

Longterm Generalized Actions for
Smart, Autonomous Robot Agents

by

Jan Winkler

A Dissertation Presented to the
FACULTY 3
UNIVERSITY OF BREMEN, GERMANY
In Partial Fulfillment of the
Requirements of the Degree
Dr.-Ing.
(Doctor of Engineering)

Committee:

Prof. Michael Beetz Computer Science
Prof. David Michael Lane Engineering

Date of doctoral defence: 5th of February, 2018

Affidavit

I hereby confirm that my thesis entitled “Longterm Generalized Actions for Smart, Autonomous Robot Agents” is the result of my own work. I did not receive any help or support from commercial consultants. All sources and materials applied are listed and specified in the thesis.

Furthermore, I confirm that this thesis has not yet been submitted as part of another examination process neither in identical nor in similar form.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, die Dissertation “Longterm Generalized Actions for Smart, Autonomous Robot Agents” eigenständig, d.h. insbesondere selbständig und ohne Hilfe eines kommerziellen Promotionsberaters, angefertigt und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet zu haben.

Ich erkläre außerdem, dass die Dissertation weder in gleicher noch in ähnlicher Form bereits in einem anderen Prüfungsverfahren vorgelegen hat.

Signature / Unterschrift

Acknowledgements

I would like to express my special appreciation and thanks to my advisor Professor Michael Beetz, PhD. You have been a tremendous mentor for me. Your vision and seemingly endless optimism have greatly inspired me, and your advice changed the way I see the field of robotics. I would also like to thank Professor David Michael Lane for serving as my committee member. I am honored that he co-examined my thesis.

I would especially like to thank Ferenc Bálint-Benczédi for the great collaboration throughout the whole time, and for lending his expertise in image understanding. Without him, many aspects of my work would have lacked vision. Also, I thank Lorenz Mösenlechner and Moritz Tenorth for introducing me to CRAM and KnowRob, and for patiently answering my countless questions concerning both. Furthermore, I thank Mihai Pomarlan, Georg Bartels, Zhou (Yuen) Fang, Daniel Nyga, Daniel Bessler, Alexis Maldonado, Gheorghe Lisca, Jan-Hendrik Worch, and Gaya Kozhayan for many fruitful discussions and for their invaluable comments on this document. You did a great deal to help me when I was stuck and brought new ideas to my attention.

Last but not least, a special thanks to the institute's management and secretariat for the great constant support. The institute rests on your shoulders, and you do a great job keeping everything up and running.

Dedication

To my parents for all their love and support and putting me through the best education possible. I appreciate their sacrifices and I would not have been able to get to this stage without them.

To Rebekka for her support, patience, and unending love, and for putting my head right again more than once. I would not have gotten through this doctorate if it was not for her.

Table of Contents

Affidavit / Eidesstattliche Erklärung	i
Acknowledgements	iii
Dedication	v
List of Tables	xi
List of Figures	xii
List of Algorithms	xiv
List of Acronyms	xv
Abstract	xvii
Zusammenfassung	xix
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Research Scenario	2
1.3 The Challenge	4
1.3.1 What is the Challenge?	4
1.3.2 Why is it interesting? Why is it important?	6
1.3.3 Why is it a hard problem? Why has it not been solved yet?	7
1.4 Approach	7
1.5 Contributions	9
1.6 Reader's Guide	11

Chapter 2	Related Work	13
2.1	Planning and Robot Architectures	13
2.1.1	Robot Plan Design and Behavior Control	13
2.1.2	Classical Planning Approaches	14
2.1.3	Robot Architectures	14
2.2	Behavior	15
2.2.1	State Automata	15
2.2.2	Task Networks	15
2.2.3	Behaviour Trees	15
2.3	Autonomy	16
2.3.1	Episodic Memories in Autonomous Systems	16
2.3.2	Vague Task Execution in Real World Scenarios	16
2.3.3	Longterm Autonomy	17
Chapter 3	Overview and Foundations	19
3.1	Generalized Plan Design and Representation for Robots	19
3.2	Data Logging and Learning from Episodic Memories	19
3.2.1	What are Robot Episodic Memories	20
3.3	Embodiment of Autonomous Robot Control Programs	21
Chapter 4	Generalized Plan Design and Representation for Robots	23
4.1	Principles of Generalized Plan Design	23
4.1.1	Action Description vs. Goal Definition	25
4.1.2	Knowledge-based Strategy Selection	25
4.1.3	Experience-based Behavior Enhancement	27
4.1.4	Implicit Modular Task Recovery	28
4.2	CRAM as a Robot Plan Language	29
4.2.1	Robot Plan Design	32
4.2.2	Representing Robot Plans: Behaviour Trees	33
4.3	Mobile Manipulation	34
4.3.1	Common Tasks for Mobile Manipulators	35
4.3.2	Overcoming Space Constraints when Accessing Containers	38
4.3.3	Task Stability: Pre- and Post-Poses in Robot Manipulation	39

4.3.4	Plan Design for Reactive Task Monitoring in Mobile Manipulators	39
4.3.5	Example: Autonomous Object Search Tasks	40
4.3.6	Example: Autonomous Fetch Tasks	41
4.4	Contextual Knowledge in Autonomous Robot Agents	42
4.4.1	Contextually Constraining Generalized Plans	43
4.4.2	Static and Dynamic Knowledge	44
4.5	Planning for Generalized Fetch and Place Activities	45
4.5.1	An A [*] -based Planner for Generalized Actions	46
4.5.2	Example: Re-arrangement of Objects in Retail Shopping Racks	46
4.6	Context-dependent Failure Handling Strategy Selection	48
4.7	Summary	49
Chapter 5 Data Collection and Experience-Based Learning		51
5.1	Performance Enhancement through Experience	51
5.2	Machine Learning	52
5.3	Methods in AI based Robotics	52
5.4	Anatomy of Episodic Memories	53
5.4.1	How Episodic Memories are Recorded	55
5.5	Generating Experience Models from Episodic Memories	57
5.5.1	Expectation Models: Task Outcome Prediction	57
5.5.2	Prototypical Experiences: Informed Strategy Exploration	65
5.6	Multi-modal Analysis of Robot Experiences	70
5.6.1	Approach and Comparison	70
5.6.2	Multi-modal Data Analysis	71
5.6.3	Evaluation	72
5.7	Learning PDDL Domain Knowledge from Experience	73
5.7.1	Identifying PDDL Actions from Experience	74
5.8	Automated Experiments for Data Collection	76
5.8.1	Requirements for Experiment Scenarios	77
5.8.2	AutoExperimenter: A tool for generating meaningful data	77
5.8.3	Adaptation to other Scenarios	80
5.9	Summary	80

Chapter 6	Embodiment of Autonomous Robot Control Programs	83
6.1	Failure Handling and Recovery in the Real World	83
6.1.1	Exhaustingly Repeating Actions as most Naive Handling Strategy	86
6.1.2	Global Failure Taxonomy	87
6.1.3	Uncertainty for Mobile Robots in the Real World	88
6.2	Raw Interfaces to the External World	90
6.2.1	ROBOSHERLOCK: Semantic Perception	91
6.2.2	MoveIt!: Abstract, Constrained Motion Planning	91
6.3	Translating Symbolic and Subsymbolic Information	93
6.3.1	Perception Queries and Results	93
6.3.2	Motion Planning Queries and Results	93
6.4	Maintaining a Dynamic Planning Scene for Manipulation	96
6.5	Navigation in Semantically Known Environments	97
6.6	Summary	98
Chapter 7	Evaluation	99
7.1	Experiment Scenario	99
7.1.1	Experimental Data	101
7.2	The Role of Generalized Plans	102
7.3	Insights about what did not work	105
7.4	Automated Experiments	107
7.4.1	Simulated vs. Real Experiments	108
Chapter 8	Conclusion	109
8.1	The Need for Abstract Activity Descriptions	109
8.1.1	Human Household Scenarios	109
8.2	Summary of the Approach	109
8.2.1	Discussion	110
8.3	Future Research	111
8.3.1	Significant Amounts of Episodic Memories on Real Robots	111
8.3.2	Plan Verification for Generalized Plans	111
8.3.3	Generalizing over Multiple Domains	111
8.3.4	Experience Transfer between Different Robots	112

8.3.5	New Machine Learning Methods	112
Appendix A	Research Platforms	113
A.1	The Willow Garage PR2: A Capable Mobile Manipulator	113
A.2	Gazebo: Simulation-based Manipulation	114
Appendix B	Plans	115
	Keyword Index	117
	References	119

List of Tables

1.1	Preferences of individual meal participants	5
1.2	Meal object characteristics	5
3.1	Components contained in an Episodic Memory	20
4.1	Atomic action primitives for mobile manipulation	35
4.2	Common types of static and dynamic information	45
4.3	Action primitives for re-arranging shopping rack	47
4.4	Action weights for re-arranging shopping rack	48
5.1	Most prominent SEMREC RPC calls	56
5.2	Example training data for decision trees	61
5.3	Implemented Plan Scoring Methods	69
6.1	Dimensions for categorizing robot plan failures	86
6.2	Symbolic and subsymbolic uncertainty types	89
7.1	Resulting set of fetch and place tasks	101
7.2	Numerical analysis of 425 experiments	103
7.3	Ignoring flow control operators helps generalizing experience models	106

List of Figures

1.1	A PR2 robot picks items from a refrigerator	2
1.2	Precise environment knowledge is an enabler for reasoning-intensive tasks	2
1.3	Example situation from table setting experiments	3
1.4	Impression of the experiment kitchen environment and area map	4
1.5	Table seat structure	5
1.6	Decision models improve over time through variation	8
1.7	Example analyses from aggregated Episodic Memories	9
1.8	Object picking scenes: From drawer, fridge, full table	9
3.1	Architecture model separated by main entities	20
3.2	Governing architecture of the presented topics	21
4.1	Trade-off between specialized and Generalized Plans	24
4.2	Action Description vs. Goal Definition	26
4.3	Schematics of hierarchical, modular plan elements	29
4.4	Step-wise generalization of action descriptions	30
4.5	Designator driven grasp strategy decision	30
4.6	BT representation of simple plan	31
4.7	Individual CRAM language elements represented as BT constructs	34
4.8	Common tasks for mobile manipulation robots	36
4.9	Opening containers under space constraints	38
4.10	Decision flow of grasping an object	45
4.11	Architecture overview of modified A* planner	46
4.12	Planning representation for shopping rack re-arrangement	47
5.1	Task tree for “serve milk” and fitting concept ontology	53
5.2	Example task tree visualization of recorded experience	55
5.3	Structure and Content of a single EM instance	58
5.4	The architecture of a robot’s prediction capabilities, based on ExMods. This figure highlights their distinct role in the overall architecture.	59
5.5	Combining different EM task trees into a compound model	60
5.6	Virtual branches for unknown task types in ExMods	64
5.7	Architecture embedding Prototypical Experiences	66
5.8	PE of a fetch and place task	66
5.9	Example of PEs based strategy generation	68
5.10	Calculation of confidence intervals for expected task time consumption	69
5.11	Processing pipeline for Multi-modal Analysis of EMs	70
5.12	Probability distributions for successful grasping learned from EMs	73
5.13	Architecture overview for cascaded planning and learning from experience	74
5.14	Task tree of a robot experience	76
6.1	Failure Taxonomy	87
6.2	Object information encoded in KNOWROB	94
6.3	Relevant coordinate systems for grasping an object	95

7.1	Problem inference for Mary as the only guest on a Wednesday breakfast	100
7.2	Selection of final experiment states	101
7.3	Reiteration of Figure 5.12	102
7.4	Composition of hierarchical generalized plans	104
7.5	Typical phases during table setting	104
A.1	The Willow Garage PR2 Robot and one of its grippers	113
A.2	Different manipulation tasks as performed by a PR2 in the Gazebo simulator . . .	114
B.1	Behavior Tree version of a simple CRAM grasping plan	115
B.2	Behavior Tree version of a monitoring plan	115
B.3	Behavior Tree version of a search object plan	116
B.4	Behavior Tree version of a fetch object plan	116
B.5	Behavior Tree version of a simple object grasping plan	116

List of Algorithms

5.1	Plan definition macro that adds logging capabilities	57
5.2	Simple plan for annotating RTPs during a decision making process	60
5.3	Task Tree Generalization	61
5.4	Decision tree from Episodic Memories	62
5.5	Example plan using the choose operator	63
5.6	Computing Task Result Probabilities	64
5.7	Inverted decision tree for choosing parameters using the choose operator	65
5.8	Recursively creating Prototypical Experiences	67
6.9	Example CRAM grasping code	84
6.10	Concurrently monitored object transport task	85
7.11	Problem generator for table setting tasks	100

List of Acronyms

ADL Action Description Language	OMPL The Open Motion Planning Library
AE AutoExperimenter	OWA Open World Assumption
AI Artificial Intelligence	OWL Web Ontology Language
AMCL Adaptive Monte Carlo Localization	PDDL Planning Domain Definition Language
AR Augmented Reality	PDM Point Distribution Model
AUV Autonomous Underwater Vehicle	PM Process Module
BDI Belief-Desire-Intention	PTU Pan/Tilt Unit
BT Behaviour Tree	PE Prototypical Experience
CAD Computer Aided Design	RANSAC Random Sample Consensus
CRAM Cognitive Robot Abstract Machine	RAP Reactive Action Package
CV Computer Vision	REP Reduced Error Pruning
CWA Closed World Assumption	ROS Robot Operating System
DOF Degree of Freedom	RPC Remote Procedure Calling
DSL Domain Specific Language	RTP Relevant Task Parameter
EM Episodic Memory	SBCL Steelbank Common Lisp
ExMod Expectation Model	SEMREC Semantic Hierarchy Recorder
FSM Finite State Machine	SIFT Scale-invariant Feature Transform
GPS General Problem Solver	STRIPS STanford Research Institute Problem Solver
HTN Hierarchical Task Network	SVM Support Vector Machine
IK Inverse Kinematics	TCP Tool Center Point
JSON JavaScript Object Notation	TF The Transform Library
MMVG Mixed Multivariate Gaussian Distribution	UAV Unmanned Autonomous Vehicle
MVG Multivariate Gaussian Distribution	UIMA Unstructured Information Management Architecture
NPC Non-Player Character	YAML Yet Another Markup Language
OIR Object Identity Resolution	

Abstract

Creating intelligent artificial systems, and in particular robots, that improve themselves just like humans do is one of the most ambitious goals in robotics and machine learning. The concept of robot experience exists for some time now, but has up to now not fully found its way into autonomous robots.

This thesis is devoted to both, analyzing the underlying requirements for enabling robot learning from experience and actually implementing it on real robot hardware. For effective robot learning from experience I present and discuss three main requirements:

- (o) *Clearly expressing what a robot should do, on a vague, abstract level*
I introduce Generalized Plans as a means to express the intention rather than the actual action sequence of a task, removing as much task specific knowledge as possible.
- (o) *Defining, collecting, and analyzing robot experiences to enable robots to improve*
I present Episodic Memories as a container for all collected robot experiences for any arbitrary task and create sophisticated action (effect) prediction models from them, allowing robots to make better decisions.
- (o) *Properly abstracting from reality and dealing with failures in the domain they occurred in*
I propose failure handling strategies, a failure taxonomy extensible through experience, and discuss the relationship between symbolic/discrete and subsymbolic/continuous systems in terms of robot plans interacting with real world sensors and actuators.

I concentrate on the domain of human-scale robot activities, specifically on doing household chores. Tasks in this domain offer many repeating patterns and are ideal candidates for abstracting, encapsulating, and modularizing robot plans into a more general form. This way, very similar plan structures are transformed into parameters that change the behavior of the robot while performing the task, making the plans more flexible.

While performing tasks, robots encounter the same or similar situations over and over again. Albeit humans are able to benefit from this and improve at what they do, robots in general lack this ability. This thesis presents techniques for collecting and making robot experiences accessible to robots and outside observers alike, answering high level questions such as “*What are good spots to stand at for grasping objects from the fridge?*” or “*Which objects are especially difficult to grasp with two hands while they are in the oven?*”. By structuring and tapping into a robot’s memory, it can make more informed decisions that are not based on manually encoded information, but self-improved behavior. To this end, I present several experience-based approaches to improve a robot’s autonomous decisions, such as parameter choices, during execution time.

Robots that interact with the real world are bound to deal with unexpected events and must properly react to failures of any kind of action. I present an extensible failure model that suits the structure of Generalized Plans and Episodic Memories and make clear how each module should deal with their own failures rather than directly handing them up to a governing cognitive architecture. In addition, I make a distinction between discrete parametrizations of Generalized Plans and continuous low level components, and how to translate between the two.

