain, based on the blocks world domain, for illustrating the presented concepts and definitions. The environment of the Stacking-Blocks domain consists of several locations each one equipped with a hoist, tables and piles. For each pile there is a pallet that sits at the bottom of the pile. There are a number of red and blue blocks in each location that can be on the tables or in the piles. The blocks are picked up, put down and moved between the tables and piles in a location, by a hoist attached to the location. The state of a problem in the Stacking-Blocks domain is described by the following predicates:

- pile(x), table(x), blue(x), red(x), pallet(x): x is a pile, table, blue block, red block, or pallet, respectively.

- `attached(p,l)`: pile `p` is attached to location `l`.
- `belong(h,l)`: hoist `h` belongs to location `l`.
- `at(h,p)`: hoist `h` is at place `p`.
- `holding(h,x)`: hoist `h` is holding block `x`.
- `empty(a)`: hoist `h` is empty.
- `on(x,y)`: block `x` is on block `y`.
- `ontable(x,t)`: block `x` is on table `t`.
- `top(x,p)`: block `x` is the top of pile `p`.

The `Stacking-Blocks` domain has the following actions:

- `move(h,x,y,l)`: hoist `h` moves from place `x` to place `y` at location `l`.
- `unstack(h,x,y,p,l)`: hoist `h` picks block `x` from block `y` on pile `p` at location `l`.
- `stack(h,x,y,p,l)`: hoist `h` puts block `x` on block `y` on pile `p` at location `l`.
- `pickup(h,x,t,l)`: hoist `h` picks up block `x` from table `t` at location `l`.
- `putdown(h,x,t,l)`: hoist `h` puts down block `x` on table `t` at location `l`.

3.2 Architectural Overview

The general procedure for obtaining experiences, acquiring task knowledge and planning in EBPDs is built on the conceptual framework in Figure 3.1. In this system, a *user interface* allows a human user to instruct a robot how to carry out complex tasks by providing the robot with sequences of actions. Alternatively, a general purpose standard planner is also integrated into this system to generate solutions for tasks, when a human user is not present or providing a solution is too complex for the human user. When a task is successfully carried out, an *experience extractor* collects and records the world information and the applied actions, during the execution of the task, as a plan-based robot activity experience into an *experience memory*. A *conceptualizer* retrieves experiences from the experience memory and generates and stores activity schemata, i.e., generic methods obtained from single experiences, into a *concept memory*. Activity schemata are abstract semantic structures that are used later during planning to find solutions for similar problems. A *planning* system is used to generate plan solutions to given task problems using the learned activity schemata. The *robot platform* executes the plans

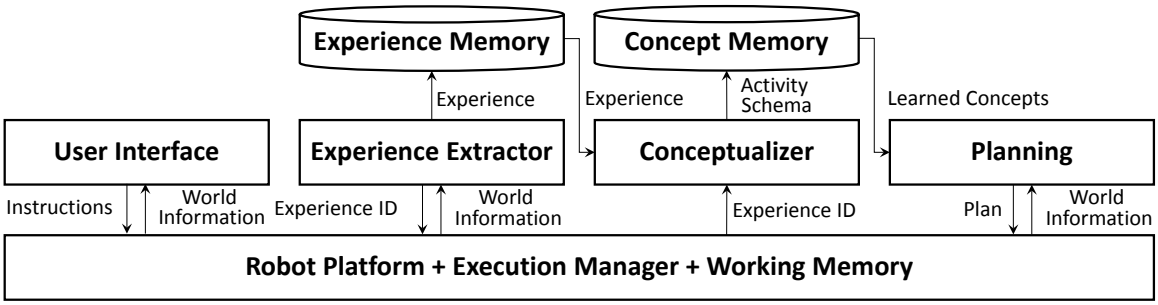


Figure 3.1: An overview of the learning and planning framework underlying an EBPD.

generated by the planner. The experience memory and the concept memory in this framework can be related to episodic memory and semantic memory in cognitive science (Wood et al., 2012). As a prerequisite of EBPDs, it is assumed that the robot is equipped with a set of basic skills (i.e., primitive operators) and we focus on techniques for the robot to acquire generic methods of achieving complex tasks (e.g., serve a coffee to a guest) based on the repertoire of primitive skills.

In this thesis, we address human-robot interaction and experience extraction in Chapter 4, the conceptualization approach in Chapter 5, and the planning approach in Chapter 6. The robot platform, execution manager and perception system are out of scope of this thesis (Stock et al., 2014; Š. Konečný et al., 2014; Kasaei et al., 2015).

3.3 Planning Domains

EBPDs rely on a *first-order language*, $\mathcal{L}$, including finitely many predicate, function and constant symbols to represent various properties of the world. The properties of the world are represented by two sets of predicates, namely the *static world information* and the *planning state*:

Definition 3.1 (**Static World Information**)**.** The *static world information*, Σ, is a set of ground predicates of $\mathcal{L}$ representing the static or invariant properties of the world, i.e., all properties of the world that are always true. ■

Definition 3.2 (**Planning State**)**.** A *planning state* (or, for short, a *state*), s, is a set of ground predicates of $\mathcal{L}$, such that $s \cap \Sigma \neq \emptyset$, representing all dynamic or transient properties of the world, i.e., relevant properties of the world that are true in a given moment but may become false later. The set of all possible states of the world is represented as $\mathcal{S}$. ■

The truth values of some predicates may not vary from state to state. These predicates are called *static* or *state-invariant* relations (Definition 3.1). However, some predicates are intended to vary from state to state. These predicates are called *transient* or *state-variant* relations (Definition 3.2). This distinction between static and transient world information, already explored in (Seabra Lopes, 1999, 2007), allows a planner to only retain or reproduce the transient world information during planning, and retrieves the static information when it is needed. This idea improves the efficiency of planning in terms of both time and memory, as well as the intelligibility of a planning domain representation. A similar idea for static and transient predicates is also addressed in (Ghallab et al., 2004) with the names *flexible* and *rigid* relations.

Problem solving in general can be viewed as transforming an *initial state* into a *final state* using a sequence of *planning operators* (Ghallab et al., 2004). Planning operators are formulas, used by a planner, that specify how the truth values of predicates change from state to state.

Definition 3.3 (**Planning Operator**). A *planning operator*, o, is a tuple,

$$o = (h, S, P, E),$$

where h is the planning operator's head, a functional expression of the form $n(x_1, ..., x_k)$ in which n is the name of the operator and $x_1, ..., x_k$ are the variables appearing anywhere in o, e.g., `(move ?robot ?from ?to)` [1], S is the static precondition of o, a set of predicates that must be proved in the static world information Σ, P is the precondition of o, a set of literals that must be proved in a state $s \in \mathcal{S}$ in order for o to be applicable in s, and E is the effect of o, a set of literals specifying the changes on s effected by o.

The set of predicates whose negations are in P is denoted by P^- and referred to as the negative precondition. The set of the remaining predicates in P is denoted by P^+ and referred to as the positive precondition. The negative effect, E^-, and the positive effect, E^+, are similarly defined. ■

A planning operator in the `RACE` domain, represented using the EBPD syntax, is shown in Listing 3.1.

The purpose of the operator's head is to provide an unambiguous way to refer to the operator without having to write its preconditions and effects explicitly. If

[1] The notation in the Planning Domain Definition Language (PDDL) is used to represent EBPDs. All terms starting with a question mark (?) are variables, and the rest are constants or function symbols.

```
(:action move_base_blind
 :parameters   (?robot - robot ?from ?to - area)
 :static       (and (manipulationarea ?from)
                    (premanipulationarea ?to)
                    (haspremanipulationarea ?from ?to))
 :precondition (robotat ?robot ?from)
 :effect       (and (robotat ?robot ?to)
                    (not (robotat ?robot ?from))))
```

Listing 3.1: The EBPD representation of a planning operator in the RACE domain. With respect to the standard PDDL, static is a new property.

o is an operator, then its head $h = n(x_1,...,x_k)$ refers unambiguously to o. Thus, when it is clear from the context, we will write $n(x_1,...,x_k)$ to refer to the entire operator o.

Actual actions, executed by the agent, are obtained by instantiating planning operators in specific states.

Definition 3.4 (**Action**)**.** Let *action* $a = (n(c_1,...,c_k), S, P, E)$ be the result of instantiating a planning operator $n(x_1,...,x_k) \in \mathcal{O}$, where each c_i is a constant symbol of $\mathcal{L}$ that instantiates a variable x_i in the operator description. In any state $s \in \mathcal{S}$, if $S \subset \Sigma$, $P^+ \subset s$ and $P^- \cap s = \emptyset$, then a is applicable to s and the result of applying a to s is a new state given by the following state-transition function:

$$\gamma(s, a) = (s - E^-) \cup E^+.$$

■

Based on the above definitions, a planning domain for problem solving is defined as follows:

Definition 3.5 (**Planning Domain**)**.** A *planning domain*, $\mathcal{D}$, is a tuple,

$$\mathcal{D} = (\mathcal{L}, \Sigma, \mathcal{S}, \mathcal{O}),$$

where $\mathcal{L}$ is the first-order logic language, Σ is the static world information, $\mathcal{S}$ is the set of all possible states, and $\mathcal{O}$ is the set of planning operators. ■

3.4 Task Planning Problems and Solutions

The goal of task planning is to determine sequences of actions to achieve specific tasks or goals. Here we present the definitions of task planning problems and plan solutions.

Definition 3.6 (**Task Planning Problem**). A *task planning problem*, $\mathcal{P}$, is a tuple of ground structures,

$$\mathcal{P} = (t, \sigma, s_0, g),$$

where t is the target task, a functional expression of the form $n(c_1,...,c_k)$ with n being the task name and each c_i a constant symbol of $\mathcal{L}$, e.g., `(ServeACoffee mug1 counter1 guest1 table1)`, $\sigma \subseteq \Sigma$ is a subset of the static world information, $s_0 \in \mathcal{S}$ is the initial state, and g is the goal, i.e., a set of propositions to be satisfied (g can be empty). ■

EBPDs are intuitively designed for robot task learning and planning. They support two different problem formulations: (*i*) the goal of task planning problems can be explicitly described as a set of goal propositions in a class of task planning problems; and (*ii*) the description of the goal propositions can be empty in a class of task planning problems. In the latter problem formulation, a set of inferred goal propositions, included in the methods (activity schemata), is used for solving a given task planning problem (see Sections 3.7 and 5.6).

Listing 3.2 shows part of a task planning problem in the `RACE` domain [2]. The problem contains 40 constants and objects, 86 static predicates and 12 transient predicates. There is no explicit goal description in this problem, however, the target task is to serve a coffee, initially on a counter, to a guest sitting at a table in a restaurant environment, specified by the head `(ServeACoffee mug1 counter1 guest1 table1)`.

A solution to a task planning problem is given in the form of a plan:

Definition 3.7 (**Plan and Solution**). Any sequence of actions $\pi = \langle a_1, \ldots, a_k \rangle$ for $k \geqslant 0$, is a *plan* if there exists a sequence of states $\langle s_0, \ldots, s_k \rangle$ such that $s_i = \gamma(a_i, s_{i-1})$ for $1 \leqslant i \leqslant k$. The plan π is a *solution* for a planning problem $\mathcal{P} = (t, \sigma, s_0, g)$ if $g \subseteq s_k$. ■

3.5 Abstraction Hierarchies

Problem solving in EBPDs is achieved using a hierarchical problem solver which includes a *concrete planning domain* and an *abstract planning domain*. The concrete planning domain provides the concrete descriptions of a set of states of the environment and a set of actions for making the transitions between the states, and

[2] The full specification of this task planning problem can be found in an online repository: `https://github.com/mokhtarivahid/prletter2017/`.

```
(define (problem ServeACoffee)
(:domain race)
(:parameters mug1 counter1 guest1 table1)
(:constants leftarm1 rightarm1 - arm counter1 - counter ···)
(:objects armposture0 armposture1 - armtuckedposture ···)
(:static (counter counter1)
         (table table1)
         (table table2)
         (manipulationarea manipulationareaeastcounter1)
         (placingarea placingareaeastrightcounter1)
         (premanipulationarea premanipulationareaeastcounter1)
         (haspremanipulationarea manipulationareaeastcounter1
             premanipulationareaeastcounter1)
         (hasmanipulationarea counter1 manipulationareaeastcounter1)
         (hasmanipulationarea placingareaeastrightcounter1
             manipulationareaeastcounter1)
         (hasplacingarea counter1 placingareaeastrightcounter1)
         (manipulationarea manipulationareaeasttable2)
         (manipulationarea manipulationareawesttable2)
         (placingarea placingareasouthrighttable2)
         (placingarea placingareasouthlefttable2)
         (placingarea placingareanorthrighttable2)
         (placingarea placingareanorthlefttable2)
         (premanipulationarea premanipulationareaeasttable2)
         (premanipulationarea premanipulationareawesttable2)
         (sittingarea sittingareanorthtable2)
         (sittingarea sittingareasouthtable2)
         (haspremanipulationarea table2 premanipulationareaeasttable2)
         (haspremanipulationarea table2 premanipulationareawesttable2)
         (haspremanipulationarea manipulationareaeasttable2
             premanipulationareaeasttable2)
         (haspremanipulationarea manipulationareawesttable2
             premanipulationareawesttable2)
         (hasmanipulationarea table2 manipulationareaeasttable2)
         (hasmanipulationarea table2 manipulationareawesttable2)
         (hasmanipulationarea placingareasouthrighttable2
             manipulationareaeasttable2)
         (hasmanipulationarea placingareasouthlefttable2
             manipulationareawesttable2)
          ...
         (mug mug1)
         (guest guest1)
         (torso torso1)
         (at guest1 sittingareawesttable1)
         (arm leftarm1)
         (arm rightarm1)
         (gripper leftgripper1)
         (gripper rightgripper1)
         (hasgripper leftarm1 leftgripper1)
         (hasgripper rightarm1 rightgripper1))
(:init   (on mug1 placingareaeastrightcounter1)
         (robotat trixi1 floorareatamsrestaurant1)
         (armtuckedposture armposture0)
         (armtuckedposture armposture1)
         (hasarmposture leftarm1 armposture0)
         (hasarmposture rightarm1 armposture1)
         (gripperclosedposture gripperposture0)
         (gripperclosedposture gripperposture1)
         (hasgripperposture rightgripper1 gripperposture0)
         (hasgripperposture leftgripper1 gripperposture1)
         (torsodownposture torsoposture0)
         (hastorsoposture torso1 torsoposture0)))
```

Listing 3.2: Part of a task problem for serving a guest in the RACE domain (some information is omitted due to limited space). With respect to the standard PDDL, a distinction between `static` and `init` makes the task planning problem more intuitive. Moreover, the goal description of the problem can be empty (in this case the goal is specified by an activity schema relevant for solving this problem, see Definition 3.16).

the abstract planning domain is intended for an abstract level of problem solving using abstract states and abstract actions.

Definition 3.8 (**Concrete Planning Domain**). A *concrete planning domain* $\mathcal{D}_c = (\mathcal{L}_c, \Sigma_c, \mathcal{S}_c, \mathcal{O}_c)$ is a planning domain that represents the concrete or physical level of problem solving in an application. ■

Definition 3.9 (**Abstract Planning Domain**). An *abstract planning domain* $\mathcal{D}_a = (\mathcal{L}_a, \Sigma_a, \mathcal{S}_a, \mathcal{O}_a)$ is a planning domain that represents an abstract level of problem solving in an application, where less relevant features of a concrete planning domain are ignored. ■

Abstraction is achieved by dropping or transforming predicates and operators of the concrete planning domain $\mathcal{D}_c$ into predicates and operators of the abstract planning domain $\mathcal{D}_a$. In the current state of the research, this transformation involves two independent abstraction hierarchies: a *predicate abstraction hierarchy* and an *operator abstraction hierarchy*:

Definition 3.10 (**Predicate Abstraction Hierarchy**). A *predicate abstraction hierarchy* is a set of abstraction relations, each one relating a concrete predicate $p_c(u_1, \ldots, u_n)$ to: an abstract predicate $p_a(v_1, \ldots, v_m)$ such that $p_c \in \mathcal{L}_c$, $p_a \in \mathcal{L}_a$, $m \leqslant n$ and $\{v_1, \ldots, v_m\} \subseteq \{u_1, \ldots, u_n\}$; or $\varnothing$ (nil). ■

Definition 3.11 (**Operator Abstraction Hierarchy**). An *operator abstraction hierarchy* is a set of abstraction relations, each one relating a concrete operator $o_c \in \mathcal{O}_c$, with arguments $(u_1, \ldots, u_n)$ to: an abstract operator $o_a \in \mathcal{O}_a$, with arguments $(v_1, \ldots, v_m)$, such that $m \leqslant n$ and $\{v_1, \ldots, v_m\} \subseteq \{u_1, \ldots, u_n\}$; or $\varnothing$ (nil). ■

In the predicate abstraction hierarchies, a concrete predicate might: map onto an abstract predicate, by replacing predicate symbols and excluding some arguments of the concrete predicate, e.g., (`holding ?hoist ?block`) $\rightarrow$ (`holding ?block`); or map onto $\varnothing$ (nil), that is, it won't be represented at the abstract level, e.g., (`at ?hoist ?pile`) $\rightarrow \emptyset$. Similarly, in the operator abstraction hierarchy, a concrete operator maps onto an abstract operator, e.g., (`pickup ?hoist ?block ?table ?loc`) $\rightarrow$ (`pickup ?block ?table`); or maps onto $\varnothing$ (nil), e.g., (`move ?hoist ?from ?to ?loc`) $\rightarrow \emptyset$. Tables 3.1 and 3.2 present the predicate and operator abstraction hierarchies in the `RACE` domain, and Tables 3.3 and 3.4 present the predicate and operator abstraction hierarchies in the `Stacking-Blocks` domain.

Table 3.1: Predicate abstraction hierarchy in the RACE EBPD.

Abstract predicate	Concrete predicate
(table ?table)	(table ?table)
(counter ?table)	(counter ?table)
(haspremanipulationarea ?table ?area)	(haspremanipulationarea ?table ?area)
(hasmanipulationarea ?table ?area)	(hasmanipulationarea ?table ?area)
(hasplacingarea ?table ?area)	(hasplacingarea ?table ?area)
(hassittingarea ?table ?sittingarea)	(hassittingarea ?table ?sittingarea)
(manipulationarea ?area)	(manipulationarea ?area)
(manipulationareasouth ?area)	(manipulationareasouth ?area)
(manipulationareanorth ?area)	(manipulationareanorth ?area)
(manipulationareaeast ?area)	(manipulationareaeast ?area)
(manipulationareawest ?area)	(manipulationareawest ?area)
(premanipulationarea ?area)	(premanipulationarea ?area)
(premanipulationareasouth ?area)	(premanipulationareasouth ?area)
(premanipulationareanorth ?area)	(premanipulationareanorth ?area)
(premanipulationareaeast ?area)	(premanipulationareaeast ?area)
(premanipulationareawest ?area)	(premanipulationareawest ?area)
(haspremanipulationarea ?area ?area)	(haspremanipulationarea ?area ?area)
(hasplacingarea ?sittingarea ?area)	(hasplacingarea ?sittingarea ?area)
(placingarealeft ?area)	(placingarealeft ?area)
(placingarearight ?area)	(placingarearight ?area)
(guest ?guest)	(guest ?guest)
(at ?guest ?sittingarea)	(at ?guest ?sittingarea)
(mug ?object)	(mug ?object)
(on ?object ?area)	(on ?object ?area)
(robot ?robot)	(robot ?robot)
(robotat ?robot ?area)	(robotat ?robot ?area)
∅	(objectobserved ?object ?area)
∅	(leftarm ?arm)
∅	(rightarm ?arm)
∅	(hasarmposture ?arm ?armposture)
∅	(armtuckedposture ?armposture)
∅	(armcarryposture ?armposture)
∅	(armtosideposture ?armposture)
∅	(armunnamedposture ?armposture)
∅	(gripper ?gripper)
∅	(hasgripper ?arm ?gripper)
∅	(hasgripperposture ?gripper ?posture)
∅	(gripperclosedposture ?gripperposture)
∅	(gripperholdingposture ?gripperposture)
∅	(gripperopenedposture ?gripperposture)
∅	(torso ?torso)
∅	(hastorsoposture ?torso ?posture)
∅	(torsodownposture ?torsoposture)
∅	(torsoupposture ?torsoposture)
∅	(torsomiddleposture ?torsoposture)

Table 3.2: Operator abstraction hierarchy in the RACE EBPD. *

Abstract operator	Concrete operator
(move/3)	(move_base/3)
(move/3)	(move_base_blind/3)
(pick_up/4)	(pick_up_object/8)
(place/4)	(place_object/8)
∅	(observe_object_on_area/4)
∅	(tuck_arms/6)
∅	(move_arms_to_carryposture/6)
∅	(move_arm_to_side/3)
∅	(move_torso/3)

* Parameters are omitted due to limited space. The numbers indicate arities.

In the remainder of this thesis, a functional expression parent(x), where x is a concrete predicate or operator, represents the parent of x in the respective abstraction hierarchy. Moreover, states and operators from the concrete domain are denoted by s_c and o_c, respectively, while states and operators from the abstract domain are denoted by s_a and o_a, respectively.

As a prerequisite in this work, we assume that descriptions of the concrete and abstract planning domains $(\mathcal{D}_c, \mathcal{D}_a)$ with operators and predicates abstraction hierarchies are given by a domain expert or a knowledge engineer. Research into knowledge acquisition already describes approaches and tools for learning action models (Walsh and Littman, 2008; Zhuo et al., 2008) and for generating abstraction in planning (Knoblock, 1994; Bacchus and Yang, 1994). Nonetheless, the automatic definition of abstract and concrete planning domains is beyond the scope of this thesis. In Chapters 5 and 6, we present how the abstraction is used for learning planning knowledge and for improving the performance of problem solving.

3.6 Plan-Based Robot Activity Experiences

Experiences are episodic descriptions of plan-based robot activities including environmental perception, sequences of applied actions and achieved tasks. By "plan-based robot activity experience" we mean an experience that resulted from the execution of a sequence of actions (see Definitions 3.4 and 3.7). In this work, we seek to generalize and abstract experiences for reuse in new situations. Let $\mathcal{L}$ be

Table 3.3: Predicate abstraction hierarchy in the Stacking-Blocks EBPD.

Abstract predicate	Concrete predicate
(table ?table)	(table ?table)
(pile ?pile)	(pile ?pile)
(block ?block)	(block ?block)
(blue ?block)	(blue ?block)
(red ?block)	(red ?block)
(pallet ?pallet)	(pallet ?pallet)
(on ?block1 ?block2)	(on ?block1 ?block2)
(ontable ?block ?table)	(ontable ?block ?table)
(top ?block ?pile)	(top ?block ?pile)
(holding ?block)	(holding ?hoist ?block)
∅	(location ?location)
∅	(hoist ?hoist)
∅	(attached ?pile ?location)
∅	(belong ?hoist ?location)
∅	(at ?hoist ?pile)
∅	(empty ?hoist)

Table 3.4: Operator abstraction hierarchy in the Stacking-Blocks EBPD.

Abstract operator	Concrete operator
(unstack ?block1 ?block2 ?pile)	(unstack ?hoist ?block1 ?block2 ?pile ?loc)
(stack ?block2 ?block1 ?pile)	(stack ?hoist ?block2 ?block1 ?pile ?loc)
(pick ?block ?table)	(pickup ?hoist ?block ?table ?loc)
(put ?block ?table)	(putdown ?hoist ?block ?table ?loc)
∅	(move ?hoist ?from ?to ?loc)

a first-order language that has finitely many predicate and constant symbols, such that $(\mathcal{L}_c \cup \mathcal{L}_a) \subseteq \mathcal{L}$:

Definition 3.12 (**Key-Property**). A *key-property*, $\tau(p)$, is a property of the world in an experience, where τ is a temporal symbol and $p \in \mathcal{L}$ is a predicate. Temporal symbols specify the temporal extent of predicates in experiences. Three types of temporal symbols are used in key-properties, namely `init`—true at the initial state, `during`—always true during an experience, and `end`—true at the final state, e.g., (`end`(`on` `mug1` `placingareawestrighttable1`)). ■

Definition 3.13 (**Plan-Based Robot Activity Experience**). A *plan-based robot activity experience* (or, for short, an *experience*), e, is a triple of ground structures,

$$e = (t, K, \pi),$$

representing how a task was achieved, where $t = n(c_1,...,c_k)$ is a functional expression representing the achieved task, n is the name of the task, $c_1,...,c_k$ are the arguments of the task, e.g., (ServeACoffeemug1counter1guest1table1), K is a set of key-properties extracted from the robot's memory during the task execution, and π is a solution plan, i.e., a sequence of concrete actions that achieves t. ■

Listing 3.3 shows an experience for a 'ServeACoffee' task in the RACE domain [3]. The head ServeACoffee(mug1,counter1,guest1,table1) specifies the task, performed in this experience. Key-properties specify important properties of the world captured in this experience. This properties include the initial and final state of the experience (properties wrapped in init and end timestamps) as well as the properties that are always true during the experience (properties wrapped in during timestamps). The plan is a solution to this experience (either provided by a human user or automatically generated using a planner).

Experiences are collected through human-robot interaction and instruction-based teaching. In Chapter 4, we present methods and approaches for teaching a robot how to achieve a task as well as for extracting and recording experiences. Extracted experiences are the main inputs for acquiring task knowledge.

3.7 Robot Activity Schemata

Activity schemata are generic methods for achieving tasks. They are obtained from experiences. The learned knowledge is represented as a set of activity schemata where each activity schema is generated from a specific experience. The knowledge stored in an activity schema is a generic solution to a class of tasks. The following definitions are provided for the representation of an activity schema:

Definition 3.14 (**Feature**)**.** Features are properties of abstract operators in learned planning knowledge. Given a task t and an abstract operator o, a *feature* of o can be one of the following:

- A key-property $\tau(p)$ such that p contains only arguments of o, i.e., $\mathtt{args}(p) \subseteq \mathtt{args}(o)$;
- A key-property $\tau(p)$ such that p contains at least one argument of o and at least one argument of t, i.e., $\mathtt{args}(p) \cap \mathtt{args}(o) \neq \emptyset$ and $\mathtt{args}(p) \cap \mathtt{args}(t) \neq \emptyset$;

[3] The full specification of this experience can be found in the online repository: https://github.com/mokhtarivahid/prletter2017/.

```
(:experience ServeACoffee
 :parameters (mug1 counter1 guest1 table1)
 :key-properties
  ((during(table table1))
   (during(counter counter1))
   (during(manipulationarea manareasouthtable1))
   (during(manipulationarea manareaeastcounter1))
   (during(premanipulationarea premanareaeastcounter1))
   (during(premanipulationarea premanareasouthtable1))
   (during(floorarea floorareatamsrestaurant1))
   (during(placingarealeft placingareaeastlefttable1))
   (during(placingarearight placingareaeastrightcounter1))
   (during(placingarearight placingareawestrighttable1))
   (during(leftarm leftarm1))
   (during(rightarm rightarm1))
   (during(gripper rightgripper1))
   (during(gripper leftgripper1))
   (during(torso torso1))
   (during(mug mug1))
   (during(guest guest1))
   (during(robot trixi1))
   (during(sittingarea sittingareawesttable1))
   (during(hasmanipulationarea counter1 manareaeastcounter1))
   (during(haspremanipulationarea counter1 premanareaeastcounter1))
   (during(hasplacingareaeastright counter1 placingareaeastrightcounter1))
   (during(hasmanipulationarea placingareaeastrightcounter1 manareaeastcounter1))
   (during(haspremanipulationarea manareaeastcounter1 premanareaeastcounter1))
   (during(hasmanipulationareasouth table1 manareasouthtable1))
   (during(haspremanipulationareasouth table1 premanareasouthtable1))
   (during(hasplacingareawestright table1 placingareawestrighttable1))
   (during(hassittingareawest table1 sittingareawesttable1))
   (during(hasplacingarearight sittingareawesttable1 placingareawestrighttable1))
   (during(haspremanipulationarea manareasouthtable1 premanareasouthtable1))
   (during(hasmanipulationarea placingareawestrighttable1 manareasouthtable1))
   (during(hasplacingareaeastleft table1 placingareaeastlefttable1))
   (during(hasmanipulationarea placingareaeastlefttable1 manareasouthtable1))
   (during(at guest1 sittingareawesttable1))
   (during(hasgripper leftarm1 leftgripper1))
   (during(hasgripper rightarm1 rightgripper1))
   (init(hastorsoposture torso1 torsoposture0))
   (init(hasarmposture leftarm1 armposture1))
   (init(hasarmposture rightarm1 armposture0))
   (init(robotat trixi1 floorareatamsrestaurant1))
   (init(on mug1 placingareaeastrightcounter1))
   (init(armunnamedposture armposture0))
   (init(armunnamedposture armposture1))
   (init(torsodownposture torsoposture0))
   (end(hastorsoposture torso1 torsoposture6))
   (end(hasarmposture leftarm1 armmovingposture62))
   (end(hasarmposture rightarm1 armposture63))
   (end(robotat trixi1 manareasouthtable1))
   (end(on mug1 placingareawestrighttable1))
   (end(armmovingposture armmovingposture62))
   (end(armtosideposture armposture63))
   (end(torsoupposture torsoposture6)))
 :plan
  ((tuck_arms leftarm1 rightarm1 armposture1 armposture0 armposture7 armposture13)
   (move_base trixi1 floorareatamsrestaurant1 premanareaeastcounter1)
   (move_torso torso1 torsoposture0 torsoposture2)
   (tuck_arms leftarm1 rightarm1 armposture7 armposture13 armposture17 armposture19)
   (move_arm_to_side leftarm1 armposture17 armposture21)
   (move_arm_to_side rightarm1 armposture19 armposture22)
   (move_base_blind trixi1 premanareaeastcounter1 manareaeastcounter1)
   (pick_up_object mug1 rightarm1 manareaeastcounter1 placingareaeastrightcounter1
        rightgripper1 torsoposture2 armposture22 armposture23)
   (move_base_blind trixi1 manareaeastcounter1 premanareaeastcounter1)
   (move_arms_to_carryposture leftarm1 rightarm1 armposture33 armposture35 armposture43
        armposture45)
   (move_torso torso1 torsoposture2 torsoposture4)
   (move_base trixi1 premanareaeastcounter1 premanareasouthtable1)
   (move_torso torso1 torsoposture4 torsoposture6)
   (move_arm_to_side rightarm1 armposture57 armposture61)
   (move_base_blind trixi1 premanareasouthtable1 manareasouthtable1)
   (place_object mug1 rightarm1 manareasouthtable1 placingareawestrighttable1
        rightgripper1 torsoposture6 armposture61 armposture63)))
```

Listing 3.3: An experience for the task of serving a guest in the RACE domain.

- A pair of key-properties $(\tau_1(p), \tau_2(q))$ such that p contains at least one argument of o, q contains at least one argument of t, and p and q have at least one argument in common, i.e., $\texttt{args}(p) \cap \texttt{args}(o) \neq \emptyset$, $\texttt{args}(q) \cap \texttt{args}(t) \neq \emptyset$, and $\texttt{args}(p) \cap \texttt{args}(q) \neq \emptyset$.

■

Features are intended to improve the performance of problem solving by guiding a planner toward a goal state and reducing the probability of backtracking. During problem solving, objects that satisfy the features are preferable to instantiate actions. Moreover, when there are different alternatives to achieve a goal, features are useful to capture preferable alternatives based on different factors, such as social norms, physical constraints (see more in Section 5.3). Features are integrated into enriched abstract operators:

Definition 3.15 (**Enriched Abstract Plan**). An *enriched abstract plan*, denoted by Ω, is a sequence of enriched abstract operators. Each *enriched abstract operator* is a pair,

$$\omega = (o, F),$$

where o is the head of an abstract operator, and F is the set of features of o. ■

Definition 3.16 (**Robot Activity Schema**). A *robot activity schema* (or, for short, an *activity schema*), m, is a triple of unground structures,

$$m = (t, G, \Omega),$$

where t is the target task to be achieved, e.g., `(ServeACoffee ?mug ?counter ?guest ?table)`, G is a set of predicates representing the *goal* of m, and Ω is an enriched abstract plan to achieve the task t. ■

Listing 3.4 shows part of an activity schema in the `RACE` domain [4]. This activity schema is automatically obtained from the 'ServeACoffee' experience in Listing 3.3. During problem solving, this schema is used to solve problem instances of the 'ServeACoffee' task. The target task is specified by the head `(ServeACoffee ?mug ?counter ?guest ?table)`. The ungrounded predicate `(on ?mug ?pawrt)` is the goal of the task, and is instantiated and achieved during problem solving for a given task problem. The (enriched) abstract plan is a generic solution to

[4] The full specification of this activity schema can be found in the online repository: `https://github.com/mokhtarivahid/prletter2017/`.

```
(:activity-schema ServeACoffee
 :parameters (?mug ?counter ?guest ?table)
 :goal (on ?mug ?pawrt)
 :abstract-plan
     (((!move ?robot ?fatr ?pmaec)
         ())
      ((!move ?robot ?pmaec ?maec)
         ((during(hasmanipulationarea ?counter ?maec))
         ((during(hasplacingarea ?counter ?paerc))
          (during(hasmanipulationarea ?paerc ?maec)))···))
      ((!pick_up ?robot ?mug ?maec ?paerc)
         ((end(on ?mug ?pawrt))
          (during(hasmanipulationarea ?counter ?maec))···))
      ((!move ?robot ?maec ?pmaec)
         (((during(haspremanipulationarea ?maec ?pmaec))
           (during(hasmanipulationarea ?counter ?maec)))))
      ((!move ?robot ?pmaec ?pmast)
         ())
      ((!move ?robot ?pmast ?mast)
         ((during(hasmanipulationarea ?pawrt ?mast))
          (during(hasmanipulationarea ?table ?mast))···))
      ((!place ?robot ?mug ?mast ?pawrt)
         (((during(hasplacingarearight ?sawt ?pawrt))
           (during(at ?guest ?sawt)))···))
      ((!move ?robot ?mast ?pmast)
         ((during(haspremanipulationarea ?table ?pmast))
         ((during(haspremanipulationarea ?table ?pmast))
          (during(hasplacingarea ?table ?pawrt)))···))))
```

Listing 3.4: Part of an activity schema for the 'ServeACoffee' task in the RACE domain. (some information is omitted due to limited space).

serve a coffee task. It includes a sequence of enriched abstract operators (starting with an exclamation mark !) associated with features. Some abstract operators may not have features depending on the information included in an experience, as well as task parameters. The features are used during problem solving for quickly instantiating the abstract operators, as well as for capturing alternatives to correctly achieve tasks such as social norms and physical constraints, e.g., the feature `((during(hasplacingarearight ?sawt ?pawrt))(during(at ?guest ?sawt)))` of the abstract operator `(!place ?robot ?mug ?mast ?pawrt)` captures principles that the guest should be served on the right side by social convention.

The goal of an activity schema is automatically inferred from an experience. It may become empty depending on the information included in an experience. In Section 5.6, we present the method of inferring a set of goal propositions from an experience. As the main contribution of this thesis, we will present, in Chapter 5, a *conceptualization* methodology to derive activity schemata from concrete experiences, and, in Chapter 6, a planning system for solving problems using the learned activity schemata.