Zusammenfassung

Künstliche intelligente Systeme zu erschaffen, speziell Roboter, die sich nach dem Vorbild von Menschen selbst verbessern ist eines der ambitioniertesten Ziele in sowohl der Robotik als auch dem Maschinellen Lernen. Das Konzept von Robotererfahrungen existiert zwar schon seit einiger Zeit, hat seinen Weg bisher aber noch nicht vollständig in die autonome Robotik gefunden.

Diese Arbeit ist beiden Aspekten gewidmet: Der Analyse der unterliegenden Anforderungen, um Robotern das Lernen aus Erfahrung zu ermöglichen und der tatsächlichen Umsetzung auf echter Roboter-Hardware. Für tatsächliches Roboter-Lernen aus Erfahrungen präsentiere und diskutiere ich drei Hauptpunkte:

- (o) *Eine klare Darstellung was ein Roboter tun soll, vage und abstrakt beschrieben*
Ich stelle Generalisierte Pläne (Generalized Plans) vor, die zum Ausdruck der Absichten eines Roboters dienen und weniger der tatsächlichen Aktionsabfolge, um sie zu erreichen. Es wird so viel aufgabenspezifisches Wissen wie möglich entfernt.
- (o) *Definieren, sammeln und analysieren von Robotererfahrungen, um Robotern zu ermöglichen, sich zu verbessern*
Ich präsentiere Episodische Erinnerungen (Episodic Memories) als Aggregat für alle gesammelten Robotererfahrungen zu einer Aufgabenstellung. Basierend darauf erstelle ich ausgeklügelte Prädiktionsmodelle, die einen Roboter bessere Entscheidungen treffen lassen.
- (o) *Abstraktion von der Realität und der Umgang mit Fehlern in der Domäne, in der sie aufgekommen sind*
Ich stelle Fehlerbehandlungsstrategien auf, sowie eine Fehlertaxonomie, die durch Erfahrungen erweitert werden kann. Ich diskutiere außerdem den Zusammenhang zwischen symbolischen/diskreten und subsymbolischen/kontinuierlichen Systemen in Bezug auf Roboterpläne, die mit realweltlichen Sensoren und Aktuatoren interagieren.

Ich konzentriere mich auf die Domäne von menschnahen Roboteraktivitäten, speziell Haushaltssarbeiten. Aufgaben in diesem Bereich eröffnen viele sich wiederholende Muster und sind ideale Kandidaten für Abstraktion, Kapselung und Modularisierung von Roboterplänen in eine allgemeinere Form. Auf diesem Wege werden sehr ähnliche Planstrukturen in eine flexible Parameterform gebracht, die das Verhalten des Roboters während der Ausführung verändern kann.

Während der Planausführung begegnen Roboter immer wieder der gleichen oder ähnlichen Situationen. Obwohl Menschen in der Lage sind, davon zu profitieren und sich in dem verbessern, was sie tun, fehlt Robotern im Allgemeinen diese Fähigkeit. Diese Arbeit stellt Techniken vor, um Robotererfahrungen zu sammeln und für sowohl Roboter als auch externe Beobachter zugänglich zu machen. Es können Fragen beantwortet werden wie *“Von wo aus lassen sich Gegenstände gut aus dem Kühlschrank nehmen?”* oder *“Welche Gegenstände sind besonders schwierig mit zwei Händen zu greifen, wenn sie im Ofen sind?”*. Durch Strukturierung und Zugänglichmachen des Gedächtnisses von Robotern können diese besser fundierte Entscheidungen treffen, die nicht rein auf manuell eingebundenem Wissen basieren, sondern auf selbstverbessertem Verhalten. Zu diesem Zweck präsentiere ich mehrere Ansätze zur erfahrungsbasierten Verbesserung von Entscheidungsprozessen in autonomen Robotern, wie z.B. Parameterwahlen, während der Planausführung.

Roboter, die mit der realen Welt interagieren, müssen mit unerwarteten Ereignissen rechnen und geeignet auf Fehler jeglicher Art reagieren können. Ich präsentiere ein erweiterbares Fehlermodell, das der Struktur von Generalisierten Plänen und Episodischen Erinnerungen gerecht wird und stelle klar, wie einzelne Module ihre eigenen Fehler behandeln sollten, bevor sie diese an eine nächst höhere Instanz weitergeben. Zusätzlich unterscheide ich zwischen einer diskreten Parametrisierung von Generalisierten Plänen und kontinuierlichen, niedrigstufigeren Komponenten, und wie zwischen beiden übersetzt wird.

Introduction

“The key to good decision making is not knowledge. It is understanding. We are swimming in the former. We are desperately lacking in the latter.”

— Malcolm Gladwell [39]

This thesis is dedicated to making autonomous robots able to robustly and efficiently perform vaguely described tasks, and to learn from their experience to improve task success rates. To this end, three major components are presented, discussed, and evaluated: The design and representation of *Generalized Robot Action Plans* that very vaguely formulate what a task should achieve. A framework for recording, semantically defining, and accessing robot experiences, or *Robot Episodic Memories* from which vague Generalized Plans are improved and high level questions about the tasks performed are answered. Episodic Memories represent all details of a task, and allow reconstruction of every step the robot took. *Embodiment* mechanisms that execute vaguely described tasks in the real world with the help of transparently added knowledge, dealing with the (un-)foreseen failures encountered in mobile manipulation scenarios. Within the context of Embodiment, another cross-functional topic is presented: Identifying known failures from a failure taxonomy, or extending the taxonomy if new failures are encountered.

I discuss both, the conceptual groundwork and the actual implementation of all components. With the results of my research, robots have access to prediction mechanisms driven by decision trees or probability distributions learned from their experience. Their Generalized Plans can automatically adapt to whether a parametrization was a good or a bad fit in the past, and can on the fly choose a new strategy that fits the current situation. Robots that perform actions using my Generalized Plans cover a wide range of situations as the plans describe the task’s intention rather than an action sequence.

All examples are built around the topic of household robots in human-scale environments, and center at the task of autonomously setting a meal table. After reading this thesis, the reader will know what generality in robot action plans is, why generalized plans are superior to specialized ones when learning is involved, and why this generality must be properly translated when dealing with the real world by connecting vital low-level skills such as motion control and perception. Furthermore, the reader will have a firm grasp on what Robot Episodic Memories are, what information they hold, what kinds of questions they can answer, and what type of models can be generated from them to improve robot behavior and action success rates.

1.1 Motivation

In my research, I focus on the area of today’s service robots. More precisely, I develop high level control mechanisms for robots helping in the household, allowing them to get better over time using experience. Figure 1.1 shows the robot which I use for my research — the Willow Garage PR2 — while it is collecting items from a refrigerator. It also shows the same robot in simulation, as I validate the functionality of all my robot action plans in simulation first. The latter is also a great source for training data in machine learning, which I use to improve the performance of robot plans.

In Figure 1.2 a situation from a supermarket shelf is shown. Items occlude each other, and there is more than one of the items needed. Choosing the right one and moving others out of the way requires knowledge, which my plans transparently add to whatever the robot is supposed to do. The necessary low level functionality is also handled transparently, so the action plan really only describes what the robot should do, not how to do it exactly.

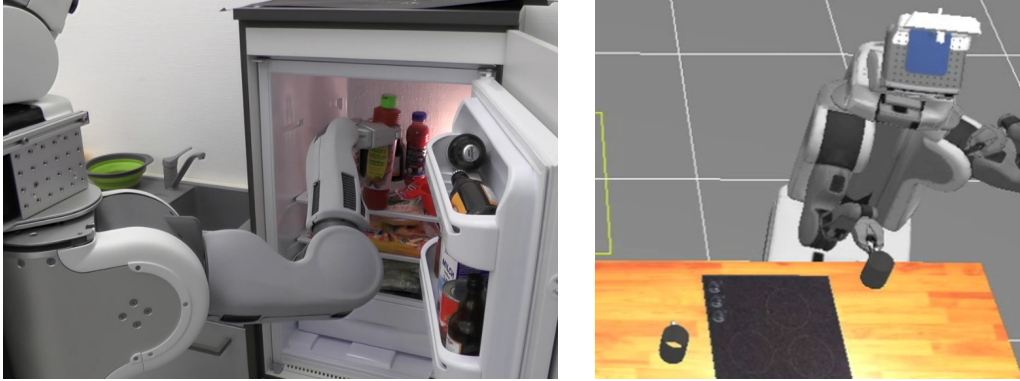


Figure 1.1: A PR2 robot picks items from a refrigerator. This requires knowledge about the items to pick, the surroundings, the other items, and applicable motion constraints from the task description and inherent task requirements (e.g. keep the bottle upright because its lid is open). Also, all plans are executable in a simulated environment; next to testing, this is a great source for machine learning data.



Figure 1.2: Precise environment knowledge is an enabler for reasoning-intensive plans. It does not only store and infer correct object identities and poses, but also discovers multiple instances of the item to pick — triggering decision processes.

All of my approaches are based on the archetype of human activities and how they improve: Humans have the ability to learn from everyday tasks, and become very efficient at doing them. I take this ideal and mix three elements to enable robots to do the same: Complex and rich manipulation activities, longterm experiments to collect a large amount of experience from these activities, and learning methods to produce meaningful, plan improving data from these experiences. In human everyday activities, I exploit strong regularities: Shared actions between different tasks, or repeatedly doing the same task with slight variations every time. Humans are able to solve tasks very elegantly after doing them a number of times, even for very complex actions. My research motivation thus is: How can this be achieved in robots? My research is based on this sole question, which also defines my overall goal: Robots that do the same household activity for two weeks should become much better at doing it afterwards. By learning the ins and outs of the task, the robot becomes more flexible and robust through learning, and effectively programs itself according to the tasks needs.

1.2 Research Scenario

As an experimental platform for describing my concepts and showcasing a possible implementation, I chose the activity of setting a meal table. The scenario's environment, a kitchen, is quasi-static and offers a wide variety of manipulation tasks for robots: Finding, picking and placing human-scale objects, opening and closing doors and drawers, and navigating around obstacles usually found in human homes.

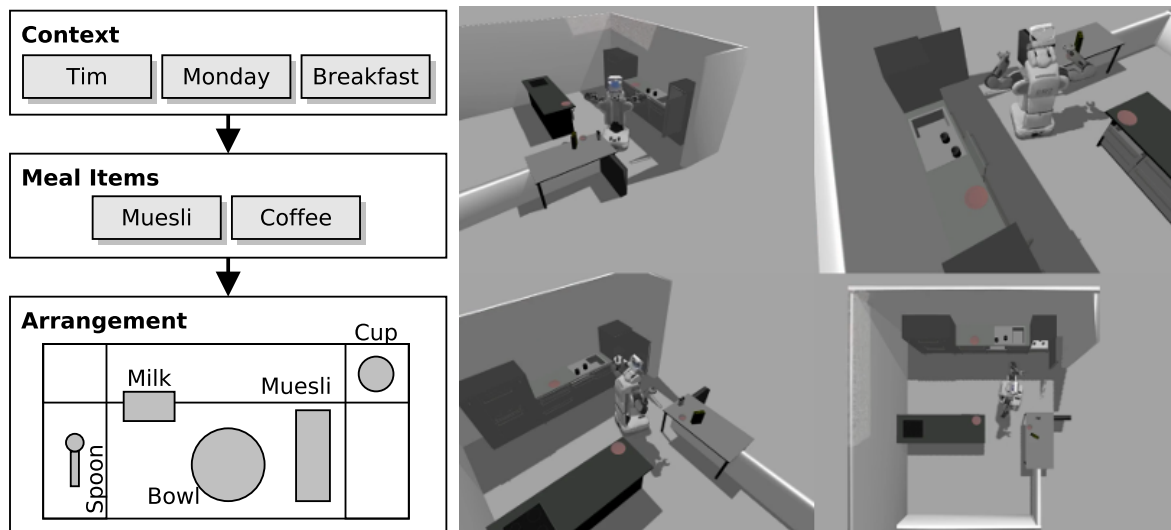


Figure 1.3: Example situation from table setting experiments conducted in simulation. The overall context is that Tim wants to have breakfast on a Monday. From background knowledge, the arrangement shown above is inferred (left). The respective tasks are executed in simulation (right).

Figure 1.3 depicts an example situation from the experiments I conducted in simulation. Figure 1.4 shows an impression of the real laboratory kitchen environment used for experiments and a top-down view of the area’s map, including some of the relevant semantically annotated furniture. The environment holds a number of challenging details: (1) The accessible floor area is concave, making 2D navigation more difficult. (2) The available containers feature horizontal (fridge) and vertical (dishwasher) doors, as well as prismatic drawer joints. Each of the three requires a separate strategy for opening and closing it. (3) The three countertops (sink area, kitchen island, meal table) can be used for setting a table. They have different heights, and an autonomous robot needs to be able to set a meal table on any of them.

The task of setting a meal table itself bears two main challenges: Acquiring the objects, and actually arranging them on the table. Acquiring the objects includes searching for them at hypothetical whereabouts, opening and closing containers on the way as necessary, and moving obstructing objects out of the way. Arranging them includes taking into account the preferences of the meal participants for whom the table is set, and finding suitable, free regions for putting down the objects in an efficient way.

The table setting (and subsequent tidying up) task relies on a very vague task description, filling in any missing information from static background knowledge, contextual hints, or experience. In the example scenario I constructed, the initial task parametrization is generated by a task sampler that resolves a very vague context description into a concrete series of tasks an autonomous robot can perform. The task sampler produces these based on who takes part in the meal, whether it is a weekday or the weekend, and the type of the meal (breakfast, lunch, dinner). Tables 1.1a and 1.1b depict the ground truth of the task sampler. Figure 1.5 shows what a typical meal seat looks like, showing its different regions. Table 1.2 defines which part of tableware goes where in that seat structure.

I chose the meal table setting scenario as it

- (o) Features rich variations of repetitive tasks: Fetching objects with very different handling requirements from various places in a kitchen and placing them at appropriate places on a table. The task’s structure is very clear and encapsulated, making it a prime candidate for generalization.
- (o) Does not change structurally, but only from a parametrization point of view: This property

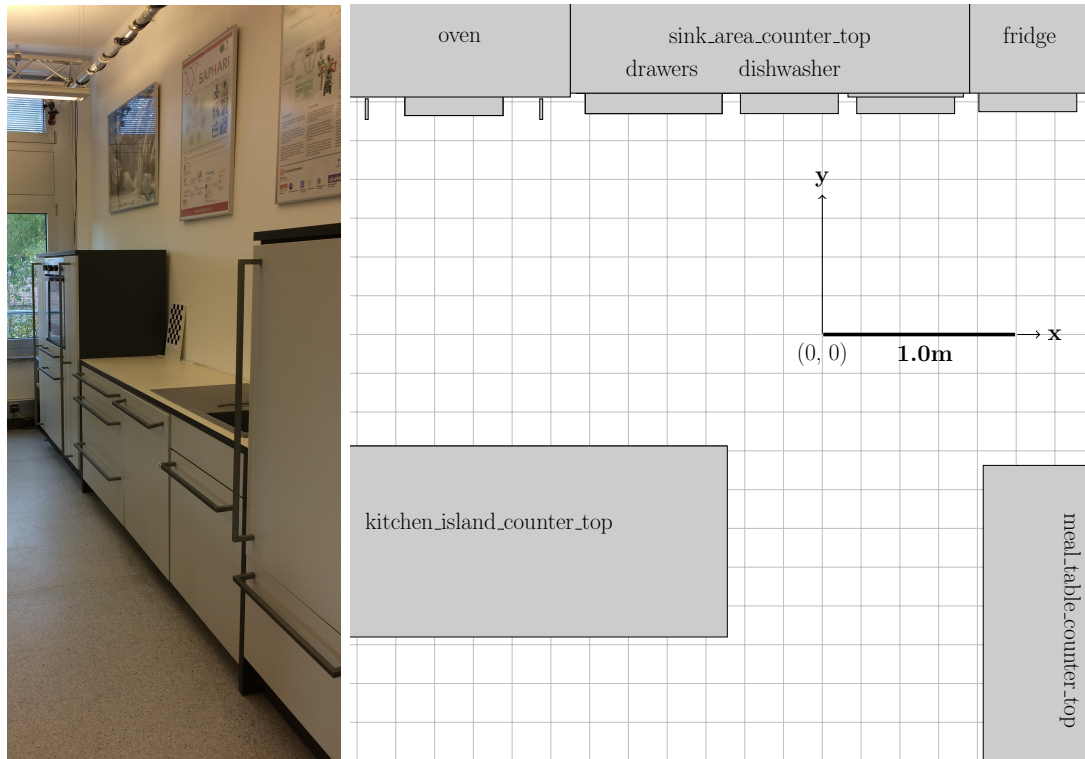


Figure 1.4: Impression of the experiment kitchen environment (left) and map of the relevant area, including furniture (right).

makes it an ideal subject for learning from experience. Predicting success of strategies and parameters for parts of the task, using experience data and the current context, are perfectly suited for this kind of scenario. Fetch and place can be modeled as an open-ended, repetitive scenario and can thus be used for longterm robot experiments.

- (o) Requires proper translation of abstract plans in real actions, based on the environment's and the task's needs: Although not explicitly described in its plans, a robot has to properly execute every abstract action on real objects in a real environment.

1.3 The Challenge

To position the research presented in this thesis, in this section I discuss what the challenges of my research area are, why they matter, and why they have not been solved before. To put the challenges in perspective, I give a number of concrete examples of situations that are very difficult or even impossible to solve for current robot control programs without additional help or information.

1.3.1 What is the Challenge?

The main challenge I address in my research is the execution of robot manipulation activities of natural complexity and variability, in particular fetch and place tasks in human household environments. At the same time, this execution must result in the collection of experience about the task — in all its detail. The most difficult aspect of this topic is to properly translate very vague task descriptions in actionable items a robot can perform in the real world — flexibly, robustly, and efficiently. These activities must be executed over long periods of time in potentially unknown environments. This should result in representative collections of experiences that properly reflect what a robot did, when it failed, and why.

Person	Day	Time	Meal	Meal	Required Objects
Tim	Workday	Breakfast	Muesli, Coffee	Muesli	Bowl
		Lunch	Eats out		Spoon
		Dinner	Bread		Muesli Box
	Weekend	Breakfast	Muesli, Coffee	Milk Box	
		Lunch	Fish	Bowl	
		Dinner	Bread	Spoon	
Mary	Workday	Breakfast	Bread, Coffee	Coffee	Cup
		Lunch	Soup	Tea	Cup
		Dinner	Bread	Bread	Plate
	Weekend	Breakfast	Muesli, Tea	Butter Knife	
		Lunch	Fish	Plate	
		Dinner	Bread	Fish Knife	

(a) Participant Meal Preferences

(b) Meal Tableware Requirements

Table 1.1: Preferences of meal participants and required objects for serving individual meal types. The preferences change based on the day of the week and on the time of the day.

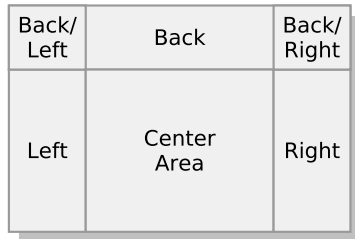


Figure 1.5: Named areas relative to a meal participant’s seat, divided semantically: Left, right, front, back, and combinations thereof.

Object	Place
Bowl	Center
Spoon	Left
Cup	Right, Back
Plate	Center
Butter Knife	Left
Fish Knife	Left

Table 1.2: Objects used for setting a meal table. They have designated places where they need to be placed relative to the participant’s seat. Unmentioned objects can go anywhere free.

A robot needs a large deal of knowledge to decide how to act robustly and efficiently: How to identify an object, how to grasp it properly, what to take into account when putting it down again, and how to react if any of these steps fail. The knowledge gap for robots starts at a very basic level: Where to stand in order to reach an object, which part of the object to grasp, and how much space is necessary when putting it down again. Neither exists a common database of these details, nor are they so obvious that a robot can infer them on the fly, in general making informed decisions from static knowledge included in a robot action plan practically impossible.

Humans easily solve a large number of very complex tasks. We do not only do this when everything works, but can on the fly deal with a large amount of variations and problems that arise. The questions that surface now are: How can robots be enabled to do the same? What are the steps a robot needs to take to make everything alright, despite unexpected events? While the former is the larger research question covered in this work, the latter can be exemplified well using very basic (mobile) manipulation skills:

- (o) *Put a plate on the table*: Fetching a plate from one place and placing it on a certain table
- Find out where plates are: What is the environment map of where objects (or object classes) are located; if no plates can be found, a suitable substitute must be identified
 - Find a way to get to that place: Which path to take, how to evade obstacles
 - Make the plate accessible: How to open any closed containers that (might, if no specific location is known) contain a plate (requires knowledge about handles and

- joints, and how to use them); what to do if the plate is obstructed by other objects
- Identify the plate: Which plate matches the context (dinner plate, dessert plate); requires knowledge about what plates look like and what purpose they serve
- Pick the plate: Where to stand in order to reach the plate; which arm to use; what parts of the plate are graspable; what is the collision environment of the containing furniture; what is the model of the plate
- Transport the plate to the destination: How to hold the plate along the way; which instance of “table” must the plate be placed on; how to get to that table
- Placing the plate on the table: What to do if there is no space on the table; where exactly must the plate be placed

(o) *Serve a glass of water*: Pouring water from a container into a glass

- Position yourself to reach both, the source container and the target glass: What is the reachability of the robot’s arms; which arm is used for pouring; where to stand
- Grasping the source container: Where is the object graspable; what parts are the opening used to pour from
- Determine how much to pour: How much can the target container hold; how much is in the source container; what is a appropriate amount to serve
- Pouring the liquid: Where to position the source container; what angle to use to pour; what are the dynamics of water while pouring; when to stop tilting and thus stop pouring

Both examples show: A robot needs a fair deal of knowledge that depends on both, the context, the expectations of the person ordering the task, and the circumstances in which the task is done. For a robot to make everything alright when ordered to perform a task, static plans and a limited static knowledge base does not suffice — and if unexpected interruptions and failures occur, even more knowledge, robustness and flexibility are required.

To implement the above characteristics and overcome the current challenges described, a major requirement is a robot plan language with a suitable level of expressiveness. This language must be able to cover all three areas: abstract descriptions of tasks, meta programming capabilities for transparently augmenting plans with experience collection and usage mechanisms, and a mature failure propagation and handling mechanism.

1.3.2 Why is it interesting? Why is it important?

Fetch and place activities are a very important topic in robotics, as robust and efficient fetch and place skills allow a robot to abstract away from the actual environment it operates in. They are used as building blocks for more complex behavior — thus it is this knowledge intensive abstraction that makes them valueable, but also very difficult to perform in real world settings. At the same time, they are a prime candidate for experience based learning, as their structure is mostly fixed, and their parametrization depends on factors shared between use-cases: What is the object to be grasped, where is it, how should it be grasped, and what are the relevant obstacles in the environment.

With the above problem solved, robots would only need a very crude description of how to perform a task. They would then repeat it over and over again, in a multitude of variable situations, and collect lots of experience on the way. They would then get better at each and every try, and eventually master the task. With their collected experiences, they could even explain under which circumstances a task is feasible and make predictions about their own chances for success — which in turn speeds up tasks and reduces fatal failures. Robots would then be able to act more flexibly, robustly, and elegantly, and get closer to the expectations humans have of their actions.

1.3.3 Why is it a hard problem? Why has it not been solved yet?

The more generic and abstract a task becomes, the less concrete information about how to do it is known per se. A generic object fetch activity would contain just enough information to identify what to do — and in no way how. An autonomous robot needs to add all extra information for grounding the vague task in its current environment, and for figuring out how to execute the single steps it needs to take. Thus for a robot “*fetch me a glass of water*” is much more difficult than “*pick an empty glass from the kitchen cabinet, fill it with water from the tap, and bring it to me*”. These vague descriptions can easily lead to ambiguities that the robot needs to resolve in a trial-and-error fashion. Robots in general lack the ability to draw from experience to foresee when a decision leads to bad results. Even when the right decision was made, unforeseen events (dropping an object, knocking it over, a path being blocked, having no space to put down an object) can interrupt the robot’s activity. The most difficult aspects in this problem are:

- (o) The proper definition of vague task descriptions, leaving out all task and domain specific knowledge, and the transparent addition of necessary knowledge when executing them
- (o) Collecting and analyzing robot experiences over long periods of time, and improving a robot’s decision-making process through them to increase the overall task success rate, making it able to semantically answer high level questions about what it did, why it did it, and what went wrong
- (o) Properly detecting and handling foreseen and unforeseen failures in the real world, and adapting the task strategy accordingly

These points pose a number of research challenges in themselves, which is why they have not been solved before. Robot programming languages oftentimes directly connect the plan and the execution view of a task, resulting in rather specialized plans than does not generalize well. Based on this, collected robot experiences map onto the concrete, specialized plans — and not on the more generic, abstract intention of the task. The missing separation between the different layers also leads to failure detection and handling being a direct part of the plan itself: Even low level failures or failures in sub plans (like fetching a glass when setting a table → no glass was found in the cabinet) are handled by the specialized plan version. This again breaks the separation between abstract failures and concrete problems in the environment the plan is embodied in by the robot.

1.4 Approach

In this thesis I present my research on the design of a smart, robust plan- and experience-based robot control architecture that allows longterm autonomy and experience-based learning. To this end, I lay the conceptual groundwork for static knowledge-based robot plans and equip them with extensive failure-handling capabilities and task-knowledge. Robots that perform tasks using my plans collect Episodic Memories of their doings. These memories — or experiences — enable the robots to predict what will happen next, to choose proper parametrizations for their actions in order to succeed, and to plan ahead which steps are helping them to achieve their goals. From experience, they can semantically answer abstract questions such as “*what did you do*”, “*why did you do it*”, and “*what went wrong*”. Ultimately they improve at what they do by continuing to do it, through experience and an evolving intuition. This process is shown in Figure 1.6: Based on what went well and what did not work, a robot forms and improves models predicting its own success, learning from its failures and preventing future ones.

I define robot plan language elements for describing tasks as abstractly as possible, and move the actual embodiment and handling of real world failures into a completely separate conceptual layer. This way, my plans convey only their actual intentions, as free as possible from implementation, hardware platform, and environment details.

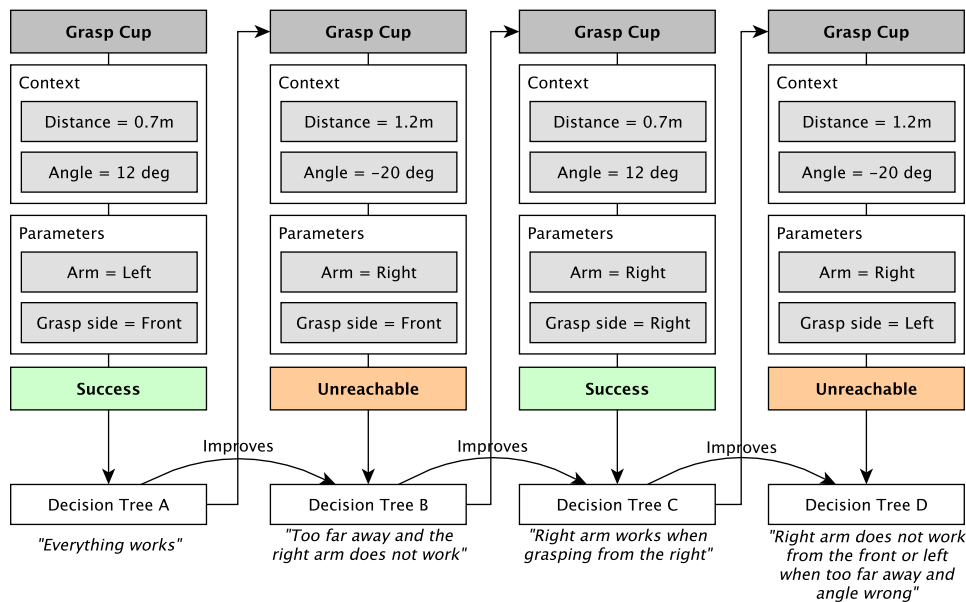


Figure 1.6: Through experiencing different variations of the same task, the decision models of the executing robot improve over time.

I must put special emphasize on the fact that task intelligence (or competence) does not come from nowhere. We build and program robots for purposes manifested in our own needs, namely we want them to do tasks for us exactly as we expect each other to do them, or better. Thus, in this thesis I does not let robots bootstrap core abilities or well-understood task knowledge. A robot's behavior is based on a solid, manually written plan that acts as a heuristic prior — and with experience, the robot gets better at it and changes the plan's parameters, adapting to the task's needs, and mastering the task itself. That said, a primitive heuristic starter plan does not nearly cover all necessary situations. In my experiments, common problems for the uninformed robot were:

- (o) Not knowing where objects usually are located: The task took very long and had a rather high probability of failure as all locations had to be searched in an arbitrary order.
- (o) Not knowing where to stand in order to properly grasp an object: Without experience models that dictate good locations to stand at, the robot in my system relied on a linear gaussian distribution around the object to grasp, often repositioning itself before actually being able to reach it.
- (o) Not knowing where placing an object fails more often: Given a degree of freedom when placing an object, the robot often chose angles or positions on the table that were syntactically correct, but were situationally unreachable.

These problems were remedied when the experience of the robot grew, but even with a heuristic prior plan, bootstrapping its parametrization needs these failure experiences to actually develop expertise.

Throughout the whole thesis, I concentrate on the broad area of autonomous mobile manipulation, and more particularly on robots that perform tasks in human households. Tasks in households are well-understood and we have a definite set of expectations about what tasks are important, how they should be done, and what is contextually appropriate to achieve them. To explain my concepts in a real world scenario, I exemplarily chose the activity of setting a meal table which I already introduced in Section 1.2.

To give the reader an idea of my work's results, Figure 1.7 shows an example data analysis of aggregated travel paths of the same task, as well as the multivariate mixed gaussian distribution

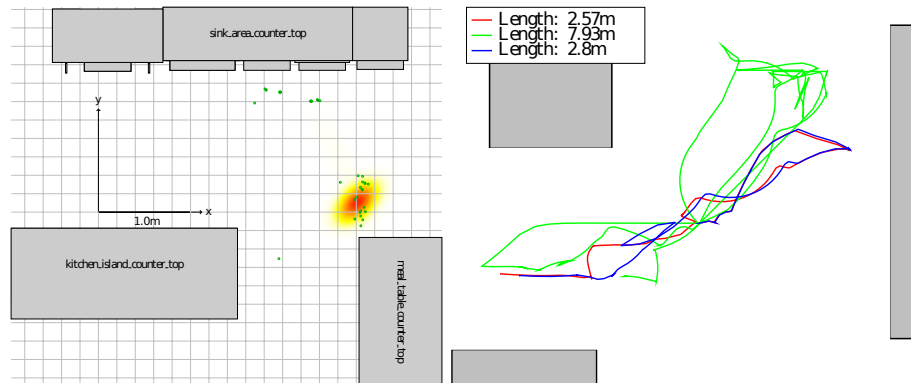


Figure 1.7: Example analyses from aggregated Episodic Memories: Aggregated travel paths during execution of the same task in multiple instances, and multivariate mixed gaussian distributions over places a robot stood at while it was grasping successfully.



Figure 1.8: Picking objects in different situations: Objects can be stored in drawers or fridges that need to be opened and that act as collision objects; places can be crowded with objects, and every detected object needs to be identified, and accounted for during collision checking.

over places a robot stood at while grasping objects. The former is used for predicting the probability of passing along other interesting areas, while the latter is used for predicting good places to stand at for grasping.

1.5 Contributions

In my research, I focus on the design of Generalized Plans that record, and make use of robot Episodic Memories (EMs) to improve their own performance [103]. To alleviate the challenges described above, I introduce design principles and language constructs for generalizing robot action plans. My considerations give them semantic meaning to allow a robot to reason about the plan’s role during its current task, and enable the ability to transparently make use of external knowledge sources and the current context as sources of information. I present my findings in Chapter 4: *Generalized Plan Design and Representation for Robots* in which I introduce requirements for plans to be considered generalized, identify and decompose common tasks in mobile manipulation, and pinpoint how generalized plans deal with vague parametrizations — especially using experience. Additionally, I describe the various knowledge sources that robots can be parametrized from, and give a case in point for planning using the identified common manipulation tasks. My main contribution in terms of Generalized Plan Design and Representation thus is:

- (o) **Design Principles for Generalized Robot Plans for supporting Modularity, Knowledge integration, and Recovery in Autonomous Behavior**
 - (o) Encapsulating and separating plans by semantic meaning (for mobile manipulation)
 - (o) Transparently accessing knowledge sources, including the task’s context

- (o) Hierarchically organizing plans for modular recovery and proper context composition

To validate my Generalized Plans, I execute the same abstract plans in two different scenarios, namely on a real PR2 in an actual kitchen environment, and in a simulated version of the kitchen with slightly different semantics. Additionally, I generate a large number of variations for the latter and let the same generic fetch and place plans set a meal table over and over again. Since the actual execution in both environments is handled by low level modules that specialize on the respective environment, the qualitative behavior based on the Generalized Plans should be the same. My experiments show that this indeed is the case, and variations only happen during parametrization of the low level skills, while showing comprehensible, expected behavior of the executed high level task.