3.8 Experience-Based Planning Domains

The conceptual framework for planning and learning from experiences involves planning domains, experiences and learned methods. Therefore, based on the definitions provided above, we now define:

Definition 3.17 (**Experience-Based Planning Domain**). An *Experience-Based Planning Domain* (EBPD), Δ, is a tuple,

$$\Delta = (\mathcal{L}, \mathcal{D}_a, \mathcal{D}_c, \mathcal{A}, \mathcal{E}, \mathcal{M}),$$

where $\mathcal{L}$ is a first-order logic language, $\mathcal{D}_c$ is a concrete planning domain, $\mathcal{D}_a$ is an abstract planning domain, $\mathcal{A}$ is a set of abstraction hierarchies that relates $\mathcal{D}_c$ to $\mathcal{D}_a$, $\mathcal{E}$ is a set of experiences, and $\mathcal{M}$ is a set of activity schemata (methods) for solving problems. The concrete language, $\mathcal{L}_c$, and the abstract language, $\mathcal{L}_a$, are subsets of $\mathcal{L}$. ■

The abstract planning domain is intended to support an abstract level of problem solving in which less relevant features of a problem are ignored and abstract (skeletal) solutions are derived in a coarse fashion with less effort. The abstract solutions are then refined to become concrete solutions to problems using the concrete planning domain. The experiences are records of previous problem solving episodes, automatically collected through human-robot interaction and experience extraction procedures (see Chapter 4). The activity schemata are automatically derived from experiences and serve as methods for solving classes of problems (see Chapters 5 and 6).

3.9 Summary

In this chapter, we proposed a new planning representation — Experiences-Based Planning Domains (EBPDs) — which allows for representing plan-based robot activity experiences and robot task knowledge (i.e., activity schemata). EBPDs provide a powerful and expressive representation based on the PDDL notation and integrate concepts required for learning and planning into a distinct planning language.

EBPDs rely on abstract and concrete planning domains for problem solving. As a prerequisite, we assume that the representations of concrete and abstract domains (state description and operators) are given by a domain expert. An abstract

language which is given by the user has the advantage that abstraction is expressed in a language with which the user is familiar. Consequently, understandability and explainability, which are always important issues in this kind of systems, can be achieved more easily. Compared to approaches in which abstraction hierarchies are generated automatically, more effort is required to specify the abstract language, but we feel that this is a price we have to pay to make planning more tractable in certain situations.

Chapter 4

Human-Robot Interaction and the Extraction of Experiences

Robots are today becoming part of our everyday life and starting to provide close physical assistance to humans. This evolution implies the ability of robots to learn semantic task knowledge from humans. Any approach based on preprogramming all possible courses of actions at the design stage is infeasible for fairly unstructured settings. Cognitive approaches are essential to endow robots with intelligent abilities to learn the appropriate behavior from *human teachers* and *experiences*, and to reason about how to deal with complex environments. One promising approach is *interactive task learning* which takes inspiration from how we teach humans new tasks (Merrill, 2002; Laird et al., 2017).

This involves a human-robot interaction between a robot that learns new tasks and a human instructor available to teach the tasks. The principal objective of Human-Robot Interaction (HRI) is to develop the algorithms making robots capable of direct, safe and effective interaction with humans. Many facets of HRI research relate to and draw from insights and principles from psychology, communication, anthropology, philosophy, and ethics, making HRI an inherently interdisciplinary endeavor (Goodrich and Schultz, 2007; Feil-Seifer and Matarić, 2009).

The other important asset to intelligent robots is the ability to exploit past experiences. Experiences provide a rich resource for learning and problem solving. They help avoiding difficulties, predicting the effects of activities, and obtaining commonsense insights. In order to leverage experiences in a meaningful way, a robot must be able to reason about several aspects of the domain in which it operates. These aspects pertain to different types of knowledge including: *ontological knowledge* about the entities in the world and the concepts used to describe them; *factual knowledge* about the state of the world and the contingent properties of the

objects in it; and *action knowledge* about the causes and effects of situations and actions in order to achieve a goal.

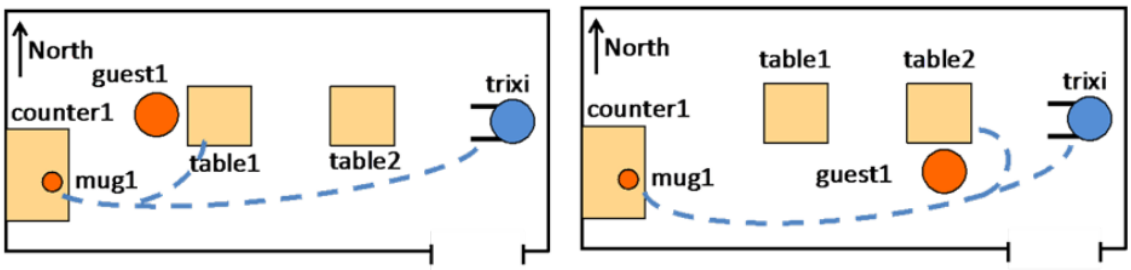
In this chapter, we focus on a specific type of interaction: human-robot collaborative task achievement supported by an instruction-based communication interface. In this interaction, a task is described by a human teacher using step-by-step instructions. We expect the robot to immediately learn from every interaction with a human instructor, which requires one-shot learning instead of repeated practice over large datasets (although practice might help tune the learned knowledge). In Section 4.2, we first present the knowledge representation framework relevant for interactive experience gathering. In Sections 4.3 and 4.4, we present human-robot interaction protocols and functionalities for teaching new tasks and recording experiences of robot activities. The conceptual framework and respective implementation were developed in the EU RACE project where a PR2 robot was employed for recording experiences, learning and applying the learned concepts (see Section 1.3).

In the RACE architecture, experiences are stored at multiple levels of abstraction, from high-level descriptions in terms of goals, tasks and behaviors, to sensory and actuator skills at the lowest level. Experiences provide a detailed account of how the robot has achieved past goals or how it has failed, and what sensory events have accompanied the activities. Robot competence is obtained by abstracting and generalizing from experiences, extending task planning and execution beyond preconceived situations. To achieve these, a common conceptual framework for representing robot experiences, planning and learning was establish.

A central component of the RACE architecture was a semantic memory system implemented as an RDF database. It was mainly used in RACE to keep track of the evolution of both the internal state of the robot, the executed actions and the events observed in the environment. Every unit of data recorded into the memory is called *fluent*, which describes an instance of some concept in the ontology with time points representing the time in which that instance has occurred.

4.1 Running Example: Teaching Tasks in the RACE Domain

One of the demonstrations in RACE focused on coffee serving. This demonstration includes two Scenarios A and B sketched in Figure 4.1. In Scenario A, the robot



(a) Scenario A (b) Scenario B

Figure 4.1: Initial states of the restaurant floor for 'ServeACoffee' tasks with `trixi` PR2 robot in Scenarios A and B.

– here called `trixi` – receives detailed instructions on how to serve a coffee to a guest and learns that this activity constitutes a 'ServeACoffee' task:

- Instructions for Scenario A: "Move to counter1, grasp mug1, move to south of table1, place mug1 at right west placement area of guest1 — this is a 'ServeACoffee'."

In Scenario B, it is assumed that the robot has already learned a concept from the example and will serve a coffee to right placement area at south of guest1.

- Instructions for Scenario B: "Do a 'ServeACoffee' to guest1 at table2."

In both scenarios, we assume that the robot knows the location of the guest and the placement areas on the table. However, it does not know which placement area to approach for guest1.

We explore this simple example, in this chapter, for interactive teaching and experience extraction.

4.2 Knowledge Representation Aspects

Different concepts relevant for planning and execution of robot tasks are covered in the RACE OWL2 ontology (Pecora et al., 2012; Rockel et al., 2013). In OWL, an ontology contains class definitions consisting of a class name and relations to other classes: unary relations for class membership and binary relations for the definition of properties. Due to the scope of this thesis, we only present the ontological concepts for representing experiences and user instructions in the RACE project.

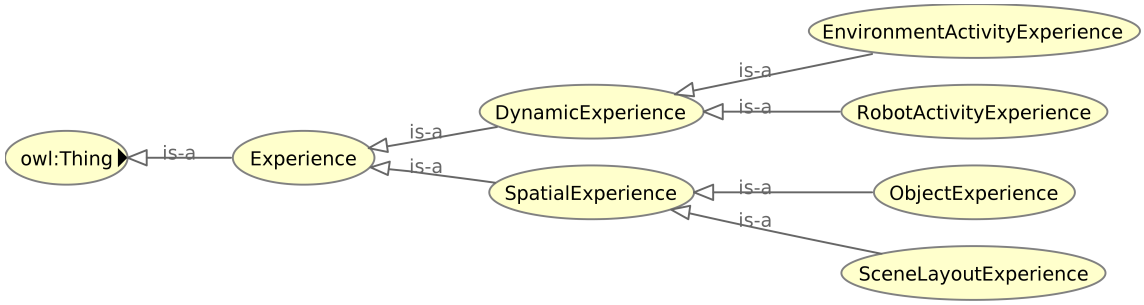


Figure 4.2: Partial representation of the experience ontology in the RACE domain.

4.2.1 Ontology of Experiences

Experiences are the main source of information used for learning how to achieve a more robust robot behavior. They are records of past happenings stored by a robot and interpreted according to the robot's conceptual framework. Experiences arise from robot observations and activities and are typically abstract from low-level data. In this section, we present the format adopted in RACE for representing experiences, collected through a human-robot interaction. These data play a central role in RACE as experimental input for the learning processes. A single experience represents events and physical objects as perceived or inferred by the robot in a coherent time span and confined to the restaurant environment. In the RACE domain ontology, a basic distinction was introduced between *spatial* and *dynamic* experiences. Spatial experiences can be related to the evolutionary roots of human semantic memory (Tulving, 2005). They are relevant for conceptualizing the spatial and relational structure of passive physical objects and scenes, in particular the shape and appearance of objects and the layout of scenes. Spatial experiences do not include the time dimension. By contrast, dynamic experiences contain data that describes activities as sets of occurrences, i.e., temporally extended relations or properties. Figure 4.2 shows the taxonomical organization of the experience concepts in the RACE upper ontology. The root concept, `Thing`, represents every entity which is needed for the field of application.

Considering the scope of this thesis, the ontological concepts only related to dynamic experiences are presented here (see Listing 4.1). We use the Manchester Syntax[1] for textual concept representations. The `Experience` concept represents an experience record containing data that describes an instance of a category to be learned along with a name (`CategoryLabel`) provided by an instructor or an

[1] `https://www.w3.org/TR/owl2-manchester-syntax/`

```
Class: Experience
SubClassOf: Thing
  that hasCategoryLabel some String
  and hasExperienceStartTimeLowerBound some time
  and hasExperienceStartTimeUpperBound some time
  and hasExperienceEndTimeLowerBound some time
  and hasExperienceEndTimeUpperBound some time

Class: DynamicExperience
SubClassOf: Experience
  that hasSceneObject some SceneObject

Class: RobotActivityExperience
SubClassOf: DynamicExperience
  that hasPlanObject some PlanObject
```

Listing 4.1: Ontology of experiences in OWL2 Manchester Syntax.

unsupervised internal process. `DynamicExperience` is an experience relevant for conceptualizing activities in terms of sets of occurrences in a coherent subset of space-time. It contains data about conceptual objects used to represent behavior concepts which are either states or occurrences, and physical objects to present existing entities. `RobotActivityExperience` is a dynamic experience containing data about an activity of the robot, extracted from the occurrence history during the execution of that activity. The experience data includes the underlying goals, executed plan, a set of occurrences and success information. Robot activity experiences are relevant for conceptualizing a given category of robot activities.

4.2.2 Human-Robot Interaction Ontology

A Human-Robot Interaction (HRI) ontology was developed for supervised experience gathering. HRI is used in situations where the robot fails to autonomously deal with exceptional situations as well as for teaching knowledge relevant for new tasks. Figure 4.3 shows the taxonomical composition of the instructor concept in the RACE domain.

The human-robot interaction ontology is centered on the instructor activities. Listing 4.2 shows part of the `InstructorActivity` ontology considering the scope of this thesis. The concept `InstructorAchieve` is a dialog move of imperative nature. It is used to request the robot to achieve some goal or to perform some activity. `InstructorTell` is a dialog move of general-purpose declarative nature. `InstructorTeach` is used for providing information in terms of assertions about the scene. A special case is `TeachActivityCategory` which is useful for teaching a new task.

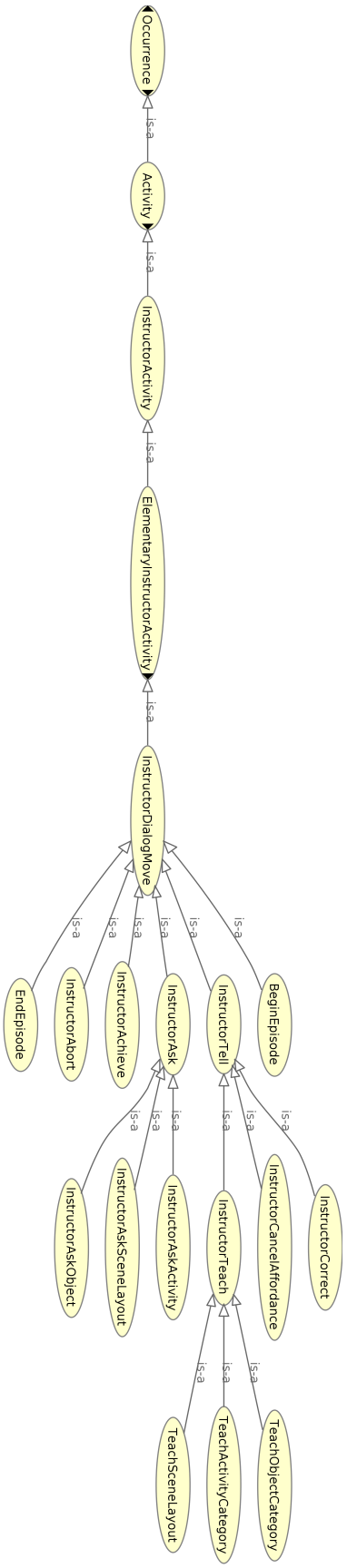


Figure 4.3: Representation of the instructor ontology of the RACE domain.

```
Class: ElementaryInstructorActivity
SubClassOf: InstructorActivity

Class: InstructorDialogMove
SubClassOf: ElementaryInstructorActivity
  that hasSentence some string

Class: InstructorAchieve
SubClassOf: InstructorDialogMove
  that hasTask exactly 1 Task
  and hasPlanConstraint MIN 0 PlanConstraint

Class: InstructorTell
SubClassOf: InstructorDialogMove
  that hasConstrainedState exactly 1 ConstrainedState
  and hasRelationSubject max 1 (SceneObject or PlanObject)
  and hasRelationObject max 1 SceneObject

Class: InstructorTeach
SubClassOf: InstructorTell
  that hasRelationObjectName some string

Class: TeachActivityCategory
SubClassOf: InstructorTeach
```

Listing 4.2: Ontology for instructions related to robot activities in OWL2 Manchester Syntax.

4.3 Interactive Teaching

During the robot's performance in an environment, different data are added and recorded into the robot's memory to represent the evolution of the robot's perception of the environment. Robot activity experiences are subsets of this information in the robot's memory that are extracted based on cues from the human user. In RACE, a command line interface was developed and used to support the communication between the robot and the human user (Chauhan et al., 2013). The interface allows the user to provide instructions for the successive steps in a task, such as driving to locations and picking or placing objects. The user can also teach a set of performed activities with a given name. The following types of instructions are mainly used for teaching a robot a new task and gathering plan-based robot activity experiences:

- **achieve**: this instruction corresponds to the `InstructorAchieve` concept and takes as input a task to be performed with respective arguments:
  Format: `achieve <task name> <task arguments>`
- **teach_task**: this instruction corresponds to the `TeachActivityCategory` concept and takes as input the name of the task that user wishes to teach and the respective arguments:
  Format: `teach_task <task name> <task arguments>`

```
achieve drive_robot_Task preManipulationAreaEastCounter1
achieve grasp_object_w_arm_Task mug1 rightArm1
achieve drive_robot_Task preManipulationAreaSouthTable1
achieve put_object_Task mug1 placingAreaWestRightTable1
teach_task ServeACoffee mug1 guest1 table1 counter1
```

Listing 4.3: A sequence of textual instructions to teach a robot how to serve a coffee to a guest.

This interface is tightly integrated with the robot's memory. When the robot carries out a task or achieves a goal, a `teach_task` instruction triggers the experience extraction. We consider *temporal segmentation heuristics*, i.e., the start time of the first `achieve` instruction and the end time of the `teach_task` instruction, to extract a subset of data related to the given task. All data active (started and/or ended) within this interval become part of the experience pertaining to the new activity taught by an instructor. Listing 4.3 shows an example of a sequence of instructions for teaching a robot the standard method of serving a coffee to a guest. In this example, `achieve` instructions specify instances of the `Task` concept to be achieved. The user interface creates a fluent for every `achieve` instruction which is dispatched for execution. Note that in the RACE project, the achieve instructions included compound HTN tasks for which an HTN planner was employed to generate the respective plans (Stock et al., 2014; Š. Konečný et al., 2014). Experience gathering is immediately carried out after `teach_task` instruction.

Providing a robot with instructions and plan solutions through human-robot interaction has some advantages over using a standard planner. For instance, when there are different alternatives to achieve a goal, some alternatives may be preferable to correctly achieve the goal based on different factors that have not been encoded in the domain specification, such as social norms, physical constraints, etc. Nevertheless, a standard planner can be alternatively integrated in the robot system to generate a plan solution for a given task demonstration, when a human user is not present or providing a suitable sequence of actions to achieve the task is too complex for the human user.

4.4 Robot Activity Experience Extraction

The robot's memory contains fluents deriving from perception and proprioception. It also contains the executed actions, the fluents representing instructions posted by users through the user interface, and the results of reasoning performed by the various reasoning services, e.g., plans to achieve the posted instructions. Each fluent is an instance of a concept in the OWL ontology. Fluents can describe different

```
---
!Fluent
Class_Instance: [On, on4]
StartTime: [0, 0]
FinishTime: [17.513, 17.513]
Properties:
  - [hasArea, Area, placingAreaEastRightCounter1]
  - [hasPhysicalEntity, PhysicalEntity, mug1]
```

Listing 4.4: An example of a fluent in the RACE robot's memory represented in YAML syntax.

kinds of information as part of an experience. Listing 4.4 shows an example of a fluent in the RACE robot's memory represented in the YAML syntax (Ben-Kiki et al., 2005). In this fluent, the field `Class_Instance` contains the concept in the ontology and the name of the instance. It is followed by two intervals, indicating the start and finish times, and a number of properties of the `Fluent`. Time points are specified as uncertainty ranges with a minimum and maximum, which may be empty if unknown. Each property takes one line, starting with a dash and consisting of the triple `[<property name>, <filler class>, <filler name>]`. Fillers may introduce new fluents with new names which can be used in other fluents.

An experience is an episodic description of the execution of a plan-based robot activity including environment perception, sequence of executed actions and achieved task. The robot's memory is a semantic network, i.e., a graph where nodes represent objects or entities and edges represent relations. Hence, experience extraction is the problem of finding a sub-graph in the robot's memory pertaining to the achieved task. Relations (i.e., predicates) are defined over the fluents, for instance in Listing 4.4, one relation is defined as `(On mug1 placingAreaEastRightCounter1)`. Since the robot's memory may contain insignificant and irrelevant data for task learning, we employ a graph simplification approach based on *ego networks* (Newman, 2003) to determine which piece of information are relevant and to extract a sub-graph of the robot's memory for representing the experience. An ego network is a network centered on a specific focal node which is called *ego*. To construct an ego network, the neighbor nodes of the ego are decided based on the length of the paths to the ego. The result is a sub-graph describing the neighborhoods surrounding the egos, which may reveal something important from the egos' perspective.

We present our approach for extracting and representing an experience in the robot's memory in Algorithm 1. First, all relations pertaining to the static world information, initial state and final state are extracted and wrapped with the appropriate temporal symbols `during`, `init` and `end` (lines 4-11). Since experience

Algorithm 1 Experience Extraction

input:
- t ▷ the task achieved in an experience (taught by a user)
- π ▷ sequence of actions to achieve the task t
- F ▷ set of fluents representing the world during an experience

output:
- $e = (t, K, \pi)$ ▷ a robot activity experience (Def. 3.13)

1 $t_1 \leftarrow$ start time of the first action in π
2 $t_2 \leftarrow$ finish time of the last action in π
3 $R \leftarrow \emptyset$
4 **for each** fluent f in F **do**
5 $p \leftarrow \textsc{Predicate}(f)$ ▷ make a predicate for f
6 **if** $(f.StartTime < t_1 \wedge f.FinishTime > t_1 \wedge f.FinishTime < t_2)$ **then**
7 $R \leftarrow R \cup \{\texttt{init}(p)\}$ ▷ fluents pertaining to the initial state
8 **else if** $(f.FinishTime > t_2 \wedge f.StartTime < t_2 \wedge f.StartTime > t_1)$ **then**
9 $R \leftarrow R \cup \{\texttt{end}(p)\}$ ▷ fluents pertaining to the final state
10 **else if** $(f.StartTime \leqslant t_1 \wedge f.FinishTime \geqslant t_2)$ **then**
11 $R \leftarrow R \cup \{\texttt{during}(p)\}$ ▷ fluents pertaining to the static world
12 $\mathcal{E} \leftarrow \texttt{args}(t) \cup \{x \mid x \in \texttt{args}(a), \forall a \in \pi\}$ ▷ set of egos
13 $O \leftarrow \{$objects involved in $R\}$
14 $\mathcal{N} \leftarrow \{w \in O \mid \exists u, v \in \mathcal{E}, \exists r, s \in R, \{u, w\} \subseteq \texttt{args}(r), \{w, v\} \subseteq \texttt{args}(s)\}$ ▷ set of neighbors of at least two egos
15 $K \leftarrow \{r \in R \mid \texttt{args}(r) \subseteq (\mathcal{E} \cup \mathcal{N})\}$
16 **return** (t, K, π)

extraction is based on plan-based robot activities, the arguments of the taught task (i.e., experience parameters) and the arguments of the plans' operators constitute the ego nodes (line 12). Then, we compute the one-step and two-step neighbors of the ego nodes (line 14). Finally, all relations over these nodes are decided as the set of key-properties (line 15) and recorded with the taught task name and provided task's arguments, by the human user, as well as the plan solution as an experience (line 16).

Table 4.1 presents a summary of the concepts and instances that can be found in the robot's memory at the end of the 'ServeACoffee' task. A total of 710 concept instances were found, the most common ones are poses, bounding boxes, arm postures and torso postures. Table 4.2 presents all relations (2281 in total) between instances in the robot's memory at the end of the mentioned task, along with the frequency of use of each relation.

In Figures 4.4 and 4.5, we use graph visualization to help the readers compre-

hend the significance of the proposed experience extraction approach based on ego networks. Figure 4.4 shows the robot's memory content for the 'ServeACoffee' task and Figure 4.5 shows the extracted experience content for the same task.

Listing 4.5 shows the EBPD representation (Definition 3.13) of the extracted experience for the 'ServeACoffee' task in Scenario A.

4.5 Summary

In this chapter, we presented the first steps in the direction of building cognitive functionalities required for teaching robots to perform complex tasks and acquiring experiences. We proposed ontological concepts for internally representing experiences. We used a simple instruction-based interface to support human-robot interaction and experience gathering. The interface allows the human user to command the robot to perform primitive behaviors, as well as to teach a compound task consisting of a sequence of actions taken by the robot. A graph simplification approach, based on ego networks, was developed to filter out irrelevant information in the robot memory and to extract important information relevant for conceptualizing the experience. In this approach, some objects in an experience are selected as egos, and their neighbors are computed based on length of the path to the egos. The resulting sub-graph describes the neighborhoods surrounding the egos, which contain important information from the egos' perspective.

Recorded experiences are the source of information used for learning. The next step is to generate useful high-level planning knowledge from the acquired experiences. The following chapter will present a conceptualization approach consisting of several techniques to construct robot activity schemata from experiences.

Table 4.1: Concepts and respective number of instances in the robot's memory for a 'ServeACoffee' task.

Concept	Count	Concept	Count
Pose	266	ManipulationAreaWest	1
BoundingBox	133	ManipulationConstraintEastCounter	1
ArmMovingPosture	38	ManipulationConstraintEastVerticalTable	1
RobotAt	22	ManipulationConstraintNorthHorizontalTable	1
PrimitiveTask	16	ManipulationConstraintSouthHorizontalTable	1
ArmUnnamedPosture	14	ManipulationConstraintWestVerticalTable	1
ArmToSidePosture	13	NearAreaCounter	1
ArmCarryPosture	10	NearAreaDoor	1
GripperOpenedPosture	8	NearConstraint	1
ArmUntuckedPosture	5	NearConstraintCounter	1
InstructorAchieve	4	PickUpObject	1
On	4	PlaceObject	1
TorsoMiddlePosture	4	PlacingAreaEastLeft	1
ArmTuckedPosture	3	PlacingAreaNorthLeft	1
GripperClosedPosture	3	PlacingAreaNorthRight	1
GripperHoldingPosture	3	PlacingAreaWestLeft	1
MoveArmToSide	3	PlacingAreaWestRight	1
MoveBaseBlind	3	PlacingConstraintEastLeftHorizontalTable	1
TopArea	3	PlacingConstraintEastRightCounter	1
Arm	2	PlacingConstraintEastRightHorizontalTable	1
Gripper	2	PlacingConstraintNorthLeftVerticalTable	1
ManipulationAreaEast	2	PlacingConstraintNorthRightVerticalTable	1
MoveBase	2	PlacingConstraintSouthLeftVerticalTable	1
MoveTorso	2	PlacingConstraintSouthRightVerticalTable	1
NearAreaTable	2	PlacingConstraintWestLeftHorizontalTable	1
NearConstraintTable	2	PlacingConstraintWestRightHorizontalTable	1
PlacingAreaEastRight	2	PlacingAreaSouthLeft	1
PreManipulationAreaEast	2	PlacingAreaSouthRight	1
PreManipulationAreaNorth	2	PR2	1
PreManipulationConstraintEast	2	PreManipulationAreaWest	1
TorsoMovingPosture	2	PreManipulationConstraintNorth	1
TorsoUpPosture	2	PreManipulationConstraintSouth	1
TuckArms	2	PreManipulationConstraintWest	1
At	1	Restaurant	1
Counter	1	SittingAreaEast	1
Door	1	SittingAreaNorth	1
FloorArea	1	SittingAreaSouth	1
Guest	1	SittingAreaWest	1
HorizontalTable	1	SittingConstraintEastHorizontalTable	1
InstructorTeach	1	SittingConstraintNorthVerticalTable	1
Mug	1	SittingConstraintSouthVerticalTable	1
TorsoDownPosture	1	SittingConstraintWestHorizontalTable	1
ManipulationAreaNorth	1	Torso	1
ManipulationAreaSouth	1	VerticalTable	1

Table 4.2: Relations and respective frequency of use in the robot's memory for a 'ServeACoffee' task.

Relation	Count	Relation	Count
hasPose	266	hasPlacingAreaWestRight	2
hasX	269	hasPlacingConstraintEastLeft	2
hasY	269	hasPlacingConstraintEastRight	2
hasZ	269	hasPreManipulationConstraintEast	2
hasYaw	253	hasFloorArea	1
hasBoundingBox	133	hasManipulationAreaEast	1
hasXSize	134	hasManipulationAreaNorth	1
hasYSize	134	hasManipulationAreaSouth	1
hasZSize	132	hasManipulationAreaWest	1
hasArmPosture	79	hasManipulationConstraintNorth	1
hasTask	42	hasManipulationConstraintSouth	1
hasArea	28	hasManipulationConstraintWest	1
hasRobot	18	hasNearAreaCounter	1
hasResult	16	hasNearConstraintCounter	1
hasManipulationArea	15	hasPlacingConstraintNorthLeft	1
hasPhysicalEntity	14	hasPlacingConstraintNorthRight	1
hasGripperPosture	12	hasPlacingConstraintSouthLeft	1
hasPlacingArea	12	hasPlacingConstraintSouthRight	1
hasPreManipulationArea	11	hasPlacingConstraintWestLeft	1
hasTorsoPosture	7	hasPlacingConstraintWestRight	1
hasNearArea	5	hasPreManipulationAreaEast	1
hasIntent	4	hasPreManipulationAreaNorth	1
hasOn	4	hasPreManipulationAreaSouth	1
hasSittingArea	4	hasPreManipulationAreaWest	1
hasPlacingAreaEastRight	3	hasPreManipulationConstraintNorth	1
hasTopArea	3	hasPreManipulationConstraintSouth	1
hasArm	2	hasPreManipulationConstraintWest	1
hasManipulationConstraintEast	2	hasSittingAreaEast	1
hasNearAreaTable	2	hasSittingAreaNorth	1
hasNearConstraintTable	2	hasSittingAreaSouth	1
hasPlacingAreaEastLeft	2	hasSittingAreaWest	1
hasPlacingAreaNorthLeft	2	hasSittingConstraintEast	1
hasPlacingAreaNorthRight	2	hasSittingConstraintNorth	1
hasPlacingAreaSouthLeft	2	hasSittingConstraintSouth	1
hasPlacingAreaSouthRight	2	hasSittingConstraintWest	1
hasPlacingAreaWestLeft	2		

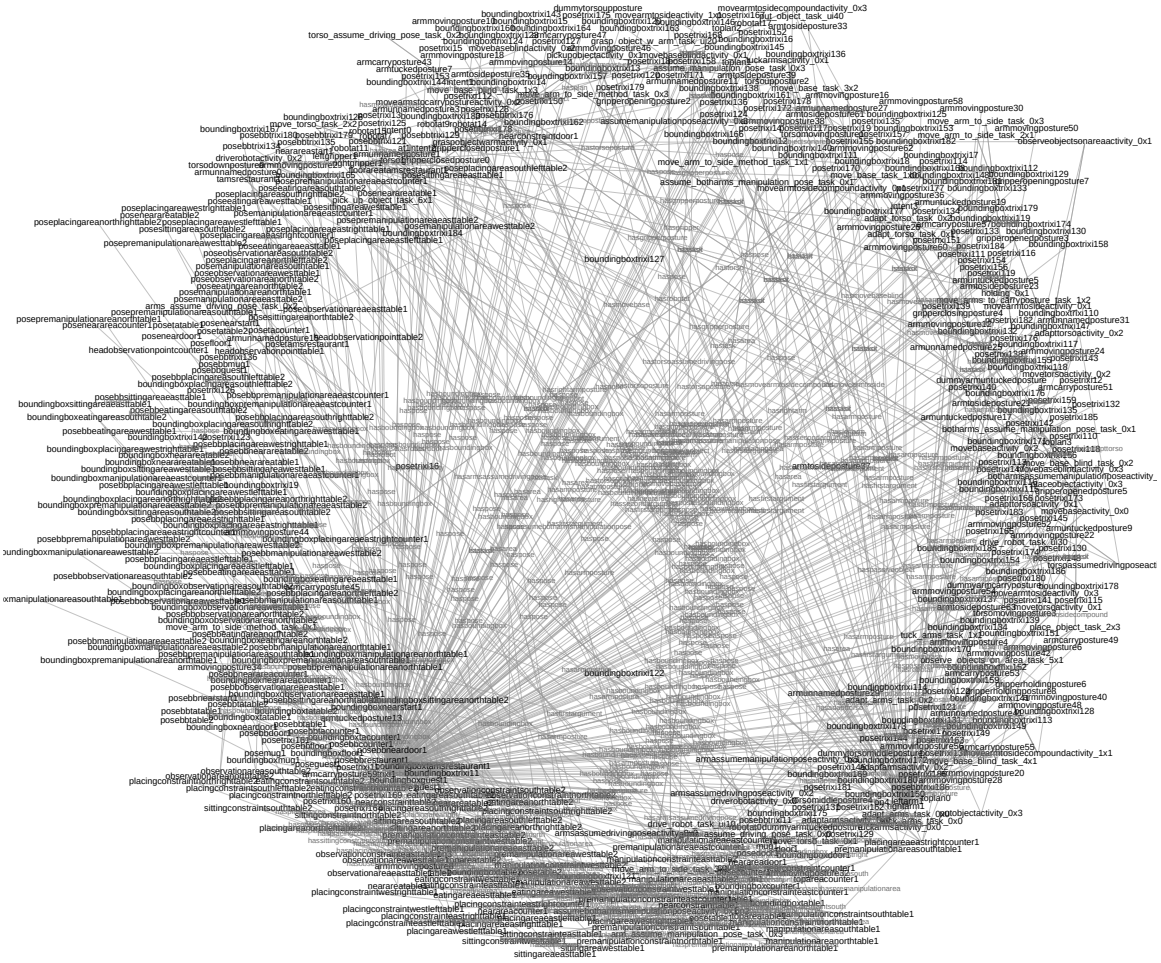


Figure 4.4: The robot's memory content during the 'ServeACoffee' task (710 instances and 2281 relations).