Both, the result from and the input to Generalized Plans is data contained in Episodic Memories [105] recorded while performing robot actions. In Chapter 5: *Data Collection and Experience-Based Learning* I introduce these memories themselves and how they benefit Generalized Plans through action effect prediction and informed guessing of good task parameters. Robot Episodic Memories allow access to all past decision-making processes and their perceived effects. Aggregated, longterm collected amounts of Episodic Memories allow to generalize over the executed tasks and identify common patterns, or outliers in parametrizations that impact the task's success rate. In the designated Chapter, I go into detail about how to generate experience models for predicting the success of symbolic task parameters, and how multivariate gaussian mixture models are generated for finding suitable parameter ranges for continuous parameters. Additionally, I describe how PDDL domains can be extended automatically through learning from experience and automated classification of tasks. My main contribution in terms of data collection and behavior improvement thus is:

- (o) **Robot Behavior Improvement from Longterm Experience-based Learning using Episodic Memories**
 - (o) Acquiring Episodic Memories that fully reflect the executed task's structure and contain all decision-making processes and their perceived effects
 - (o) Interpreting Episodic Memories by semantically accessing them and answering high level, abstract questions about the task from the robot's point of view
 - (o) Learning from aggregated Episodic Memories by forming prediction models and robot facilities for informed parameter guessing to improve task success based on longterm experience collection

To validate the content of the collected Episodic Memories, I inspect the experience models generated from aggregated memories. Their content shows the provided knowledge gain. If their content (which is human-readable: decision trees with nominal classes, gaussian distributions over good places to pick up objects, etc.) reflects the intrinsics of the executed task and the environment it was performed in, the underlying experience data must be correct.

Autonomous robots must act flexibly and robustly enough to translate their vague task descriptions into proper physical actions in the real world [101]. Humans can perform marvelously complex tasks from a very abstract description — even despite unexpected interruptions, wrong information, and the plain need to find an alternative solution to what we want to achieve. To date, this is a major challenge for robots that lack both, a proper intuition and the skills needed to properly react to the real world's difficulties. In Chapter 6: *Embodiment of Autonomous Robot Control Programs* I present my approach to abstract vague plans away from actual robot actions. For the chapter's main part, I introduce and discuss two of my research topics: Failure handling and recovery for known and for unknown failures, and the implicit translation between

discrete parametrizations of plans and continuous parametrizations of sensor and actuator systems. Finally, I discuss common issues that autonomous robots have to deal with in the real world. My main contribution in terms of autonomous robots in real world scenarios thus is:

- (o) **Embodiment of Autonomous Robot Control Programs through Sensor/Actuator Abstraction, Failure Categorization, Experience-based Skill Improvement**
 - (o) Low level skill improvement through experience
 - (o) Failure categorization, handling, and recovery, supported by Episodic Memories that form a global failure taxonomy
 - (o) Translation between discrete, abstract plan parametrizations and concrete real-world sensor and actuator values

To validate the Embodiment layer of my architecture, I inspect the performance of my plans in the real world, how they decide to enact vague instructions, and how well failure handling performs. The Episodic Memories of all plan executions provide enough evidence about which reason led to which decision and what failures were handled, and how. This indeed shows that robust and flexible behavior stems at least in part from properly handling low level failures on the respective module levels.

To exemplify my contributions, I developed fetch and place plans in which an autonomous mobile robot sets a meal table. All topics, the generalized plans, the experience-based learning that improves plans, and the embodiment of said plans in the actual scenario are explained in detail in their respective chapters. Ultimately, I present vital elements of longterm fetch and place manipulation activities on a Willow Garage PR2 robot using my developed components in both, actual robot hardware (see Appendix A.1: *The Willow Garage PR2: A Capable Mobile Manipulator*), and simulation (see Appendix A.2: *Gazebo: Simulation-based Manipulation*).

1.6 Reader's Guide

This thesis is structured as follows. Chapter 2 gives an outlook of related work and already existing approaches to adjacent research problems. In Chapter 3, an overview of my overarching system architecture is presented and each of the three main component-centric chapters is introduced. Chapter 4 introduces and explains principles for generalized plan design and representation, and includes both conceptual and code examples for the ideas presented. In Chapter 5 I present and discuss the role of data collection and machine learning mechanisms when used in conjunction with robot Episodic Memories and Generalized Plans. Furthermore, in Chapter 6 I go into the importance of a dynamic failure taxonomy, interfaces to the real world from abstract generalized plans, and how to translate data between both view points. In Chapter 7 I evaluate my concepts based on experiments I concluded on a real PR2 robot and in simulation, and conclude my work in Chapter 8 with a brief summary of what was done and an outlook towards open research aspects.

Related Work

“If I have seen further, it is by standing on the shoulders of giants.”

— Isaac Newton [60]

To make the context of my work more clear, I added related work from a number of research fields similar, or adjacent to my own. For controlling and defining a robot’s behavior, I go into more detail about (1) Planning and Robot Architectures, (2) Behavior Definition, and (3) Autonomy.

2.1 Planning and Robot Architectures

2.1.1 Robot Plan Design and Behavior Control

The very essence of controlling a robot to perform a task is to relieve humans from the same job. Further down that road lies the obvious goal of completely relinquishing human intervention until the task is done. A very interesting line of research that pursues very similar goals to my own is the work of Lane *et al.*, with special emphasis on the Pandora project [52, 53]. They develop and implement a robot architecture that relies on a strong encapsulation of abstraction layers, and on a strong formalization of knowledge using Web Ontology Language (OWL). At the same time, they have a focus on embodiment in the real world and go to great lengths to properly detect and handle failures. Their main goal is *Persistent Autonomy* — making Autonomous Underwater Vehicles (AUVs) able to deal with the majority of problems themselves over extended periods of time. Different, though, is the domain of application (underwater maintenance vs. household chores), and the kinds of tasks pursued: Their work is strongly influenced by periods without communication in which the AUV has to work on its own, while I focus on the plan-based control part of autonomy, namely knowledge integration and plan design.

Behavior control using robot planning capabilities has been tackled with diverse architecture approaches. Following an analysis of common architectural challenges such as the context problem, failing upwards, and least commitment — the same ones discussed in this thesis — , Berger *et al.* [14] presented their *double pass architecture*, a Belief-Desire-Intention (BDI) inspired framework that specializes on highly dynamic environments. Their proposed solution to the common challenges is to abolish the strictly layered hierarchy of plans and let the (planned) Intention part of BDI act as an action initiator, effectively decoupling the deliberate and reactive parts. This approach greatly increases a plan’s performance in environments with high dynamics, at the cost of a strict task tree structure.

Behavior design for autonomous robots has been approached with various means. Popular examples are the Robot Task Commander (RTC) by Hart *et al.* [40] created for the NASA-JSC robots Robonaut-2 and Valkyrie. RTC is effectively a mixture of a framework for creating behavior *“building blocks”* (*process nodes*) and an IDE for building state machines out of these blocks, and executing them. Representing behavior as state machines makes it relatively easy to understand, predict, and debug, but lacks features such as using failure escalation to enable local failure handling, or the notion of *“contextual parametrization”*.

Another engine for describing robot behavior was introduced by Niemüller *et al.* [62] to control the humanoid robot Nao. They explicitly position themselves in a *“middle layer”* between the deliberate top level control and the bottom layer of hardware interfaces. Using hybrid state machines (HSMs), they account for a strong hierarchical task structure, relying on layered

execution and monitoring. For them, triggered behaviors are "*skills*", and any behavior cannot call other behaviors above its own abstraction layer.

A major bias towards one specific activity in household robots' prospective has been found and studied by Taipalus *et al.* [91]. According to their study, "*[...] fetching of objects from table-tops or shelves in home environment is one of the key functionalities for a mobile manipulator type of home robot.*". At the same time, they find that the main challenges are "*[...] that the environment is dynamic, poorly defined and some times even unknown.*". Both claims are well in line with the argumentation and results of this thesis. Their proposed solution to these challenges is rather different, though: They implement a remote control framework that allows a user to control a mobile manipulator's fundamental skills (navigation, grasping objects, etc.). They skip the top level autonomy and replace it by a human that takes care of all "*planning*" work.

2.1.2 Classical Planning Approaches

Commonly, classical planning approaches allow the definition of a problem in an axiomatic description language. This description is transformed into a search problem, allowing a sufficiently efficient search algorithm to find a (minimal) path from a given initial state to a given goal state. From any state to any other state, a (possibly empty) set of transition operators exists that allow moving from one state to the other, resulting in a graph search problem. In my work, I do not do planning explicitly using a first order planning language, but implicitly through automated selection of the next, most appropriate action from a number of actions available at that time (e.g. when accessing a location, open it if it is a drawer or fridge in the most appropriate way). This selection is based on static rules in a knowledge base, contextual hints, and prior knowledge learned from experience.

Currently, the most common abstract framework for describing planning problems is the Planning Domain Definition Language [58]. While PDDL itself is not a planner, it is the base for implementations of the STanford Research Institute Problem Solver (STRIPS) planner [31], and its spiritual successor, the Action Description Language (ADL) planner [68]. A significant difference between them is that STRIPS is strictly constrained by the Closed World Assumption (CWA), while ADL allows an Open World Assumption (OWA). Both planners produce a complete path from start to goal state based on known state transitions, planned symbolically. Given the very high degree of uncertainty and strong subsymbolic character of decisions my plans need to make, a symbolic planner's state space suffers from intractability when applied to them. Thus, my approaches structurally need to fall into a second category besides the PDDL based, analytic planners: A heuristic action selection.

Even when — in a sufficiently simple scenario — a valid action sequence has been calculated with a symbolic planner, it is still a sequential description of symbolic, atomic actions that rely on a conforming environment. Given the need for reactivity when the environment can drastically change without prior notice, the plans need to show a high degree of concurrency, monitoring the course of action. It has been proven though that mapping a partially ordered sequence of abstract actions on concurrent reactive skills is extremely difficult [82, 83].

2.1.3 Robot Architectures

In my work, the execution of actual robot plans and reacting to their various outcomes or interruptions plays a central, crucial role. Albeit largely being labelled a plan language, Cognitive Robot Abstract Machine (CRAM) really serves as a complete robot architecture model: It has the authority and ability to make goal-directed decisions, collects sensor information to maintain a sufficiently precise world model, and executes plans to minimize the delta between a current, and a desired state.

A series of well-known robot architectures exists that focus on the purely architectural aspects. The ones being most closely related to my own work are 3T [3, 17, 38], the subsumption architecture [18], and PRS/BDI [45].

2.2 Behavior

2.2.1 State Automata

A rather common, and by way of comparison rather easy way to define straight forward behavior are State Automata — or state machines. They consist of states an intelligent agent can be in, and possible transitions (edges) that lead to other states. In terms of robots, edges are usually associated with actions that either change the state of the robot, the environment, or both.

Two characteristics in state machines stand out: (1) To control robot behavior, a state machine is usually finite [75], and are called Finite State Machines (FSMs). The robot has a finite number of states it can be in. Although infinite state machines have their application in theoretical logic and planning [1], they currently yield no advantages to robot behavior definition.

(2) The other characteristic is hierarchy: Different "*strains*" of behavior are encapsulated into one state machine, and a higher level state machine exists for which the encapsulated behavior is merely a state. They can be expressed explicitly [2], or implicitly [15, 84].

While the behavior resulting from generalized plans can be represented by hierarchical FSM a posteriori, a behavior definition a priori is unfeasible. Tasks can succeed or fail for a multitude of reasons not directly apparent from the plans' syntactical structure. Thus, the entirety of knowledge stored in external knowledge bases and experiences used for action effect prediction and implicit parametrization would need to be encoded into such an FSM to represent all possible edges. Additionally, the FSM would need to be updated every time new knowledge becomes available.

2.2.2 Task Networks

Another established concept to define behavior for robots are task networks, and more generally, Hierarchical Task Networks (HTNs). They rely on the definition of named tasks that need doing, and that can be combined into a Reactive Action Package (RAP). Inside the RAP, tasks can be structured sequentially, or in parallel [32]. HTNs in particular allow STRIPS like definition of preconditions and effects of a task, but are more expressive [28].

HTNs pose a very elegant and abstract way to define behavior, which is well aligned with the principles of generalized plans. Since HTN planning is an interesting research field by itself and offers new ways to define behavior beyond the scope of this thesis, a future addition to my work could well be an integration with HTN style behavior definition and planning.

2.2.3 Behaviour Trees

To express structured, partially non-linear behavior, Behaviour Trees have proven to be a vital mechanism for programming robot activities, and Non-Player Characters (NPCs) in games. Interesting applications include increasing a UAV's modularity by encapsulating partial Behaviour Trees (BTs) [65], and helping to improve robustness and safety in hybrid systems [21]. In this thesis, I use BTs solely for representational purposes of abstract robot plans. Generalized plans in their entirety are difficult to represent this way for the same reasons given under "*State Automata*", although the operators available in BTs allow to encode the general mechanisms used (retrying actions, sequences, conditions, etc.).

2.3 Autonomy

2.3.1 Episodic Memories in Autonomous Systems

The concept of Episodic Memories is not entirely new. Nuxoll and Liard [63] have shown that EMs — although largely ignored up to that point — can be used to improve a task’s performance in the cognitive architecture SOAR. They present concepts for encoding, storing, and retrieving memories, and developed a computational model of a working memory that extrapolates knowledge from past memories. One of their prime examples for behavior enhancement through memories is TankSoar [64].

The presented approach — while spiritually pointing in exactly the same direction — differs from mine in both, type and application of memories. The authors focus mainly on integrating their EM concept into the SOAR cognitive architecture while I use EMs as an external, optional tool for deducing parametrizations. EMs as presented in this thesis are the *"raw"* data used for model generation, which in turn improves plan performance. One major difference between their and my EMs is the role of the hierarchical task tree: My reasoning and model building techniques use the task tree as the prime reference point. The authors’ live knowledge retrieval mechanisms — prior to using the knowledge — select explicit features from individual tasks to improve their agents’ behavior.

2.3.2 Vague Task Execution in Real World Scenarios

Performing tasks just like humans is easier said than done for autonomous robots. Most of the time, they lack a proper task model or strategy to handle obscure situations. I found that the three *"grand challenges"* for autonomous robots in human-scale environments as summarized by Kemp *et al.* [46] clearly show why these problems are complex, difficult to solve, and require lots of background information today’s robots do not have per se. Each of these challenges is a very specific scenario on its own, but requires robots to be very skilled in a certain, more general task. The challenges they suggest and the required robot skills therein are:

- (o) **Tidying up a House:** Fetching and placing objects of unknown nature, using knowledge of common object placement, handling large variance in room and furniture structure
- (o) **Preparing and Delivering Food:** Manipulating flexible materials, using tools made for humans, performing small but complex assembly tasks
- (o) **Outdoor Party Preparation:** Collaborating with humans, taking instructions from them, moving large outdoor furniture together

The main research fields that need to contribute to these feats are perception, learning, platform design, control, and collaboration between robots and humans. While all of these domains are distinctively different, one of their common problems is the uncertainty of information. Perception systems rely on accurate data for object classification, and learning algorithms produce robot behavior based on the training data available. Control algorithms require correct sensor input to operate within acceptable tolerances, and platform design needs to compensate for uncertainty structurally, supporting sensor information gathering and recovering from failed task attempts. Finally, collaboration between robots and humans requires a robot to have a detailed and near-exact model of the current state and the intentions of its human partner, wherein uncertainty in the robot system can lead to fatal decisions when the human is close by.

Since robots act in increasingly dynamic environments, they can barely ever know the complete state of the world around them to a sufficient degree of accuracy at all times. The challenge for autonomous mobile robots therefore is not so much to prevent or remove uncertainty, but recognizing and handling it properly. Given the domain of household chores, this means objects

are often not where they belong, their state is not as expected (e.g. clean vs. dirty), and symbolic task and object descriptions are highly ambiguous due to being strongly underspecified.

Examples of complex tasks that robots are in principle capable of include making pancakes [9], cookie baking [16], laundry folding [51], and meal preparation [66]. More individual tasks are especially challenged in the RoboCup@Home context [88, 89]. All of these tasks have a definite, judgeable outcome and are over once that outcome was reached. Their environment specifically reflects the task they need to solve. In a fully autonomous system, the environment and contextual knowledge would only vaguely correlate to the task to perform, requiring much more reasoning effort by the autonomous robot itself to find out what to do, when it is complete, and how well it was performed.

2.3.3 Longterm Autonomy

A major difficulty in today's robot systems is longterm autonomy — stable plan execution and proper failure handling and recovery without human intervention over prolonged stretches of time. Lane *et al.* [52] have pursued longterm autonomy for underwater robots. They identified three key areas necessary for making robots fit for longterm autonomy:

- (o) Correctly and thoroughly **describing the world** (world model)
- (o) Directing and Adapting **Intentions** (goal-driven, reactive behavior)
- (o) Acting **Robustly** (considering expected or unexpected failures)

While their work specializes in AUVs, these key areas are valid for autonomous service robots as well. All three areas require consistent data definitions and updates, and smart, transparent incorporation in action plans. Especially the last point, robustness, is very hard to achieve when done manually by a human: Not all situations can be foreseen in the real world.

Overview and Foundations

*“Design is not just what it looks like and feels like.
Design is how it works.”*

— Steve Jobs [97]

In the following chapters, I will present a number of topics central to my research, how they are intertwined, and what their ultimate benefits for autonomous robots are. Figure 3.1 shows the overall architecture of my work, featuring all major components. I will present the currently presented area at the example of this architecture. At its core are Generalized Robot Action Plans and their representation. These plans are embodied by a number of low-level skills for sensing the environment, and after making sense of the results, enacting their intentions — an Embodiment layer. While these plans are executed, divergences from the expected path of action are either identified as known failures or classified as new failure classes. These are then inserted into a global failure taxonomy, being mapped to failure handling heuristics. All of this activity is recorded into Episodic Memories that serve as the basis for machine learning approaches that improve the performance of said Generalized Plans by the means of Expectation Models. Figure 3.2 shows the respective execution and data flow paths in great detail, further clarifying each component’s role.

3.1 Generalized Plan Design and Representation for Robots

The core element of my research are Generalized Plans for autonomous robots. These plans are very abstract descriptions of a — usually fairly complex — task and have the distinction of being underspecified and vague in their parametrization. They are parametrized based on a concrete task description, static facts from a knowledge base, the current context, and memorized data from earlier executions.

The idea of Generalized Plans is to make them applicable to as many variations of their abstract task as possible. To meet the lack of specialization for one particular variation, they employ heuristic fallbacks for any missing parameter. This way, robots are initially rather bad at their task, but over time collect experience in performing it, ultimately mastering the task and being on par with, or even exceeding specialized, non-generalized plans.

Along with the description of tasks, the description and classification of failures is also abstracted to a point where real failure circumstances cannot be clearly identified. Like the parameters necessary for executing a plan, failures are learned over time after performing the same plan over and over again. Differences between the expected and the actual path of action are noted, and are used to either identify a known failure class, or to classify and add a new failure class to a global failure taxonomy.

Generalized Plans, their prerequisites and design criteria, as well as their representation are discussed in Chapter 4.

3.2 Data Logging and Learning from Episodic Memories

Autonomous robots make lots of decisions while performing a plan without human help or intervention. These decisions are mostly based on momentary information, such as intermediate inference results or sensor input, that is lost afterwards. This makes retracing the reasons for decisions — or the decisions themselves — very difficult, and therefore raises two issues: Firstly,

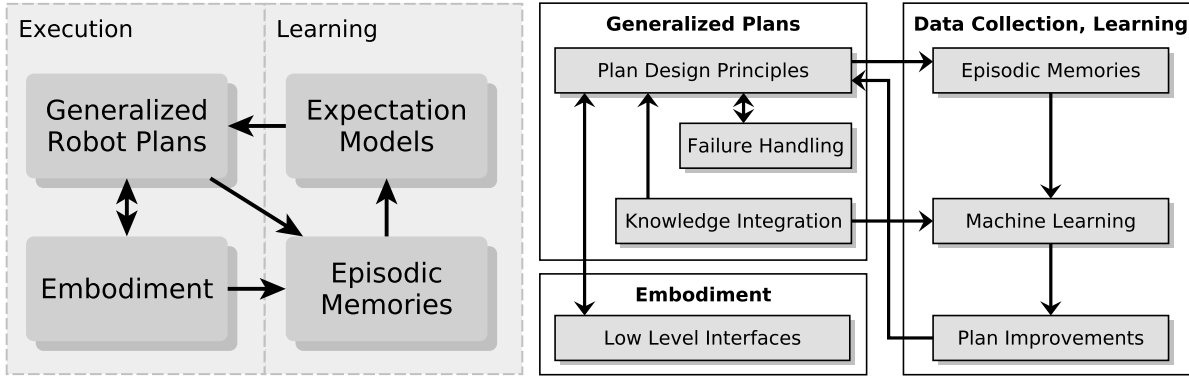


Figure 3.1: The overall, high level architecture includes Generalized Robot Plans, a conglomerate of Episodic Memories as machine learning data source and Expectation Models as the learning subject, and an Embodiment layer that lets an autonomous robot interact with the real world.

Low Level Information	High Level Information
Raw Camera Images (<i>RGBD</i>)	Goals and Intentions
Sensor Values (Fingertip Pressure, ...)	Task Parameters and Results
Kinematic Transformations (<i>tf</i>)	Hierarchical Task Tree
Detected Object Characteristics (Poses, Extents, ...)	Thrown and Caught Failure Instances
	Performed Atomic Actions
	Perception Request and Result Pairs

Table 3.1: Components contained in Episodic Memories, separated by low and high level information.

debugging of complicated decision-making processes during development or maintenance phases is obscured. Secondly, autonomous robots themselves do not have access to former evidence of which decisions led to what results, effectively ruling out machine learning applications based on this evidence.

To address these issues, I developed the concept of Episodic Memories for autonomous robots. Episodic Memories are collected while a robot performs a task, and reflect all symbolic plan data (plans, parametrizations, outcomes) and all relevant sensor data from a task [105]. Using this conglomerate of data, I developed a system for reconstructing the course of action of any formerly performed task and comparing it to a current execution, resulting in task Expectation Models [103]. These models can predict the outcome of a current task based on its parameterization and the overall current context. Furthermore, Episodic Memories can answer high level, semantic questions about the tasks a robot performed: *“how was a task done”*, *“what was the context and what was different in the environment than usually”*, *“did everything go well or were there unexpected interruptions”*.

3.2.1 What are Robot Episodic Memories

Episodic Memories (EMs) represent both, the low level, high volume data that continuous sensors in a robot system produce, and the high level, low volume intentional data that symbolically describes action parametrizations, a hierarchical task tree, and the result of any action a robot performs.

Table 3.1 shows an excerpt of components that are usually included in such an EM. The low level, subsymbolic part includes raw camera images, continuous kinematic transformations, and other sensor values for all time steps. The high level, symbolic part describes the robot’s

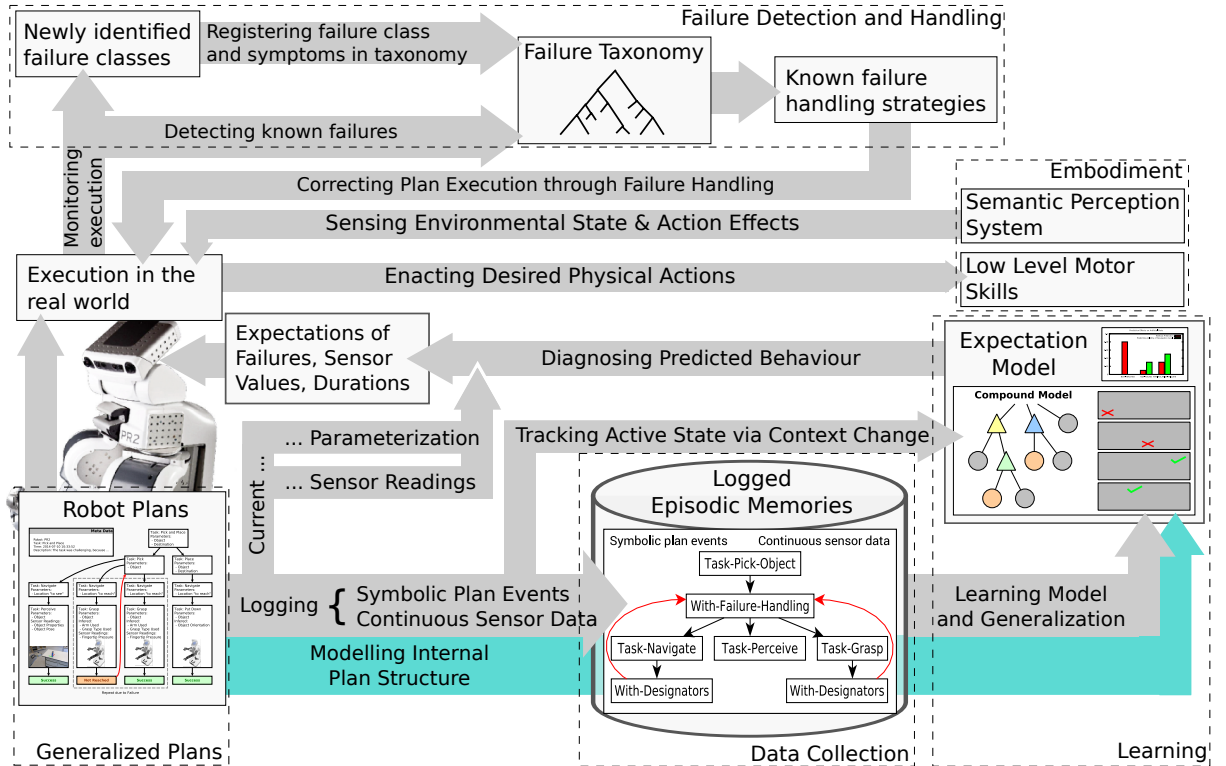


Figure 3.2: Governing architecture of the presented topics: Generalized Plans act as the central element for embodying tasks in the real world, collecting and classifying data in the process. Failures are either identified as known or new, and in the latter case are inserted into a global failure taxonomy that maps them to failure handling heuristics. From the executed plans, episodic memory data is recorded and facilitated into expectation models allowing a robot to make predictions about a task’s outcome while executing it, and choosing optimal parameters from experience.

intentions and goals, its task knowledge, action parameterizations and results, and the general hierarchical task tree of what happens while the robot is performing its work. Both levels are linked via unique identifiers and — in the case of continuous data — ranges of time points. Autonomous robots use both, symbolic as well as subsymbolic data to make decisions: The current context, action parameterization, sensor values, and world belief can fundamentally change the decision making process. EMs reflect this volatile information and make it accessible to model building, machine learning, and data analysis techniques after an action episode concluded.

The data logging infrastructure and learning applications are explained in detail in Chapter 5.

3.3 Embodiment of Autonomous Robot Control Programs

To enact either hand-crafted or dynamically planned behavior of autonomous robots efficiently and effectively, a robust layer for interacting with the real world is a must. Since in the nature of Generalized Plans the actual connection to the real world is not considered, an embodiment component is required. I present this component in the form of mechanisms for controlling motion control and navigation, as well as for sensors required by the robot. This layer also transparently handles failures that arise during interaction with the environment as far as possible. As part of the embodiment, I present a global failure taxonomy that identifies known failures and classifies new ones, selecting proper handling strategies. Embodiment also covers the translation of discrete, abstract parametrizations of Generalized Plans into — context dependent — continuous parameters for low level skills. The required knowledge to properly parametrize these

actions can be dictated by the more abstract higher level plans. Following a least commitment approach, I push these decisions as far down the queue as possible — ideally letting the actually executing module decide on the concrete parameters, only escalating the decision if it cannot find a solution on its own.

All of these aspects are discussed in Chapter 6.

Generalized Plan Design and Representation for Robots

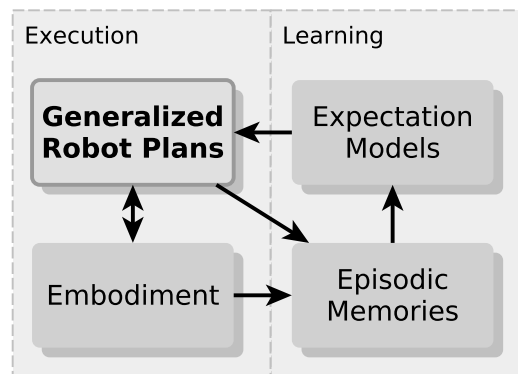
“Intelligence is the ability to adapt to change.”

— Stephen Hawking [41]

To make a robot able to handle diverse and quickly changing contexts it was not explicitly programmed for, a very generic way for describing its action plans and concurrently running monitoring mechanisms is required. This very abstract plan is then grounded in explicit knowledge about the current situation, making it executable.

This chapter describes the fundamental principles I use in designing generalized robot plans. I introduce the principles of generalized plan design and their benefits, raise the role of uncertainty in real world settings, and introduce the most common tasks for mobile manipulators as well as some strategies for performing them. Furthermore, I describe the importance of robot knowledge required for grounding very abstract plans in concrete situations to make them executable. Finally, I go into detail about the focus of my research on this topic and summarize the most important aspects of my work.

From a robot programming point of view, I introduce the CRAM plan language that has foundational character to my work. Therein, I explain why I chose CRAM, what my contributions to the system are, and how I represent the encoded robot action plans using Behaviour Trees.



4.1 Principles of Generalized Plan Design

Generalized robot plans are supposed to perform their specific task in very different contexts. At the same time, they need to convey functionality and semantic meaning of what their purpose is. This provides two requirements when designing generalized robot plans:

- (o) **Knowledge-integration** to support decision-making in different contexts, making the plan more flexible and tolerant to changes in task description and constraints, and
- (o) **Conveying semantic meaning** through design of encapsulated plans with well-defined behaviors and interfaces

These very high level requirements are underpinned by a list of concrete design aspirations, their requirements, and how to achieve them:

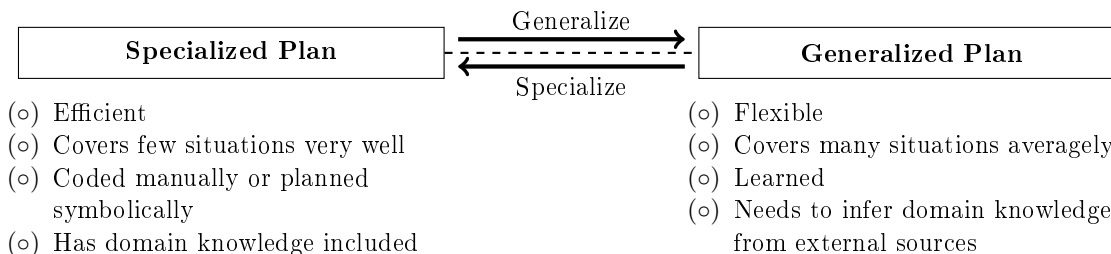


Figure 4.1: Trade-off between specialized and Generalized Plans. Specialized plans solve one task very well and efficient, while Generalized Plans cover a large area of similar plans very flexibly, but are in general less efficient.

- (o) **Accounting for anticipated and unanticipated failures:** Failures must be clearly classified into cases that a robot can know, and cannot know about. The former is countered with careful, explicit consideration in generalized plans while the latter must be satisfied by heuristics that cover as much ground as possible.
- (o) **Selecting actions based on hierarchical knowledge:** All action sequences and their concrete instantiation must be based on the available knowledge, or default to a common fallback solution. Knowledge is to be acquired through concrete parametrizations, static rules in a knowledge base, inference, or the context.
- (o) **Self-specialization through experience:** The very idea of generalized plans is that they — ultimately — contain no task and domain specific knowledge at all. In the simplest case, they draw this knowledge from external rule sets (e.g. a knowledge base). Their final form should allow them to draw upon (processed) experiences from earlier executions, learning from mistakes and improving by themselves.
- (o) **Encapsulating functionality:** Generalized plans are hierarchical constructs. As such, every layer must take full responsibility for taking track of, managing, and cleaning up their own effects on the internal belief state and the external world. Every generalized plan must thus be accompanied by a generalized clean-up routine. This aspect is well in sync with Beetz' structured reactive controllers [5] and structured plans [6], applied to a more abstract type of plan.