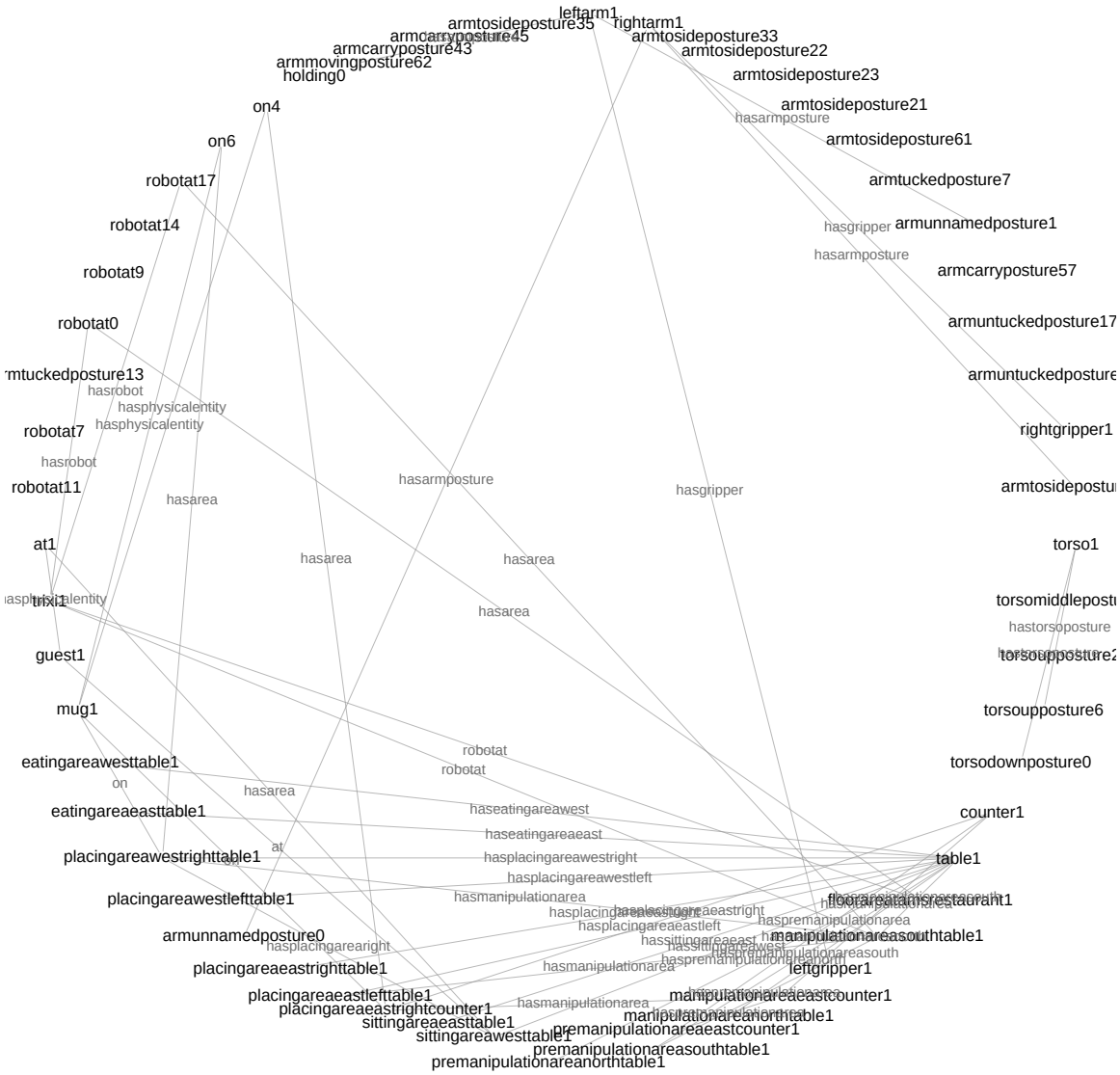


Figure 4.5: Content of the extracted 'ServeACoffee' experience after the simplification approach based on the ego networks (56 instances and 54 relations).

```
(:experience ServeACoffee
 :parameters (mug1 counter1 guest1 table1)
 :key-properties
  ((during(table table1))
   (during(counter counter1))
   (during(manipulationarea manipulationareasouthtable1))
   (during(manipulationarea manipulationareaeastcounter1))
   (during(premanipulationarea premanipulationareaeastcounter1))
   (during(premanipulationarea premanipulationareasouthtable1))
   (during(floorarea floorareatamsrestaurant1))
   (during(placingarealeft placingareaeastlefttable1))
   (during(placingarearight placingareaeastrightcounter1))
   (during(placingarearight placingareawestrighttable1))
   (during(leftarm leftarm1))
   (during(rightarm rightarm1))
   (during(gripper rightgripper1))
   (during(gripper leftgripper1))
   (during(torso torso1))
   (during(mug mug1))
   (during(guest guest1))
   (during(robot trixi1))
   (during(sittingarea sittingareawesttable1))
   (during(hasmanipulationarea counter1 manipulationareaeastcounter1))
   (during(haspremanipulationarea counter1 premanipulationareaeastcounter1))
   (during(hasplacingareaeastright counter1 placingareaeastrightcounter1))
   (during(hasmanipulationarea placingareaeastrightcounter1 manipulationareaeastcounter1))
   (during(haspremanipulationarea manipulationareaeastcounter1 premanipulationareaeastcounter1))
   (during(hasmanipulationareasouth table1 manipulationareasouthtable1))
   (during(haspremanipulationareasouth table1 premanipulationareasouthtable1))
   (during(hasplacingareawestright table1 placingareawestrighttable1))
   (during(hassittingareawest table1 sittingareawesttable1))
   (during(hasplacingarearight sittingareawesttable1 placingareawestrighttable1))
   (during(haspremanipulationarea manipulationareasouthtable1 premanipulationareasouthtable1))
   (during(hasmanipulationarea placingareawestrighttable1 manipulationareasouthtable1))
   (during(hasplacingareaeastleft table1 placingareaeastlefttable1))
   (during(hasmanipulationarea placingareaeastlefttable1 manipulationareasouthtable1))
   (during(at guest1 sittingareawesttable1))
   (during(hasphysicalentity at1 guest1))
   (during(hasarea at1 sittingareawesttable1))
   (during(hasgripper leftarm1 leftgripper1))
   (during(hasgripper rightarm1 rightgripper1))
   (init(hastorsoposture torso1 torsodownposture0))
   (init(hasarmposture leftarm1 armunnamedposture1))
   (init(hasarmposture rightarm1 armunnamedposture0))
   (init(robotat trixi1 floorareatamsrestaurant1))
   (init(on mug1 placingareaeastrightcounter1))
   (init(armunnamedposture armunnamedposture0))
   (init(armunnamedposture armunnamedposture1))
   (init(torsodownposture torsodownposture0))
   (end(hastorsoposture torso1 torsoupposture6))
   (end(hasarmposture leftarm1 armmovingposture62))
   (end(hasarmposture rightarm1 armtosideposture63))
   (end(robotat trixi1 manipulationareasouthtable1))
   (end(on mug1 placingareawestrighttable1))
   (end(armmovingposture armmovingposture62))
   (end(armtosideposture armtosideposture63))
   (end(torsoupposture torsoupposture6)))
 :plan
  ((tuck_arms leftarm1 rightarm1 armunnamedposture armunnamedposture armunnamedposture1
      ↪ armunnamedposture0 armtuckedposture armtuckedposture armtuckedposture7 armtuckedposture13)
   (move_base trixi1 floorareatamsrestaurant1 premanipulationareaeastcounter1)
   (move_torso torso1 torsodownposture torsodownposture0 torsoupposture torsoupposture2)
   (tuck_arms leftarm1 rightarm1 armtuckedposture armtuckedposture armtuckedposture7 armtuckedposture13
       ↪  armuntuckedposture armuntuckedposture armuntuckedposture17 armuntuckedposture19)
   (move_arm_to_side leftarm1 armuntuckedposture armuntuckedposture17 armtosideposture21)
   (move_arm_to_side rightarm1 armuntuckedposture armuntuckedposture19 armtosideposture22)
   (move_base_blind trixi1 premanipulationareaeastcounter1 manipulationareaeastcounter1)
   (pick_up_object mug1 rightarm1 manipulationareaeastcounter1 placingareaeastrightcounter1
       ↪ rightgripper1 torsoupposture2 armtosideposture22 armtosideposture23)
   (move_base_blind trixi1 manipulationareaeastcounter1 premanipulationareaeastcounter1)
   (move_arms_to_carryposture leftarm1 rightarm1 armtosideposture armtosideposture armtosideposture33
       ↪ armtosideposture35 armcarryposture43 armcarryposture45)
   (move_torso torso1 torsoupposture torsoupposture2 torsomiddleposture torsomiddleposture4)
   (move_base trixi1 premanipulationareaeastcounter1 premanipulationareasouthtable1)
   (move_torso torso1 torsomiddleposture torsomiddleposture4 torsoupposture torsoupposture6)
   (move_arm_to_side rightarm1 armcarryposture armcarryposture57 armtosideposture61)
   (move_base_blind trixi1 premanipulationareasouthtable1 manipulationareasouthtable1)
   (place_object mug1 rightarm1 manipulationareasouthtable1 placingareawestrighttable1 rightgripper1
       ↪ torsoupposture6 armtosideposture61 armtosideposture63)))
```

Listing 4.5: An experience for 'ServeACoffee' task in Scenario A, in the RACE EBPD.

Chapter 5

Learning Planning Knowledge

The goal of learning is to obtain new domain knowledge that can be used in solving future problems. We propose to use plan-based robot activity experiences, extracted through human-robot interaction (see Chapter 4), for acquiring task knowledge in the form of activity schemata. In this chapter, we present a *conceptualization* methodology for learning activity schemata from experiences. The proposed conceptualization approach is a combination of different techniques including deductive generalization, different forms of abstraction and feature extraction. Conceptualizing of plan-based experiences involves loop detection, scope inference and goal inference. The overall procedure for learning activity schemata from experiences in EBPDs is depicted in Figure 5.1. In Section 5.1, we employ a deductive generalization approach, comparable to Explanation-Based Generalization (EBG), to obtain a new concept from an experience which forms the basis of an activity schema. In Section 5.2, we integrate an abstraction technique to reduce the level of detail in the concept obtained by generalization. In Section 5.3, a feature extraction approach is proposed to extract and integrate useful information into the obtained concept. The features capture essential aspects of an experience and are used during problem solving to improve the efficiency of search. A loop detection approach is presented in Section 5.4 to find the possible loops of actions in an experience and include them in the learned activity schema. The loops are useful for increasing the flexibility and adaptability of the system, widening the class of problems that can be solved with varying sets of objects. In Section 5.5, we present a method, based on canonical abstraction and 3-valued logic, for inferring the scope of applicability of the activity schema, i.e., an abstract structure used to verify weather an activity schema is applicable to solving a given problem. Finally in Section 5.6, a method is proposed for inferring and integrating a goal description into the activity schema.

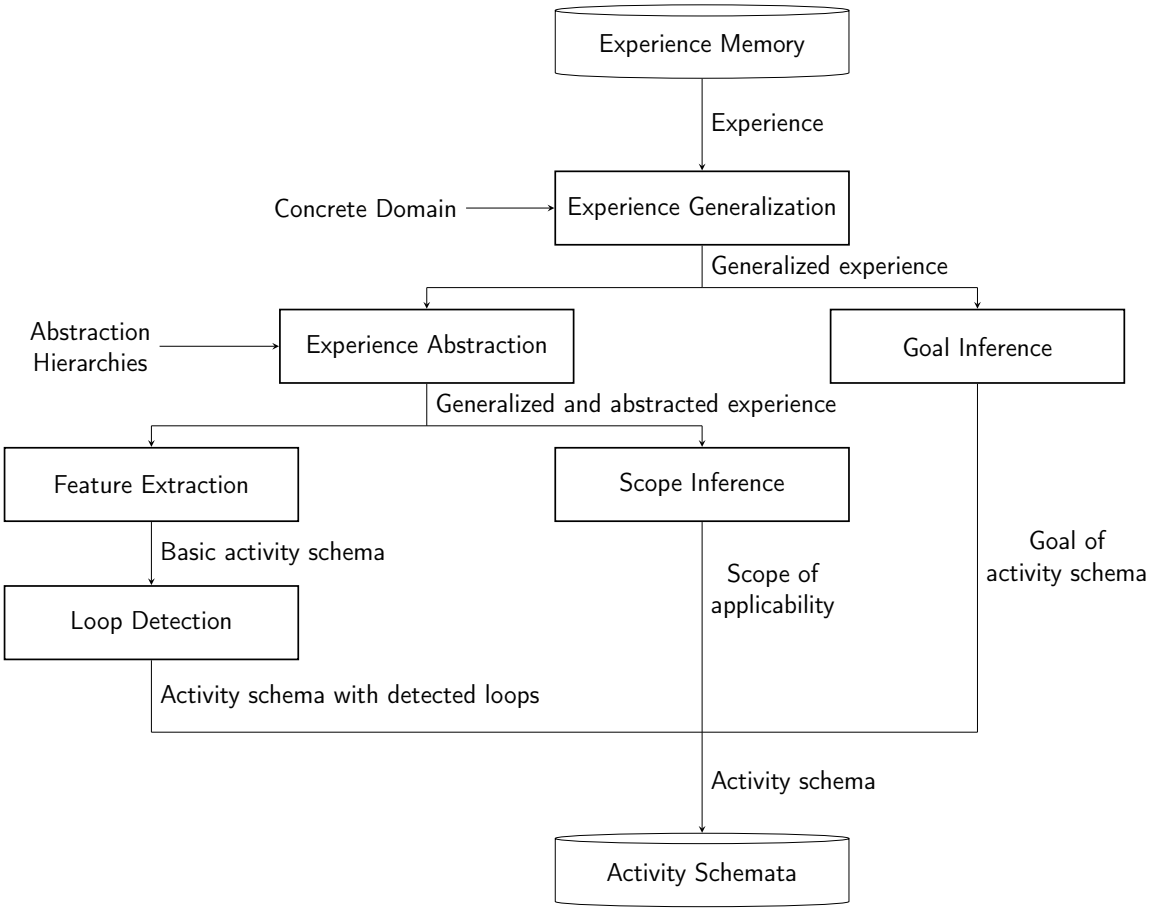


Figure 5.1: Flowchart representing the activity schema learning process.

To illustrate how the conceptualization approach generates an activity schema, we use an experience in a class of task problems, namely `'Stack_N_Blue'`, in the `Stacking-Blocks` domain (see Section 3.1.2). The goal of this class of task problems is to stack a number of blue blocks from a table on a pile. Listing 5.1 shows an experience for the `'Stack_N_Blue'` task containing 5 blue blocks. The key-properties describe the initial, final and static world information of the experience. The plan solution to this problem contains 20 primitive actions.

5.1 Experience Generalization

The first stage, applied to an experience, in order to extract its basic principles, is a deductive generalization method comparable to Explanation-Based Generalization (EBG) (Mitchell et al., 1986; DeJong and Mooney, 1986) (see Section 2.3.4). The basic idea of EBG is to simulate the human learning capability. That is, from a

```
(:experience Stack_N_Blue
 :parameters     (table1 pile1)
 :key-properties ((during(pile pile1))
                  (during(table table1))
                  (during(location location1))
                  (during(hoist hoist1))
                  (during(attached pile1 location1))
                  (during(attached table1 location1))
                  (during(belong hoist1 location1))
                  (during(pallet pallet1))
                  (during(block block1))
                  (during(block block2))
                  (during(block block3))
                  (during(block block4))
                  (during(block block5))
                  (during(blue block1))
                  (during(blue block2))
                  (during(blue block3))
                  (during(blue block4))
                  (during(blue block5))
                  (init(top pallet1 pile1))
                  (init(ontable block1 table1))
                  (init(ontable block2 table1))
                  (init(ontable block3 table1))
                  (init(ontable block4 table1))
                  (init(ontable block5 table1))
                  (init(at hoist1 table1))
                  (init(empty hoist1))
                  (end(on block1 pallet1))
                  (end(on block2 block1))
                  (end(on block3 block2))
                  (end(on block4 block3))
                  (end(on block5 block4))
                  (end(top block5 pile1))
                  (end(at hoist1 pile1))
                  (end(empty hoist1)))
 :plan           ((pickup hoist1 block1 table1 location1)
                  (move hoist1 table1 pile1 location1)
                  (stack hoist1 block1 pallet1 pile1 location1)
                  (move hoist1 pile1 table1 location1)
                  (pickup hoist1 block2 table1 location1)
                  (move hoist1 table1 pile1 location1)
                  (stack hoist1 block2 block1 pile1 location1)
                  (move hoist1 pile1 table1 location1)
                  (pickup hoist1 block3 table1 location1)
                  (move hoist1 table1 pile1 location1)
                  (stack hoist1 block3 block2 pile1 location1)
                  (move hoist1 pile1 table1 location1)
                  (pickup hoist1 block4 table1 location1)
                  (move hoist1 table1 pile1 location1)
                  (stack hoist1 block4 block3 pile1 location1)
                  (move hoist1 pile1 table1 location1)
                  (pickup hoist1 block5 table1 location1)
                  (move hoist1 table1 pile1 location1)
                  (stack hoist1 block5 block4 pile1 location1)
                  (move hoist1 pile1 table1 location1)))
```

Listing 5.1: An experience for the 'Stack_N_Blue' task in the Stacking-Blocks domain.

single example and background knowledge about a domain, people can learn new concepts.

In our proposed conceptualization approach, the generalization is carried out over the plan of an experience. In the generalization, the constants in the plan are

replaced with variables, hence the plan becomes free from the specific constants and could be used in situations involving arbitrary constants. However, this is not just naively replacing each constant in the plan by a variable. The proposed generalization approach follows from the tradition of PLANEX (Fikes et al., 1972). The specific algorithm used in this work and its respective implementation have been proposed and described in (Seabra Lopes, 1997, 1999, 2007). Given a concrete plan-based robot activity experience, the generalization of the experience is carried out over the plan of the experience and achieved by the following steps:

- *Variabilization*: build a variabilized version of the plan, i.e., every occurrence of a constant in the plan is replaced by a new variable.
- *Explanation*: use the primitive operators in the planning domain to explain the experienced plan, i.e., generate a sequence of partial descriptions of states in the experience that support the proof of the preconditions of the applied actions.
- *Generalization*: compute the weakest conditions on the variabilized plan such that the explanation structure, obtained for the concrete plan, still holds for the variabilized plan, i.e., unify the variables representing the same entities in the explanation of the concrete plan. Note that the generalization is carried out in parallel with the explanation of the concrete plan.
- *Criticism*: criticize over-generalization, i.e., establish additional bindings on the variables.

The used generalization method consistently variablizes all constants appearing in the actions of the plan in an experience, and when it reaches the last action in the plan, propagates the variables for constants in the whole experience, i.e., the constants in the key-properties of the experience are also replaced with the variables obtained by the generalization of the plan. Listing 5.2 shows the obtained generalized experience for the input `'Stack_N_Blue'` experience in Listing 5.1. The variables (starting with a question mark ?) are obtained by the generalization of the plan and substituted for the constants in the whole experience. The generated generalized experience forms the basis of a new activity schema.

Deductive generalization methods are useful for improving problem solving efficiency. However, they do not warrant the problem solving efficiency because the learned knowledge has a "hidden" cost of matching the conditions to check their applicability, i.e., the *utility problem* (Minton, 1990). To overcome with this

```
(:experience Stack_N_Blue
 :parameters     (?table ?pile)
 :key-properties ((during(pile ?pile))
                  (during(table ?table))
                  (during(location ?location))
                  (during(hoist ?hoist))
                  (during(attached ?pile ?location))
                  (during(attached ?table ?location))
                  (during(belong ?hoist ?location))
                  (during(pallet ?pallet))
                  (during(block ?block1))
                  (during(block ?block2))
                  (during(block ?block3))
                  (during(block ?block4))
                  (during(block ?block5))
                  (during(blue ?block1))
                  (during(blue ?block2))
                  (during(blue ?block3))
                  (during(blue ?block4))
                  (during(blue ?block5))
                  (init(top ?pallet ?pile))
                  (init(ontable ?block1 ?table))
                  (init(ontable ?block2 ?table))
                  (init(ontable ?block3 ?table))
                  (init(ontable ?block4 ?table))
                  (init(ontable ?block5 ?table))
                  (init(at ?hoist ?table))
                  (init(empty ?hoist))
                  (end(on ?block1 ?pallet))
                  (end(on ?block2 ?block1))
                  (end(on ?block3 ?block2))
                  (end(on ?block4 ?block3))
                  (end(on ?block5 ?block4))
                  (end(top ?block5 ?pile))
                  (end(at ?hoist ?pile))
                  (end(empty ?hoist)))
 :plan           ((pickup ?hoist ?block1 ?table ?location)
                  (move ?hoist ?table ?pile ?location)
                  (stack ?hoist ?block1 ?pallet ?pile ?location)
                  (move ?hoist ?pile ?table ?location)
                  (pickup ?hoist ?block2 ?table ?location)
                  (move ?hoist ?table ?pile ?location)
                  (stack ?hoist ?block2 ?block1 ?pile ?location)
                  (move ?hoist ?pile ?table ?location)
                  (pickup ?hoist ?block3 ?table ?location)
                  (move ?hoist ?table ?pile ?location)
                  (stack ?hoist ?block3 ?block2 ?pile ?location)
                  (move ?hoist ?pile ?table ?location)
                  (pickup ?hoist ?block4 ?table ?location)
                  (move ?hoist ?table ?pile ?location)
                  (stack ?hoist ?block4 ?block3 ?pile ?location)
                  (move ?hoist ?pile ?table ?location)
                  (pickup ?hoist ?block5 ?table ?location)
                  (move ?hoist ?table ?pile ?location)
                  (stack ?hoist ?block5 ?block4 ?pile ?location)
                  (move ?hoist ?pile ?table ?location)))
```

Listing 5.2: The experience of Listing 5.1 after deductive generalization.

limitation, we present methods to extract and integrate useful information into the generalized experience.

5.2 Experience Abstraction

Abstraction is one of the most challenging and also promising approaches to improve complex problem solving and it is inspired by the way humans seem to solve problems (Saitta and Zucker, 2013; Yang, 1997). Abstraction reduces the level of detail in a representation allowing it to be more easily adapted to new situations. An abstract representation allows, during problem solving, to more easily solve the given problems, i.e., with a reduced computational effort. Abstraction also makes the learned concepts broader, more widely applicable. We propose to use an abstraction methodology for transforming the obtained generalized experience into an abstracted generalized experience. The problem of experience abstraction can be described as transforming an experience from the concrete level into an experience in an abstract level. Based on the given concrete and abstract domains $(\mathcal{D}_c, \mathcal{D}_a)$ and predicate and operator abstraction hierarchies (Definitions 3.10 and 3.11) in an EBPD, the abstraction of an experience is achieved by transforming the concrete predicates and operators into abstract predicates and operators, which results in reducing the level of detail in the generalized experience, i.e., results in an abstracted and generalized experience (see Algorithm 2). In this process, while some of the concrete predicates/operators can be skipped, each abstract predicate/operator must result from a particular concrete predicate/operator. The predicate/operator abstraction hierarchies specify which of the concrete predicates/operators are mapped and which are skipped.

Algorithm 2 Experience Abstraction

input:
- $\Delta = (\mathcal{L}, \mathcal{D}_a, \mathcal{D}_c, \mathcal{A}, \mathcal{E}, \mathcal{M})$ ▷ an experience-based planning domain (Def. 3.17)
- $e = (t, K, \pi)$ ▷ a generalized experience (Def. 3.13)

output:
- $e_a = (t, K_a, \pi_a)$ ▷ a generalized and abstracted experience

1 $K_a \leftarrow \emptyset$ ▷ K_a is the set of generalized and abstracted key-properties
2 $\pi_a \leftarrow \emptyset$ ▷ π_a is the sequence of generalized and abstracted operators (abstract plan)
3 **for all** $\tau(p)$ in K **do**
4 **if** $\texttt{parent}(p) \neq \emptyset$ **then**
5 $K_a \leftarrow K_a \cup \{\tau(\texttt{parent}(p))\}$ ▷ get parent of p (Def. 3.10)
6 **for all** o in π **do**
7 **if** $\texttt{parent}(o) \neq \emptyset$ **then**
8 $\pi_a \leftarrow \pi_a \cdot \texttt{parent}(o)$ ▷ get parent of o (Def. 3.11)
9 **return** (t, K_a, π_a)

Listing 5.3 shows the generalized 'Stack_N_Blue' experience after abstraction. Compared to Listing 5.2, some key-properties are excluded from the generalized experience (predicate abstraction) and actions in the plan are replaced with their abstract operator parents (operator abstraction). In this example, the abstraction is based on the predicate and operator abstraction hierarchies presented in Tables 3.3 and 3.4.

```
(:experience Stack_N_Blue
 :parameters     (?table ?pile)
 :key-properties ((during(pile ?pile))
                  (during(table ?table))
                  (during(pallet ?pallet))
                  (during(block ?block1))
                  (during(block ?block2))
                  (during(block ?block3))
                  (during(block ?block4))
                  (during(block ?block5))
                  (during(blue ?block1))
                  (during(blue ?block2))
                  (during(blue ?block3))
                  (during(blue ?block4))
                  (during(blue ?block5))
                  (init(top ?pallet ?pile))
                  (init(ontable ?block1 ?table))
                  (init(ontable ?block2 ?table))
                  (init(ontable ?block3 ?table))
                  (init(ontable ?block4 ?table))
                  (init(ontable ?block5 ?table))
                  (end(on ?block1 ?pallet))
                  (end(on ?block2 ?block1))
                  (end(on ?block3 ?block2))
                  (end(on ?block4 ?block3))
                  (end(on ?block5 ?block4))
                  (end(top ?block5 ?pile))))
 :plan           ((pick ?block1 ?table)
                  (stack ?block1 ?pallet ?pile)
                  (pick ?block2 ?table)
                  (stack ?block2 ?block1 ?pile)
                  (pick ?block3 ?table)
                  (stack ?block3 ?block2 ?pile)
                  (pick ?block4 ?table)
                  (stack ?block4 ?block3 ?pile)
                  (pick ?block5 ?table)
                  (stack ?block5 ?block4 ?pile)))
```

Listing 5.3: The experience of Listing 5.1 after generalization and abstraction.

5.3 Feature Extraction

The performance of concept learning algorithms may degrade when supplied with data attributes that are not directly and independently relevant to the learned concept (John et al., 1994; Markovitch and Rosenstein, 2002). The discovery of meaningful features can contribute to the creation of a more concise and accurate

learned concept. The theory of feature generation for concept learning in problem solving systems has been traditionally studied to measure the degree of achievement of important goals and subgoals (Fawcett and Utgoff, 1991, 1992). A feature can be represented as a conjunction or disjunction of first-order terms. These terms are called the conditions of the feature. A feature is evaluated with respect to a problem solving state by determining whether the conditions are satisfiable in the state.

Whilst abstraction reduces the level of detail in an experience, extracting some other features would help to capture the essence of the experience. The idea for the features is extended from the previous work (Seabra Lopes, 1999, 2007) where a set of hand coded domain-specific features were used for describing the learned concepts. In the current work, features (see Definition 3.14) are key-properties of objects involved in an experience that are automatically extracted and integrated into an activity schema. For example in Listing 5.3, the key-property `(init(ontable ?block1 ?table))`, on line 17, is a feature that connects the `?block1`, an argument of the abstract operator `pick`, on line 28, to the `?table`, an argument of the task `'Stack_N_Blue'`, on line 2.

The objective of feature extraction is twofold: first, to improve the performance of problem solving by guiding the planner toward a goal state and reducing the probability of backtracking, that is, during problem solving objects that satisfy the features are preferable to instantiate actions; and, second, when there are different alternatives to achieve a goal, some alternatives may be preferable based on different factors, such as social norms, physical constraints, etc., e.g., although a guest in a café could be served on the left side, a feature can capture the social norm that the guest should be served on the right (see an example in Listing 3.4, on page 53, where the feature `((during(hasplacingarearight ?sawt ?pawrt)) (during(at ?guest ?sawt)))` of the abstract operator `(!place ?robot ?mug ?mast ?pawrt)` captures the principle that the guest should be served on the right side by social convention).

We develop a feature extraction procedure (see Algorithm 3) in which, for each abstract operator in the abstract plan of a generalized and abstracted experience, all possible relations between the arguments of the abstract operator and the task arguments, according to the Definition 3.14, are extracted and associated to the abstract operator. That is, for each abstract operator o, we extract a set of key-properties that contain only the arguments of o (0-step features); or at least one

Algorithm 3 Feature Extraction

input:
– $e = (t, K, \pi)$ ▷ a generalized and abstracted experience (Def. 3.13)
output:
– Ω ▷ an enriched abstract plan (Def. 3.15)

```
1  Ω ← ∅
2  T ← args(t)
3  for all o in π do
4      A ← args(o)
5      F ← ∅                                              ▷ set of features
6      for all τ1(p) in K do
7          U ← args(p)
8          if (U ⊆ A) ∨ (U ∩ A ≠ ∅ ∧ U ∩ T ≠ ∅) then       ▷ 0-step and 1-step features
9              F ← F ∪ {τ1(p)}
10         else
11             for all τ2(q) in K, τ2(q) ≠ τ1(p) do
12                 V ← args(q)
13                 if (U ∩ A ≠ ∅) ∧ (V ∩ T ≠ ∅) ∧ (U ∩ V ≠ ∅) then   ▷ 2-step features
14                     F ← F ∪ {(τ1(p), τ2(q))}
15     Ω ← Ω · (o, F)                                     ▷ associate o with F
16 return Ω
```

argument of o and one argument of the task (1-step features); or pairs of key-properties that mutually link the arguments of o with the arguments of the task (2-step features). The basic activity schema learned for the ‘Stack_N_Blue’ experience after generalization, abstraction and feature extraction is shown in Listing 5.4. The abstract-plan includes a sequences of enriched abstract operators, i.e., abstract operators associated with features (see Definition 3.15).

5.4 Loop Detection

Solutions to a class of problems with varying number of objects may differ in the repetition of some actions or sequences of actions. In a generalized and abstracted experience there might be some abstract operators with the same features that are repeated for different variables. Therefore detecting and representing possible loops of enriched abstract operators in an activity schema would help to improve the compactness and to increase the applicability of the final concept. A *loop* is a sequence of enriched abstract operators that is repeated for several sets of objects in an experience. Two or more contiguous subsequences of enriched abstract

```
1  (:activity-schema Stack_N_Blue
2   :parameters (?table ?pile)
3   :abstract-plan
4      (((!pick ?block1 ?table)
5           ((init(ontable ?block1 ?table))(during(block ?block1))
6            (during(blue ?block1))(during(table ?table))))
7       ((!stack ?block1 ?pallet ?pile)
8           ((init(ontable ?block1 ?table))(init(top ?pallet ?pile))
9            (during(block ?block1))(during(blue ?block1))
10           (during(pallet ?pallet))(during(pile ?pile))))
11      ((!pick ?block2 ?table)
12          ((init(ontable ?block2 ?table))(during(block ?block2))
13           (during(blue ?block2))(during(table ?table))))
14      ((!stack ?block2 ?block1 ?pile)
15          ((init(ontable ?block1 ?table))(init(ontable ?block2 ?table))
16           (during(block ?block1))(during(block ?block2))(during(blue ?block1))
17           (during(blue ?block2))(during(pile ?pile))))
18      ((!pick ?block3 ?table)
19          ((init(ontable ?block3 ?table))(during(block ?block3))
20           (during(blue ?block3))(during(table ?table))))
21      ((!stack ?block3 ?block2 ?pile)
22          ((init(ontable ?block2 ?table))(init(ontable ?block3 ?table))
23           (during(block ?block2))(during(block ?block3))(during(blue ?block2))
24           (during(blue ?block3))(during(pile ?pile))))
25      ((!pick ?block4 ?table)
26          ((init(ontable ?block4 ?table))(during(block ?block4))
27           (during(blue ?block4))(during(table ?table))))
28      ((!stack ?block4 ?block3 ?pile)
29          ((init(ontable ?block3 ?table))(init(ontable ?block4 ?table))
30           (during(block ?block3))(during(block ?block4))(during(blue ?block3))
31           (during(blue ?block4))(during(pile ?pile))))
32      ((!pick ?block5 ?table)
33          ((end(top ?block5 ?pile))(init(ontable ?block5 ?table))
34           (during(block ?block5))(during(blue ?block5))
35           (during(table ?table))))
36      ((!stack ?block5 ?block4 ?pile)
37          ((end(top ?block5 ?pile))(init(ontable ?block4 ?table))
38           (init(ontable ?block5 ?table))(during(block ?block4))
39           (during(block ?block5))(during(pile ?pile))(during(blue ?block4))
40           (during(blue ?block5))))))
```

Listing 5.4: The activity schema learned thus far for the 'Stack_N_Blue' task after generalization, abstraction and feature extraction. Each abstract operator is associated with a set of features. See the previous steps of the process in Listings 5.1, 5.2 and 5.3.

operators belong to a loop if in each subsequence the following properties hold: (*i*) the names and the order of the abstract operators are the same; (*ii*) the sets of features describing the abstract operators are the same; and (*iii*) the variables appearing in the corresponding abstract operators and in their respective features play the same role, i.e., they appear at the same positions. For example in Listing 5.4, subsequences of abstract operators `((!pick ...) ...) ((!stack ...) ...)`, on lines (11, 14), (18, 21) and (25, 28), with the same features, and the corresponding variables playing the same roles are repeated for the variables `?block2`, `?block3` and `?block4`, hence belonging to a loop.

We propose a loop detection approach reminiscent of the standard algorithms for computing the Suffix Array (SA) of a string and the Longest Common Prefix

(LCP) of two strings. Manber and Myers (1993) introduced the LCP array, for the first time, alongside the suffix array, in order to improve the running time of the string search algorithm in the suffix array. The main intuition to employ SA and LCP is the efficiency of those algorithms for finding a pattern P of length m in a string S of length n in $O(m + \log n)$ complexity. The problem is also known as finding the longest repeated substring (Hirschberg, 1977). We first provide the standard definitions for SA, LCP and LCP array as follows:

Definition 5.1 (**Suffix Array**). A *Suffix Array* (SA) of a string is an array of integers providing the starting positions of all suffixes of the string, sorted in lexicographical order. ■

An LCP array is an array of integers storing the lengths of the longest common prefixes between all pairs of consecutive suffixes in a suffix array:

Definition 5.2 (**Longest Common Prefix**). Let A and B be two strings, and $A[i:j]$ and $B[i:j]$ denote the substrings of A and B ranging from i to $j-1$ respectively. The *Longest Common Prefix* (LCP) of A and B, denoted by $\texttt{lcp}(A, B)$, is the length, l, of the longest substring, S, such that $A[0:l] = B[0:l] = S$. ■

Definition 5.3 (**Longest Common Prefix Array**). Let S be a string and SA the suffix array of S. The *Longest Common Prefix* array (LCP array) of S is an array of integers of size $n = \texttt{len}(S)$ such that $LCP[0]$ is undefined and $LCP[i] = \texttt{lcp}(S[SA[i-1] : n], S[SA[i] : n])$, for $1 \leqslant i < n$. ■

The LCP array cannot be used to detect potential loops in a string, since it also selects overlapping repeated substrings. For example, in the string `'ababa'`, the longest repeated substring is `'aba'`, but selecting that one causes overlapping. To avoid overlapping, we restrict the size of the longest common prefix of two strings to the difference in length between the two strings. For example, the non-overlapping longest common prefix of two strings `'ababa'` and `'aba'` is `'ab'` of the length $\texttt{abs}(\texttt{len}(\texttt{'ababa'}) - \texttt{len}(\texttt{'aba'})) = 2$. We modify the definition of LCP to build a Non-overlapping Longest Common Prefix (NLCP) array from a string, which gives a list of potential patterns in the string:

Definition 5.4 (**Non-overlapping Longest Common Prefix**). Let A and B be two strings. The *Non-overlapping Longest Common Prefix* (NLCP) of A and B, denoted by $\texttt{nlcp}(A, B)$, is the length, l, of the longest substring, S, such that $l \leqslant \texttt{abs}(\texttt{len}(A) - \texttt{len}(B))$ and $A[0:l] = B[0:l] = S$. ■

Table 5.1: The enriched abstract plan in Listing 5.4 is represented as the string 'abacacacdf'.

enriched abstract operator	symbol
((!pick ?block1 ?table) ...)	a
((!stack ?block1 ?pallet ?pile) ...)	b
((!pick ?block2 ?table) ...)	a
((!stack ?block2 ?block1 ?pile) ...)	c
((!pick ?block3 ?table) ...)	a
((!stack ?block3 ?block2 ?pile) ...)	c
((!pick ?block4 ?table) ...)	a
((!stack ?block4 ?block3 ?pile) ...)	c
((!pick ?block5 ?table) ...)	d
((!stack ?block5 ?block4 ?pile) ...)	f

Definition 5.5 (**Non-overlapping Longest Common Prefix Array**). Let S be a string and SA the suffix array of S. The *Non-overlapping Longest Common Prefix* array (NLCP array) is an array of integers of size $n = \texttt{len}(S)$ such that NLCP[0] is undefined and $\text{NLCP}[i] = \texttt{nlcp}(S[SA[i-1] : n], S[SA[i] : n])$, for $1 \leqslant i < n$. ■

In Algorithm 4, based on the above definitions, given an abstract plan Ω (of a generalized and abstracted experience) we first build a suffix array, SA, for Ω and then build an NLCP array (lines 1–4). The function `nlcp` (line 13) computes the NLCP of two given suffixes according to the Definition 5.4.

Note that in the implementation level, the abstract operators with the same features are consistently represented by single characters. For example, in Listing 5.4 the sequence of enriched abstract operators with their corresponding features (abstract-plan) is represented as the string 'abacacacdf' according to the Table 5.1. The computed SA, LCP and NLCP arrays for the string 'abacacacdf' is shown in Table 5.2.

The NLCP array does not warrant that the obtained non-overlapping longest common prefixes are consecutive in Ω. Hence we define the Contiguous Non-overlapping Longest Common Prefix (CNLCP) array which is obtained form the NLCP array.

Definition 5.6 (**Contiguous Non-overlapping Longest Common Prefix Array**). A *Contiguous Non-overlapping Longest Common Prefix* array (CNLCP array) is an array of structures, constructed from the SA and NLCP arrays of a string, such that each CNLCP[i], for $i \geqslant 0$, contains a substring, representing a pattern that consecutively occurs in the string, and a list of starting positions of the pattern in the string.

Algorithm 4 Contiguous Non-overlapping Longest Common Prefixes (CNLCP)

input:
– Ω ▷ an enriched abstract plan (represented as a string)
output:
– CNLCP ▷ a contiguous non-overlapping longest common prefixes array (Def. 5.6)

```
1  SA ← sorted(range(len(Ω)),key = lambda i : Ω[i :])
                          ▷ build a suffix array from Ω (coding in python) (Def. 5.1)
2  NLCP[0] ← 0
3  for i in range(len(SA) − 1) do
                          ▷ build a non-overlapping longest common prefixes array, NLCP (Def. 5.4)
4      NLCP[i + 1] ← nlcp(Ω[SA[i] :], Ω[SA[i + 1] :])          ▷ see line 13
5  CNLCP ← ∅                                                   ▷ an empty dictionary
6  for maxlen in sorted(NLCP) do                               ▷ build a CNLCP array (Def. 5.6)
7      for i in range(len(Ω)) do
8          if NLCP[i] == maxlen then
9              if abs(SA[i]−SA[i − 1]) == maxlen then
                                  ▷ keep only consecutive occurrences in NLCP
10                 k ← Ω[SA[i] :SA[i] + maxlen]
11                 CNLCP[k] ← CNLCP[k] ∪ {SA[i − 1]}            ▷ starting position of pattern k
12 return CNLCP

13 function nlcp(suf1, suf2)   ▷ find non-overlapping longest common prefix between two given suffixes
14     maxlen ← min(abs(len(suf1) − len(suf2)), len(suf1), len(suf2))
15     for i in range(maxlen) do
16         if suf1[i] ≠ suf2[i] then
17             return len(suf1[0 : i])
18     return len(suf1[0 : maxlen])
```

A non-overlapping longest common prefix between NLCP[i] and NLCP[i − 1] is consecutive if NLCP[i] = abs(SA[i]−SA[i − 1]) for $1 \leqslant i < n$. ■

In Algorithm 4, the main function constructs the CNLCP array (lines 5–11). Line 9 reflects the proposed condition in the Definition 5.6 to find the contiguous non-overlapping longest common prefixes in the NLCP array. When a contiguous CNLCP is detected, its starting position is added to the CNLCP array (line 11). Table 5.3 shows the computed CNLCP array for the string `'abacacacdf'`. CNLCP gives the pattern (the iteration of a loop) `'ac'` happening at the positions (2, 4, 6) respectively. The Kleene closure of the input string is `'ab(ac)*df'`.

When the CNLCP array is constructed for the abstract plan of a generalized and abstracted experience, we start by selecting an iteration (i.e., a substring) with the

Table 5.2: The computed SA, LCP and NLCP arrays for the string 'abacacacdf'.

i	suffix	SA[i]	LCP[i]	NLCP[i]*
0	abacacacdf	0	0	0
1	acacacdf	2	1	1
2	acacdf	4	4	2
3	acdf	6	2	2
4	bacacacdf	1	0	0
5	cacacdf	3	0	0
6	cacdf	5	3	2
7	cdf	7	1	1
8	df	8	0	0
9	f	9	0	0

* Each number in i^{th} row specifies the length of the non-overlapping longest common prefix between two suffixes in rows i and $(i-1)$ for $i \geqslant 1$. For example, in rows 1 and 2, 'ac' is the non-overlapping longest common prefix between two consecutive suffixes 'acacacdf' and 'acacdf'.

Table 5.3: The computed CNLCP array for the same string 'abacacacdf'.

k*	CNLCP[k]
ac	2, 4, 6

* Each suffix in the CNLCP array is a pattern (an iteration of a loop) with its starting positions in a given string.

largest length in the CNLCP array and construct a loop by merging all iterations of the loop. More specifically, the loop iterations are merged, an intersection of their corresponding features is computed, and a new variable represents the different variables playing the same role in the corresponding abstract operators and in their corresponding features in each subsequence. We continue this process for all other iterations in the CNLCP array until no more loops are formed. For example in Listing 5.4, three subsequences of abstract operators `((!pick ...)` `((!stack ...)` (lines 11–28) with the variables `?block2`, `?block3` and `?block4` are merged into a loop and the variable `?block2` represents for all objects with the same features in a given task problem. Note that, in the current loop detecting approach, we do not address the possibility of nested loops.

For the sake of simplicity, some complex cases are not covered in the Algorithm 4. For example, the CNLCP array may also provide overlapping patterns, e.g., in the string 'banana', CNLCP gives the two potential overlapping patterns 'an' and 'na' as iterations of loops; or may provide the same patterns of the same loops, e.g., in the string 'ababxabab', CNLCP gives the pattern 'ab' which occurs at two distinct loops. Implementing a strategy to analyze the CNLCP array for

```
(:activity-schema Stack_N_Blue
 :parameters (?table ?pile)
 :abstract-plan
   (((!pick ?block1 ?table)
       ((init(ontable ?block1 ?table))(during(block ?block1))
        (during(blue ?block1))(during(table ?table))))
    ((!stack ?block1 ?pallet ?pile)
       ((init(ontable ?block1 ?table))(init(top ?pallet ?pile))
        (during(block ?block1))(during(blue ?block1))
        (during(pallet ?pallet))(during(pile ?pile))))
    (loop
       ((!pick ?block2 ?table)
           ((init(ontable ?block2 ?table))(during(block ?block2))
            (during(blue ?block2))(during(table ?table))))
       ((!stack ?block2 ?block1 ?pile)
           ((init(ontable ?block1 ?table))(init(ontable ?block2 ?table))
            (during(block ?block1))(during(block ?block2))
            (during(blue ?block1))(during(blue ?block2))(during(pile ?pile))))
    )
    ((!pick ?block5 ?table)
       ((end(top ?block5 ?pile))(init(ontable ?block5 ?table))
        (during(block ?block5))(during(blue ?block5))(during(table ?table))))
    ((!stack ?block5 ?block4 ?pile)
       ((end(top ?block5 ?pile))(init(ontable ?block4 ?table))
        (init(ontable ?block5 ?table))(during(block ?block4))
        (during(block ?block5))(during(pile ?pile))(during(blue ?block4))
        (during(blue ?block5))))))
```

Listing 5.5: The activity schema learned thus far for the 'Stack_N_Blue' experience now with a loop of actions (see Listings 5.1 and 5.4).

building loops in more complex cases is not addressed here.

To represent the obtained loops of actions in an activity schema, we refine the notion of the enriched abstract plan (Definition 3.15) to include loops as follows:

Definition 5.7 (**Enriched Abstract Plan with Loops**)**.** An *enriched abstract plan with loops*, denoted by Ω, is a sequence of elements each of which is either an enriched abstract operator or a loop. A *loop* is represented as an enriched abstract plan without loops (Definition 3.15), which is repeated continually while there is an object in a planning state that satisfy the features of the abstract operators in the loop. ■

The activity schema built thus far for the 'Stack_N_Blue' experience, now with a loop of actions, is shown in Listing 5.5. For a given task planning problem in later problem solving situation, the loop is expanded for all objects that verify the features of the enriched abstract operators inside the loop.

5.5 Scope Inference

After detecting possible loops in an experience, we propose to infer the *scope of applicability* of the learned activity schema. The scope is a logical structure that

allows for testing the applicability of the activity schema to solve a given problem. The developed approach relies on Canonical Abstraction (Sagiv et al., 2002), a procedure for creating finite representations of (possibly infinite) sets of logical structures. Canonical abstraction is based on Kleene's 3-valued logic (Kleene, 1952), which extends Boolean (2-valued) logic by introducing an indefinite value $1/2$, to denote either 0 or 1. We infer the scope of an activity schema from the key-properties of a generalized and abstracted experience in the form of a 3-valued logical structure. Since the key-properties of an experience use, not only the `init` temporal symbol, but also the `during` and `end` temporal symbols, the inferred scope is more than a simple precondition.

The 3-valued logical structure can be used as an abstraction of a larger 2-valued logical structure. Standard definitions of 2-valued and 3-valued logical structures are adapted to accommodate the temporal symbols used in key-properties of experiences. We first represent the key-properties of a generalized and abstracted experience using a 2-valued logical structure:

Definition 5.8 (2-Valued Logical Structure with Temporal Symbols). A *2-valued logical structure*, also called a *concrete structure*, over a set of temporal symbols T and a set of predicate symbols P is a pair,

$$C = (U, \iota),$$

where U is a set of individuals called the universe of C and ι is an interpretation for T and P over U. For every temporal symbol $\tau \in T$ and predicate symbol $p^{(k)} \in P$ of arity k, the interpretation is a function $\iota(\tau, p) : U^k \to \{0, 1\}$. ■

We convert a set of key-properties K to a 2-valued logical structure, denoted by $\texttt{Struct}(K) = (U, \iota)$, as follows:

$$\begin{aligned}
P &= \{p \mid \tau(p(t_1, \ldots, t_k)) \in K\} \ , \\
T &= \{\tau \mid \tau(p(t_1, \ldots, t_k)) \in K\} \ , \\
U &= \bigcup_{\tau(p(t_1,\ldots,t_k)) \in K} \{t_1, \ldots, t_k\} \ , \\
\iota &= \lambda \tau \in T, p^{(k)} \in P \,.\, \lambda(t_1, \ldots, t_k) \in U^k. \begin{cases} 1, & \text{if } \tau(p(t_1, \ldots, t_k)) \in K \\ 0, & \text{otherwise.} \end{cases}
\end{aligned}$$

That is, the universe of $\texttt{Struct}(K)$ consists of the objects appearing in the arguments of the key-properties in K. The interpretation of a temporal symbol $\tau \in T$, where $T = \{\texttt{during}, \texttt{init}, \texttt{end}, \texttt{none}\}$[1], and predicate symbol $p^{(k)} \in P$ of arity k, for

[1] Note that the types of objects, in typed planning domains descriptions, are also used as unary predicates in constructing a 2-valued logical structure. Hence, the temporal symbol `none` is used for these unary predicates.

a tuple of objects $(t_1,\ldots,t_k) \in U$ is 1, i.e., $\iota(\tau,p)(t_1,\ldots,t_k) = 1$, if a corresponding key-property $\tau(p(t_1,\ldots,t_k))$ appears in K.

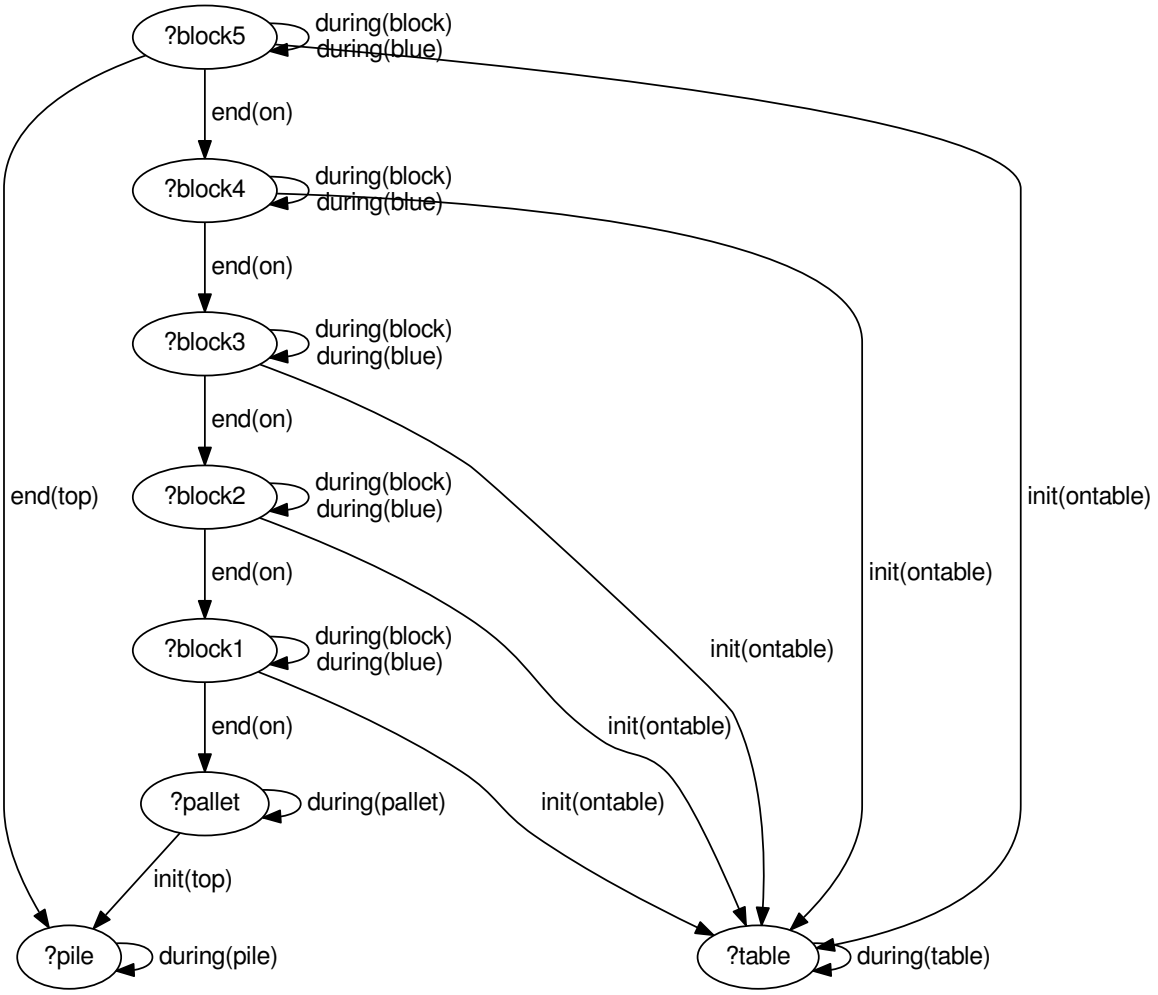
2-valued logical structures can be represented graphically as directed graphs. The individuals of the universe are drawn as nodes and true key-properties are drawn as directed solid edges.

Example 5.1. Figure 5.2(a) shows a 2-valued logical (concrete) structure C representing the generalized and abstracted 'Stack_N_Blue' experience in Listing 5.3. In this example, the universe and the truth-values (interpretations) of the key-properties over the universe of C are as follows.

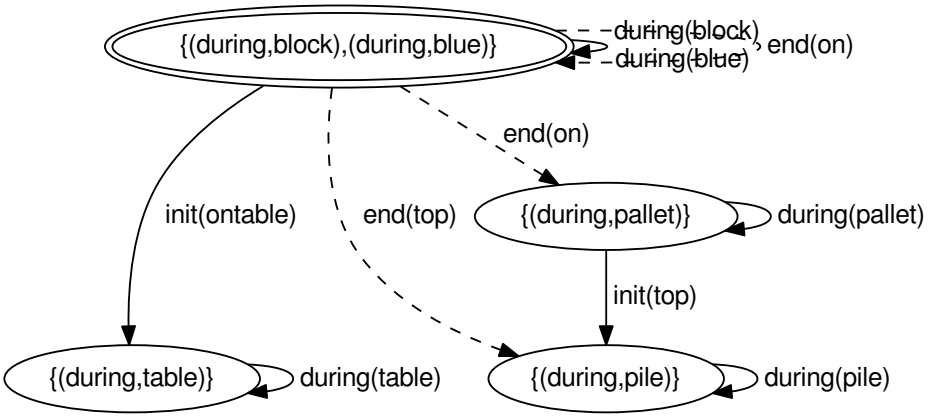
```
P  = {pile,table,pallet,block,blue,top,ontable,on,top} ,
T  = {during,init,end},
U  = {?pile,?table,?pallet,?block1,?block2,?block3,?block4,?block5} ,
ι  = {(during(pile ?pile)),
      (during(table ?table)),
      (during(pallet ?pallet)),
      (during(block ?block1)),
      (during(block ?block2)),
      (during(block ?block3)),
      (during(block ?block4)),
      (during(block ?block5)),
      (during(blue ?block1)),
      (during(blue ?block2)),
      (during(blue ?block3)),
      (during(blue ?block4)),
      (during(blue ?block5)),
      (init(top ?pallet ?pile)),
      (init(ontable ?block1 ?table)),
      (init(ontable ?block2 ?table)),
      (init(ontable ?block3 ?table)),
      (init(ontable ?block4 ?table)),
      (init(ontable ?block5 ?table)),
      (end(on ?block1 ?pallet)),
      (end(on ?block2 ?block1)),
      (end(on ?block3 ?block2)),
      (end(on ?block4 ?block3)),
      (end(on ?block5 ?block4)),
      (end(top ?block5 ?pile))} .
```

□

The scope of applicability of an activity schema will be represented by a 3-valued logical structure:



(a) A concrete structure C.



(b) An abstract structure $\mathfrak{S} = \beta(C)$.

Figure 5.2: Canonical abstraction of key-properties in the (generalized and abstracted) 'Stack_N_Blue' experience (in Listing 5.3). (a) Graphical representation of the 2-valued structure generated from the key-properties. (b) Graphical representation of the 3-valued structure obtained through canonical abstraction.

Definition 5.9 (**3-Valued Logical Structure with Temporal Symbols**). A 3-*valued logical structure*, also called an *abstract structure*, over a set of temporal symbols T and a set of predicate symbols P is a pair,

$$\mathfrak{S} = (\mathsf{U}, \iota),$$

where U is a set of individuals called the universe of $\mathfrak{S}$ and ι is an interpretation for T and P over U. For every temporal symbol $\tau \in \mathsf{T}$ and predicate symbol $p^{(k)} \in \mathsf{P}$ of arity k, the interpretation is a function $\iota(\tau, p) : \mathsf{U}^k \to \{0, 1, 1/2\}$, where $1/2$ means indefinite. ■

The transformation form a 2-valued logical structure into a 3-valued logical structure is based on canonical names and the join operation (Lev-Ami and Sagiv, 2000) as follows:

Definition 5.10 (**Canonical Name**). Let (U, ι) be a (2-valued/3-valued) logical structure over a set of temporal symbols T and a set of predicate symbols P. The *canonical name* of an object $u \in \mathsf{U}$, also called an *abstraction predicate*, denoted by $\mathsf{canon}(u)$, is a set of pairs of temporal symbols and unary predicate symbols that hold for u in the structure:

$$\mathsf{canon}(u) = \{(\tau, p) \mid \tau \in \mathsf{T}, p \in \mathsf{P}, \iota(\tau, p)(u) = 1\} \ .$$

■

Example 5.2. The canonical names of the objects in U in Example 5.1 are the following:

$$\begin{aligned}
\mathsf{canon}(\texttt{?table}) &= \{(\texttt{during},\texttt{table})\} \\
\mathsf{canon}(\texttt{?pile}) &= \{(\texttt{during},\texttt{pile})\} \\
\mathsf{canon}(\texttt{?pallet}) &= \{(\texttt{during},\texttt{pallet})\} \\
\mathsf{canon}(\texttt{?block1}) &= \{(\texttt{during},\texttt{block}),(\texttt{during},\texttt{blue})\} \\
\mathsf{canon}(\texttt{?block2}) &= \{(\texttt{during},\texttt{block}),(\texttt{during},\texttt{blue})\} \\
\mathsf{canon}(\texttt{?block3}) &= \{(\texttt{during},\texttt{block}),(\texttt{during},\texttt{blue})\} \\
\mathsf{canon}(\texttt{?block4}) &= \{(\texttt{during},\texttt{block}),(\texttt{during},\texttt{blue})\} \\
\mathsf{canon}(\texttt{?block5}) &= \{(\texttt{during},\texttt{block}),(\texttt{during},\texttt{blue})\}
\end{aligned}$$

□

Definition 5.11 (**Join Operation**). In Kleene's 3-valued logic, we say the values 0 and 1 are definite values and $1/2$ is an indefinite value. For $l_1, l_2 \in \{0, 1, 1/2\}$, l_1 has more definite information than l_2, denoted by $l_1 \preceq l_2$, if $l_1 = l_2$ or $l_2 = 1/2$ (see also

Table 5.4: The truth table (left) and the graphical representation (right) of the information order ($\preceq$) for two Kleene's 3-valued propositions.

$\preceq$	0	1	1/2
0	1	0	1
1	0	1	1
1/2	0	0	1

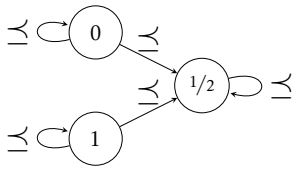


Table 5.5: The truth table of the join operation ($\sqcup$) for two Kleene's 3-valued propositions.

$\sqcup$	0	1	1/2
0	0	1/2	1/2
1	1/2	1	1/2
1/2	1/2	1/2	1/2

Table 5.4). The *join operation* of l_1 and l_2, denoted by $l_1 \sqcup l_2$, is the least-upper-bound operation with respect to the information order, $\preceq$, defined as follows (see also Table 5.5):

$$l_1 \sqcup l_2 = \begin{cases} l_1, & \text{if} \quad l_1 = l_2; \\ 1/2, & \text{otherwise.} \end{cases}$$

■

Based on the above definitions, we can now infer a 3-valued logical structure from a 2-valued logical structure using canonical abstraction as follows:

Definition 5.12 (Canonical Abstraction). Let $C = (U, \iota)$ be a 2-valued logical structure. The *canonical abstraction* of C, denoted by $\beta(C)$, is a 3-valued logical structure $\mathfrak{S} = (U', \iota')$ defined as follows:

$$U' = \{\mathsf{canon}(u) \mid u \in U\} \ ,$$
$$\iota'(\tau, p^{(k)})(t'_1, \ldots, t'_k) = \bigsqcup_{t_1,\ldots,t_k} \{\iota(\tau, p^{(k)})(t_1, \ldots, t_k) \mid \forall i = 1..k.\ t'_i = \mathsf{canon}(t_i)\} \ .$$

$\mathfrak{S}$ may contain *summary objects*. A summary object in U' for a canonical name, c, denoted by $\mathsf{summary}(c)$, represents the set of objects in U that have c as canonical name: [2]

$$\mathsf{summary}(c) = \{u \in U \mid \mathsf{canon}(u) = c\} \ .$$

■

[2] Note that we avoid corresponding a summary object to the objects appearing in the task (parameters) of an experience.

The join operation is used to determine the truth-values (interpretations) of key-properties in a 3-valued logical structure. That is, the interpretation of a key-property in the 3-valued logical structure is 1 (solid arrows in the graphical representation) if the key-property holds for all objects of the same canonical name in the 2-valued logical structure, otherwise the truth-value is $1/2$ if the key-property holds for some objects of the same canonical name (dashed arrows), and 0 if the key-property does not hold for any object (no arrow drawn). [3]

Following the same convention used for 2-valued logical structures, 3-valued logical structures are also drawn as directed graphs. In addition, summary objects are drawn as double circles and key-properties with indefinite ($1/2$) values are drawn as directed dashed edges.

Example 5.3. Figure 5.2(b) shows the 3-valued structure $\mathfrak{S}$ of the concrete structure C in Figure 5.2(a). The double circles stand for summary objects, e.g., `summary`({`(during,block)`,`(during,blue)`}) is a summary object in $\mathfrak{S}$ corresponding to the objects (`?block1..?block5`) in C with the same canonical name. Solid (dashed) arrows represent truth-values of 1 ($1/2$). Intuitively, because of the summary objects, the abstract structure $\mathfrak{S}$ represents the concrete structure C and all other '`Stack_N_Blue`' problems that have exactly one `table`, one `pile`, one `pallet`, and at least one `blue block`, such that blue blocks are initially on the table and finally in the pile. □

The inferred scope is finally integrated into the learned activity schema. Therefore, we extend the notion of robot activity schema (Definition 3.16) to include the scope of applicability as follows:

Definition 5.13 (**Robot Activity Schema with Scope of Applicability**). A *robot activity schema with scope of applicability* (or, for short, an *activity schema*), $\mathfrak{m}$, is a tuple of unground structures,

$$\mathfrak{m} = (t, \mathfrak{S}, G, \Omega),$$

where t is the target task to be performed, $\mathfrak{S}$ is a 3-valued logical structure with temporal symbols (Definition 5.9) representing the scope of applicability of $\mathfrak{m}$, G

[3] In a planning domain description, the set of unary predicates is used to build the set of abstraction predicates. The function of canonical abstraction suggests that we should have sufficient unary predicates to be able to determine if an abstract structure exists for a concrete structure. In all example domains used in this work, we provided sufficient unary predicates. However, the types of objects (in typed planning domains descriptions) are also assumed as unary predicates when unary predicates are not sufficient.

```
(:activity-schema Stack_N_Blue
 :parameters (?table ?pile)
 :scope
  ((summary ((during,block)(during,blue)))
   (during(table (during,table)))
   (during(pile (during,pile)))
   (during(pallet (during,pallet)))
   (during(block ((during,block)(during,blue))))
   (during(blue ((during,block)(during,blue))))
   (init(top (during,pallet) (during,pile)))
   (init(ontable ((during,block)(during,blue)) (during,table)))
   (maybe(end(on ((during,block)(during,blue)) (during,pallet))))
   (maybe(end(on ((during,block)(during,blue)) ((during,block)(during,blue)))))
   (maybe(end(top ((during,block)(during,blue)) (during,pile)))))
 :abstract-plan
  (((!pick ?block1 ?table)
      ((init(ontable ?block1 ?table))(during(block ?block1))
       (during(blue ?block1))(during(table ?table))))
   ((!stack ?block1 ?pallet ?pile)
      ((init(ontable ?block1 ?table))(init(top ?pallet ?pile))
       (during(block ?block1))(during(blue ?block1))
       (during(pallet ?pallet))(during(pile ?pile))))
   (loop
      ((!pick ?block2 ?table)
          ((init(ontable ?block2 ?table))(during(block ?block2))
           (during(blue ?block2))(during(table ?table))))
      ((!stack ?block2 ?block1 ?pile)
          ((init(ontable ?block1 ?table))(init(ontable ?block2 ?table))
           (during(block ?block1))(during(block ?block2))
           (during(blue ?block1))(during(blue ?block2))(during(pile ?pile)))))
   )
   ((!pick ?block5 ?table)
      ((end(top ?block5 ?pile))(init(ontable ?block5 ?table))
       (during(block ?block5))(during(blue ?block5))(during(table ?table))))
   ((!stack ?block5 ?block4 ?pile)
      ((end(top ?block5 ?pile))(init(ontable ?block4 ?table))
       (init(ontable ?block5 ?table))(during(block ?block4))
       (during(block ?block5))(during(pile ?pile))(during(blue ?block4))
       (during(blue ?block5))))))
```

Listing 5.6: An activity schema learned for the 'Stack_N_Blue' task with loops of actions and its scope of applicability.

is the goal of m, and Ω is an enriched abstract plan with loops (Definition 5.7) to achieve the task t. ■

Listing 5.6 shows the activity schema learned in the Stacking-Blocks domain for the 'Stack_N_Blue' task. The inferred scope of applicability specifies a class of problems that can be solved with this activity schema. Summary objects are represented as (summary ?c) where ?c is a canonical name. They represent arbitrary sets of objects with the same properties in a task planning problem, e.g., (summary(during,block)(during,blue)) specifies an arbitrary set of objects with the unary predicate symbols block and blue in a task planning problem. Key-properties with indefinite values (1/2) appear as (maybe(k)) where k represents a key-property, i.e., k is a key-property that may exist in a 2-valued logical structure represented by the scope of the activity schema. They specify predicates

that may be contained in a task planning problem for some objects of the same properties, e.g., `(maybe(end(on ((during,block)(during,blue)) ((during,block) (during,blue)))))` specifies that the predicate on may not exist for all blue blocks (at the final state). The rest are key-properties specifying predicates contained in a task planning problem, e.g., `(init(ontable ((during,block)(during,blue)) (during,table)))` specifies that the predicate ontable exists for all blue blocks (at the initial state).

5.6 Goal Inference

Depending on the problem formulation in an EBPD (see Section 3.4), the description of goal propositions may not be given in a class of task problems. In this context, the conceptualization approach infers a set of generalized (i.e., ungrounded) key-properties from an experience as the goal of an activity schema. The inferred goal of the activity schema is used, during problem solving, to describe the explicit goal of the given task.

Goal inference is performed on a generalized experience. The parameters of a taught task, identified by an instructor, define the main target for inferring goal propositions. In Algorithm 5, we present a breadth first search approach (GoalInference), which takes as input a set of task arguments and a set of key-properties in a generalized experience. For every pair of task arguments, the key-properties are explored to find the shortest connecting path between the arguments. Each path is a set of key-properties that reveals a relation between two task arguments in the experience. Finally, a union of the shortest paths computed for all pairs of arguments is used for describing the inferred goal. Note that the key-properties with temporal symbols `init` and `during` cannot be part of the goal in an experience. During problem solving, they are essentially used to instantiate the goal predicates for a given task problem based on the world information included in the problem description. The inferred goal is finally represented in the learned activity schema.

Example 5.4. We use the (generalized) experience for 'ServeACoffee' task in RACE domain (shown in Listing 4.5 on page 72) to infer a set of goal propositions for the 'ServeACoffee' task. The task parameters are used as cues for inferring goal propositions. All paths between the parameters (`?mug ?counter ?guest ?table`) in the 'ServeACoffee' experience after generalization and abstraction are shown in Figure 5.3. Therefore, the inferred goal includes the following key-properties:

Algorithm 5 Goal Inference

input:
– $e = (t, K, \pi)$ ▷ a generalized experience (Def. 3.13)
output:
– G ▷ a set of goal predicates

```
1  G ← ∅
2  for all distinct pairs (u, v) of args(t) do
3      G ← G ∪ FindPaths(u, v, K)          ▷ find the shortest path from u to v in K
4  return G

5  function FindPaths(u, v, K)   ▷ a breadth first search returning the shortest path from u to v in K
6      U ← {τ(p) ∈ K | u ∈ args(p), |args(p)| == 1}   ▷ set of unary predicates containing u
7      Q ← [(u, U)]                          ▷ initialize a queue with u and U
8      while Q ≠ ∅ do
9          (x, path) ← remove(Q)
10         if x == v then
11             return path                   ▷ v is reached (return the path)
12         for all τ(p) in {K − path} do
13             if x ∈ args(p) then
14                 path ← path ∪ {τ(p)}
15                 for all y ∈ {args(p) − x} do
16                     U ← {τ(p) ∈ K | y ∈ args(p), |args(p)| == 1}
17                     Q ← Q + [(y, path ∪ U)]   ▷ add to Q and continue to reach v
18     return ∅
```

```
G = {(init(on ?mug ?paerc)),
     (during(mug ?mug)),
     (during(guest ?guest)),
     (during(table ?table)),
     (during(counter ?counter)),
     (during(placingarearight ?paerc)),
     (during(hasplacingareaeastright ?counter ?paerc)),
     (during(sittingarea ?sawt)),
     (during(placingarearight ?pawrt)),
     (during(hasplacingarearight ?sawt ?pawrt)),
     (during(hasplacingareawestright ?table ?pawrt)),
     (during(at ?guest ?sawt)),
     (end(on ?mug ?pawrt))} .
```

□

When the goal is instantiated for a task problem, only key-properties with temporal symbol end are considered as the goal of the task problem, e.g., (on ?mug

Figure 5.3: All paths between the task arguments (`?mug ?guest ?table ?counter`) in the (generalized) experience 'ServeACoffee'. The key-properties with temporal symbols `init`, `during` and `end`, involved in the shortest paths (solid edges), are included in the description of the inferred goal.

`?pawrt`) in the above example. This inferred goal was shown as part of the activity schema for the 'ServeACoffee' task in Listing 3.4 on page 53.

The current goal inference approach does not warrant that a set of goal propositions can always be extracted from experiences. The existence of a set of goal propositions depends on the information included in the experience.

5.7 Summary

Machine learning enables intelligent systems to perform tasks and improve their capabilities by learning from examples over the time. Over the years, machine learning techniques have had great success in many applications such as robotics (Chernova and Thomaz, 2014) and planning (Jiménez et al., 2012). In this chapter, we used several learning techniques to improve domain theory by acquiring new knowledge from experiences. We specifically proposed methods, within the formal framework of EBPDs (Chapter 3), for generating activity schemata, i.e., generic solutions to a class of tasks including possible loops of actions, from robot activity experiences. We used an explanation-based generalization approach to form a concept from a single experience. Abstraction was used to reduce computation efforts and achieve more compact and widely applicable activity schemata. Loops are detected and integrated into the learned activity schemata to increase the applicability of the learned concepts by repeating sequences of actions for different sets of objects in the target task problems. The computed scope of applicability for the resulting activity schemata may not be sufficient, but can still provide useful information to decide if a learned activity schema is relevant for a particular problem. Using the proposed techniques, the obtained activity schemata can be applied more generally to solve many problems.

Chapter 6

Planning Using the Learned Knowledge

Problem solving in EBPDs is achieved using a hierarchical problem solver which includes an abstract planner and a concrete planner. The abstract planner derives an abstract solution to a given task problem by following a learned activity schema. Then the concrete planner follows the abstract solution and fills the necessary details to derive a concrete solution. This chapter presents the developed planning system. First, we present a method of transforming a given concrete task problem into an abstract task problem (Section 6.1). Then, a learned activity schema is retrieved for solving the given task problem taking into account the target task and the scope of applicability of the activity schema (Section 6.2). We present an abstract planner which transforms possible loops in the activity schema into sequences of abstract actions for the objects involved in the abstracted task problem and generates an abstract plan without loops (Section 6.3). Finally, we present a concrete planner to generate the final solution plan by refining the abstract plan towards a concrete plan (Section 6.4). The overall planning process in EBPDs is depicted in Figure 6.1.

6.1 Problem Abstraction

We propose a problem abstraction methodology for transforming a concrete task problem into an abstract task problem. As explained in previous chapters, abstraction is intended to improve problem solving performance by initially ignoring less relevant features of a problem and solving the problem in a coarse fashion with less effort. Then, the derived abstract (skeletal) solution serves as a guide for solving the original concrete problem.