In their initial design phase, these plans act as if they were static. They are designed completely by human developers or are planned symbolically by an automated planner. In principle, a designer could distinguish two static cases for robot plans: (1) Generalized Plans that cover a wide range of applications, featuring a mediocre performance, or (2) specialized plans that do one thing very well, impinging upon application generality. Figure 4.1 depicts this trade-off. To allow true self-specialization for robot plans, generalizing them requires a special plan structure: Task parameters are abstracted to a purely symbolic level, leaving no traces of actual task or domain intelligence. Which action steps to perform, and how to parametrize them solely depends on external information and is similar to Beetz' declarative goals [10]. The same accounts for concurrent failure handling, the choices of which failures to monitor for which action, and what the respective recovery strategies are.

The more general a plan is, the fewer domain-specific knowledge is encoded into it. To make a robot able to improve its own behavior and thus its own plans over time, these plans need to draw rules for their decision-making from a knowledge base, which is populated with static task-related background information and robot experience data.

Since domain-specific knowledge is encoded programming-language agnostically in a well-defined knowledge format, it is easier to maintain and understand for developers on a conceptual

level. Debugging and maintaining their behavior in explicit situations requires analysis of the inferred domain knowledge though. The effort for this depends on the amount of knowledge and the interdependencies involved. I will go into detail about debugging this volatile information when I cover Episodic Memories.

In the following sections, I will point out elements of major importance in generalized plan design: (1) The role of strategy selection based on knowledge about the task and the current context, and (2) enhancing a robot's behavior through experience. Additionally, I will pinpoint the role of these elements in concrete examples during autonomous object search, and fetch and place tasks.

A Plan's Degree of Generality

The degree of generality of an existing plan reflects whether my above principles were obeyed when designing and implementing it. To make any statements about its generality, the following questions about the plan have to be answered:

- (o) Does the plan include no traces of explicit task domain knowledge?
- (o) Does the plan make no assumptions about the environment or the robot platform that are not vital to the task performed?
- (o) Does the plan scale to all situations that could be encountered during its task?
- (o) Does the plan rely on learnable parameters?
- (o) Does the plan use general modules that have overlap with other tasks?

So for a fully generalized plan all questions would be answered with a yes. Violations to these criteria generally result in less performance gain through experience, and make the plan more of a niche solution than a generalized task module.

4.1.1 Action Description vs. Goal Definition

Instructions given by humans often involve a mixture of two representations: A task is either explicitly described as an action, or as the definition of a goal state, starting from an initial state. Both representations are shown schematically in Figure 4.2. While it is relatively simple to perform a given action description in a start state, deducing the required actions when only the start and goal states are known is more challenging. Traditionally, this is the task performed by planning algorithms such as STRIPS (and more generally: PDDL planners).

Planning action sequences — or at least the next step — is vital for cognitivist autonomous agents in dynamic environments. In this thesis, I concern myself further with Planning Domain Definition Language (PDDL) based planning and PDDL domain extension through learning in Section 5.7.

4.1.2 Knowledge-based Strategy Selection

The same general task often needs to be executed in different fashions, depending on what is known about the context and the task itself. One of the most common examples in mobile manipulation is grasping an object: The overall sequence of how to grasp an object stays the same: Approach an object, open the hand, enclose the object (or parts of it), close the hand, carry off the object. This sequence is driven by both, symbolic and subsymbolic parameters that define the actual execution:

Symbolic:

- (o) Which arm to use

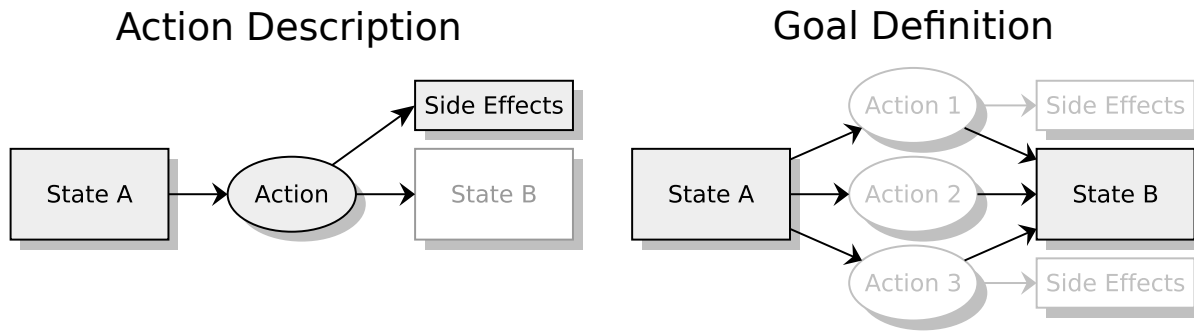


Figure 4.2: Action Descriptions with an initial state A result, when successful, in a definite state B. Goal Definitions with an initial state A and a goal state B can have arbitrarily many ambiguous transition paths, which can cause unintended side effects. Action Descriptions are the disambiguated form of Goal Definitions, i.e. the concrete selection of one transition path, accepting all of its side effects.

- (o) Which approach strategy to use: From the front, top, side, etc. [19, 22]
- (o) Which part of the object to grasp (task dependent [81])

Subsymbolic:

- (o) Relative position of the grasp point on the object
- (o) Force to apply, closing radius of the hand

Given an incomplete description such as (an action (to grasp) (object ?o)), an autonomous robot has to infer all the required information from its own knowledge base and the context. When designing generalized plans, as little of this information as possible should be present in the plans. Which arm to use for grasping is inferred e.g. based on heuristics like which hand is free, and how far each hand would have to travel to reach the object. More elaborate approaches use knowledge about the purpose of the grasp, e.g. from subsequent plan steps or the parent task's current goal. The object part to grasp depends on the task for which purpose the grasp is performed: For pouring, a bottle must be grasped at its body and not at its cap. For only transporting it, all parts are fine. The approach strategy is constrained by the part chosen: For a bottle standing upright, the body can only be approached from the sides, front and back. The cap can be approached from the top as well.

The required information — at the example of grasping an object — can come from a number of distinct sources, with each (if present) superimposing its predecessors:

- (o) **Static knowledge base, based on task type**
The task always has a certain default way of handling objects.
- (o) **Static knowledge base, based on object type**
The object type to act upon is per default always handled in a certain way.
- (o) **Static knowledge base, based on task/object type combinations**
The object is grasped for pouring from it, so some parts may not be used for grasping.
- (o) **Contextual constraints**
Parent tasks constrain how to grasp this object, or objects in general (e.g. `with-context`)
- (o) **Description of the current task**
The explicit description of the task to perform states how to handle the object.
- (o) **Requirements and constraints observed while performing the task**
The object is not approachable from the left and back side, so only approach directions from the right and front are possible.

In some cases, the ambiguity of vague task descriptions cannot be resolved using the available information. There are two ways to overcome this problem:

(o) **Assuming heuristic default behavior**

For every action parameter, a fallback value, and thus a default behavior is specified. While this ensures executability of the task, it might result in suboptimal, or even disastrous behavior: Grasping a very fragile object with a default, average grasping force can easily destroy it, possibly rendering the task (and its supertasks) impossible.

(o) **Signalling insufficient information and escalating the problem**

The next-higher supertask is informed that too little information is available, and is supposed to re-parameterize or provide additional hints for execution. This is handed up along the task tree until either the missing information becomes available, or a human operator (the final parent node) is reached and asked to provide it.

Depending on the task, and how crucial its success (or disastrous its failure) is, both strategies can be intermixed in the same generalized plan. Section 4.4 will further elaborate on the source and nature of contextual knowledge for autonomous robots.

4.1.3 Experience-based Behavior Enhancement

The whole concept and idea of generalized plans is to take explicitly modeled situations out of action descriptions, resulting in schematic, abstract plans. While this, given enough supporting knowledge, makes the plans very versatile it has two major drawbacks for plan designers:

(1) **Identifying and repairing plan problems is much more difficult**

The more general a plan is, the less information is available about why a certain decision was made, and which parameterization led to a certain outcome. How control structures are intertwined strongly depends on the tasks defined by external knowledge (possibly unknown to the designer).

(2) **Accounting for specific situations is very cumbersome**

As in generalized plans decisions are made based on external knowledge, it gets significantly more difficult to introduce new behavior for a specific situation than it would in an imperatively programmed, task-specific action sequence. In the former, appropriate rules need to be defined that are compatible with all other task descriptions, while in the latter an additional branch of code can explicitly account for the situation.

Problem number one is addressed by equipping the robot's plan architecture with comprehensive *plan logging capabilities*: Each language construct is augmented by emitters that signal when the construct starts, ends, or when significant atomic actions within it are triggered. Example constructs are (sub)plans, queries to external components, or failure handling elements:

```
;; Definition of plans
```

```
(defmacro def-plan (name parameters code)
  '(let ((id (signal-start :plan ,name ,parameters))
        (return-value nil)) ;; Return value is relevant
    (unwind-protect (setf return-value (progn ,@code))
      (signal-end id return-value))))
```

```
;; Definition of failure handlers
```

```
(defmacro def-failure-handler (name &rest handling-code)
  (let ((fnc-name (symbol (concatenate 'string "handle-failure-"
```

```

                                (write-to-string name))))))
‘(progn
  ;; Defines the actual handler function
  (defun ,fnc-name (failure)
    (let ((id (signal-start :failure-handling ,name failure)))
      (unwind-protect (progn ,@handling-code)
        (signal-end id))))
  ;; Globally stores named reference to handler
  (global-register-handler fnc-name))))

;; Example usage of def-failure-handler
(def-failure-handler :simple-retry
  (do-retry))

;; with-failure-handlers catches failures and calls the respective
;; globally registered handlers.
(with-failure-handlers ((:manipulation-failure :simple-retry)
                       (:navigation-failed :other-handler))
  (perform-some-failure-prone-task))

```

The functions `signal-start` and `signal-end` record their activity into a hierarchical tree that represents the structure of the tasks executed. Additionally, this tree can hold information about start and end times of nodes, return values of functions (success/failure), and any occurring meta data (e.g. plan parameters, requests and results of a call to the perception system, etc.). The resulting data can be inspected for when a given failure happened, what execution trace led to it, and which parameter values were involved in the decision-making process leading to this failure. A plan designer therefore has full access to all static as well as volatile information from the plan execution. Section 3.2.1 explains in-depth what the structure of this resulting data is.

The second problem emerges from the fact that knowledge rules for defining generalized plan’s behavior can become too complicated to manage manually. This is remedied by applying machine learning techniques to the logged execution data from the previous problem: Decision trees, confidence intervals for probabilistic parameter choices, and static rules can be learned from sufficient amounts of Episodic Memory data. In Chapter 5, this process is explained in detail, together with comprehensive code examples.

4.1.4 Implicit Modular Task Recovery

One of the main advantages of modular, context-less plan design is that with these plan primitives, a large variety of complex behavior can be defined [102]. The same principle is advantageous for failure handling. Ingham *et al.* [43] have developed the programming approach RMPL that allows constraining of concurrently running processes while performing an otherwise sequential task. Their task execution monitors a set of variables and fires upon meeting predefined requirements. I extended this principle by introducing mechanisms for interrupting plan performance when unusual conditions arise, following a take-down routine. As each of my (sub)plans is accompanied by a take-down strategy to rewind its own effects, each layer in the hierarchy only has to revert its own actions. This strategy allows for encapsulating implicit recovery mechanisms to properly unwind tasks created by complex plan hierarchies. Figure 4.3 gives an intuition of how the resulting plan structure elements are encapsulated and can be combined again.

Such “building block” plans have well-defined behaviors and capabilities. They are semantically annotated with required input parameters and possible outcomes. Nesting them entails

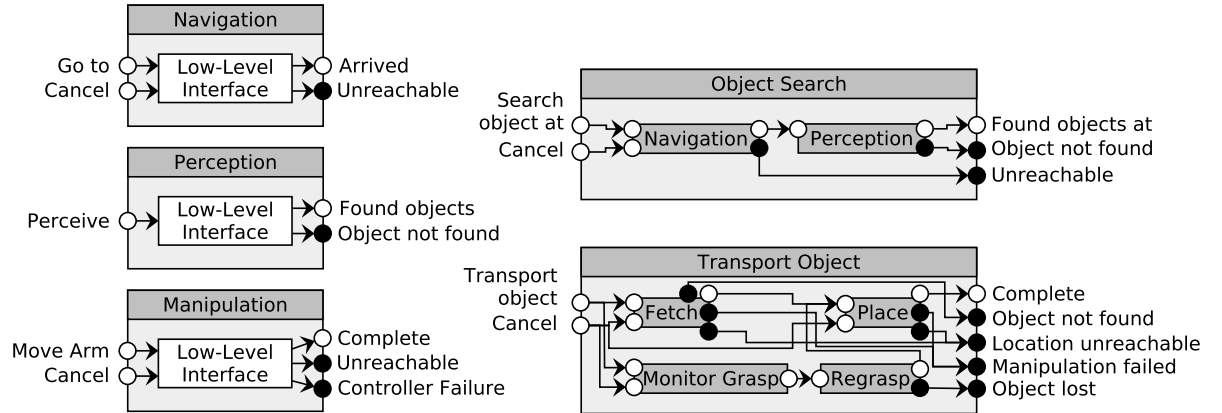


Figure 4.3: Schematics of hierarchical, modular plan elements. On the left are basic primitive functions: Navigation, Perception, and Manipulation. On the right are compositions of primitives: “Object Search” as a linear sequence of Navigation and Perception, and “Transport Object”, featuring a parallel monitoring process to trigger regrasping when necessary. Every element has defined in- and outputs.

advantages in code reusability, modularity and function encapsulation, but also grants semantic meaning to the contained structures and the resulting overall plan.

4.2 CRAM as a Robot Plan Language

In this section I give an overview of what CRAM is, why I chose it in my work, and how I extend it. CRAM has foundational character to my work, as many of my techniques are implemented using its language constructs or design concepts.

CRAM defines a comprehensive programming environment for writing flexible action plans for autonomous robots [12]. It is suited for reasoning about its own plan execution [101] and allows robot control in the real world, as well as in simulated environments (e.g. Gazebo, see Appendix A.2: *Gazebo: Simulation-based Manipulation*).

The main purpose of CRAM is to design and implement robot action plans on a very abstract level that are grounded in the real world as late as possible during execution, following a least-commitment approach. This way, the executing robot commits to actual action only when really needed, not giving up degrees of freedom in reasoning and decision making too early. To this end, CRAM offers a number of operators and environments that allow vague description of objects, locations, and actions, in the form of designators [57]. Designators are used throughout all generalized robot plans in CRAM and are its main means for defining degrees of freedom for decision making. They are hierarchical trees of key-value pairs, with the values being any valid Lisp constructs, including other designators. Typical designators are defined as follows:

```
(an object (type cup) (color red))           ;; Object designator
(a location (on table) (in kitchen))         ;; Location designator
(an action (to grasp) (an object (type cup))) ;; Action designator
```

To resolve these vague constructs, CRAM makes use of its internal Prolog-esque reasoning engine, using backtracking over multiple hypotheses for vague descriptions to find a proper grounding in the current situation. Since their ambiguity can lead to multiple solutions, designator solutions are tried iteratively, ordered from the most specific fit to the most general one. Specific solutions are based on specialized, possibly learned knowledge about the current situation and are highly efficient. General solutions are fallbacks that almost always work, at the cost of speed, efficiency, and task competence. Figure 4.4 depicts this process.

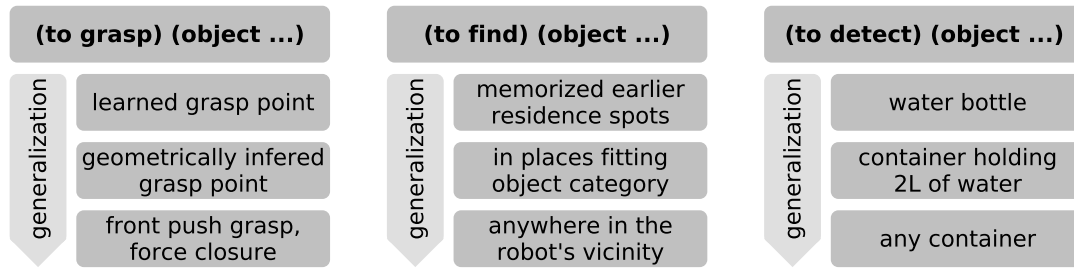


Figure 4.4: Action descriptions are first grounded in the most specialized facts known to the robot, and are generalized iteratively when the specialized rules do not work (learned knowledge, static knowledge base content, generic rules).