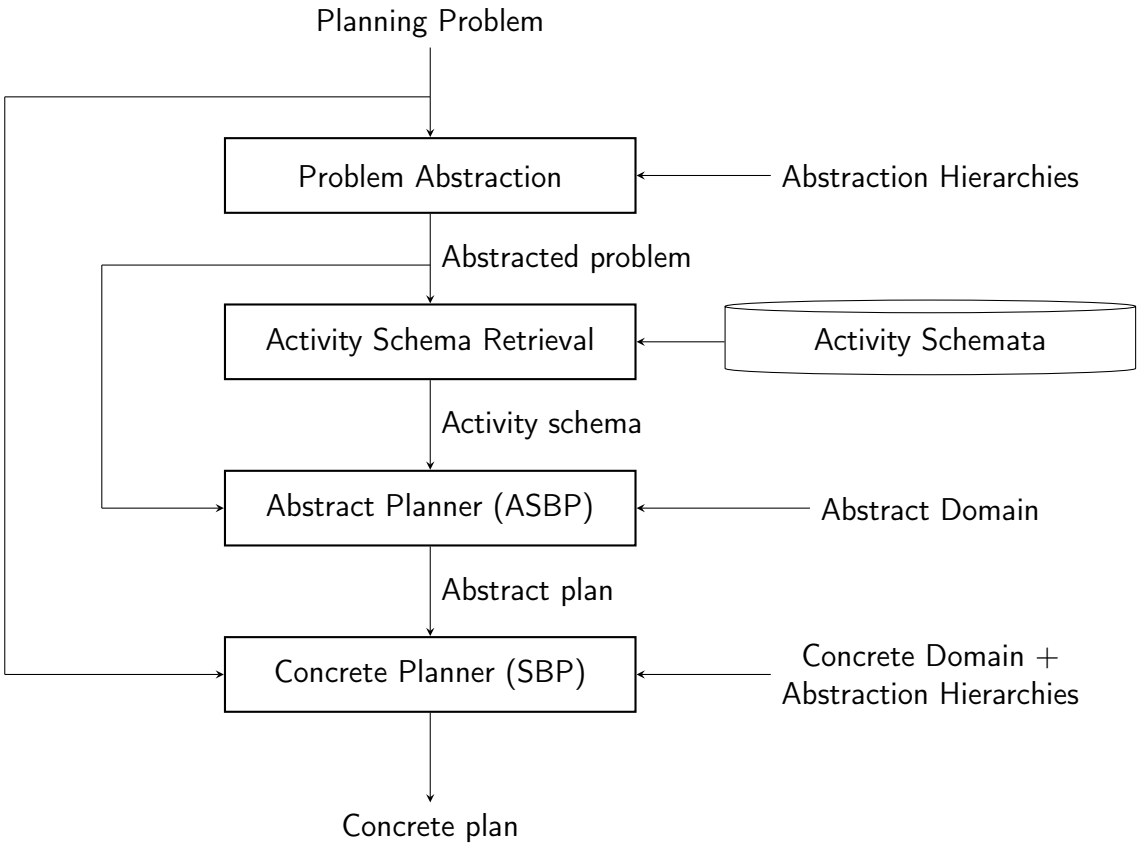


Figure 6.1: Flowchart of the planning system in EBPDs.

Based on the given concrete and abstract planning domains $(\mathcal{D}_c, \mathcal{D}_a)$ and the predicate abstraction hierarchy (Definition 3.10) in an EBPD, the abstraction of a concrete task planning problem is achieved by transforming the concrete predicates into abstract predicates, which results in reducing the level of detail in the concrete task planning problem (see Algorithm 6).

A concrete task planning problem for '`Stack_N_Blue`' in the `Stacking-Blocks` EBPD, containing 10 blue blocks, is shown in Listing 6.1. Listing 6.2 shows this task planning problem after abstraction. This abstraction is achieved using the predicate abstraction hierarchy in Table 3.3. The derived abstract task planning problem is the key input for retrieving an applicable activity schema and generating an abstract solution.

Algorithm 6 Problem Abstraction (ProbAbstraction)

input:
- $\Delta = (\mathcal{L}, \mathcal{D}_a, \mathcal{D}_c, \mathcal{A}, \mathcal{E}, \mathcal{M})$ ▷ an experience-based planning domain (Def. 3.17)
- $\mathcal{P} = (t, \sigma, s_0, g)$ ▷ a concrete task planning problem (Def. 3.6)

output:
- $\mathcal{P}_a = (t, \sigma_a, s_{0a}, g_a)$ ▷ an abstracted task planning problem

```
σ_a ← ∅
s_0a ← ∅
g_a ← ∅
for all p in σ do
    if parent(p) ≠ ∅ then
        σ_a ← σ_a ∪ {parent(p)}
for all p in s_0 do
    if parent(p) ≠ ∅ then
        s_0a ← s_0a ∪ {parent(p)}
for all p in g do
    if parent(p) ≠ ∅ then
        g_a ← g_a ∪ {parent(p)}
return (t, σ_a, s_0a, g_a)
```

6.2 Activity Schema Retrieval

After generating an abstracted task problem from a concrete task problem, the planning system retrieves an applicable activity schema for solving the (abstracted) task problem. An activity schema is applicable for solving a task planning problem if: (*i*) the task achieved by the activity schema is unifiable with the task of the target problem, e.g., the task of the '`Stack_N_Blue`' activity schema (`Stack_N_Blue ?table ?pile`) in Listing 5.6 unifies with the task of the abstracted task problem (`Stack_N_Bluetable1pile1`) in Listing 6.2 with `?table` bound to `table1` and `?pile` bound to `pile1`; and (*ii*) the abstracted task planning problem is *embedded* in the scope of the activity schema, i.e., the abstracted task planning problem maps onto the scope of the activity schema.

To see how the abstracted task planning problem $\mathcal{P}_a = (t, \sigma_a, s_{0a}, g_a)$ is embedded in the scope of an activity schema, we first simulate the set of key-properties K for $\mathcal{P}_a$ by wrapping the predicates of σ_a, s_{0a} and g_a with temporal symbols *during*, *init* and *end*, respectively:

$$K \leftarrow \{\texttt{during(p)} \mid p \in \sigma\} \cup \{\texttt{init(p)} \mid p \in s_0\} \cup \{\texttt{end(p)} \mid p \in g\} \ . \tag{6.1}$$

```
(define (problem Stack_N_Blue)
(:domain stacking-blocks)
(:parameters table1 pile1)
(:objects location1 hoist1 pallet1 block1 block2 block3 block4 block5 block6
     block7 block8 block9 block10)
(:static  (pile pile1)
          (table table1)
          (location location1)
          (hoist hoist1)
          (attached pile1 location1)
          (attached table1 location1)
          (belong hoist1 location1)
          (pallet pallet1)
          (block block1)
          (block block2)
          (block block3)
          (block block4)
          (block block5)
          (block block6)
          (block block7)
          (block block8)
          (block block9)
          (block block10)
          (blue block1)
          (blue block2)
          (blue block3)
          (blue block4)
          (blue block5)
          (blue block6)
          (blue block7)
          (blue block8)
          (blue block9)
          (blue block10))
(:init    (top pallet1 pile1)
          (ontable block1 table1)
          (ontable block2 table1)
          (ontable block3 table1)
          (ontable block4 table1)
          (ontable block5 table1)
          (ontable block6 table1)
          (ontable block7 table1)
          (ontable block8 table1)
          (ontable block9 table1)
          (ontable block10 table1)
          (at hoist1 table1)
          (empty hoist1))
(:goal    (on block1 pallet1)
          (on block2 block1)
          (on block3 block2)
          (on block4 block3)
          (on block5 block4)
          (on block6 block5)
          (on block7 block6)
          (on block8 block7)
          (on block9 block7)
          (on block10 block7)
          (top block10 pile1)))
```

Listing 6.1: A concrete task planning problem for the 'Stack_N_Blue' task, containing 10 blue blocks, in the Stacking-Blocks domain.

Then K is converted to a 2-valued logical structure (as described in Section 5.5) and, for each activity schema available for the target task, the 2-valued logical structure is checked to see if it is embedded in the 3-valued logical structure rep-

```
(define (problem Stack_N_Blue)
(:domain stacking-blocks)
(:parameters table1 pile1)
(:objects pallet1 block1 block2 block3 block4 block5 block6 block7 block8
     block9 block10)
(:static  (pile pile1)
          (table table1)
          (pallet pallet1)
          (block block1)
          (block block2)
          (block block3)
          (block block4)
          (block block5)
          (block block6)
          (block block7)
          (block block8)
          (block block9)
          (block block10)
          (blue block1)
          (blue block2)
          (blue block3)
          (blue block4)
          (blue block5)
          (blue block6)
          (blue block7)
          (blue block8)
          (blue block9)
          (blue block10))
(:init    (top pallet1 pile1)
          (ontable block1 table1)
          (ontable block2 table1)
          (ontable block3 table1)
          (ontable block4 table1)
          (ontable block5 table1)
          (ontable block6 table1)
          (ontable block7 table1)
          (ontable block8 table1)
          (ontable block9 table1)
          (ontable block10 table1))
(:goal    (on block1 pallet1)
          (on block2 block1)
          (on block3 block2)
          (on block4 block3)
          (on block5 block4)
          (on block6 block5)
          (on block7 block6)
          (on block8 block7)
          (on block9 block7)
          (on block10 block7)
          (top block10 pile1)))
```

Listing 6.2: The abstracted task problem for the 'Stack_N_Blue' task of Listing 6.1. Compared to the input task problem, predicates related to the objects `location` and `hoist` are omitted in the abstracted task problem.

resenting the scope of applicability of that activity schema:

Definition 6.1 (**Embedding**). Let $\mathcal{C} = (\mathcal{U}, \iota)$ be a 2-valued logical structure, representing an abstracted task problem, and $\mathcal{S} = (\mathcal{U}', \iota')$ a 3-valued logical structure, representing the scope of an activity schema, over a set of temporal symbols $\mathcal{T}$ and a set of predicate symbols $\mathcal{P}$. We say that $\mathcal{C}$ is *embedded* in $\mathcal{S}$, denoted by $\mathcal{C} \sqsubseteq \mathcal{S}$, if

there exists a surjective function $f : U \to U'$ such that for every temporal symbol $\tau \in T$, predicate symbol $p^{(k)} \in P$ of arity k, and objects $u_1, ..., u_k \in U$, one of the following conditions holds:

$$\begin{gathered} \iota(\tau, p)(u_1, ..., u_k) = \iota'(\tau, p)(f(u_1), ..., f(u_k)) \quad \text{or} \\ \iota'(\tau, p)(f(u_1), ..., f(u_k)) = 1/2 \ . \end{gathered} \tag{6.2}$$

■

Using the above embedding function, the planning system finds an applicable activity schema $m = (t, \mathfrak{S}, G, \Omega)$ to an abstracted task planning problem $\mathcal{P}_a = (t, \sigma_a, s_{0a}, g_a)$ by first checking if the tasks in m and $\mathcal{P}_a$ are syntactically unifiable, and then building a set of key-properties K for $\mathcal{P}_a$ (6.1) and checks if K maps onto $\mathfrak{S}$, i.e., checks whether $\texttt{Struct}(K) \sqsubseteq \mathfrak{S}$ holds.

Note that if there are several learned activity schemata for $\mathcal{P}_a$, the most recent one is used by the planner. In case of failure, the next recent one is used, and so on. If there exists no activity schema for $\mathcal{P}_a$, the planning system cannot generate a plan solution. In this case, supervised task teaching is used to teach a new activity schema.

When an activity schema is retrieved for a task planning problem, the planning system attempts to generate a solution plan for the task. Generating a solution plan involves abstract planning to derive an abstract solution to the abstracted task planning problem by following the retrieved activity schema. The abstract planning may involve loops extension, i.e., determining concrete iterations of a loop.

6.3 Abstract Planning

We develop an abstract planner, *Abstract Schema-Based Planner* (ASBP), to transform the loops in a retrieved activity schema into sequences of abstract operators and generate an abstract plan to solve the target abstracted problem (see Algorithm 7). The planner follows the general approach proposed by Seabra Lopes (1999). The idea is to search forward while following the activity schema. ASBP takes as input an abstract planning domain $\mathcal{D}_a = (\mathcal{L}_a, \Sigma_a, \mathcal{S}_a, \mathcal{O}_a)$, an abstracted task problem $\mathcal{P}_a = (t, \sigma_a, s_{0a}, g_a)$ and an activity schema $m = (t, \mathfrak{S}, G, \Omega)$, and returns a grounded abstract plan π without loops. Line 1 creates the initial node of the search tree. Each node retains a state s, the grounded abstract plan π without loops built so far, the remaining part of the enriched abstract plan Ω, the total cost

Algorithm 7 Abstract Schema-Based Planner (ASBP)

input:
- $\mathcal{D} = (\mathcal{L}, \Sigma, \mathcal{S}, \mathcal{O})$ ▷ an abstract planning domain (Def. 3.5)
- $\mathcal{P} = (t, \sigma, s_0, g)$ ▷ an abstracted task planning problem (Def. 3.6)
- $m = (t, \mathfrak{S}, G, \Omega_0)$ ▷ an applicable activity schema (Def. 3.16)

output:
- π ▷ a grounded abstract plan without loops (Def. 3.7)

```
1  Open ← {(s_0, ∅, Ω_0, 0, 0)}                                  ▷ initialize the root
2  while Open ≠ ∅ do
3      (s, π, Ω, f, c) ← pick a node with the lowest f from Open
4      if (g ≠ ∅ ∧ g ⊆ s) ∨ (g == ∅ ∧ Ω == ∅) then
5          return π
6      if head(Ω) is a loop then                                  ▷ create two parallel branches
7          STEP(s, π, head(Ω) + Ω, f, c)                          ▷ continue loop
8          STEP(s, π, tail(Ω), f, c)                              ▷ terminate loop
9      else
10         STEP(s, π, Ω, f, c)
11 return failure

12 procedure STEP(s, π, Ω, f, c)                                  ▷ generate successors of the front abstract operator in π
13     (o, F) ← head(Ω)                                           ▷ pick the front enriched abstract operator (Def. 3.15)
14     A ← {abstract actions applicable to s instantiated from o}
15     for each abstract action a ∈ A do
16         s_n ← γ(s, a)                                          ▷ state-transition function
17         F' ← extract features of a                             ▷ Alg. 3
18         k ← number of features in F
19         ν ← number of features in F verified in F'
20         c_n ← c + cost(s, a) · (k + 1)/(ν + 1)                 ▷ cost(s, a) = 1
21         if no node with the same s and Ω, and lower f is in Open then
22             Open ← add (s_n, π · a, tail(Ω), c_n + h⁺(s_n), c_n)
```

f, and the cost c of the path from the start node to the current node (line 1). In each planning iteration, a leaf node with the lowest f value is retrieved (line 3). ASBP selects the front enriched abstract operator, $\mathsf{head}(\Omega)$, and instantiates it, i.e., develops the search tree (lines 6–10). If $\mathsf{head}(\Omega)$ is a loop (line 6), the planner generates successors for two alternatives: (*i*) it adds an iteration of the loop to the front of Ω, i.e, $\mathsf{head}(\Omega) + \Omega$ (line 7); and, (*ii*) skips the loop and moves on to the next abstract operator in Ω, i.e, $\mathtt{tail}(\Omega)$ (line 8). This way, successors are generated by applying, not only abstract actions instantiated from the front enriched abstract operator inside the loop, but also abstract actions instantiated from the following enriched abstract operator after the loop. The key idea is to generate all successors

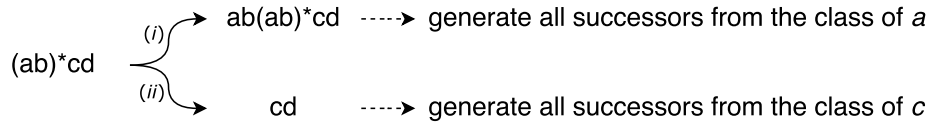


Figure 6.2: Assume that (ab)*cd is an enriched abstract plan, in which each letter represents an enriched abstract operator and (ab)* represents a loop. ASBP generates two classes of successors when gets to a loop in an abstract plan: (*i*) ASBP generates an iteration of the loop and appends it to the beginning of the abstract plan and generates the successors for the obtained abstract plan, i.e., ab(ab)*cd in the figure; and (*ii*) ASBP skips the loop and generates the successors for the rest of the abstract plan, i.e., cd in the figure. ASBP then picks a successor with the lowest cost to develop. This procedure either extends or skips a loop.

of the front enriched abstract operator inside the loop (line 7), and of the following enriched abstract operator after the loop (line 8), and to develop a successor with the lowest cost, hence the planner either continues or terminates the loop, based on the cost of the generated successors. This idea is illustrated in Figure 6.2. On line 10, if $\mathsf{head}(\Omega)$ is not a loop, successors are generated by applying abstract actions instantiated from $\mathsf{head}(\Omega)$.

The core of ASBP is the procedure STEP, used to generate all successors for the class of the front enriched abstract operator in a given enriched abstract plan. On line 13, STEP selects the front enriched abstract operator (o, F) and, on line 14, generates applicable abstract actions instantiated from o. On lines 15–20, the features F' of each abstract action are verified with the features in F, and for each abstract action a a cost c_n is computed as,

$$c_n = c + \mathsf{cost}(s, a) \cdot (k+1)/(\nu+1), \tag{6.3}$$

where k is the total number of features in F, ν is the number of features in F' that are verified in F (i.e., the unground features in F should be syntactically unifiable with ground features in F'), and $\mathsf{cost}(s, a)$ is the real cost of a. This means the applicable abstract actions that verify all features in F, gain the lower cost, and abstract actions with the lower percentage of verified features, gain higher cost. On line 22, the planner appends a to the current grounded abstract plan ($\pi \cdot a$); moves forward on the rest of the enriched abstract plan, i.e., $\mathtt{tail}(\Omega)$; and computes the total cost as $c_n + h^+(s_n)$. The heuristic function h^+ estimates the cost from the current state to the goal g (i.e., we used an *additive heuristic* function). Note that the h^+ is extremely efficient since the ASBP works on abstract planning domain and abstract task problem. On line 4, ASBP stops when, either the goal g is achieved

in a goal state s_g (if g is not empty, depending on the problem formulation of the EBPD (see Section 3.4)); or Ω is empty, i.e., ASBP reaches the end of the enriched abstract plan (if g is empty).

6.4 Concrete Planning

A grounded abstract plan produced by ASBP forms the main skeleton of the final solution. The basic idea is to refine the grounded abstract plan by generating and substituting concrete actions for the abstract actions. This also involves generating and inserting actions from the $\varnothing$ (nil) class (see Definition 3.11 and Section 3.5). We present the *Schema-Based Planner* (SBP) in Algorithm 8, which takes as input an EBPD, $\Delta = (\mathcal{L}, \mathcal{D}_a, \mathcal{D}_c, \mathcal{A}, \mathcal{E}, \mathcal{M})$, and a concrete task planning problem $\mathcal{P} = (t, \sigma, s_0, g)$, and generates a concrete solution plan π for task t. Abstracted task planning problem $\mathcal{P}_a$ is generated from $\mathcal{P}$ using the ProbAbstraction function (see Algorithm 6). Using the Embedding function, introduced in Section 6.2 (see Definition 6.1), SBP looks for an activity schema $m \in \mathcal{M}$ applicable to solving task t (line 2). Note that when an activity schema is found applicable to a task planning problem, the parameters of the activity schema are instantiated based on the parameters of the task planning problem, i.e., they are bounded to the objects appearing as arguments of the task, and propagated throughout the activity schema. For example, the variables `?table` and `?pile`, the arguments of the `'Stack_N_Blue'` activity schema, shown in Listing 5.6, are bounded to `table1` and `pile1`, the arguments of the `'Stack_N_Blue'` task problem, in Listing 6.2, respectively. Depending on the problem formulation in an EBPD (see Section 3.4), if the set of goal propositions in $\mathcal{P}$ is empty, the GoalInstantiate function (see Algorithm 9) generates a set of goal propositions for $\mathcal{P}$ by grounding the goal propositions in m based on $\mathcal{P}$. Goal instantiation is achieved by unifying the unground goal propositions in m (see Section 5.5) with the world information in $\mathcal{P}$. On line 4, the abstract planner ASBP is called to generate a grounded abstract plan π_a without loops (see Algorithm 7). Line 5 creates the root node. In SBP, each node of the search tree retains a state s, the concrete plan π built so far, the remaining part of the grounded abstract plan π_a, the total cost f, and the cost c of the path from the root node to the current node. On each planning iteration, a node with the lowest f value is retrieved (line 7). On lines 9–18, the planner selects the front grounded abstract action, $\mathsf{head}(\pi_a)$, and expands the search tree. SBP on each iteration either generates all concrete actions that have the $\mathsf{head}(\pi_a)$ as parent (line 9), and moves

Algorithm 8 Schema-Based Planner (SBP)

input:
- $\Delta = (\mathcal{L}, \mathcal{D}_a, \mathcal{D}_c, \mathcal{A}, \mathcal{E}, \mathcal{M})$ ▷ an experience-based planning domain (Def. 3.17)
- $\mathcal{P} = (t, \sigma, s_0, g)$ ▷ a concrete task planning problem (Def. 3.6)

output:
- π ▷ a concrete solution plan (Def. 3.7)

1 $\mathcal{P}_a \leftarrow$ ProbAbstraction$(\Delta, \mathcal{P})$ ▷ generate an abstracted task problem (Alg. 6)
2 $m \leftarrow$ Embedding$(\mathcal{P}_a, \mathcal{M})$ ▷ find an applicable activity schema m (Def. 6.1)
3 $g \leftarrow$ GoalInstantiate$(m, \mathcal{P})$ ▷ instantiate the goal of m based on the information in $\mathcal{P}$ (Alg. 9)
4 $\pi_a \leftarrow$ ASBP$(\mathcal{D}_a, \mathcal{P}_a, m)$ ▷ extend loops in m and generate a grounded abstract plan (Alg. 7)
5 Open $\leftarrow \{(s_0, \emptyset, \pi_a, 0, 0)\}$ ▷ initialize the root
6 **while** Open $\neq \emptyset$ **do**
7 $(s, \pi, \pi_a, f, c) \leftarrow$ pick a node with the lowest f from Open
8 **if** $(g \neq \emptyset \wedge g \subseteq s) \vee (g == \emptyset \wedge \pi_a == \emptyset)$ **then return** π
9 $A \leftarrow \{a \in \mathcal{O} \mid$ action a is applicable to s, $\texttt{parent}(a) = \texttt{head}(\pi_a)\}$
▷ concrete actions applicable to s having $\texttt{head}(\pi_a)$ as their parents (Def.3.11)
10 **if** $A \neq \emptyset$ **then**
11 $\pi_a \leftarrow \texttt{tail}(\pi_a)$ ▷ move forward on π_a
12 **else**
13 $A \leftarrow \{a \in \mathcal{O} \mid$ action a is applicable to s, $\texttt{parent}(a) = \varnothing\}$
▷ concrete actions applicable to s from the $\texttt{nil}$ class (Def.3.11)
14 **for each** concrete action $a \in A$ **do**
15 $c_n \leftarrow c + \texttt{cost}(s, a)$ ▷ $\texttt{cost}(s, a) = 1$
16 $h \leftarrow \alpha \cdot \texttt{length}(\pi_a)$ ▷ heuristic based on the length of π_a
17 **if** no node with the same s and π_a, and lower f is in Open **then**
18 Open $\leftarrow$ add $(\gamma(s, a), (\pi \cdot a), \pi_a, (c_n + h), c_n)$
19 **return** failure

forward on π_a (line 11); or generates all concrete actions from the $\varnothing$ ($\texttt{nil}$) class (line 13). In either case, SBP computes a cost $c_n \leftarrow c + \texttt{cost}(s, a)$ (line 15), and estimates a heuristic h as the length of the remaining abstract plan π_a multiplied by a factor α, i.e., $h = \alpha \cdot \texttt{length}(\pi_a)$ (line 16). The factor α is used to calibrate the heuristic value to maximize search efficiency while ensuring admissibility. Empirically, α can be estimated as the average number of actions in a real plan per abstract action in the abstract plan, for example, in the `Stacking-Blocks` domain we estimated $\alpha = 3$ (see the number of concrete actions in Listing 5.2 compared to the number of abstract actions in Listing 5.3). Note that the lowest value of α may cause to underestimate the cost of reaching the goal, and the highest value may cause to overestimate the actual cost. Determining a suitable value of α may require some experimentation depending on the used domain.

Algorithm 9 Goal Instantiation (GoalInstantiate)

input:
- $\mathcal{P} = (t, \sigma, s_0, g)$ ▷ a task planning problem (Def. 3.6)
- $m = (t, \mathfrak{S}, G, \Omega)$ ▷ an activity schema (Def. 3.16)

output:
- g ▷ the set of grounded goal propositions

```
1 if g == ∅ then
2     g ← G with the grounded variables corresponding to the args(t)
3     for all τ(p) in G where τ = during do
4         g ← g ∪ {unify(τ(p), σ)}
5     for all τ(p) in G where τ = init do
6         g ← g ∪ {unify(τ(p), s_0)}
7     g ← {p | τ(p) ∈ g, τ = end}
8 return g
```

On line 18, the planner appends the current action a to the concrete plan $(\pi \cdot a)$ and computes a total cost $c_n + h$, and inserts a new node in the search tree. Finally, on line 8, a concrete solution plan is successfully generated when either the goal g is achieved in a state s, if g is not empty; or Ω is empty, if the goal g is empty.

6.5 Summary

This chapter completes our development of methods for experience-based problem solving in EBPDs. We proposed a hierarchical problem solver which derives a solution plan for a problem using two levels of abstract and concrete planning. Abstract planning applies an activity schema to derive an abstract skeletal solution plan to the given task problem by extending the possible loops to objects in the problem description. During abstract planning, an abstract problem description is achieved by dropping sentences of the concrete problem description. Ignoring such sentences leads to an abstract solution useful to reduce the search at the more concrete planning level. The obtained abstract solution serves as the main skeleton of the final concrete solution during the concrete planning.

Chapter 7

Implementation, Demonstration and Evaluation

In this chapter, we present the system demonstrations and the results of our experiments with different classes of problems in both real and simulated environments. The results involve different aspects of the system in the course of its development. Each experiment involves three phases, namely experience gathering, activity schema learning and planning to solve a new problem. We used two real robotic platforms to demonstrate the utility of our system. The first demonstration was performed within the RACE project[1] on a real PR2 (Section 7.3). The second demonstration, performed in a robotic arm JACO, shows the utility of loops in solving a class of problems (Section 7.4). In addition to real robot demonstrations, we carry out systematic performance evaluation experiments in EBPDs based on classical planning domains well-known in automated planning literature (Section 7.5).

7.1 Prototyping and Implementation

We implemented a prototype of our system in SWI-Prolog, which is a general-purpose logic programming language for fast prototyping artificial intelligence techniques, and used TVLA (Lev-Ami et al., 2004) as an engine, implemented in Java, for computing the scope of applicability of activity schemata. However, some modules are based on old resources including the generalization algorithm and its respective implementation in SWI-Prolog, from (Seabra Lopes, 1997), and the textual interface implemented in Python, from (Chauhan et al., 2013). We ran all experiments in simulated domains and simulated robot platforms, e.g., PR2, on a machine 2.20GHz Intel Core i7 with 12G memory.

[1] RACE (Robustness by Autonomous Competence Enhancement) http://project-race.eu/

7.2 Evaluation Metrics

We used the metrics *penetrance*, *average branching factor* and *effective branching factor* to evaluate the performance of the learning and planning system in solving new problems.

The penetrance is the extent to which the search has focused toward the goal (Russell and Norvig, 2010). It shows how the extracted features guide the planner to avoid developing irrelevant nodes during the search:

Definition 7.1 (**Penetrance**)**.** The penetrance ratio, denoted by $\mathcal{P}$, of a search is:

$$\mathcal{P} = \frac{L}{X} \ , \tag{7.1}$$

where L is the length of the generated plan, and X is the total number of nodes expanded during search, i.e., nodes popped out of the open queue for expansion. ■

The average branching factor is the average number of successors generated for each node in a search problem:

Definition 7.2 (**Average Branching Factor**)**.** The average branching factor, denoted by R, of a search is:

$$R = \frac{N-1}{X} \ , \tag{7.2}$$

where N is the total number of nodes generated during the search and X is the number of expanded nodes. ■

The effective branching factor (Nilsson, 1980; Manickam, 1985) is a measure of the heuristic's usefulness.

Definition 7.3 (**Effective Branching Factor**)**.** The effective branching factor B is the average number of branches at any given node in a uniform search tree, specified by:

$$N = \frac{B^{L+1}-1}{B-1} \ , \tag{7.3}$$

where L is the plan length and N is the total number of nodes generated during the search. ■

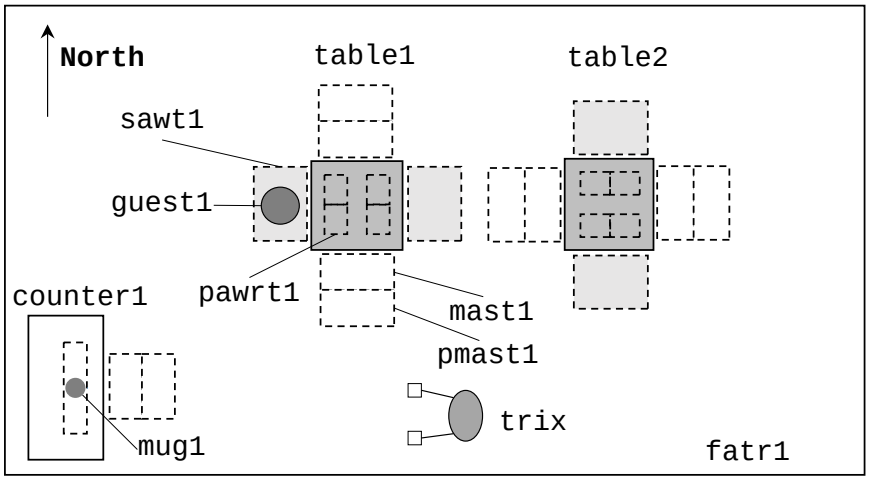
7.3 The RACE Demonstrations and Results

The proposed system was initially developed in the framework of the RACE project (see Sections 1.3 and 3.1.1). We present here the main demonstrations carried out during the course of the project. This demonstration mainly focused on the integration of our system on a real PR2. It shows the utility of the learning and planning system in its early stage of development. The loop detection and scope inference procedures are not included in this experiment. The goal is not included in task problems, therefore, in the conceptualization stage, the goal is automatically inferred from experiences. In Section 3.1.1, and Tables 3.1 and 3.2 we have already described and presented the abstract and concrete planning domains as well as the abstraction hierarchies in the RACE EBPD.

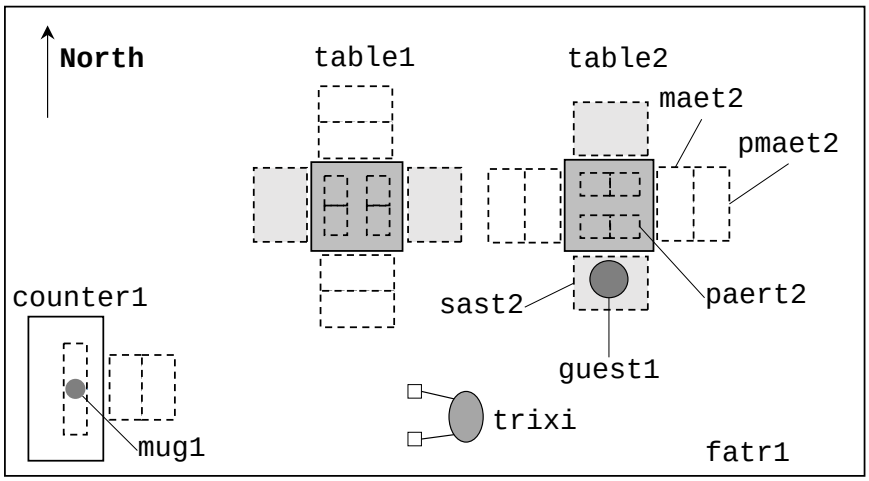
In Section 4.1, we presented a description of the 'ServeACoffee' task scenarios used within the RACE project: a scenario A for experiences-gathering, which is then followed by experience-exploitation in a scenario B. Figure 7.1 shows the initial states of the restaurant environment for scenarios A and B.

Experience gathering is based on the human-robot interaction mechanisms introduced in Chapter 4. This interaction involves a sequence of human instructions to carry out and teach a task. The sequence of instructions in Listing 4.3 (on page 64 of Chapter 4) was used for teaching the robot the scenario A of the 'ServeACoffee' task. Each achieve instruction requests the robot to carry out a specific task. In RACE, the achieve instructions included compound HTN tasks for which an HTN planner was used to generate a plan for each requested task in achieve instructions, e.g, `drive_robot`, `grasp_object`, etc. (Stock et al., 2014; Š. Konečný et al., 2014). The teach task instruction triggers the experience extractor module which generates plan-based robot activity experiences. Figure 7.2 shows snapshots of the main steps in executing a 'ServeACoffee' task in a fully physically simulated Gazebo environment with a PR2 in a restaurant environment. The extracted 'ServeACoffee' experience in this scenario was shown in Listing 4.5 on page 72 of Chapter 4 (see Chapter 4 for detailed description of human-robot interaction and experience gathering). A video of this teaching scenario on a real PR2 is available online: `https://youtu.be/5Z6PJX6Ucfg`.

Conceptualization is carried out immediately after extracting a plan-based robot activity experience (as described in Chapter 5). The resulting activity schema for



(a) Scenario A.



(b) Scenario B.

Figure 7.1: The initial states of the restaurant floor for the 'ServeACoffee' task in scenarios A and B with `trixi` PR2. In both scenarios, `trixi` is taught to place `mug1` on the right side of `guest1` at `table1` and `table2` respectively.

'ServeACoffee' experience was shown in Listing 3.4 on page 53 in Chapter 3. [2]

After teaching the 'ServeACoffee' task, we set up the scenario B and demonstrated the performance of the planning system on a real PR2. A video of the operation of the planning system on the real PR2 is available online: `https://youtu.be/mjrP3hiMRnw`.

In addition to the 'ServeACoffee' task, we used a 'ClearATable' task in the `RACE` domain, including a scenario A to teach and learn the task, and several test scenarios to evaluate the performance of the learned concept in new contexts. Figure 7.3

[2] The original experiences, learned activity schemata and given task problems used in this experiment are available online: `https://github.com/mokhtarivahid/icaps2016/`.

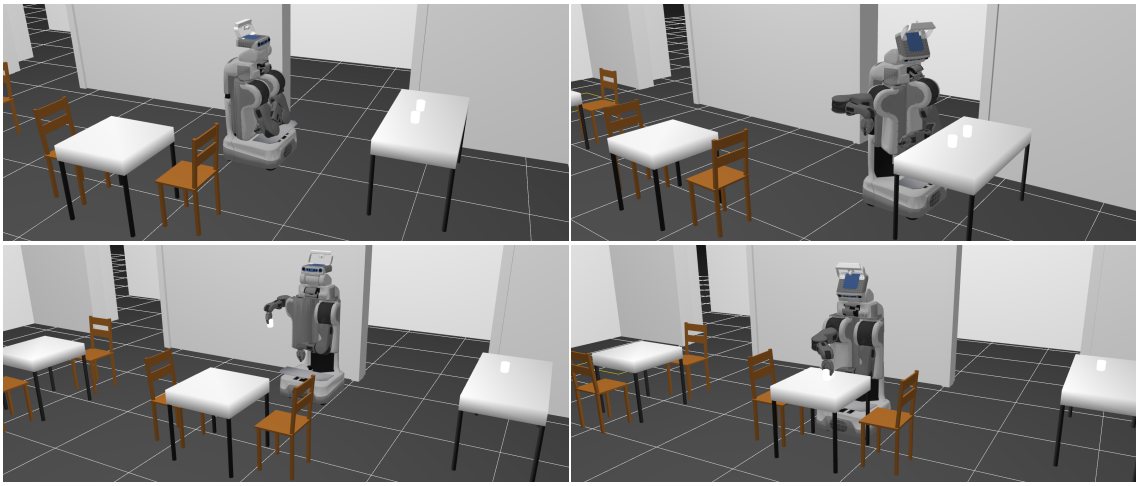


Figure 7.2: An example of the execution of a 'ServeACoffee' task with a PR2 robot in Gazebo simulated environment according to the instructions in Listing 4.5 (in Chapter 4). In this scenario, robot moves to the `counter1` (top-left), picks up `mug1` from the counter (top-right), moves to the `table1` (bottom-left), and puts the mug on the table in front of a guest (bottom-right).

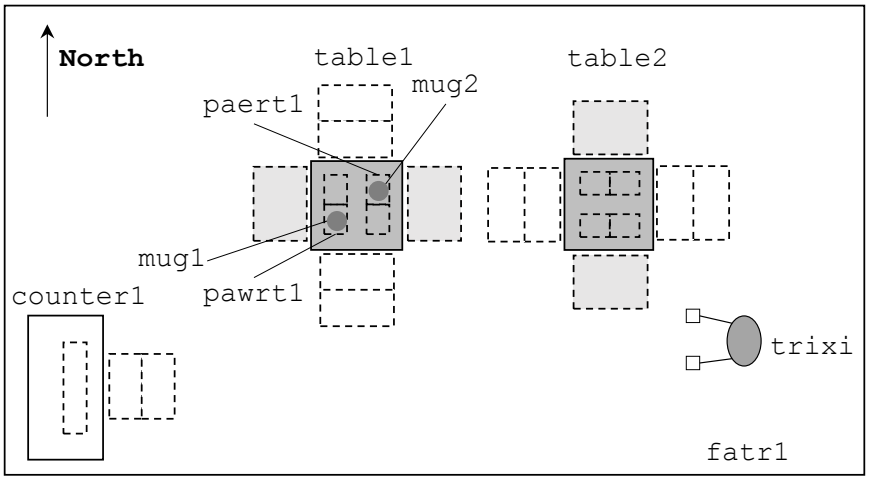


Figure 7.3: The initial state of the restaurant floor for the 'ClearATable' task in scenario A. The `trixi` PR2 robot is taught to clear `table1`.

shows the initial state of the restaurant environment for the 'ClearATable' task in scenario A. To evaluate the planning system over the learned activity schemata, we used three test scenarios in both 'ServeACoffee' and 'ClearATable' tasks. The test scenarios differ in the initial location and configuration of the robot and in the number of objects and their positions in the world state. Listing 7.1 shows the plan executed in the teaching scenario of 'ServeACoffee' as well as the plans generated for the three test scenarios (based on the learned activity schema). Table 7.1 presents the performance metrics for the 'ServeACoffee' and 'ClearATable' tasks in test scenarios.

```
Original experience of: (ServeACoffee ?mug1 ?guest1 ?table ?counter)

1.   (tuck_arms aunp aunp aunp0 aunp1 atp atp atp7 atp13)
2.   (move_base fatr1 pmaec1)
3.   (move_torso torso1 tdp tdp0 tup tup2)
4.   (tuck_arms atp atp atp7 atp13 autp atp autp5 atp1002)
5.   (move_arm_to_side leftarm1 autp autp5 atsp21)
6.   (move_arm_to_side rightarm1 autp autp9 atsp23)
7.   (move_base_blind_to_ma pmaec1 maec1)
8.   (pick_up_object mug1 rightarm1)
9.   (move_base_blind_to_pma maec1 pmaec1)
10.  (move_arms_to_carryposture atsp atsp atsp21 atsp33 acp37 acp39)
11.  (move_torso torso1 tup tup2 tmp tmp4)
12.  (move_base pmaec1 pmast)
13.  (move_torso torso1 tmp tmp4 tup tup6)
14.  (move_arm_to_side rightarm1 acp acp39 atsp53)
15.  (move_base_blind_to_ma pmast mast1)
16.  (place_object mug1 rightarm1 pawrt1)
17.  (move_base_blind_to_pma mast1 pmast)

--------------------------------------------------------------------------------
Experiment #1

1.   (move_base fatr1 pmaec1)
2.   (move_torso torso1 tdp tdp0 tup tup307)
3.   (move_base_blind_to_ma  pmaec1 maec1)
4.   (tuck_arms atp atp atp0 atp1 autp autp autp1101 autp1102)
5.   (move_arm_to_side leftarm1 autp autp1101 asp1174)
6.   (pick_up_object mug1 leftarm1)
7.   (move_base_blind_to_pma maec1 pmaec1)
8.   (move_torso torso1 tup tup307 tmp tmp2404)
9.   (move_arms_to_carryposture asp autp asp1174 autp1102 acp2414 acp2415)
10.  (move_base pmaec1 pmast1)
11.  (move_torso torso1 tmp tmp2404 tup tup4595)
12.  (move_base_blind_to_ma pmast1 mast1)
13.  (move_arm_to_side leftarm1 acp acp2414 asp5011)
14.  (place_object mug1 leftarm1 pawrt1)
15.  (move_base_blind_to_pma mast1 pmast1)

--------------------------------------------------------------------------------
Experiment #2

1.   (tuck_arms aunp aunp aunp0 aunp1 atp atp atp106 atp107)
2.   (move_base fatr1 pmaec1)
3.   (move_torso torso1 tdp tdp0 tup tup398)
4.   (move_base_blind_to_ma pmaec1 maec1)
5.   (tuck_arms atp atp atp106 atp107 autp autp autp1192 autp1193)
6.   (move_arm_to_side leftarm1 autp autp1192 asp1278)
7.   (pick_up_object mug2 leftarm1)
8.   (move_base_blind_to_pma maec1 pmaec1)
9.   (move_torso torso1 tup tup398 tmp tmp2626)
10.  (move_arms_to_carryposture asp autp asp1278 autp1193 acp2636 acp2637)
11.  (move_base pmaec1 pmant1)
12.  (move_torso torso1 tmp tmp2626 tup tup5822)
13.  (move_base_blind_to_ma pmant1 mant1)
14.  (move_arm_to_side leftarm1 acp acp2636 asp6358)
15.  (place_object mug2 leftarm1 paert1)
16.  (move_base_blind_to_pma mant1 pmant1)

--------------------------------------------------------------------------------
Experiment #3

1.   (move_torso torso1 tup tup0 tdp tdp110)
2.   (tuck_arms aunp aunp aunp0 aunp1 atp atp atp132 atp133)
3.   (move_base fatr1 pmaec1)
4.   (move_torso torso1 tdp tdp110 tup tup554)
5.   (move_base_blind_to_ma pmaec1 maec1)
6.   (tuck_arms atp atp atp132 atp133 autp autp autp1348 autp1349)
7.   (move_arm_to_side leftarm1 autp autp1348 asp1460)
8.   (pick_up_object mug3 leftarm1)
9.   (move_base_blind_to_pma maec1 pmaec1)
10.  (move_torso torso1 tup tup554 tmp tmp4058)
11.  (move_arms_to_carryposture asp autp asp1460 autp1349 acp4068 acp4069)
12.  (move_base pmaec1 pmaet2)
13.  (move_torso torso1 tmp tmp4058 tup tup9249)
14.  (move_base_blind_to_ma pmaet2 maet2)
15.  (move_arm_to_side leftarm1 acp acp4068 asp9875)
16.  (place_object mug3 leftarm1 pasrt2)
17.  (move_base_blind_to_pma maet2 pmaet2)
```

Listing 7.1: The plan followed in the original experience of the 'ServeACoffee' task, and the plans generated for three given task problems of the same class. Depending on the given task problems, plans with different lengths and different sequences of actions were generated.

Table 7.1: Evaluation metrics in the 'ServeACoffee' and 'ClearATable' tasks.

Task	ServeACoffee			ClearATable		
Experiment	#1	#2	#3	#1	#2	#3
Plan length (L)	15	16	17	15	16	17
Expanded nodes (X)	210	255	485	224	204	230
Search tree size (N)	787	960	1792	705	732	895
Penetrance (%) ($\mathcal{P}$)	7.12	6.27	3.50	6.69	7.84	7.39
Average branching factor (R)	3.74	3.76	3.69	3.14	3.58	3.88
Effective branching factor (B)	1.44	1.42	1.45	1.42	1.39	1.38

7.4 Robotic Arm Demonstration and Results

We also demonstrated the integration of our system into a real robotic arm JACO, delivered by Kinova Robotics. This is a lightweight assistive robotic device that is used to perform various functions in many applications. For this purpose, we developed a `Jaco` EBPD. Tables 7.2 and 7.3 show the predicate and operator abstraction hierarchies in the `Jaco` EBPD. Full descriptions of the abstract and concrete planning domains can be found in: `https://github.com/mokhtarivahid/prletter2017/tree/master/extra/jaco`. We use a '`JacoClearATable`' task in this EBPD with three different scenarios: one for teaching a task to the robot, and two for evaluating the performance of the learned activity schema. The main objective in this task is to clear a table by removing objects from the table and placing them on a tray. This demonstration is intended to show the utility of loops in solving a class of problems with varying sets of objects in a real robot platform. Loop detection and scope inference procedures are included in this experiment. The goal inference procedure is not included since it is assumed that the goal descriptions are given in the task problem descriptions. The sequence of instructions (Listing 7.2) provided by the human user resulted in generating an experience for this task (Listing 7.3). Conceptualization derives an activity schema (Listing 7.4) immediately after extracting the experience. Figure 7.4(a) shows the 2-valued logical structure representing the '`JacoClearATable`' experience (as in Listing 7.3), and Figure 7.4(b) shows the 3-valued logical structure representing the scope of the '`JacoClearATable`' activity schema (as in Listing 7.4). Figure 7.5 shows snapshots of teaching the '`JacoClearATable`' task to the robot with two objects on the table.

To demonstrate the performance of the system in solving problems, we used two test scenarios, one with three objects on the table and the other with four

Table 7.2: Predicate abstraction hierarchy in the Jaco EBPD.

Abstract predicates	Concrete predicates
(table ?table)	(table ?table)
(tray ?table)	(tray ?table)
(object ?object)	(object ?object)
(holding ?object)	(holding ?arm ?object)
(on ?object ?table)	(on ?object ?table)
(in ?object ?tray)	(in ?object ?tray)
∅	(arm ?arm)
∅	(reach ?arm ?table)
∅	(at ?arm ?table)
∅	(free ?arm)
∅	(detected ?arm ?object ?table)

Table 7.3: Operator abstraction hierarchy in the Jaco EBPD.

Abstract operators	Concrete operators
(pick ?object ?table)	(pick_up_object ?arm ?object ?table)
(place ?object ?tray)	(place_object ?arm ?object ?tray)
∅	(detect_pose_object ?arm ?object ?table)
∅	(move_arm ?arm ?from ?to)
∅	(carry_object ?arm ?object ?from ?to)

objects. We observed that the robot successfully carried out these two tasks. The system was integrated with the work of Shafii et al. (2016), where a robotic arm learns to grasp different objects through kinesthetic teaching and object category learning (Kasaei et al., 2015). Videos of this demonstration are available online in https://goo.gl/zFhm30.

```
1  achieve detect_pose_object jaco1 obj1 table1
2  achieve pick_up_object jaco1 obj1 table1
3  achieve carry_object jaco1 obj1 table1 tray1
4  achieve detect_pose_object jaco1 obj1 tray1
5  achieve place_object jaco1 obj1 tray1
6  achieve move_arm jaco1 tray1 table1
7  achieve detect_pose_object jaco1 obj2 table1
8  achieve pick_up_object jaco1 obj2 table1
9  achieve carry_object jaco1 obj2 table1 tray1
10 achieve detect_pose_object jaco1 obj2 tray1
11 achieve place_object jaco1 obj2 tray1
12 teach_task JacoClearATable table1
```

Listing 7.2: The instructions used to teach the robotic arm the 'JacoClearATable' task.

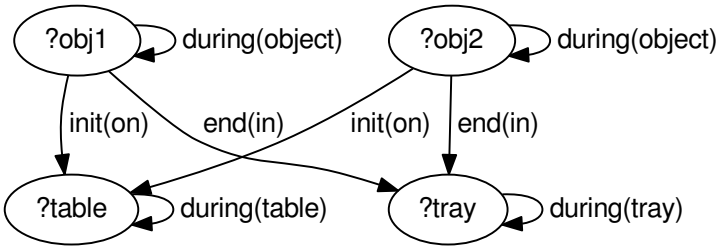
```
(:experience JacoClearATable
 :parameters       (table1 tray1)
 :key-properties   ((init(on obj1 table1))
                    (init(on obj2 table1))
                    (init(free jaco1))
                    (init(at jaco1 table1))
                    (during(object obj1))
                    (during(object obj2))
                    (during(arm jaco1))
                    (during(reach jaco1 table1))
                    (during(reach jaco1 tray1))
                    (during(table table1))
                    (during(tray tray1))
                    (end(in obj1 tray1))
                    (end(in obj2 tray1))
                    (end(free jaco1))
                    (end(at jaco1 tray1)))
 :plan             ((detect_pose_object jaco1 obj1 table1)
                    (pick_up_object jaco1 obj1 table1)
                    (carry_object jaco1 obj1 table1 tray1)
                    (detect_pose_object jaco1 obj1 tray1)
                    (place_object jaco1 obj1 tray1)
                    (move_arm jaco1 tray1 table1)
                    (detect_pose_object jaco1 obj2 table1)
                    (pick_up_object jaco1 obj2 table1)
                    (carry_object jaco1 obj2 table1 tray1)
                    (detect_pose_object jaco1 obj2 tray1)
                    (place_object jaco1 obj2 tray1)))
```

Listing 7.3: An experience for the 'JacoClearATable' task problem in the Jaco EBPD.

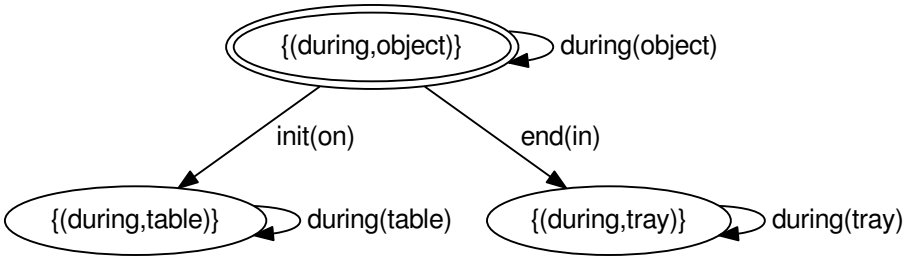
```
(:activity-schema JacoClearATable
 :parameters    (?table ?tray)
 :scope         ((summary (during(object)))
                 (init(on (during(object)) (during(table))))
                 (during(table (during(table))))
                 (during(object (during(object))))
                 (during(tray (during(tray))))
                 (end(in (during(object)) (during(tray)))))
 :abstract-plan ((loop ((!pick ?obj1 ?table)
                             ((init(on ?obj1 ?table))
                              (during(object ?obj1))
                              (during(table ?table))
                              (end(in ?obj1 ?tray))))
                       ((!place ?obj1 ?tray)
                             ((init(on ?obj1 ?table))
                              (during(object ?obj1))
                              (during(tray ?tray))
                              (end(in ?obj1 ?tray)))))))
```

Listing 7.4: The learned activity schema for the 'JacoClearATable' task with a loop of actions and its scope of applicability, in the Jaco EBPD.

For evaluating our system more systematically in the 'JacoClearATable' task, we generated a set of 'JacoClearATable' task problems with varying number of objects, ranging from 10 to 30 in each problem, and evaluated the performance of our system in this set of problems. Table 7.4 presents the performance metrics of the planning system in this experiment. We also compared the obtained results



(a) A concrete structure C.



(b) An abstract structure $\mathfrak{S} = \beta(C)$.

Figure 7.4: The canonical abstraction of the 'JacoClearATable' experience (in Listing 7.3) in the Jaco EBPD, which represents the scope of applicability of the 'JacoClearATable' activity schema. This abstract structure $\mathfrak{S}$ represents all 'JacoClearATable' problems that have exactly one table, one tray and at least one object such that objects are initially on the table and finally in the tray.

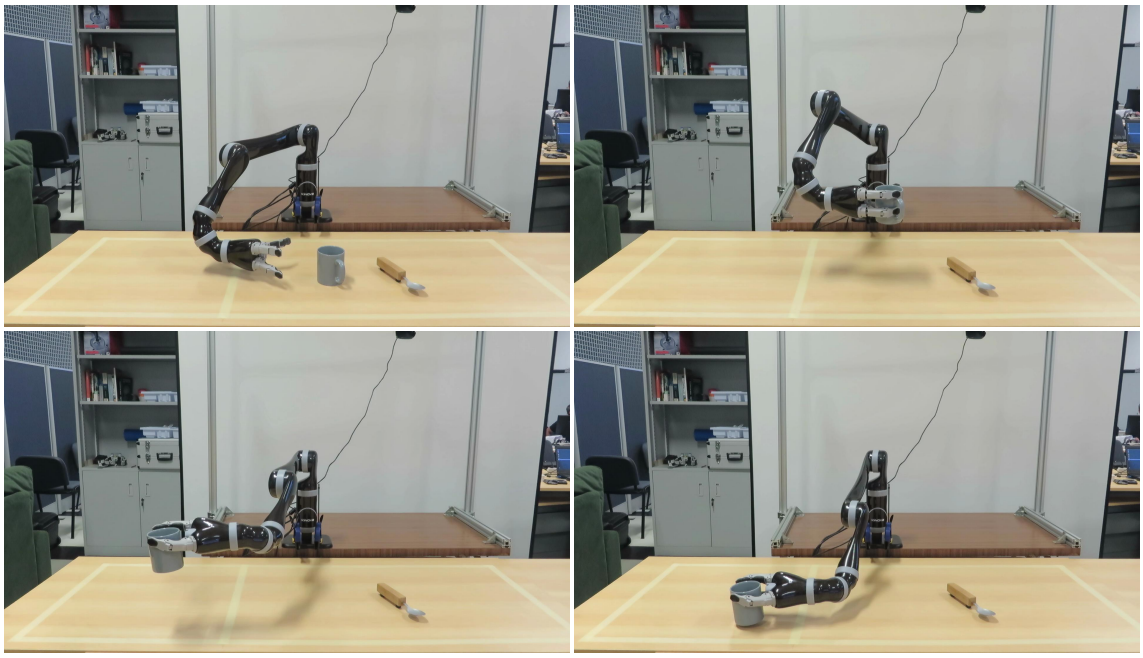


Figure 7.5: From left to right, robot moves to the *cup*, picks up the *cup* from the table, carries the *cup*, and place it on the *tray*.

Table 7.4: Evaluation metrics for SBP in the 'JacoClearATable' task in the Jaco domain.

Problem	Plan length (L)	Expanded nodes (X)	Search tree size (N)	Penetrance (%) ($\mathcal{P}$)	Average branching factor (R)	Effective branching factor (B)
p1	41	62	108	66.13	1.726	1.041
p2	45	68	124	66.18	1.809	1.039
p3	48	72	138	66.67	1.903	1.038
p4	52	78	156	66.67	1.987	1.036
p5	56	84	175	66.67	2.071	1.035
p6	61	92	198	66.30	2.141	1.033
p7	65	98	219	66.33	2.224	1.032
p8	69	104	241	66.35	2.308	1.032
p9	73	110	264	66.36	2.391	1.030
p10	77	116	288	66.38	2.474	1.029
p11	81	122	313	66.39	2.557	1.028
p12	85	128	339	66.41	2.641	1.027
p13	88	132	363	66.67	2.742	1.027
p14	93	140	394	66.43	2.807	1.026
p15	97	146	423	66.44	2.890	1.025
p16	100	150	450	66.67	2.993	1.025
p17	105	158	484	66.46	3.057	1.024
p18	108	162	513	66.67	3.160	1.024
p19	113	170	549	66.47	3.224	1.023
p20	117	176	583	66.48	3.307	1.022

with a state-of-the-art planner, Madagascar (M) (Rintanen, 2012), based on four measures: *time*, *number of expanded nodes*, *plan length* and *memory* (see Table 7.5). Note that the time comparison is not accurate (not fair for SBP) in this evaluation, since SBP has been implemented in Prolog, in contrast to Madagascar planner, which has been implemented in C++. Madagascar was slightly faster than SBP, while SBP, by contrast, is efficient in terms of memory and expanded nodes in the search tree. Figure 7.6 summarizes the performance of the two planners.

The original experience, the learned activity schema and given task problems used in this experiment are available online by the link: `https://github.com/mokhtarivahid/prletter2017/`.

7.5 Standard Planning Domains

In addition to the real robot demonstrations, we developed two EBPDs based on classical planning domains well-known in the automated planning literature, and evaluated our system in different classes of tasks in these EBPDs.

Table 7.5: Performance of the SBP and Madagascar (M) planners in the 'JacoClearATable' task in Jaco domain.

Problem/ (#objects)	Total time (s)		Memory (MB)		Expanded nodes		Plan length	
	SBP	M	SBP	M	SBP	M	SBP	M
p1 (10)	0.84	0.00	2.0	24.4	62	806	41	41
p2 (11)	0.92	0.01	2.2	24.6	68	827	45	45
p3 (12)	0.11	0.01	2.2	24.6	72	691	48	48
p4 (13)	0.10	0.00	2.6	26.9	78	1204	52	52
p5 (14)	0.11	0.01	2.9	27.2	84	1243	56	56
p6 (15)	0.12	0.02	3.3	29.8	92	1847	61	61
p7 (16)	0.15	0.02	3.6	29.8	98	1727	65	65
p8 (17)	0.16	0.03	3.9	30.0	104	1796	69	69
p9 (18)	0.16	0.03	4.2	34.6	110	2395	73	73
p10 (19)	0.18	0.04	4.2	34.6	116	2456	77	77
p11 (20)	0.18	0.04	4.6	37.5	122	3125	81	81
p12 (21)	0.21	0.05	5.0	35.7	128	3259	85	85
p13 (22)	0.22	0.05	5.2	38.0	132	3295	88	88
p14 (23)	0.24	0.07	5.7	40.8	140	3962	93	93
p15 (24)	0.25	0.06	6.1	41.4	146	3977	97	97
p16 (25)	0.27	0.09	6.4	44.5	150	5150	100	100
p17 (26)	0.30	0.09	6.9	43.7	158	5356	105	105
p18 (27)	0.31	0.10	7.5	46.2	162	5186	108	108
p19 (28)	0.34	0.13	7.3	47.8	170	6253	113	113
p20 (29)	0.37	0.12	7.8	47.0	176	6228	117	117

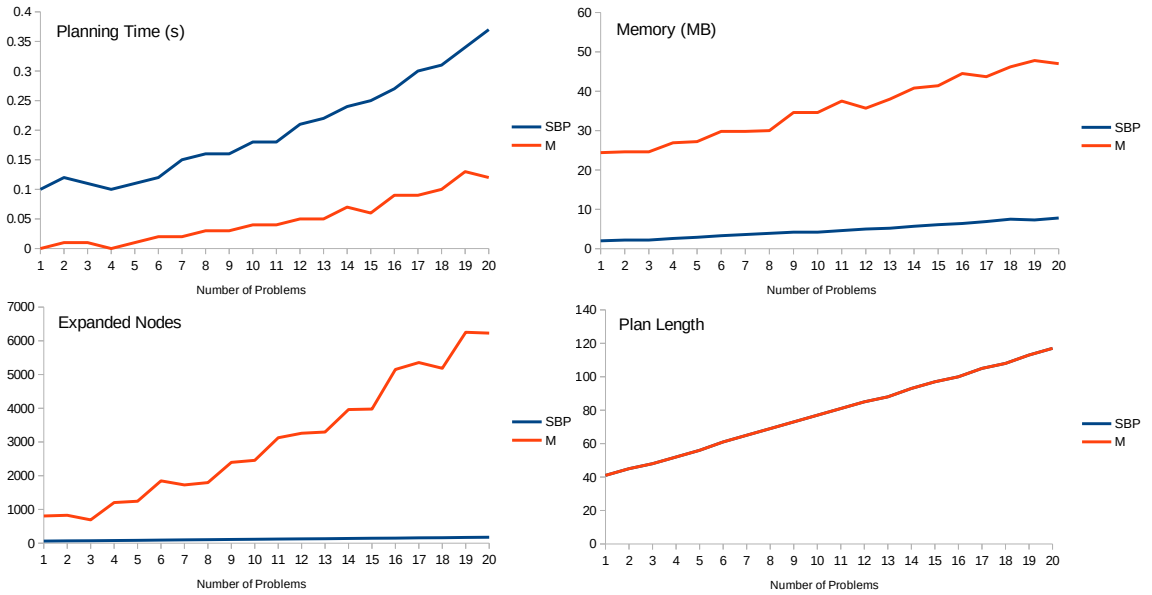


Figure 7.6: Performance of the SBP and Madagascar (M) in the 'JacoClearATable' task.

Stacking-Blocks Domain. The first domain is `Stacking-Blocks`, an extension and adaptation of the standard blocks world domain (Gupta and Nau, 1992). In Section 3.1.2 we already described and presented this domain (see also Tables 3.3 and 3.4). We develop four classes of '`Stack_N_Blue_N_Red`' task problems with the same goal but different initial configurations. The following types of initial configurations were considered:

1. a number of red and blue blocks on a table;
2. a pile of red and blue blocks, with red blocks at the bottom and blue blocks on top;
3. a pile of alternating red and blue blocks, with a blue block at the bottom and a red block on top; and
4. a pile of alternating red and blue blocks, with a red block at the bottom and a blue block on top.

In the four types of problems, the goal is always to make a new pile of red and blue blocks with blue blocks at the bottom and red blocks on top. The main objective of this experiment is to learn a set of different activity schemata (tasks) with the same goal but different scopes of applicability, and to evaluate how the scope testing (embedding) function allows the system to automatically find an applicable activity schema to a given task problem.

To show the effectiveness of the proposed scope of applicability inference, we simulated an experience (containing an equal number of 20 blocks of red and blue colors) in each of the above classes. Based on them the system generates four activity schemata with distinct scopes of applicability (see Figures 7.7, 7.8, 7.9 and 7.10). We evaluated the system over the learned activity schemata by randomly generating 60 task problems in all four classes of the '`Stack_N_Blue_N_Red`' tasks, ranging from 20 to 50 equal number of red and blue blocks in each problem. In this experiment, the system retrieved applicable activity schemata to solve the given task problems in under 50ms for testing the scope of applicability (see Table 7.7). Note that the average time of retrieving an activity schema increases linearly with the number of learned activity schemata for a specific task, i.e., in this experiment we ran among four learned activity schemata for the '`Stack_N_Blue_N_Red`' task. After finding relevant activity schemata, SBP successfully solved all the problems. The performance metrics of the planning system are presented in Table 7.6.

Table 7.6: Evaluation metrics for SBP in the 'Stack_N_Blue_N_Red' task.

Problem	Plan length (L)	Expanded nodes (X)	Search tree size (N)	Penetrance (%) ($\mathcal{P}$)	Average branching factor (R)	Effective branching factor (B)
p1	87	131	593	66.41	4.519	1.035
p2	88	133	152	66.17	1.135	1.011
p3	95	143	695	66.43	4.853	1.034
p4	96	145	166	66.21	1.138	1.011
p5	103	155	805	66.45	5.187	1.032
p6	104	157	180	66.24	1.140	1.010
p7	111	167	923	66.47	5.521	1.031
p8	119	179	1049	66.48	5.855	1.029
p9	112	169	194	66.27	1.142	1.009
p10	120	181	208	66.30	1.144	1.008
p11	127	191	1183	66.49	6.188	1.028
p12	172	271	472	63.47	1.738	1.010
p13	128	193	222	66.32	1.145	1.007
p14	135	203	1325	66.50	6.522	1.026
p15	136	205	236	66.34	1.146	1.007
p16	143	215	1475	66.51	6.856	1.026
p17	144	217	250	66.36	1.147	1.007
p18	151	227	1633	66.52	7.189	1.024
p19	152	229	264	66.38	1.148	1.006
p20	191	288	586	66.32	2.031	1.010
p21	175	264	515	66.29	1.947	1.010
p22	160	241	278	66.39	1.149	1.006
p23	188	296	539	63.51	1.818	1.009
p24	207	312	661	66.35	2.115	1.009
p25	223	336	740	66.37	2.199	1.009
p26	159	239	1799	66.53	7.523	1.024
p27	204	321	610	63.55	1.897	1.009
p28	167	251	1973	66.53	7.857	1.023
p29	168	253	292	66.40	1.150	1.006
p30	220	346	685	63.58	1.977	1.008
p31	236	371	764	63.61	2.057	1.008
p32	176	265	306	66.42	1.151	1.005
p33	175	263	2155	66.54	8.190	1.022
p34	239	360	823	66.39	2.283	1.008
p35	252	396	847	63.64	2.136	1.008
p36	255	384	910	66.41	2.367	1.008
p37	183	275	2345	66.55	8.524	1.021
p38	184	277	320	66.43	1.152	1.005
p39	271	408	1001	66.42	2.451	1.008
p40	268	421	934	63.66	2.216	1.007
p41	192	289	334	66.44	1.152	1.005
p42	287	432	1096	66.44	2.535	1.007
p43	191	287	2543	66.55	8.857	1.021
p44	199	299	2749	66.56	9.191	1.021
p45	284	446	1025	63.68	2.296	1.007
p46	200	301	348	66.45	1.153	1.005
p47	303	456	1195	66.45	2.618	1.007
p48	316	496	1120	63.69	2.376	1.007
p49	300	471	1219	63.71	2.456	1.007
p50	335	504	1298	66.46	2.702	1.007
p51	319	480	1405	66.47	2.786	1.007
p52	332	521	1322	63.72	2.536	1.007
p53	348	546	1429	63.74	2.615	1.006
p54	351	528	1516	66.48	2.869	1.007
p55	367	552	1631	66.49	2.953	1.006
p56	364	571	1540	63.75	2.695	1.006
p57	383	576	1750	66.49	3.036	1.006
p58	380	596	1655	63.76	2.775	1.006
p59	399	600	1873	66.50	3.120	1.006
p60	396	621	1774	63.77	2.855	1.006

Table 7.7: Performance of the SBP and Madagascar (M) planners in terms of applicability test (retrieval) time, search time, memory, expanded nodes and plan length in the different classes of 'Stack_N_Blue_N_Red' problems in the Stacking-Blocks EBPD.

Problem/	Retrieval time (s)	Search time (s)		Memory (MB)		Expanded nodes		Plan length	
(#blocks)	SBP	SBP	M	SBP	M	SBP	M	SBP	M
p1 (22)	0.011	0.29	0.550	10.6	57.2	131	813	87	87
p2 (22)	0.022	0.90	0.820	8.1	92.5	133	597	88	88
p3 (24)	0.010	0.37	0.820	12.4	76.9	143	1011	95	95
p4 (24)	0.021	1.44	1.250	8.9	124.9	145	985	96	96
p5 (26)	0.010	0.49	1.290	13.9	100.2	155	1k	103	103
p6 (26)	0.023	2.16	1.780	9.8	162.4	157	1k	104	104
p7 (28)	0.010	0.61	1.780	15.9	130.2	167	1k	111	111
p8 (30)	0.010	0.77	2.360	17.3	148.8	179	1k	119	119
p9 (28)	0.026	3.22	2.750	10.3	212.9	169	1k	112	112
p10 (30)	0.029	4.67	3.170	11.4	271.2	181	1k	120	127
p11 (32)	0.010	0.95	3.330	19.7	187.10	191	1k	127	127
p12 (22)	0.035	1.48	4.220	16.9	369.7	271	4k	172	172
p13 (32)	0.023	6.84	4.460	12.5	307.4	193	1k	128	128
p14 (34)	0.010	1.16	4.570	22.1	238.5	203	2k	135	135
p15 (34)	0.022	9.14	5.270	13.5	382.8	205	2k	136	136
p16 (36)	0.010	1.43	6.390	24.1	296.7	215	2k	143	143
p17 (36)	0.026	12.31	6.900	13.9	478.5	217	2k	144	144
p18 (38)	0.014	1.73	8.770	26.9	366.7	227	3k	151	151
p19 (38)	0.028	16.87	9.200	15.0	592.2	229	3k	152	152
p20 (24)	0.055	2.05	9.950	17.7	577.7	288	7k	191	184
p21 (22)	0.048	1.33	10.930	16.1	642.6	264	7k	175	168
p22 (40)	0.024	22.23	11.100	16.1	714.4	241	3k	160	160
p23 (24)	0.032	2.30	11.190	19.4	698.8	296	10k	188	188
p24 (26)	0.046	3.04	11.630	20.1	709.5	312	7k	207	200
p25 (28)	0.050	4.50	12.560	22.5	929.2	336	8k	223	216
p26 (40)	0.010	2.07	12.900	29.7	401.1	239	2k	159	159
p27 (26)	0.039	3.78	13.530	21.9	802.6	321	7k	204	204
p28 (42)	0.011	2.48	16.120	31.5	491.7	251	3k	167	167
p29 (42)	0.023	29.52	16.930	17.3	799.7	253	3k	168	168
p30 (28)	0.040	5.27	18.330	23.7	1098.0	346	10k	220	220
p31 (30)	0.037	7.50	21.680	26.6	1388.7	371	11k	236	236
p32 (44)	0.022	38.26	23.090	17.9	947.1	265	4k	176	176
p33 (44)	0.014	2.96	24.690	35.0	593.8	263	4k	175	175
p34 (30)	0.046	6.50	24.980	24.1	1537.7	360	13k	239	232
p35 (32)	0.039	10.69	25.570	28.5	1645.8	396	12k	252	252
p36 (32)	0.046	9.71	26.170	26.8	1645.7	384	11k	255	248
p37 (46)	0.010	3.48	27.970	38.4	709.6	275	4k	183	183
p38 (46)	0.022	63.59	31.420	19.1	1128.5	277	4k	184	184
p39 (34)	0.058	12.68	35.910	29.5	2140.1	408	15k	271	264
p40 (34)	0.040	15.41	36.030	31.7	2126.8	421	13k	268	268
p41 (48)	0.022	103.44	36.120	20.3	1359.2	289	5k	192	192
p42 (36)	0.054	17.65	37.100	31.5	2410.5	432	18k	287	280
p43 (48)	0.010	4.06	37.820	41.9	841.0	287	6k	191	191
p44 (50)	0.014	4.72	41.650	44.7	904.3	299	5k	199	199
p45 (36)	0.040	20.64	42.470	34.9	2397.2	446	18k	284	284
p46 (50)	0.022	131.65	45.770	21.6	1575.5	301	6k	200	207
p47 (38)	0.048	25.80	48.710	34.5	2553.2	456	21k	303	296
p48 (40)	0.034	60.81	55.480	40.0	2658.5	496	19k	316	316
p49 (38)	0.042	35.48	57.540	38.1	2586.7	471	28k	300	300
p50 (42)	0.053	47.82	72.320	40.6	2665.7	504	26k	335	328
p51 (40)	0.061	47.58	84.570	37.5	2727.0	480	35k	319	312
p52 (42)	0.039	53.22	101.080	43.7	2724.10	521	36k	332	332
p53 (44)	0.037	75.32	105.530	47.4	2817.1	546	36k	348	348
p54 (44)	0.049	62.14	114.780	42.6	2793.1	528	43k	351	344
p55 (46)	0.048	84.28	–	46.0	–	552	–	367	–
p56 (46)	0.034	95.31	–	51.1	–	571	–	364	–
p57 (48)	0.056	108.66	–	49.5	–	576	–	383	–
p58 (48)	0.038	120.24	–	53.8	–	596	–	380	–
p59 (50)	0.050	132.46	–	51.6	–	600	–	399	–
p60 (50)	0.040	147.93	–	57.9	–	621	–	396	–

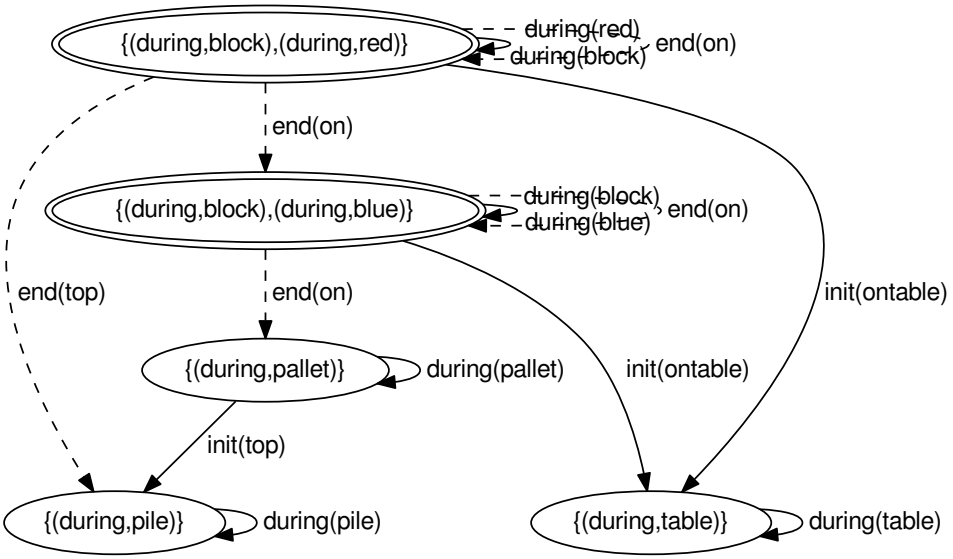


Figure 7.7: The scope of applicability for the task 1 in 'Stack_N_Blue_N_Red'. This scope represents all 'Stack_N_Blue_N_Red' problems that have exactly one table and at least one pile, one pallet, one blue block and one red block such that red and blue blocks are initially on a table and finally red blocks are on top of blue blocks (on a pallet) on a pile.