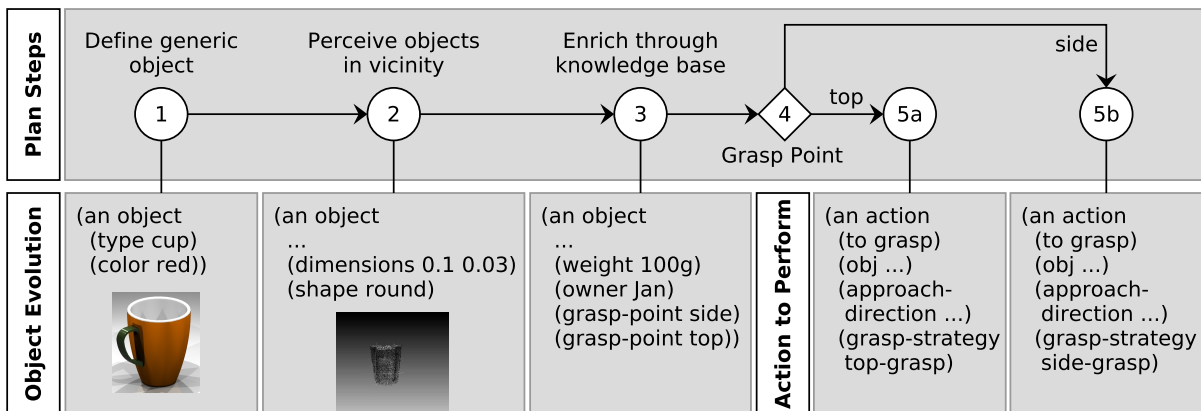


Figure 4.5: Example of grasp strategy decision based on an object designator's content. An object in vicinity is perceived given the generic description (type cup) (color red). A perception system enriches the object's description by visual characteristics (6D pose, dimensions, shape), and a knowledge base appends non-visual, but known features (weight, owner, defined grasp points and their grasp strategies). Based on the object's pose and grasp-type, different grasping-actions are performed.

For resolving designators and grounding them in the current situation, each type is handled differently: Object designators usually describe dominant object features, and are enriched using perception systems (6D pose, dimensions, further visual characteristics), and a knowledge base holding non-visual information (weight, affordances, task-specific constraints). Location designators symbolically describe superimposable geometric constraints to represent an area from which the plan system can sample actual 6D poses. Action designators describe atomic actions performable by a robot; the actions actually executed depend on the hardware platform used, and are translated into (series of) actual hardware function calls using a hardware abstraction layer (so-called *Process Modules*).

Based on the result of designator resolution, flexible plans change their behavior in qualitative, and quantitative ways. Figure 4.5 shows an example of a simple plan to grasp an object, based on the respective object designator's content. By grounding designators in the current situation, flexible plans can change their strategy to solve a task more competently. This results in varying behavior that is difficult to foresee during design time, and requires elaborate failure detection and handling mechanisms that can handle very different situation configurations. By learning a task model based on experience data, predictions about these failures can be made such that a robot can anticipate problematic situations and avoid them altogether [103].

Ultimately, these complex plans are in structure very similar to BTs: BTs are a capable tool for modeling and visualizing hierarchical action plans, but lack the ability to explicitly model

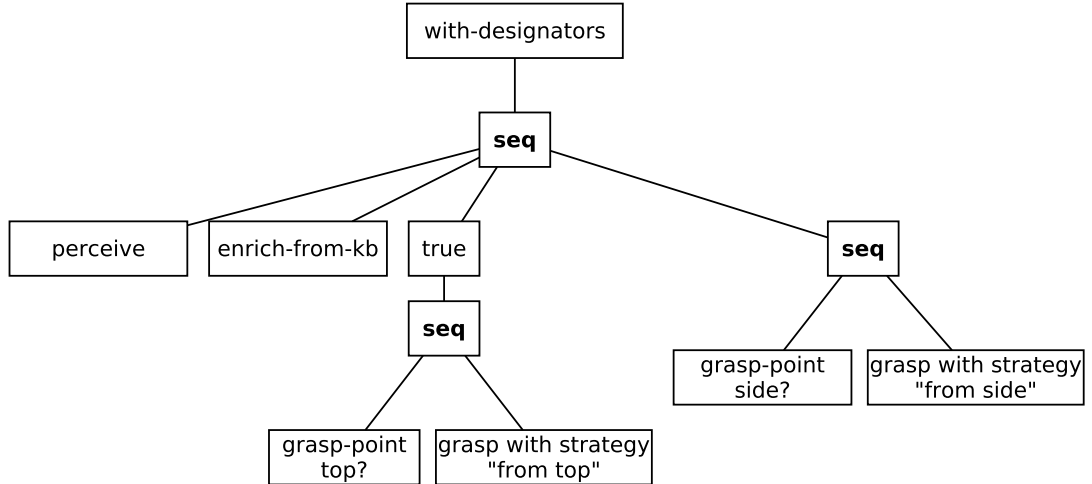


Figure 4.6: BT representation of the simple plan shown in Figure 4.5.

data, such as designators. A depiction of the simple plan shown in Figure 4.5 represented as BT primitives can be seen in Figure 4.6. Here, the two decision branches for top or side grasps are modeled using two sequences with initial checks as gatekeepers. The first one features a *"true"* decorator to keep the upper sequence running when the object is not grasped from the top. The individual primitives and their correlation to generalized robot plans in general and the CRAM language in particular will be explained in 4.2.2: *Representing Robot Plans: Behaviour Trees*.

As apparent from the above example, and explained in more detail in [102], generalized robot plans make use of different knowledge sources to make their own behavior more flexible and reactive. When executing a plan, three different knowledge types are typically used to support a robot's decision making processes:

- (o) **Task-related Knowledge:** Parameterization of the current task, as specified by either a parent plan, or ultimately by a human operator.
- (o) **Static Environment and Background Knowledge:** Semantic maps of the environment, physical models of doors and drawers, but also static facts the robot uses for inference (such as *"Perishable items reside in the fridge"*).
- (o) **Volatile Sensoric Knowledge:** Dynamic perception results, detected obstacles, dynamic triggers in the environment. All of these are unknown during design-time.

These knowledge sources are extended by a fourth one which can be assumed static for single plan executions, but affects the priors an autonomous robot has:

- (o) **Experience-based Knowledge:** Learned rules about correlations between contexts, task parameters, and their outcomes [103].

On the basis of these knowledge sources, reactive robot plans must be able to select, interrupt, and change their behavior whenever necessary. Strategies are selected based on a current situation, such as grasping with the left or right hand, depending on which one is available or closer to the object to grasp. While executing a task, a concurrent watchdog process must monitor task relevant features for significant changes, and adapt the task accordingly. An example for this would be an object slipping from the gripper during transport, requiring regrasping. When changing a task, recovery mechanisms must clean up any now-unwanted alterations in the environment, which can range from belief state updates within the robot to executing new plans that perform complex manipulation activities. In [102], these requirements are explained in detail, and the `with-context` environment is introduced:

(with-context ($c_1 \dots c_n$) *task*)

with-context allows the specification of n contextual constraints c_i that are active while executing **task**. Such constraints can be not using the robot’s left gripper, or to limit the area in which the robot is allowed to move. An important property of **with-context** is that it can be nested. A task can therefore be constrained by its own limitations, such as a maximum driving acceleration of $1m/s^2$, but also by contextual constraints from plans further up in the hierarchical chain, such as a total maximum velocity of $5m/s$.

4.2.1 Robot Plan Design

Conceptually, robot plans are a set of sequentially or hierarchically ordered directives that transform sensed information about the robot’s environment into knowledge of the current state of the world, compare them to a desired state, and perform appropriate actions to minimize the delta between both. To formulate these very abstract instructions in a way that robots can understand, a plan language is required that allows access to the robot’s sensing capabilities, and controls its actuators. Based on the principles of the chosen architecture design, a robot’s desired intentions are then encoded into this plan language.

In my research, I used the cognition-enabled architecture CRAM to define robot behavior. CRAM is based on Steelbank Common Lisp (SBCL), which allows to define extensions of the plan language easily, and enables flexible development of new language constructs. Many other prominent architectures for robots exist (for example PRODIGY [20], ESL [37], PRS [44], SOAR [49], ICARUS [54], POMDPs [70], and SmartTCL [86]). I chose CRAM over these alternatives due to a multitude of reasons, with the most prominent ones being:

- (o) Use of **Designators** as native, flexible data structures for vague entity description [11]
- (o) Proven **applicability** and existing interfaces for **embodiment** in modern real world robotics [8, 12]
- (o) Complete integration with the **Robot Operating System (ROS)**
- (o) **Actively developed and maintained** in several EU funded projects (e.g. ROBOHOW, ROBOEARTH, SHERPA, SAPHARI, ACAT)

To formalize and implement knowledge used to parametrize robot plans, I chose the popular PROLOG driven knowledge base KNOWROB by Moritz Tenorth [93]. KNOWROB is used in a large number of public projects and has excellent integration with both, CRAM and ROS.

The design process of generalized robot plans requires constant re-evaluation and reconsideration of their structure. Because evaluation requires criteria, I chose the excellent set of rules from *Introduction to AI Robotics* [59], as adapted from the original set in *Behavior-Based Robotics* [4]:

- (o) **Support for modularity**: Does it show good software engineering principles?
- (o) **Niche targetability**: How well does it work for the intended application?
- (o) **Ease of portability to other domains**: How well would it work for other applications or other robots?
- (o) **Robustness**: Where is the system vulnerable, and how does it try to reduce that vulnerability?

While originally these criteria were intended to judge the quality of robot software architectures, they serve the purpose of evaluating robot plans perfectly well. As pointed out in [59],

the niche targetability and ease of portability are often at odds with each other, requiring the designer to make trade-offs and prioritize emphasis per case. In the case of generalized robot plans though, niche targetability is enabled through knowledge-based specialization of generalized plans, making the niche target a polymorphic sub-case of the general situation. Therefore the design of generalized plans does not suffer from this trade-off, but can enable new niche targets using external knowledge made available to the robot, leaving the Generalized Plans intact.

The design of generalized robot plans is explained in more detail in the following sections.

4.2.2 Representing Robot Plans: Behaviour Trees

Symbolically planned robot behavior typically has hierarchical character: A higher order activity plan specifies atomic sub-activities which it employs while seeking to achieve its own goals. How exactly these sub-activities perform their task is beyond the control and interest of the higher order plan, leaving aside task-specific parameterizations and contextual constraints.

While from a programmatic point of view this approach yields elegant plans that are comparably easy to comprehend, it lacks a proper representation on a conceptual level: To get an idea of the big picture of a plan and all of its sub-plans, all encapsulated activities need to be expanded and inserted into one huge plan construct. Since on this level the exact implementation is usually not of interest, I represent these plans using Behaviour Trees (BTs).

In this section, I give a brief introduction to Behaviour Trees and why I use them, and present common CREAM language constructs, as the architecture of my choice, represented in that format. While most shorter plans in the following sections and chapters are spelled out in actual code, I will compile larger-scale plans into BT format for the sake of clarity. These plans can be found in Appendix *B: Plans*.

Behaviour Trees: A brief Introduction

Behaviour Trees are effectively n -ary trees with a fixed set of node types that reflect how these nodes should be evaluated when processing the tree. The basic node classes, together with some of their specializations in BTs are:

- (o) **Composition:** An aggregator node that has at least one child node. Typical specializations include Sequence nodes that perform all child nodes sequentially, and Fallback nodes that return upon the first successful child in a linear sequence.
- (o) **Decorator:** Has exactly one child, either transforming that child's return state (e.g. negation), or acting as a control flow structure such as a loop.
- (o) **Leaf:** Cannot have any children. Implements actual, atomic program behavior beyond the hierarchical structure of the BT. Examples are starting a navigation action, reading a sensor value, or picking up an object.

A BT is traversed depth-first, while Composite nodes can define an alternate order of execution of their child nodes, or prematurely return before having processed all children. Each node can have one of three return states when queried: Success, Failure, or Running. It is up to the respective Composite node to decide how to handle a child's return value. A Sequence node would continue executing child nodes until either all children returned Success, or one child returned Failure.

BTs have gained large momentum in game development, controlling NPC behavior. While the roles of most NPCs are very specific and narrow, the applicability of BTs for behavior definition based on an exogeneous world state maps perfectly to robot plan design. In my work, I do not use BTs for actual behavior control, but rather for visualization and conceptual

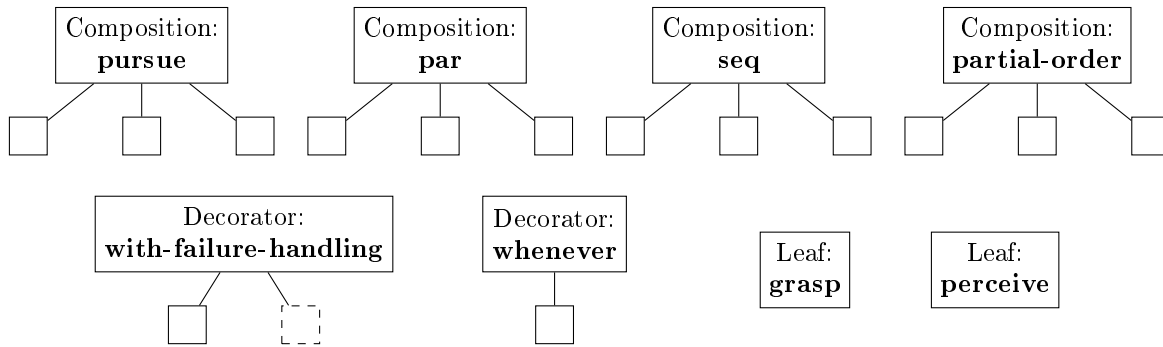


Figure 4.7: Individual CRAM language elements represented as BT constructs: Compositions having multiple children, Decorators with mostly one child, and Leafs representing atomic actions

design. The robot behavior presented here is thus implemented as CRAM plans. However, more complex behavior is depicted as BTs for the sake of clarity.

Common CRAM language constructs represented as BTs

As per Section 4.2.2: *Behaviour Trees: A brief Introduction*, the most common flow control concepts and example atomic actions from the CRAM plan language are depicted as BT constructs in Figure 4.7. The individual elements I defined in BT terminology are described as follows:

- (o) **pursue**: Runs all children in parallel, until one returns. All other children are forcefully terminated. Returns the return value of the finished child.
- (o) **par**: Runs all children in parallel. Succeeds if all succeed, fails if one fails. Returns the result of the child that finished last.
- (o) **seq**: Runs all children sequentially, until one fails, or all succeed. Returns the respective value.
- (o) **partial-order**: Runs children in designated order (potentially in parallel). Returns the result of the child that finished last.
- (o) **with-failure-handling**: Runs main child and additionally runs failure handling code (optional child) if a failure signal from the main child was caught. If no failure signal was caught, the child's return value is returned.
- (o) **whenever**: Whenever a given condition is met, the only child is run (possibly concurrently). Returns the child's return value.
- (o) **grasp**: Atomically runs code to pick up an object. Return value depends on the success of the implementation.
- (o) **perceive**: Atomically runs code to perceive an object, the environment, or other details as specified in the node's parametrization. Return value depends on the success of the implementation.

4.3 Mobile Manipulation

The most prominent task for autonomous robots is the manipulation of objects. In this section, I will discuss the role of mobility in manipulation tasks, its advantages and inherent risks. In current factory settings, robots — as well as the objects they manipulate — are relatively static with respect to their environment, requiring the robot to only move its end effectors for manipulation. Outside of factories, especially in human-scale environments, scenes are designed to fit human habits. For example in households, objects are constantly moved around, storage places are scattered around multiple rooms, and tasks require robots to operate in two or more non-adjacent places. Examples for such tasks are tablesetting and meal preparation. Both

Primitive	Description	Class	Req. Component
Enclose	Enclosing object with hand	Grasping	Hand/Gripper
Release	Releasing object from hand	Grasping	Hand/Gripper
Navigate	Navigating mobile base (2-dimensional)	Navigation	Mobile Base
Reach	Reaching cartesian coordinates	Motion	Arm Kinematics
Look	Pointing camera at coordinates	Motion	Pan/Tilt Unit (“Head”)
See	Identifying objects in field of view	Vision	Perception System

Table 4.1: Atomic action primitives required by mobile manipulation robots. Each primitive is member of a class of primitives (grasping, navigation, motion, vision), and requires certain components for operation.

require an autonomous robot to fetch objects from cupboards, drawers, and a fridge, and bring them to the working area in a kitchen or a meal table. Navigation, as well as path planning under consideration of obstacles, is thus a central part to enable manipulation.

To illustrate variability of parametrizations and strategies that a Generalized Plan should be able to cover, I list common tasks in manipulation up to implementation details.