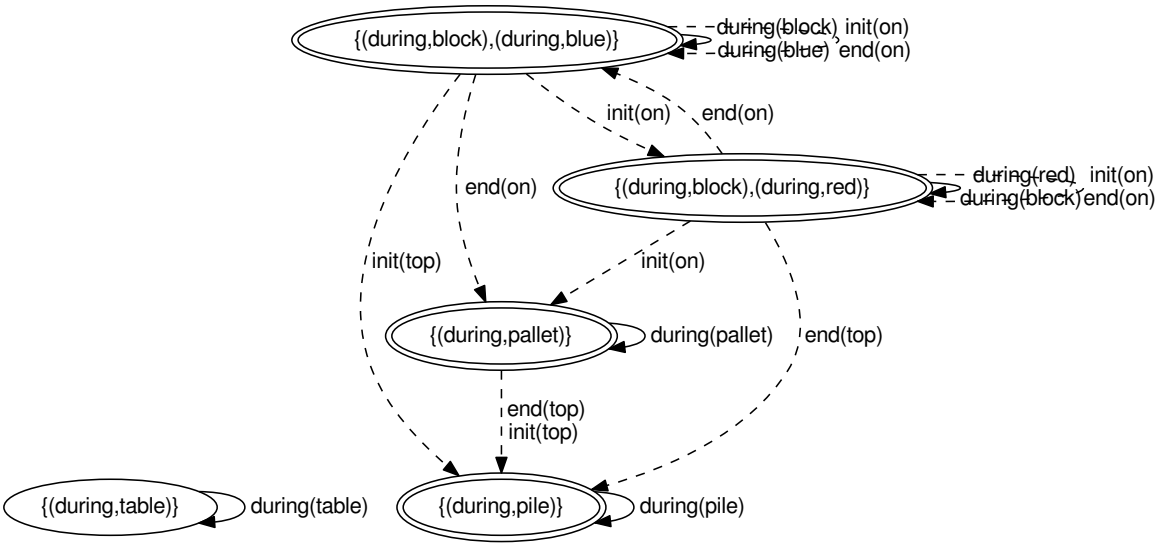


Figure 7.8: The scope of applicability for the task 2 in 'Stack_N_Blue_N_Red'. This scope represents all 'Stack_N_Blue_N_Red' problems that have exactly one table and at least one pile, one pallet, one blue block and one red block such that blue blocks are initially on top of red blocks and finally red blocks are on top of blue blocks on a pile.

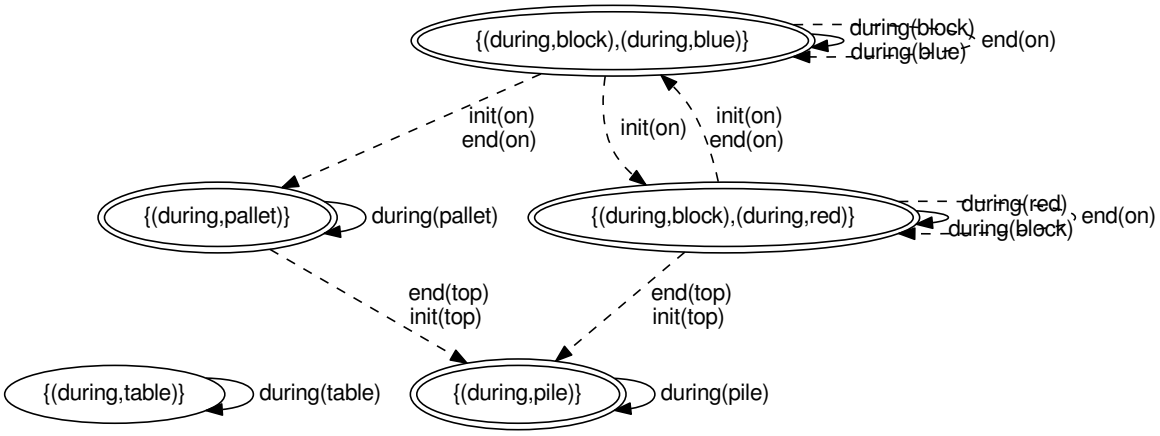


Figure 7.9: The scope of applicability for the task 3 in 'Stack_N_Blue_N_Red'. This scope represents all 'Stack_N_Blue_N_Red' problems that have exactly one table and at least one pile, one pallet, one blue block and one red block such that alternate red and blue blocks are initially on a pile with a blue block at the bottom (on a pallet) and a red block on top and finally red blocks are on top of blue blocks.

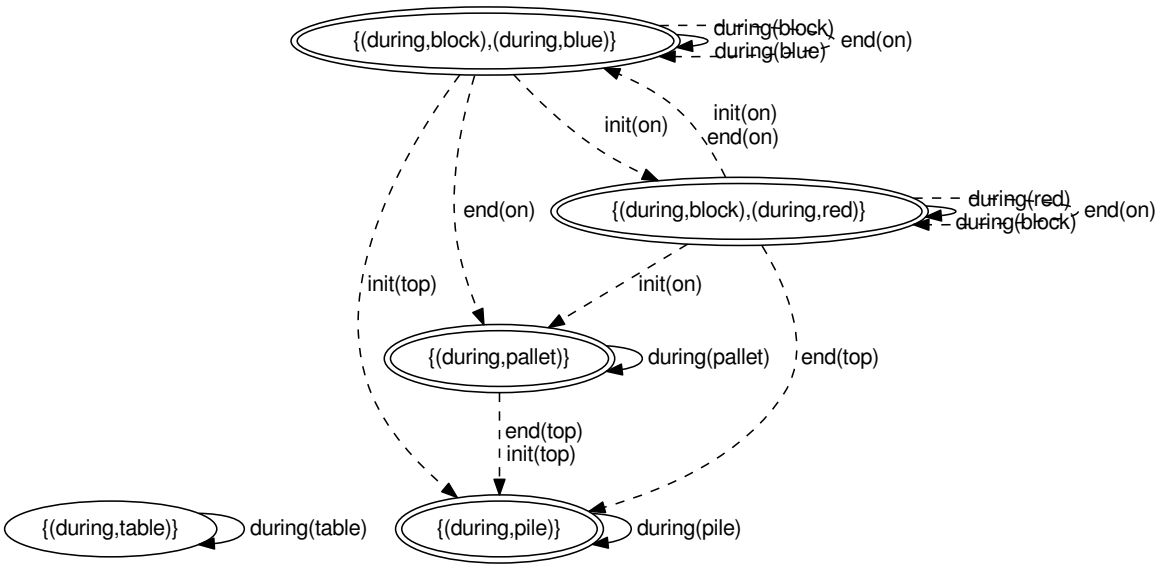


Figure 7.10: The scope of applicability for the task 4 in 'Stack_N_Blue_N_Red'. This scope represents all 'Stack_N_Blue_N_Red' problems that have exactly one table and at least one pile, one pallet, one blue block and one red block such that alternate red and blue blocks are initially on a pile with a red block at the bottom (on a pallet) and a blue block on top and finally red blocks are on top of blue blocks.

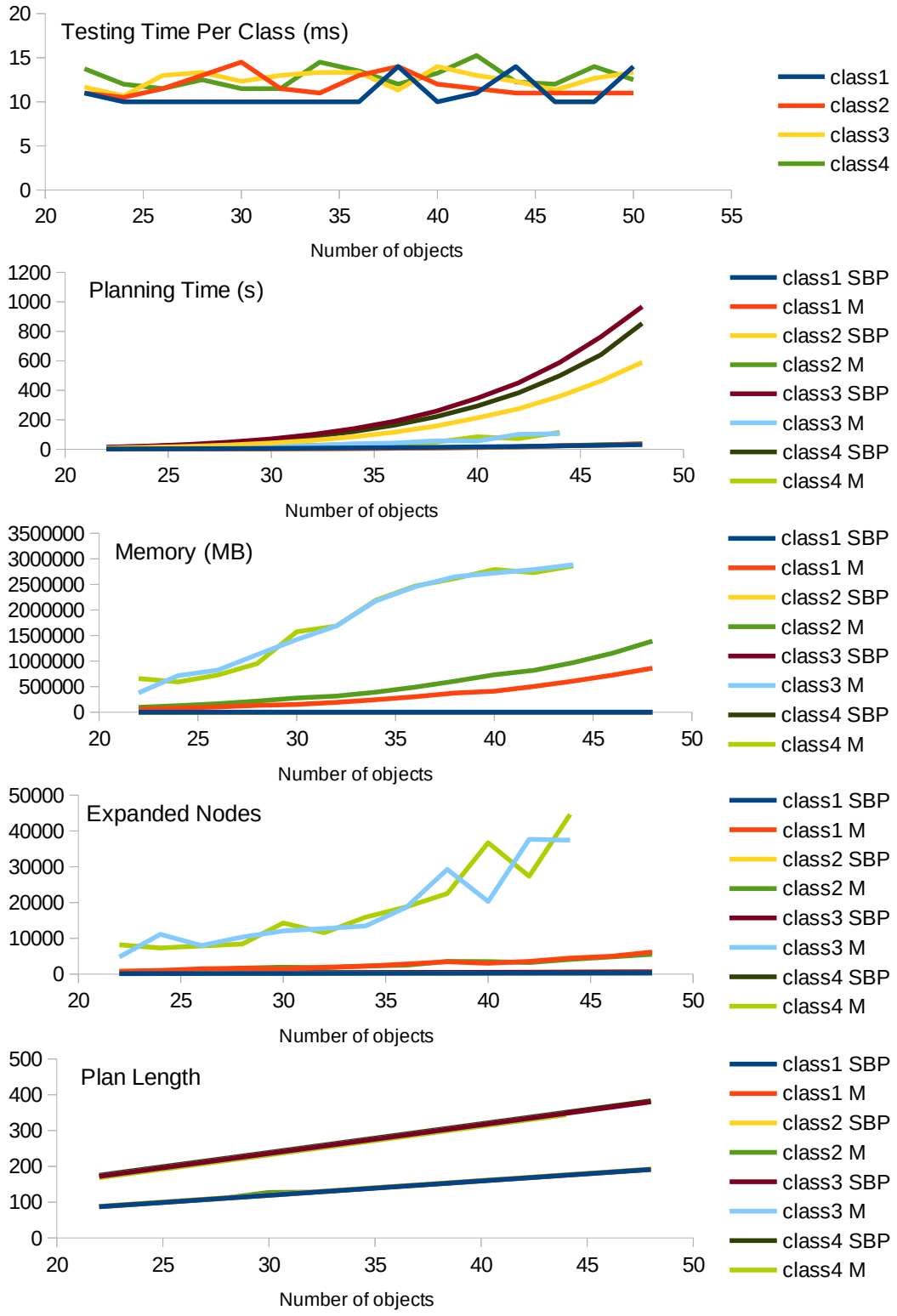


Figure 7.11: Performance of the SBP and Madagascar (M) planners in the 'Stack_N_Blue_N_Red' task.

Table 7.8: Abstract and concrete planning operators in the Satellite domain.

Abstract / Concrete operators
(turn-to ?satellite ?direction1 ?direction2)
(switch-on ?instrument ?satellite)
(switch-off ?instrument ?satellite)
(calibrate ?satellite ?instrument ?direction)
(take-image ?satellite ?direction ?instrument ?mode)

To show the efficiency of the system, we also evaluated and compared the performance of the SBP with Madagascar based on measures: time, memory, number of expanded nodes and plan length (Table 7.7). In this experiment, SBP was extremely efficient in terms of memory and expanded nodes in the search tree. However, Madagascar was slightly faster to solve some problems. Note that the time comparison is not accurate in this evaluation, since SBP has been implemented in Prolog, whereas Madagascar has been implemented in C++. Figure 7.11 summarizes the performance of the two planners.

The original experiences, learned activity schemata and given task problems used in this experiment are available online by the link: https://github.com/mokhtarivahid/ebpd/tree/master/domains/stacking-blocks.

Satellite Domain. The second EBPD is based on the standard Satellite domain (from IPC-2000). In this case, the abstract and concrete planning domains are the same, since the concrete planning domain offers no details to ignore in an abstraction process (see Table 7.8). Full specification of the abstract and concrete planning domains can be found in: https://github.com/mokhtarivahid/ebpd/tree/master/domains/satellite. This experiment is mainly intended to show that the system still works when the abstraction is not available. In this domain, there is a set of satellites equipped with different instruments, which can operate in different modes. The goal is to acquire desired images of targets by dividing the observation tasks between the satellites based on the capabilities of their instruments.

We generated 40 problems in two classes assuming one satellite, a maximum of two instruments, a number of modes ranging from 2 to 10 and a number of targets ranging from 10 to 50 in each problem. We simulated two experiences by randomly choosing one task problem in each class with different configurations (i.e., two experiences differ in the number of instruments used in solutions) and generated two activity schemata with two distinct scopes of applicability for this

class of the problems. The activity schema retrieval time was negligible in this experiment. SBP solved all problems in this class using the two learned activity schemata. Table 7.9 shows the efficiency of SBP in the `Satellite` EBPD compared to MADAGASCAR. SBP was efficient in terms of memory and expanded nodes in the search tree, and MADAGASCAR was fairly fast in time. Figure 7.12 summarizes the performance of the two planners. The original experiences, learned activity schemata and given task problems used in this experiment are available online: `https://github.com/mokhtarivahid/ebpd/tree/master/domains/satellite`.

ROVER. In this experiment, we used the `Rover` domain from the 3rd International Planning Competition (IPC-3). We adopt a different approach for evaluating the proposed scope inference technique. We randomly generated 50 problems containing exactly 1 rover and ranging from 1 to 3 waypoints, 5 to 30 objectives, 5 to 10 cameras and 5 to 20 goals in each problem. Using the scope inference procedure, the problems are classified into 9 sets of problems. That is, problems that converge to the same 3-valued structure are put together in the same set. Hence, each set of problems is identified with a distinct scope of applicability. Figure 7.13 shows the distribution of the problems in the obtained sets of problems. In each set of problems, we simulated an experience and generated an activity schema for problem solving. Figure 7.14 shows the time required to retrieve an applicable activity schema (among 9 activity schemata in this experiment) for solving given problems, i.e., the time required to check whether a given problem is embedded in the scope of an activity schema. SBP successfully solved all problems in each class.

7.6 Summary

In this chapter, we presented the experimental results of our system in different real-world and simulated tasks. We showed how the system integrates and operates in real robot platforms, e.g., a PR2 and a JACO arm robot. Through these experiments, we demonstrated different functionalities of the system, including loop detection, scope inference and goal inference. We showed the results using different evaluation metrics. We showed the timing results for test problems in these experiments. The time required for learning activity schemata, and computing and testing their scopes were negligible. While the results show good scalability, many engineering optimizations are possible on our prototype implementation of

Table 7.9: Performance of the SBP and MADAGASCAR (M) planners in terms of applicability test (retrieval) time, search time, memory, expanded nodes and plan length in the `Satellite` EBPD.

Problem/	Retrieval time (ms)	Search time (s)		Memory (MB)		Expanded nodes		Plan length	
(class#)	SBP	SBP	M	SBP	M	SBP	M	SBP	M
p1 (1)	11	0.20	0.01	0.85	7.6	20	7784	10	11
p2 (1)	10	0.52	0.01	1.78	7.9	32	8048	16	18
p3 (2)	10	0.69	0.00	1.36	8.1	28	8312	13	19
p4 (1)	10	0.91	0.00	1.87	8.6	38	8840	19	19
p5 (2)	10	0.94	0.02	1.81	8.9	32	9104	15	16
p6 (1)	10	1.05	0.01	1.94	9.1	35	9368	15	15
p7 (2)	10	1.15	0.01	1.77	8.9	38	9104	17	18
p8 (1)	10	2.26	0.03	2.56	10.7	43	10952	21	21
p9 (2)	14	2.79	0.02	2.09	10.4	39	10688	19	19
p10 (1)	10	3.19	0.02	3.20	11.2	50	11480	25	26
p11 (2)	11	3.65	0.00	3.57	8.4	67	8576	15	17
p12 (2)	14	4.53	0.05	1.88	13.0	37	13328	15	16
p14 (1)	10	4.70	0.04	3.12	12.8	56	13064	27	28
p13 (1)	10	4.94	0.04	3.04	12.5	54	12800	27	28
p15 (2)	14	6.85	0.03	3.52	13.8	59	14120	29	29
p16 (2)	13	7.60	0.02	3.25	12.0	50	12272	25	26
p17 (1)	14	8.10	0.05	3.34	14.3	58	14648	27	28
p18 (1)	12	8.70	0.07	3.25	16.1	57	16496	25	25
p19 (2)	12	9.12	0.08	4.28	16.9	63	17288	31	32
p20 (1)	11	12.48	0.08	5.14	18.4	70	18872	35	35
p21 (1)	14	14.32	0.11	5.28	19.3	72	19712	35	32
p22 (2)	12	14.58	0.08	4.71	19.2	68	19696	31	35
p23 (2)	12	17.54	0.03	5.46	12.0	94	12272	22	28
p24 (2)	13	18.45	0.10	4.50	22.6	60	23096	30	32
p25 (2)	12	18.95	0.08	3.38	18.2	58	18608	25	25
p26 (1)	12	19.46	0.08	4.64	20.5	67	20984	29	29
p27 (2)	15	25.25	0.11	3.02	22.7	40	23252	20	21
p28 (1)	14	25.76	0.11	5.71	24.4	75	25000	35	35
p29 (2)	12	28.72	0.12	5.16	24.8	73	25392	31	32
p30 (1)	13	29.90	0.15	5.69	25.6	82	26176	39	39
p31 (1)	15	41.21	0.23	6.35	32.0	86	32796	41	42
p32 (2)	12	42.45	0.16	5.89	27.8	83	28416	39	39
p33 (1)	12	46.44	0.16	5.59	28.9	76	29544	38	39
p34 (1)	14	56.43	0.26	6.29	35.0	88	35880	39	39
p35 (2)	13	63.93	0.24	6.93	37.8	91	38684	41	42
p36 (1)	13	65.36	0.23	8.17	39.1	96	39992	45	45
p37 (2)	11	76.25	0.27	6.52	41.9	83	42904	39	39
p38 (1)	14	81.29	0.35	8.78	41.9	100	42908	47	47
p39 (2)	13	114.04	0.27	9.95	38.2	189	39160	36	51
p40 (2)	12	854.11	0.51	31.53	51.7	323	52932	50	68

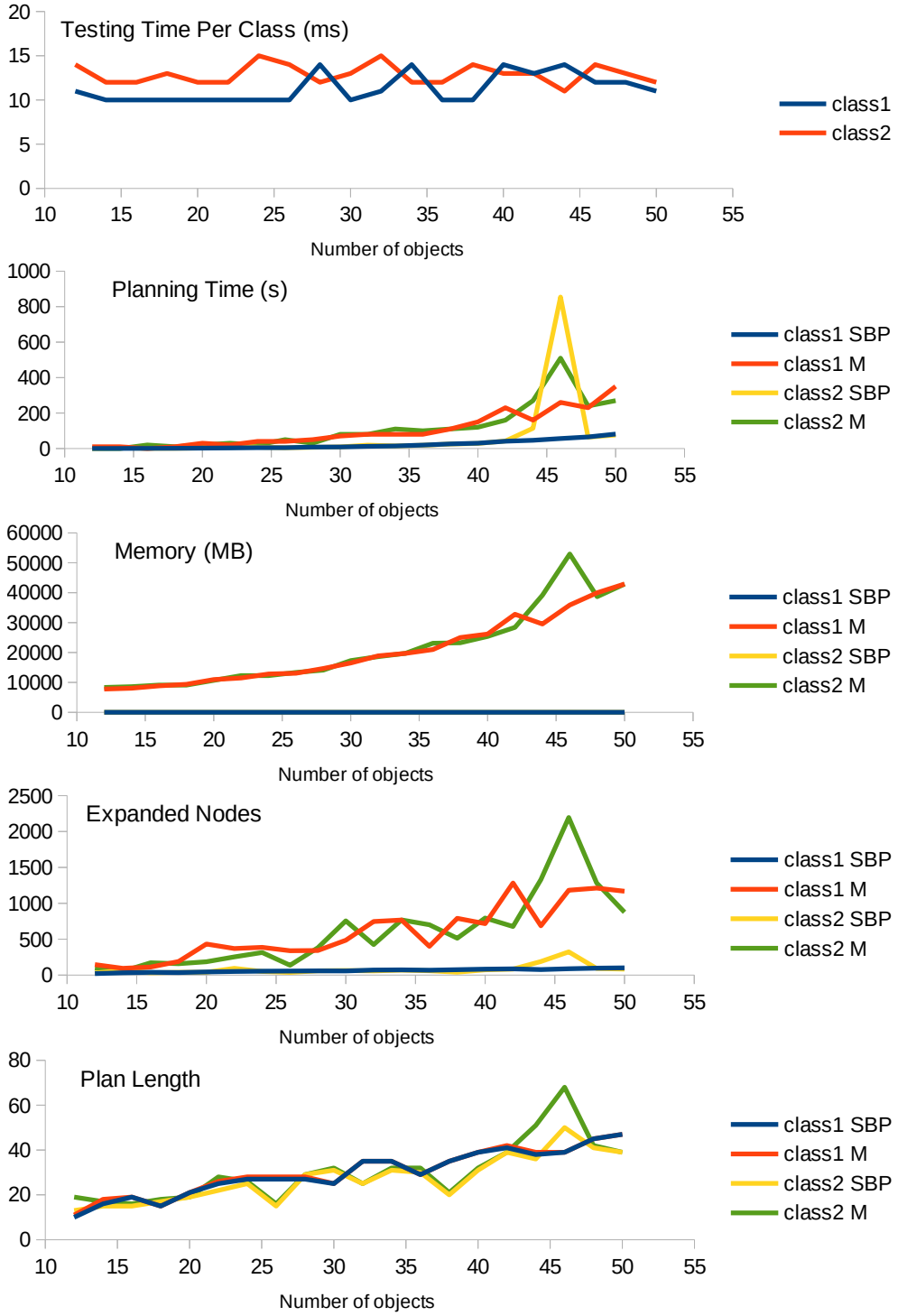


Figure 7.12: Performance of the SBP and Madagascar (M) planners in the `Satellite` domain.

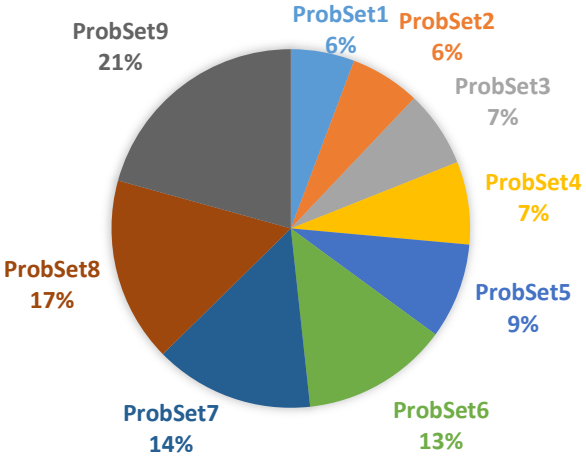


Figure 7.13: Distribution of the problems in the obtained problem sets in the Rover domain

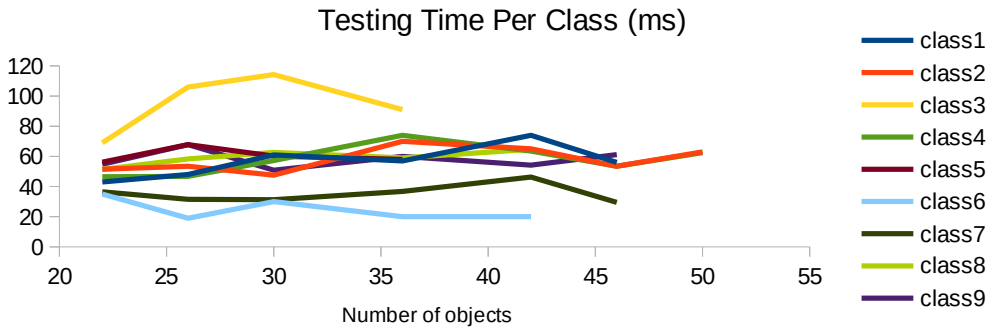


Figure 7.14: CPU time used by SBP to find an applicable activity schema (among 9) for solving problems in the Rover domain.

the presented algorithms. Faster results can be obtained from an implementation in a compiled language.

Our system learned activity schemata from single examples in under a second, in contrast to other machine learning techniques, reviewed in the related work, which usually require large sets of plan traces and longer cpu time to learn planning domain knowledge, e.g., HTN-Maker (Hogg et al., 2008) uses 75 out of 100 input problems to train the system, and CaMeL (Ilghami et al., 2002, 2005) takes about 40 seconds to train the system in the Blocks-World domain.

Chapter 8

Conclusions

The central investigation of this thesis is towards the development of robot capabilities to acquire high-level task planning models, by conceptualizing past experiences, for solving any particular instances of the same tasks. The motivation for tackling this problem centers on the belief that it is impossible to preprogram all the necessary knowledge into a robot operating in a diverse, dynamic and unstructured environment. Instead, robots must have abilities to interact with end-users, and learn and generalize from their own experiences. While significant research has been performed in the area of learning low-level skills, little work concerns learning of high-level task knowledge. Conceptualizing experiences is essential for intelligent robots since it enables them to incrementally and open-endedly acquire new task knowledge. High-level task learning makes robot programs independent from the platform and eases their exchange between robots with different kinematics. The lack of works on symbolic learning of planning algorithms and high-level task representation in robotics motivated us to develop tangible methods for acquiring high-level planning knowledge from experiences which could potentially solve classes of similar problems.

8.1 Summary of the Thesis Contributions

In many robotics applications, particularly those involving Learning from Demonstration, structures are defined and guidelines for information flow are specified in an architecture (Vernon et al., 2007). Apart from basic principles of all cognitive architectures, there are common key components in most architectures for robot learning. According to Langley et al. (2009), following principles are aspects of an agent, which are essential for all mechanisms to work in different application do-

mains: (*i*) short and long-term memories; (*ii*) representation of elements residing in these memories; and (*iii*) functional processes operating on these structures.

In Chapter 3, we proposed a robot architecture (see Section 3.2) for instruction-based task teaching, experience acquisition, learning of high-level methods and problem solving in robotics. This architecture is the underlying framework for Experience-Based Planning Domains (EBPDs) (see Section 3.8). EBPDs integrate the main elements of information involved in the long-term learning and task-level problem solving loop of an intelligent robot. The EBPD representation scheme uses notation derived from first-order logic. The textual representation is an extension and adaptation of the well-established Planning Domain Definition Language (PDDL) (Mcdermott et al., 1998). The proposed EBPD formalization captures many interesting aspects of experience-based learning and planning, which can foster the reuse of this research. It is assumed that the robot is equipped with a set of basic actions (e.g., pickup and putdown) and we aim to enable the robot to construct a high-level task representation of a complex task (e.g., serve a coffee to a guest) built from the existing behavior set.

In Chapter 4, we proposed to use a human-robot interaction interface to support task teaching and supervised experience extraction. A novel graph simplification approach, based on ego networks, was proposed to reduce the complexity and data dimensionality in an experience and to improve the performance in the learning stage (see Section 4.4). The obtained robot activity experiences contain relevant information for learning planning knowledge.

In Chapter 5, we presented our contribution towards developing a methodology for conceptualizing experiences, and generating task planning knowledge in the form of *robot activity schemata*. The conceptualization approach is based on deductive generalization which allows for one-shot-learning by generalizing from a single example (see Section 5.1). This approach is useful for real-world robotic tasks when setting up and performing similar examples is a time-consuming or annoying task. We proposed to use an abstraction methodology for removing inessential details and to identify a common "essence" in an experience (see Section 5.2). Abstraction is the key to computing (Kramer, 2007). Abstract representation allows, during problem solving, to solve more easily the given problems, i.e., with a reduced computational effort. It also makes the learned concepts broader, more compact and more widely applicable. A feature extraction methodology was proposed for selecting a subset of relevant features to include in the learned activity schema (see Section 5.3). Features improve the performance of problem solving

by guiding the planner toward a goal state and reducing the probability of backtracking. They are also useful to capture social norms, physical constraints, etc. One of the key components of the conceptualization procedure is loop detection. In Section 5.4, we proposed a novel algorithm of loop detection, based on the standard method of constructing the Longest Common Prefix (LCP) array. Manber and Myers (1993) have already proved that LCP array is computed in $O(n \log n)$ time where n is the length of the given string, i.e., the length of the plan in an experience, in our work. We also proposed to use Canonical Abstraction, originally developed as part of the TVLA system (Sagiv et al., 2002), to construct an abstract representation of the *scope of applicability* of the activity schema (see Section 5.5). The inferred scope allows for testing the applicability of a learned activity schema to solve a given problem. The computed scope of applicability for the resulting activity schemata may be not sufficient, but still provides useful information to classify the learned activity schemata. Using the proposed techniques, the obtained activity schemata can be applied more generally to solve many problems instances.

In Chapter 6, we used the *embedding* function of TVLA, to retrieve an activity schema relevant for solving a given task problem. The embedding function checks if the task problem matches the scope of applicability of the activity schema. We proposed a hierarchical problem solver which applies a learned activity schema for generating a solution plan to a task problem. Problem solving is achieved by first ignoring less relevant features of a problem description and solving an abstract problem in a coarse fashion with less effort. Then the derived abstract solution serves as a skeleton or guide for solving the original concrete problem.

We demonstrated the utility of our system in different domains, and effectively tackled complex, real world tasks, in contrast to the vast majority of existing robot learning techniques that have been applied to only a single, often unique, domain.

8.2 Directions for Future Work

Although the objectives of this dissertation were successfully achieved, there are still many issues of importance to be tackled in the future. This section is aimed at discussing new possible routes of research arising from the work presented in the previous chapters, taking into consideration the current and future requirements to be fulfilled in order to build more autonomous and smarter robot companions.

The most important prerequisite of this work is the availability of the required background knowledge, namely the concrete planning domain, the abstract planning domain, and the predicate and operator abstraction hierarchies. The formulation of an adequate abstract domain is essential to the success of the approach. If those abstract operators are missing the flexibility of the learned knowledge to solve a wider range of problems with varying initial configurations is constrained. Normally, for the construction of a planning system, the concrete planning domain must be acquired anyway, since it specifies the "language" of the problem description and the problem solution. In our work, the abstract planning domain and the (predicate and operator) abstraction hierarchies must also be acquired. However, for a stronger abstraction framework, the automatic generation of predicate and operator abstraction hierarchies is desirable and can alleviate the burden of domain knowledge engineering. Some early research on knowledge acquisition already described approaches and tools for the acquisition of abstract level operators and hierarchies in real-world domains, e.g., in (Knoblock, 1993, 1994), that can be properly integrated with our system.

We proposed a planning system to generate totally-ordered plans. However a non-linear planner that can generate partially-ordered plans is of interest to real robots. The principal use of a partially-ordered plan is to optimize the execution time by taking advantage of the agent's execution capabilities for parallel actions. In future work we plan to integrate a partial-order planner, based on lifted partial-order planning (Veloso et al., 1990), with the EBPDs' planning system for the extraction of parallel actions. This technique transforms produced sequential plans into plans with parallelism using a post-processing which uncovers the dependencies between the actions.

The proposed abstract planner (ASBP) (Section 6.3) offers a guided state-based search approach to planning. In the abstract planning, the heuristic is the key to increase the efficiency of the planner. In future work we plan to utilize the vast literature on heuristic search to refine and develop methods for guiding the proposed abstract planner.

One of the contributions of this thesis is the Contiguous Non-overlapping Longest Common Prefix (CNLCP) algorithm to identify possible loops of actions in a robot activity experience (see Section 5.4). A limitation of the CNLCP algorithm is to only detect simple loops of actions. The current implementation of the CNLCP algorithm is unable to uncover nested loops (i.e., loops inside loops) in an experience. It would be desirable to extend this algorithm to identify the loops in-

side loops. It will increase the compactness and make broader the applicability of the learned activity schemata. This extension of the CNLCP algorithm could be achieved by recursively applying the CNLCP approach to the identified loops, that is, the identified loops can be given as the new inputs to the CNLCP algorithm until no more loop can be detected. However, the abstract planner is also required to adapt to deal with and expand nested loops during problem solving.

Bibliography

Abbeel, P., Coates, A., Quigley, M., and Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. In *Advances in neural information processing systems*, pages 1–8.

Agostini, A. G., Torras, C., and Wörgötter, F. (2011). Integrating task planning and interactive learning for robots to work in human environments. In *International Joint Conference on Artificial Intelligence*, pages 2386–2391. AAAI Press.

Ahmadzadeh, S. R., Kormushev, P., and Caldwell, D. G. (2013). Visuospatial skill learning for object reconfiguration tasks. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 685–691. IEEE.

Ahmadzadeh, S. R., Paikan, A., Mastrogiovanni, F., Natale, L., Kormushev, P., and Caldwell, D. G. (2015). Learning symbolic representations of actions from human demonstrations. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 3801–3808. IEEE.

Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M., and Taysom, W. (2007). PLOW: A collaborative task learning agent. In *AAAI*, volume 7, pages 1514–1519.

Antoniou, G. and Van Harmelen, F. (2004). Web ontology language: Owl. In *Handbook on ontologies*, pages 67–92. Springer.

Argall, B. D., Chernova, S., Veloso, M., and Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469 – 483.

Bacchus, F. and Yang, Q. (1994). Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100.

Ben-Kiki, O., Evans, C., and Ingerson, B. (2005). Yaml ain't markup language (yamlâĎć) version 1.1. *yaml. org, Tech. Rep*, page 23.

Bentivegna, D. C., Ude, A., Atkeson, C. G., and Cheng, G. (2002). Humanoid robot learning and game playing using pc-based vision. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2449–2454. IEEE.

Bergmann, R. and Wilke, W. (1995). Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research*, 3:53–118.

Billard, A., Calinon, S., Dillmann, R., and Schaal, S. (2008). Robot programming by demonstration. In *Springer handbook of robotics*, pages 1371–1394. Springer.

Billard, A. and Matarić, M. J. (2001). Learning human arm movements by imitation:: Evaluation of a biologically inspired connectionist architecture. *Robotics and Autonomous Systems*, 37(2-3):145–160.

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300.

Bonet, B. (2013). An admissible heuristic for SAS+ planning obtained from the state equation. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI)*, IJCAI '13, pages 2268–2274. AAAI Press.

Bonet, B. and Geffner, H. (1999). Planning as heuristic search: New results. In *European Conference on Planning*, pages 360–372. Springer.

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

Bonet, B., Loerincs, G., and Geffner, H. (1997). A robust and fast action selection mechanism for planning. In *AAAI/IAAI*, pages 714–719.

Borrajo, D., Roubíčková, A., and Serina, I. (2015). Progress in case-based planning. *ACM Computing Surveys (CSUR)*, 47(2):35:1–35:39.

Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-FF: improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, pages 581–621.

Botea, A., Müller, M., Schaeffer, J., et al. (2007). Fast planning with iterative macros. In *IJCAI*, pages 1828–1833.

Browning, B., Xu, L., and Veloso, M. (2004). Skill acquisition and use for a dynamically-balancing soccer robot. In *AAAI*, pages 599–604.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204.

Calinon, S. and Billard, A. G. (2007). What is the teacherâĂŹs role in robot programming by demonstration?: Toward benchmarks for improved learning. *Interaction Studies*, 8(3):441–464.

Calinon, S., Guenter, F., and Billard, A. (2007). On learning, representing, and generalizing a task in a humanoid robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 37(2):286–298.

Carbonell, J. G. and Veloso, M. (1988). Integrating derivational analogy into a general problem solving architecture. In *Proceedings of the First Workshop on Case-Based Reasoning*, pages 104–124.

Chao, C., Cakmak, M., and Thomaz, A. L. (2011). Towards grounding concepts for transfer in goal learning from demonstration. In *Development and Learning (ICDL), 2011 IEEE International Conference on*, volume 2, pages 1–6. IEEE.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial intelligence*, 32(3):333–377.

Chauhan, A., Seabra Lopes, L., Tomé, A. M., and Pinho, A. (2013). Towards supervised acquisition of robot activity experiences: an ontology-based approach. In *16th Portuguese Conference on Artificial Intelligence - EPIA'2013*.

Chen, J. and Zelinsky, A. (2003). Programing by demonstration: Coping with suboptimal teaching actions. *The International Journal of Robotics Research*, 22(5):299–319.

Chernova, S. and Thomaz, A. L. (2014). Robot learning from human teachers. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(3):1–121.

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

Connell, J. H. and Mahadevan, S. (1993). *Robot learning*, volume 233 of *The Springer International Series in Engineering and Computer Science*. Springer.

De la Rosa, T., García-Olaya, A., and Borrajo, D. (2013). A case-based approach to heuristic planning. *Applied intelligence*, 39(1):184–201.

Dean, T. and Boddy, M. (1988). Reasoning about partially ordered events. *Artificial Intelligence*, 36(3):375–399.

DeJong, G. and Mooney, R. (1986). Explanation-based learning: an alternative view. *Machine learning*, 1(2):145–176.

Dillmann, R. (2004). Teaching and learning of robot tasks via observation of human performance. *Robotics and Autonomous Systems*, 47(2-3):109–116.

Dubba, K. S., De Oliveira, M. R., Lim, G. H., Kasaei, H., Lopes, L. S., Tomé, A., and Cohn, A. G. (2014). Grounding language in perception for scene conceptualization in autonomous robots. In *Proceedings of AAAI 2014 spring symposium on qualitative representations for robots*.

Edelkamp, S. and Hoffmann, J. (2004). PDDL2.2: The language for the classical part of the 4th international planning competition. *4th International Planning Competition (IPCâĂŹ04), at ICAPSâĂŹ04*.

Eén, N. and Sörensson, N. (2003). An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer.

Ehrenmann, M., Zollner, R., Rogalla, O., and Dillmann, R. (2002). Programming service tasks in household environments by human demonstration. In *Robot and Human Interactive Communication, 2002. Proceedings. 11th IEEE International Workshop on*, pages 460–467. IEEE.

Ekvall, S. and Kragic, D. (2005). Grasp recognition for programming by demonstration. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 748–753. IEEE.

Ekvall, S. and Kragic, D. (2006). Learning task models from multiple human demonstrations. In *Robot and Human Interactive Communication, 2006. RO-MAN 2006. The 15th IEEE International Symposium on*, pages 358–363. IEEE.

Ekvall, S. and Kragic, D. (2008). Robot learning from demonstration: a task-level planning approach. *International Journal of Advanced Robotic Systems*, 5(3):223–234.

Estlin, T. A. and Mooney, R. J. (1997). Learning to improve both efficiency and quality of planning. In *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2*, IJCAI'97, pages 1227–1232, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Fawcett, T. E. and Utgoff, P. E. (1991). A hybrid method for feature generation. In *Machine Learning Proceedings 1991*, pages 137–141. Elsevier.

Fawcett, T. E. and Utgoff, P. E. (1992). Automatic feature generation for problem solving systems. In *Machine Learning Proceedings 1992*, pages 144–153. Elsevier.

Feil-Seifer, D. and Matarić, M. J. (2009). Human-robot interaction. In *Encyclopedia of complexity and systems science*, pages 4643–4659. Springer.

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. (1972). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208.

Fox, M. and Long, D. (2002). PDDL+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, volume 4, page 34.

Fox, M. and Long, D. (2003). PDDL2.1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*.

Friedrich, H., Mnch, S., Dillmann, R., Bocionek, S., and Sassin, M. (1996). Robot programming by demonstration (rpd): Supporting the induction by human interaction. *Machine Learning*, 23(2-3):163–189.

Garland, A. and Lesh, N. (2003). Learning hierarchical task models by demonstration. *Mitsubishi Electric Research Laboratory (MERL), USA–(January 2002)*.

Georgievski, I. and Aiello, M. (2015). HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156.

Gerevini, A., Saetti, A., and Vallati, M. (2011). Exploiting macro-actions and predicting plan length in planning as satisfiability. In Pirrone, R. and Sorbello, F., editors, *AI*IA 2011: Artificial Intelligence Around Man and Beyond*, volume 6934 of *Lecture Notes in Computer Science*, pages 189–200. Springer Berlin Heidelberg.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated planning: theory & practice*. Elsevier.

Goodrich, M. A. and Schultz, A. C. (2007). Human-robot interaction: a survey. *Foundations and trends in human-computer interaction*, 1(3):203–275.

Gupta, N. and Nau, D. S. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254.

Hammond, K. J. (1986). CHEF: a model of case-based planning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 267–271. AAAI Press.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246.

Henry, P., Krainin, M., Herbst, E., Ren, X., and Fox, D. (2012). Rgb-d mapping: Using kinect-style depth cameras for dense 3d modeling of indoor environments. *The International Journal of Robotics Research*, 31(5):647–663.

Hertzberg, J., Zhang, J., Zhang, L., Rockel, S., Neumann, B., Lehmann, J., Dubba, K., Cohn, A., Saffiotti, A., Pecora, F., Mansouri, M., Konečný, Š., Günther, M., Stock, S., Seabra Lopes, L., Oliveira, M., Lim, G., Kasaei, H., Mokhtari, V., Hotz, L., and Bohlken, W. (2014). The RACE project. *KI - Künstliche Intelligenz*, 28(4):297–304.

Hirschberg, D. S. (1977). Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675.

Hoffmann, J. (2002). Extending ff to numerical state variables. In *ECAI*, pages 571–575. Citeseer.

Hoffmann, J. and Brafman, R. (2005). Contingent planning via heuristic forward search with implicit belief states. In *15th International Conference on Automated Planning and Scheduling (ICAPS)*, volume 2005.

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

Hoffmann, J. and Nebel, B. (2001). The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

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

Hogg, C., Kuter, U., and Muñoz-Avila, H. (2009). Learning hierarchical task networks for nondeterministic planning domains. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1708–1714.

Hogg, C., Munoz-Avila, H., and Kuter, U. (2008). HTN-MAKER: learning HTNs with minimal additional knowledge engineering required. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 950–956. AAAI Press.

Holder, L. B. (1990). The general utility problem in machine learning. In Porter, B. and Mooney, R., editors, *Machine Learning Proceedings 1990*, pages 402–410. Morgan Kaufmann, San Francisco (CA).

Hu, Y. and De Giacomo, G. (2011). Generalized planning: synthesizing plans that work for multiple environments. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, volume 22, pages 918–923.

Hu, Y. and Levesque, H. (2009). Planning with loops: Some new results. In *ICAPS Workshop on Generalized Planning*, page 37.

Hu, Y. and Levesque, H. J. (2011). A correctness result for reasoning about one-dimensional planning problems. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, volume 22, pages 26–38.

Ijspeert, A. J., Nakanishi, J., Hoffmann, H., Pastor, P., and Schaal, S. (2013). Dynamical movement primitives: learning attractor models for motor behaviors. *Neural computation*, 25(2):328–373.

Ijspeert, A. J., Nakanishi, J., and Schaal, S. (2002). Learning rhythmic movements by demonstration using nonlinear oscillators. In *Proceedings of the ieee/rsj int. conference on intelligent robots and systems (iros2002)*, number BIOROB-CONF-2002-003, pages 958–963.

Ikeuchi, K. (1995). Assembly plan from observation. In *Electronic Manufacturing Technology Symposium, 1995, Proceedings of 1995 Japan International, 18th IEEE/CPMT International*, pages 9–12. IEEE.

Ilghami, O. and Nau, D. S. (2006). Learning to do HTN planning. In *16st International Conference on Automated Planning and Scheduling (ICAPS)*, pages 390–393. AAAI Press.

Ilghami, O., Nau, D. S., Munoz-Avila, H., and Aha, D. W. (2002). CaMeL: learning method preconditions for HTN planning. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 131–142.

Ilghami, O., Nau, D. S., Munoz-Avila, H., and Aha, D. W. (2005). Learning preconditions for planning from plan traces and HTN structure. *Computational Intelligence*, 21(4):388–413.

Ingrand, F. and Ghallab, M. (2014). Robotics and artificial intelligence: a perspective on deliberation functions. *AI Communications*, 27(1):63–80.

Ingrand, F. and Ghallab, M. (2015). Deliberation for autonomous robots: a survey. *Artificial Intelligence*.

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*, 27:433–467.

John, G. H., Kohavi, R., and Pfleger, K. (1994). Irrelevant features and the subset selection problem. In *Machine Learning Proceedings 1994*, pages 121–129. Elsevier.

Kambhampati, S. and Hendler, J. A. (1992). A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2-3):193–258.

Kambhampati, S., Katukam, S., and Qu, Y. (1996). Failure driven dynamic search control for partial order planners: an explanation based approach. *Artificial Intelligence*, 88(1-2):253–315.

Kambhampati, S., Parker, E., and Lambrecht, E. (1997). Understanding and extending graphplan. In *European Conference on Planning*, pages 260–272. Springer.

Kasaei, S. H., Oliveira, M., Lim, G. H., Seabra Lopes, L., and Tomé, A. M. (2015). Interactive open-ended learning for 3d object recognition: an approach and experiments. *Journal of Intelligent & Robotic Systems*, 80(3):537–553.

Kautz, H. A., Selman, B., et al. (1992). Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer.

Kleene, S. C. (1952). *Introduction to metamathematics*, volume 483. D. Van Nostrand Co., Inc., New York, N. Y.

Knoblock, C. (1993). *Generating abstraction hierarchies*. Springer US, San Francisco, CA, USA.

Knoblock, C. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243 – 302.

Knoblock, C. A., Minton, S., and Etzioni, O. (1991). Integrating abstraction and explanation-based learning in PRODIGY. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence*, pages 541–546. AAAI Press.

Koenemann, J., Burget, F., and Bennewitz, M. (2014). Real-time imitation of human whole-body motions by humanoids. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 2806–2812. IEEE.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42.

Kulic, D., Takano, W., and Nakamura, Y. (2009). Online segmentation and clustering from continuous observation of whole body motions. *IEEE Transactions on Robotics*, 25(5):1158–1166.

Kuniyoshi, Y., Inaba, M., and Inoue, H. (1994). Learning by watching: extracting reusable task knowledge from visual observation of human performance. *IEEE transactions on robotics and automation*, 10(6):799–822.

Laird, J. E., Gluck, K., Anderson, J., Forbus, K. D., Jenkins, O. C., Lebiere, C., Salvucci, D., Scheutz, M., Thomaz, A., Trafton, G., Wray, R. E., Mohan, S., and Kirk, J. R. (2017). Interactive task learning. *IEEE Intelligent Systems*, 32(4):6–21.

Langley, P., Laird, J. E., and Rogers, S. (2009). Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160.

Lelis, L., Stern, R., Felner, A., Zilles, S., and Holte, R. C. (2012). Predicting optimal solution cost with bidirectional stratified sampling. In *22st International Conference on Automated Planning and Scheduling (ICAPS)*, pages 155–163. AAAI Press.

León, A., Morales, E. F., Altamirano, L., and Ruiz, J. R. (2011). Teaching a robot to perform task through imitation and on-line feedback. In *Iberoamerican Congress on Pattern Recognition*, pages 549–556. Springer.

Lev-Ami, T., Manevich, R., and Sagiv, M. (2004). TVLA: a system for generating abstract interpreters. In *Building the Information Society*, pages 367–375. Springer.

Lev-Ami, T. and Sagiv, M. (2000). TVLA: a framework for kleene logic based static analyses. *Master's thesis, Tel Aviv University*.

Levesque, H. J., Pirri, F., and Reiter, R. (1998). Foundations for the situation calculus. *Electronic Transactions on Artificial Intelligence*, 2:159–178.

Levine, G. and DeJong, G. (2006). Explanation-based acquisition of planning operators. In *ICAPS*, pages 152–161.

Lifschitz, V. (1987). On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9.

Lim, G. H., Oliveira, M., Mokhtari, V., Hamidreza Kasaei, S., Chauhan, A., Seabra Lopes, L., and Tome, A. (2014). Interactive teaching and experience extraction for learning about objects and robot activities. In *Robot and Human Interactive Communication, 2014 RO-MAN: The 23rd IEEE International Symposium on*, pages 153–160.

Manber, U. and Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948.

Manickam, S. (1985). *On penetrance and branching factor for search trees*, pages 381–388. Springer US, Boston, MA.

Markovitch, S. and Rosenstein, D. (2002). Feature generation using general constructor functions. *Machine Learning*, 49(1):59–98.

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. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

McDermott, D. V. (1996). A heuristic estimator for means-ends analysis in planning. In *AIPS*, volume 96, pages 142–149.

McGann, C., Py, F., Rajan, K., Ryan, J. P., and Henthorn, R. (2008). Adaptive control for autonomous underwater vehicles. In *AAAI*, pages 1319–1324.

Merrill, M. D. (2002). First principles of instruction. *Educational technology research and development*, 50(3):43–59.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3):363–391.

Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D. R., Etzioni, O., and Gil, Y. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40(1-3):63–118.

Mitchell, T. M. (1977). Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 1*, pages 305–310. Morgan Kaufmann Publishers Inc.

Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. (1986). Explanation-based generalization: a unifying view. *Machine Learning*, 1(1):47–80.

Mohseni-Kabir, A., Rich, C., Chernova, S., Sidner, C. L., and Miller, D. (2015). Interactive hierarchical task learning from a single demonstration. In *Proceedings of the Tenth Annual ACM/IEEE International Conference on Human-Robot Interaction*, HRI '15, pages 205–212, New York, NY, USA. ACM.

Mokhtari, V., Lim, G., Seabra Lopes, L., and Pinho, A. J. (2016a). Gathering and conceptualizing plan-based robot activity experiences. In Menegatti, E., Michael, N., Berns, K., and Yamaguchi, H., editors, *Intelligent Autonomous Systems 13*, volume 302 of *Advances in Intelligent Systems and Computing*, pages 993–1005. Springer International Publishing.

Mokhtari, V., Seabra Lopes, L., and Pinho, A. J. (2015a). Experience-based planning domains: An approach to robot task learning. In *Proceedings of the 21st Portuguese Conference on Pattern Recognition, RecPad 2015, Faro, Portugal, October 2015*, pages 30–31.

Mokhtari, V., Seabra Lopes, L., and Pinho, A. J. (2016b). Experience-based planning domains: an integrated learning and deliberation approach for intelligent robots. *Journal of Intelligent & Robotic Systems*, 83(3):463–483.

Mokhtari, V., Seabra Lopes, L., and Pinho, A. J. (2016c). Experience-based robot task learning and planning with goal inference. In *26st International Conference on Automated Planning and Scheduling (ICAPS)*, pages 509–517. AAAI Press.

Mokhtari, V., Seabra Lopes, L., and Pinho, A. J. (2017a). An approach to robot task learning and planning with loops. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6033–6038.

Mokhtari, V., Seabra Lopes, L., and Pinho, A. J. (2017b). Learning and planning of robot tasks with loops. In *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 296–301.

Mokhtari, V., Seabra Lopes, L., and Pinho, A. J. (2017c). Learning robot tasks with loops from experiences to enhance robot adaptability. *Pattern Recognition Letters*, 99(Supplement C):57 – 66. User Profiling and Behavior Adaptation for Human-Robot Interaction.

Mokhtari, V., Seabra Lopes, L., Pinho, A. J., and Lim, G. H. (2015b). Planning with activity schemata: closing the loop in experience-based planning. In *Autonomous Robot Systems and Competitions (ICARSC), 2015 IEEE International Conference on*, pages 9–14.

Muise, C., McIlraith, S., Baier, J. A., and Reimer, M. (2009). Exploiting n-gram analysis to predict operator sequences. In *19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 374–377.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of artificial intelligence research*, 20:379–404.

Nejati, N., Langley, P., and Konik, T. (2006). Learning hierarchical task networks by observation. In *Proceedings of the 23rd international conference on Machine learning*, pages 665–672. ACM.

Newman, M. E. (2003). Ego-centered networks and the ripple effect. *Social Networks*, 25(1):83–95.

Newton, M. A. H., Levine, J., Fox, M., and Long, D. (2007). Learning macro-actions for arbitrary planners and domains. In *17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 256–263.

Nicolescu, M. N. and Mataric, M. J. (2003). Natural methods for robot task learning: Instructive demonstrations, generalization and practice. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 241–248. ACM.

Niekum, S., Chitta, S., Barto, A. G., Marthi, B., and Osentoski, S. (2013). Incremental semantically grounded learning from demonstration. In *Robotics: Science and Systems*, volume 9. Berlin, Germany.

Niekum, S., Osentoski, S., Konidaris, G., and Barto, A. G. (2012). Learning and generalization of complex tasks from unstructured demonstrations. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5239–5246. IEEE.

Nilsson, N. J. (1980). *Principles of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Oliveira, M., Seabra Lopes, L., Lim, G. H., Kasaei, S. H., Tomé, A. M., and Chauhan, A. (2016). 3d object perception and perceptual learning in the race project. *Robotics and Autonomous Systems*, 75:614–626.

Pardowitz, M., Knoop, S., Dillmann, R., and Zollner, R. D. (2007). Incremental learning of tasks from user demonstrations, past experiences, and vocal comments. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 37(2):322–332.

Pearl, J. (1984). Heuristics: intelligent search strategies for computer problem solving.

Pecora, F., Mansouri, M., Rockel, S., Stock, S., Günther, M., Seabra Lopes, L., Tomé, A. M., Neumann, B., von Riegen, S., Hotz, L., and Dubba, K. S. R. (2012). Relevant knowledge representation tools and formalisms. Deliverable D1.1 Available from `http://race.informatik.uni-hamburg.de/wordpress/wp-content/uploads/2011/11/RACE-D1.1.pdf`.

Pednault, E. P. (1989). ADL: exploring the middle ground between STRIPS and the situation calculus. *Kr*, 89:324–332.

Pednault, E. P. (1994). ADL and the state-transition model of action. *Journal of logic and computation*, 4(5):467–512.

Penberthy, J. S., Weld, D. S., et al. (1992). UCPOP: A sound, complete, partial order planner for ADL. *Kr*, 92:103–114.

Peot, M. A. and Smith, D. E. (1992). Conditional nonlinear planning. In *Proceedings of the first international conference on Artificial intelligence planning systems*, pages 189–197.

Peters, J. and Schaal, S. (2008). Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697.

Petrick, R. P. and Foster, M. E. (2013). Planning for social interaction in a robot bartender domain. In *23rd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 389–397. AAAI Press.

Rao, R. P. N., Shon, A. P., and Meltzoff, A. N. (2007). *A Bayesian model of imitation in infants and robots*, page 217âĂŞ248. Cambridge University Press.

Reddy, S. Y., Frank, J. D., Iatauro, M. J., Boyce, M. E., Kürklü, E., Ai-Chang, M., and Jónsson, A. K. (2011). Planning solar array operations on the international space station. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(4):41.

Richter, S., Helmert, M., and Westphal, M. (2008). Landmarks revisited. In *AAAI*, volume 8, pages 975–982.

Richter, S. and Westphal, M. (2010). The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177.

Rintanen, J. (2010). Heuristic planning with sat: beyond uninformed depth-first search. In *Australasian Joint Conference on Artificial Intelligence*, pages 415–424. Springer.

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

Rockel, S., Neumann, B., Zhang, J., Dubba, K. S. R., Cohn, A. G., Š. Konečný, Mansouri, M., Pecora, F., Saffiotti, A., Günther, M., Stock, S., Hertzberg, J., Tomé, A. M., Pinho, A. J., Seabra Lopes, L., von Riegen, S., and Hotz, L. (2013). An ontology-based multi-level robot architecture for learning from experiences. In *Designing Intelligent Robots: Reintegrating AI II, AAAI Spring Symposium*, Stanford (USA).

Russell, S. and Norvig, P. (2010). *Artificial intelligence: a modern approach*. Prentice Hall, 3 edition.

Rybski, P., Yoon, K., Stolarz, J., and Veloso, M. (2007). Interactive robot task training through dialog and demonstration. In *Human-Robot Interaction (HRI), 2007 2nd ACM/IEEE International Conference on*, pages 49–56.

Sagiv, S., Reps, T. W., and Wilhelm, R. (2002). Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298.

Saitta, L. and Zucker, J.-D. (2013). *Abstraction in artificial intelligence and complex systems*, volume 456. Springer.

Saunders, J., Nehaniv, C. L., and Dautenhahn, K. (2006). Teaching robots by moulding behavior and scaffolding the environment. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, pages 118–125. ACM.

Seabra Lopes, L. (1997). *Robot Learning at the Task Level. A Study in the Assembly Domain*. PhD thesis, Universidade Nova de Lisboa, Monte da Caparica.

Seabra Lopes, L. (1999). Failure recovery planning in assembly based on acquired experience: learning by analogy. In *Assembly and Task Planning, 1999.(ISATP'99) Proceedings of the 1999 IEEE International Symposium on*, pages 294–300. IEEE.

Seabra Lopes, L. (2007). Failure recovery planning for robotized assembly based on learned semantic structures. In *IFAC Workshop on Intelligent Assembly and Disassembly (IAD'2007)*, pages 65–70.

Seabra Lopes, L. and Connell, J. (2001). Semisentient robots: routes to integrated intelligence. *Intelligent Systems, IEEE*, 16(5):10–14.

Shafii, N., Kasaei, S. H., and Seabra Lopes, L. (2016). Learning to grasp familiar objects using object view recognition and template matching. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2895–2900.

Shavlik, J. W. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5(1):39–70.

Sheridan, T. B. (1992). *Telerobotics, automation, and human supervisory control*. MIT press.

Skoglund, A., Iliev, B., and Palm, R. (2010). Programming-by-demonstration of reaching motions–a next-state-planner approach. *Robotics and Autonomous Systems*, 58(5):607–621.

Srivastava, S., Immerman, N., and Zilberstein, S. (2008). Learning generalized plans using abstract counting. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence*, pages 991–997. AAAI Press.

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

Stock, S., Guenther, M., and Hertzberg, J. (2014). Generating and executing hierarchical mobile manipulation plans. In *ISR/Robotik 2014; 41st International Symposium on Robotics; Proceedings of*, pages 1–6.

Sullivan, K., Luke, S., and Ziparo, V. A. (2010). Hierarchical learning from demonstration on humanoid robots. In *Proceedings of Humanoid Robots Learning from Human Interaction Workshop*, volume 38. Citeseer.

Tulving, E. (2005). Episodic memory and autonoesis: Uniquely human? In H. S. Terrace, . J. M., editor, *The Missing Link in Cognition*, pages 4–56. Oxford Univ. Press, NewYork, NY.

Š. Konečný, Stock, S., Pecora, F., and Saffiotti, A. (2014). Planning domain + execution semantics: a way towards robust execution? In *Qualitative Representations for Robots, AAAI Spring Symposium*.

Van Lent, M. and Laird, J. E. (2001). Learning procedural knowledge through observation. In *Proceedings of the 1st international conference on Knowledge capture*, pages 179–186. ACM.

Veeraraghavan, H. and Veloso, M. (2008). Teaching sequential tasks with repetition through demonstration. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '08, pages 1357–1360, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Veloso, M. M. (1992). *Learning by analogical reasoning in general problem solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Veloso, M. M. (1993). Prodilogy/analogy: Analogical reasoning in general problem solving. In *European Workshop on Case-Based Reasoning*, pages 33–50. Springer.

Veloso, M. M., Pérez, M. A., and Carbonell, J. G. (1990). Nonlinear planning with parallel resource allocation. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pages 207–212. Morgan Kaufmann San Diego, CA.

Vernon, D., Metta, G., and Sandini, G. (2007). A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *IEEE transactions on evolutionary computation*, 11(2):151–180.

Walsh, T. J. and Littman, M. L. (2008). Efficient learning of action schemas and web-service descriptions. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, volume 8, pages 714–719.

Winner, E. and Veloso, M. M. (2007). LoopDISTILL: learning domain-specific planners from example plans. In *ICAPS Workshop on Planning and Scheduling*.

Winner, E. Z. (2008). *Learning domain-specific planners from example plans*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Wood, R., Baxter, P., and Belpaeme, T. (2012). A review of long-term memory in natural and synthetic systems. *Adaptive Behavior*, 20(2):81–103.

Yang, Q. (1997). *Intelligent planning: a decomposition and abstraction based approach*. Springer-Verlag Berlin Heidelberg.

Yang, Y., Li, Y., Fermüller, C., and Aloimonos, Y. (2015). Robot learning manipulation action plans by watching unconstrained videos from the world wide web. In *AAAI*, pages 3686–3693.

Yoon, S., Fern, A., and Givan, R. (2008). Learning control knowledge for forward search planning. *J. Mach. Learn. Res.*, 9:683–718.

Zhuo, H., Li, L., Yang, Q., and Bian, R. (2008). Learning action models with quantified conditional effects for software requirement specification. In Huang, D.-S., Wunsch, D. C., Levine, D. S., and Jo, K.-H., editors, *Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues*, pages 874–881, Berlin, Heidelberg. Springer Berlin Heidelberg.

Zhuo, H. H., Hu, D. H., Hogg, C., Yang, Q., and Munoz-Avila, H. (2009). Learning HTN method preconditions and action models from partial observations. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1804–1810.

Zhuo, H. H., noz Avila, H. M., and Yang, Q. (2014). Learning hierarchical task network domains from partially observed plan traces. *Artificial Intelligence*, 212(0):134 – 157.

Zimmerman, T. and Kambhampati, S. (2003). Learning-assisted automated planning: looking back, taking stock, going forward. *AI Magazine*, 24(2):73–96.

Appendix A

The Stacking-Blocks EBPD

In this appendix, we illustrate a full description and representation of the concrete planning domain, the abstract planning domain, and the abstraction hierarchies for the `Stacking-Blocks` planning domain, in Experience-Based Planning Domains (EBPDs).

```
(define (domain stacking-blocks)
(:requirements :strips)
(:predicates   (location ?l)
               (pile ?p)
               (table ?t)
               (hoist ?h)
               (block ?b)
               (blue ?b)
               (red ?b)
               (pallet ?b)
               (attached ?p ?l)
               (belong ?h ?l)
               (at ?h ?p)
               (holding ?h ?b)
               (empty ?h)
               (in ?b ?p)
               (top ?b ?p)
               (on ?b1 ?b2)
               (ontable ?b ?t))

(:action move
 :parameters   (?h ?from ?to ?l)
 :parent       (nil ())
 :static       (and (location ?l)
                    (hoist ?h)
                    (belong ?h ?l)
                    (attached ?from ?l)
                    (attached ?to ?l))
 :precondition (and (at ?h ?from)
                    (not (= ?to ?from)))
 :effect       (and (at ?h ?to)
                    (not (at ?h ?from))))

(:action unstack
 :parameters   (?h ?a ?b ?p ?l)
 :parent       (unstack (?a ?b ?p))
 :static       (and (location ?l)
                    (hoist ?h)
                    (pile ?p)
                    (belong ?h ?l)
                    (attached ?p ?l))
 :precondition (and (at ?h ?p)
                    (empty ?h)
                    (in ?a ?p)
                    (top ?a ?p)
                    (on ?a ?b))
 :effect       (and (holding ?h ?b)
                    (top ?b ?p)
                    (not (in ?a ?p))
                    (not (top ?a ?p))
                    (not (on ?a ?b))
                    (not (empty ?h))))

(:action stack
 :parameters   (?h ?b ?a ?p ?l)
 :parent       (stack (?b ?a ?p))
 :static       (and (location ?l)
                    (hoist ?h)
                    (pile ?p)
                    (belong ?h ?l)
                    (attached ?p ?l))
 :precondition (and (at ?h ?p)
                    (holding ?h ?b)
                    (top ?a ?p))
 :effect       (and (in ?b ?p)
                    (top ?b ?p)
                    (on ?b ?a)
                    (not (top ?a ?p))
                    (not (holding ?h ?b))
```

```
                    (empty ?h)))

 (:action pickup
  :parameters   (?h ?b ?t ?l)
  :parent       (pick (?b ?t))
  :static       (and (location ?l)
                     (hoist ?h)
                     (table ?t)
                     (belong ?h ?l)
                     (attached ?t ?l))
  :precondition (and (at ?h ?t)
                     (empty ?h)
                     (ontable ?b ?t))
  :effect       (and (holding ?h ?b)
                     (not (ontable ?b ?t))
                     (not (empty ?h))))

 (:action putdown
  :parameters   (?h ?b ?t ?l)
  :parent       (put (?b ?t))
  :static       (and (location ?l)
                     (hoist ?h)
                     (table ?t)
                     (belong ?h ?l)
                     (attached ?t ?l))
  :precondition (and (holding ?h ?b)
                     (at ?h ?t))
  :effect       (and (ontable ?b ?t)
                     (not (holding ?h ?b))
                     (empty ?h)))
)
```

Listing A.1: The concrete planning domain in the `Stacking-Blocks` EBPD.

```
(define (domain stacking-blocks)
(:requirements :strips)
(:predicates   (pile ?p)
               (table ?t)
               (block ?b)
               (blue ?b)
               (red ?b)
               (pallet ?b)
               (holding ?b)
               (in ?b ?p)
               (top ?b ?p)
               (on ?b1 ?b2)
               (ontable ?b ?t))

(:action unstack
 :parameters   (?a ?b ?p)
 :static       (pile ?p)
 :precondition (and (in ?a ?p)
                    (top ?a ?p)
                    (on ?a ?b))
 :effect       (and (holding ?a)
                    (top ?b ?p)
                    (not (in ?a ?p))
                    (not (top ?a ?p))
                    (not (on ?a ?b))))

(:action stack
 :parameters   (?b ?a ?p)
 :static       (pile ?p)
 :precondition (and (holding ?b)
                    (top ?a ?p))
 :effect       (and (in ?b ?p)
                    (top ?b ?p)
                    (on ?b ?a)
                    (not (top ?a ?p))
                    (not (holding ?b))))

(:action pickup
 :parameters   (?b ?t)
 :static       (table ?t)
 :precondition (ontable ?b ?t)
 :effect       (and (holding ?b)
                    (not (ontable ?b ?t))))

(:action putdown
 :parameters   (?b ?t)
 :static       (table ?t)
 :precondition (holding ?b)
 :effect       (and (ontable ?b ?t)
                    (not (holding ?b))))
)
```

Listing A.2: The abstract planning domain in the `Stacking-Blocks` EBPD.

```
(define (abstraction-hierarchies)
(:domain stacking-blocks)

(:predicate-abstraction
    (table ?table)             : (table ?table)
    (pile ?pile)               : (pile ?pile)
    (block ?block)             : (block ?block)
    (blue ?block)              : (blue ?block)
    (red ?block)               : (red ?block)
    (pallet ?pallet)           : (pallet ?pallet)
    (on ?block1 ?block2)       : (on ?block1 ?block2)
    (ontable ?block ?table)    : (ontable ?block ?table)
    (top ?block ?pile)         : (top ?block ?pile)
    (holding ?hoist ?block)    : (holding ?block)
    (location ?location)       : ()
    (hoist ?hoist)             : ()
    (attached ?pile ?location) : ()
    (belong ?hoist ?location)  : ()
    (at ?hoist ?pile)          : ()
    (empty ?hoist)             : ())

(:operator-abstraction
    (unstack ?hoist ?block1 ?block2 ?pile ?loc)
                               : (unstack ?block1 ?block2 ?pile)
    (stack ?hoist ?block2 ?block1 ?pile ?loc)
                               : (stack ?block2 ?block1 ?pile)
    (pickup ?hoist ?block ?table ?loc)
                               : (pick ?block ?table)
    (putdown ?hoist ?block ?table ?loc)
                               : (put ?block ?table)
    (move ?hoist ?from ?to ?loc)
                               : ())
)
```

Listing A.3: The predicate and operator abstraction hierarchies in the Stacking-Blocks EBPD.