4.3.1 Common Tasks for Mobile Manipulators

During mobile manipulation, autonomous robots can achieve most tasks using a set of atomic action primitives as shown in Table 4.1. Using these primitives, I constructed a number of tasks very common for mobile manipulators. Their sequence and composition of primitives is detailed in Figure 4.8.

The five basic actions an autonomous mobile manipulator needs to be capable of are *closing and opening its hand*, *moving its base*, *reaching for entities in its vicinity*, and *visually identifying its surroundings*. Additionally, a sixth capability is *moving its head* — or Pan/Tilt Unit (PTU) — situating its camera system. While this is not notoriously necessary, it gives the robot additional freedom to explore the environment and relieves it of unnecessarily using its navigational capabilities. This reduces localization uncertainty from odometry drift, and the risk of collisions.

In Figure 4.8, six basic activities are decomposed into the above primitives necessary to perform them. Each of the activities can have one or more preconditions, such as first finding, localizing, and approaching an object before it can be picked up, or identifying a drawer handle before it can be grasped. These additional steps are highly context dependent and are not part of the actual task. They need to be taken care of by a higher level plan that checks and ensures the availability of all required information to perform the more basic tasks. The same plan (or its parent plans) must take care of handling failures that were emitted while performing the basic activities, and react accordingly by either retrying, choosing a different task, or by giving up.

In the following paragraphs, each of the basic activities is discussed briefly, together with their specific requirements, potential pitfalls, and information that was left out of the Figure for the sake of brevity and clarity.

Picking up an object (Figure 4.8a) An object needs to be in the robot’s vicinity in order to be picked up. The robot needs to be kinematically able to reach it without colliding with other objects. All necessary information about the object (e.g. grasp points) and task (e.g. which end effector and which grasp point to use) must be available prior to picking it up, partially acquired through e.g. a *Searching for an object* action.

First, the robot needs to approach the object with its end effector and assume a *pregrasp pose* (the role of pre- and post-poses will be further explained in Section 4.3.3). While doing

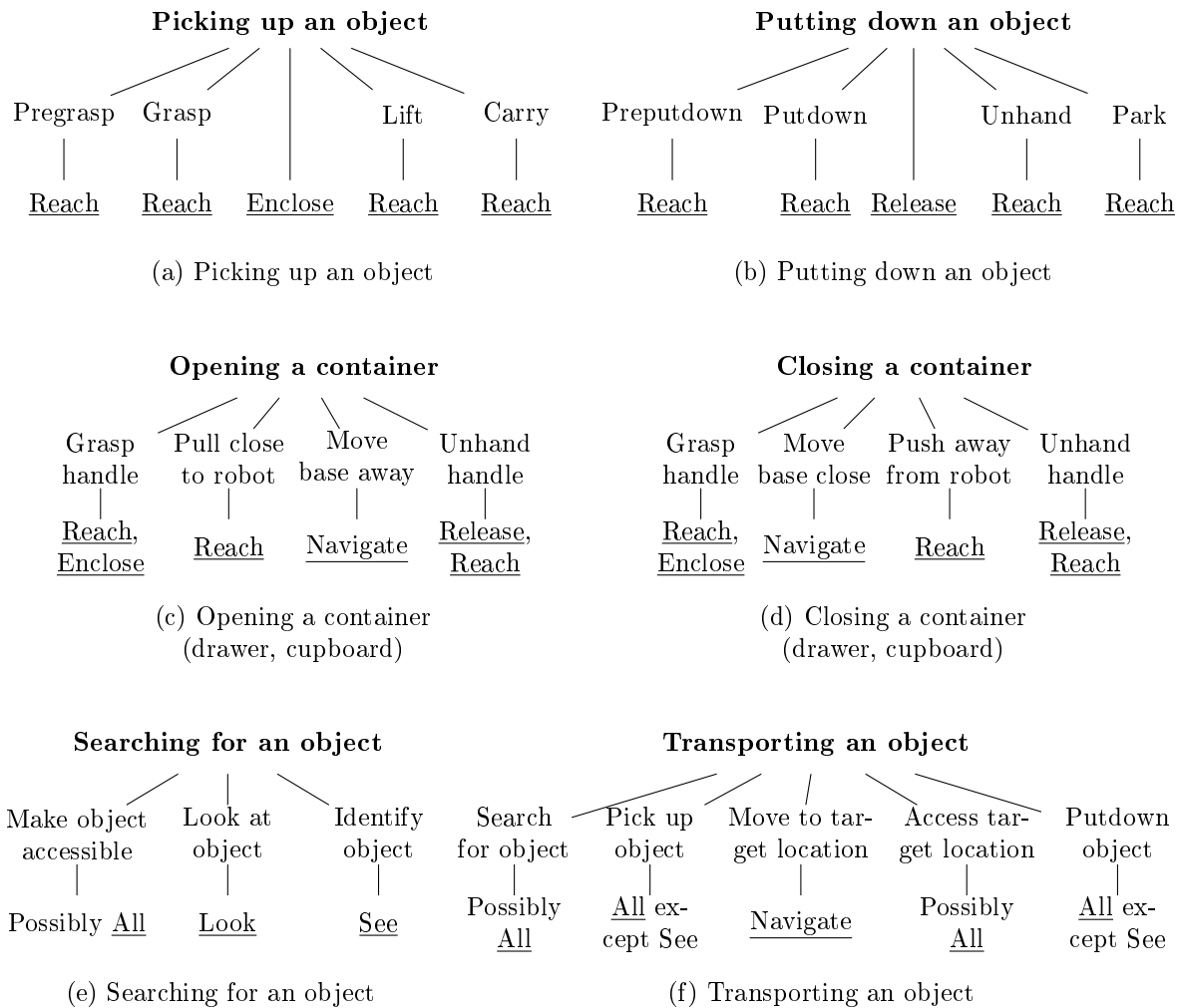


Figure 4.8: Common tasks for mobile manipulation robots, as constructed from atomic action primitives. “Possibly All” denotes tasks that may or may not require all primitives, such as accessing target locations (e.g. table top vs. drawer).

so, the end effector is positioned as close to the object as possible, while not colliding with other objects around it. It then goes into an actual *grasp pose*, linearly moving the end effector towards the object (in case of a full closure grasp). During this phase, collisions are ignored to account for object model errors that would make the object appear too large, making grasping it impossible. Alternatively, a motion planner could be allowed a collision penetration depth for the object model. The end effector is then *closed* to fixate the object. The motion planner must be informed of two changes: Attaching the collision model of the grasped object to the end effector, and adding its mass to the kinematic chain; for heavy objects, the dynamics of the robot arm will change, potentially resulting in controller failures when ignored.

To take minor model and perception errors into account, the robot then slightly *lifts* the object from the table. Possible misperception could position it inside of the table model in the collision scene, otherwise making moving an object fixated to the robot’s end effector impossible, thus preventing successful motion planning. After lifting, the robot assumes a *carry* pose for transport with the end effector holding the object, again avoiding collisions in a free space motion.

Consecutive motion planning tasks for this end effector now need to take the object’s shape into account, as it stays “attached” to the robot until it is put down, lost, or taken away.

Putting down an object (Figure 4.8b) Similar to picking up an object, putting it down requires immediate vicinity to the location where the object is to be placed. This means for most cases that this action was preceded by a *Transporting an Object* action. Additionally, the target location has to be perceived as “free”, with enough space to fit the object. The robot has to be kinematically able to reach the target location such that the object can reach its final pose.

The object to put down needs to be brought close to its target location first. To do so, under consideration of the collision environment, the robot assumes a *preputdown pose*. The object is then linearly moved into place, to the final *putdown pose*, ignoring collisions to account for possible errors in the environment model and the target location (e.g. collision model of the put down surface is geometrically higher than the target pose). The object is then *released* from the end effector, which is then linearly moved away from the object into an *unhand pose* close to the object. This is done for the same reasons that, during picking up an object, a pregrasp pose is assumed. After releasing the object, it is important to detach the object model from the robot model to inform the motion planner of the changes in the kinematic robot chain and the end effector extents. The end effector is then moved into a *park pose*, avoiding collisions in a free space motion.

Opening a container (Figure 4.8c) The container to open needs accessible handles that the robot can either pull, push, or otherwise move to uncover the container’s contents. At least one handle needs to be kinematically reachable, and its exact pose and characteristics need to be available to the robot. Additionally, enough space needs to be available for moving the robot’s corpus out of the way when opening the container.

Initially, the handle used to open the container must be *grasped*. Depending on the characteristics of the container, it is usually either attached to a prismatic joint (drawers), or a revolute joint (e.g. cupboards, fridges, dishwashers), which affects the trajectory used for opening the container. This information is either already known when the container has been identified (and the information is available in an external knowledge base), or is dynamically inferred based on the force exerted on the grasping hand when moving the container door [77]. The handle is *pulled* close to the robot to make base navigation with a grasped handle more tractable. The base is then *moved* to fulfill opening the container. Base movement is only necessary when e.g. a drawer opens farther than the robot can pull its arm back or when the initial base position is too close to the drawer already. Navigating back allows for additional space along the opening axis. Opening a cupboard with a revolute door and a long door lever can require evading the opening door, also requiring base movement. Figure 4.9 depicts some examples for this. Finally, the handle is *unhanded* to allow other navigation and manipulation activities.

Closing a container (Figure 4.8d) Closing a container is done in the exact opposite way than opening it. The robot requires information about the door handle, *grasps* it, *moves* the base such that the largest part of the opening trajectory is covered, and *pushes* the door close. The handle is then again *unhanded*.

Contrary to opening a container, handle information is not essential when closing it. Pushing the door into the appropriate direction is a faster alternative to close it and requires no precise grasping, but leaves the door dynamics uncertain and can lead to unwanted collisions.

Searching for an object (Figure 4.8e) Searching and identifying a desired object requires at least a vague description of what the performing robot needs to look for, and a perception system that can process this description to yield either success or failure upon object inspection.

In human households, objects can reside on tables, in drawers, cupboards, behind doors, or in any other enclosed, covered, or obstructed area. *Making an object accessible* means finding a potential residence location, approaching it, and opening or uncovering it if necessary (as in

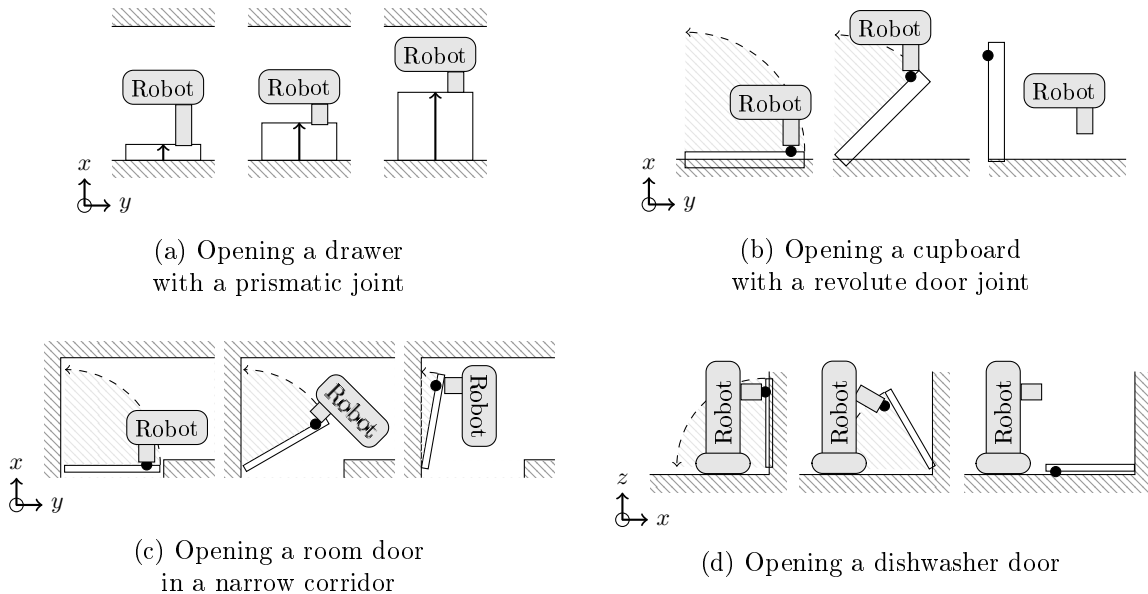


Figure 4.9: Most common cases of extra space requirements and constraints when opening containers of different kinds.

opening a container). To test available objects against the desired description, the robot *points* its camera at the object, and processes perceived visual cues to *identify* what it sees. These steps (together with *closing the container* again) are repeated until either the object is found, or a frustration tolerance is exceeded.

Transporting an object (Figure 4.8f) Besides the preconditions of its subtasks, object transport requires information about the object to deliver, and its target location. Necessary object knowledge includes constraints such as not tilting a full mug, or a maximum velocity for fragile, unstable, or stacked objects.

For transporting an object, it first has to be *searched* and localized. The robot then *picks* it up and *moves* its base close to the target location. This location needs to be *made accessible* (e.g. opening a door, etc.) before the robot can *put down* the object.

For situations that require more than one hand to carry — such as heavy objects, or objects of large dimensions — making the target location accessible needs to be done before actually acquiring the object. These two variants have to be distinguished by a higher level plan that has a priori information about the task, or that can branch into the respective strategy after the object has been found and identified.

4.3.2 Overcoming Space Constraints when Accessing Containers

Accessing containers in household environments ranges from removing the lid of a jar over swinging open a cupboard door to pulling out a drawer. Opening and closing such containers is part of the most common tasks for mobile manipulators in human households (see Section 4.3.1), and thus deserve a closer examination.

One of the biggest challenges for robots when handling containers is potential extra space required for articulating them, as for example for very long drawers or revolute door joints with a long lever. Without sufficiently sophisticated strategies, this can even make articulating them impossible. Figure 4.9 depicts some of the most common cases, which are explained in more detail in the following paragraphs.

Opening drawers (Figure 4.9a) Opening a long drawer requires extra space behind the robot standing in front of the drawer. The handle is grasped and the drawer is pulled open as far as possible without moving the base. If the drawer needs to be opened even further, the base is moved back to pull the drawer out. Alternatively, a robot can stand beside the drawer when opening it if no additional space is available to move the base back.

Opening cupboards (Figure 4.9b) Most cupboards feature a door with a revolute joint that needs to be swung open to access its contents. A robot must grasp the door's handle and then, while opening the door, simultaneously evade it. Given that enough space is available, it can back off behind the door path and then return to the opened cupboard when the door is completely open.

Opening room doors (Figure 4.9c) Rooms in human households are often connected via narrow corridors, featuring little more space than doors require to be opened. Instead of applying the same strategy as for opening cupboards that mostly have enough free space around them, robots can partially articulate the door using its handle, and at an appropriate time switch to "pushing it open". This removes full control over the door dynamics, but allows for successfully passing closed doorways in narrow corridors.

Opening dishwashers (Figure 4.9d) Similar to opening doors in narrow corridors where the handle of an open door is very difficult to access, current robots cannot reach "under" an open dishwasher door. The same strategy works here: Pulling the door open until a certain degree, from which on the door is "pushed" down until it is sufficiently wide open. For this maneuver, the acting robot requires either long enough arms to push the door down with the end effector, or a mechanism that allows moving the torso sufficiently far down.

4.3.3 Task Stability: Pre- and Post-Poses in Robot Manipulation

It is common practice in robotics to define so-called "pre-" and "post-poses" for when robot manipulators act on objects. The prime example is grasping: Before actually enclosing a robot's gripper around an object, it assumes a pose close to the object. Ideally, the remaining path to fully wrap the gripper around the object is achievable by a linear motion in cartesian space. The same accounts for unhand-poses after placing an object or when opening containers and grasping their handles. The reason for this is simple: Without these intermediate waypoints, a motion planning movement controller would have to rely on perfect object model and perception data. Even 0.5cm difference in object position can result in the robot knocking over the object while directly approaching its grasp pose. When first going into a pre-grasp pose and safely navigating around the object, only a relatively simple linear motion is left to perform, greatly lowering the chance of accidentally destroying the manipulation scene. This adds to the overall stability of the task, while not semantically being part of it. Different motion controllers, morphologies, and tasks can require varying amounts and types of intermediate waypoints.

4.3.4 Plan Design for Reactive Task Monitoring in Mobile Manipulators

The common tasks described in Section 4.3.1 consist of idealistic step sequences a robot takes to accomplish the respective task. Effectively a large number of failures can occur that can lead to unanticipated task perturbations. To detect these failures and recover from the subsequent divergence from the expected course of action, critical subtasks must be checked regularly to detect anomalies early.

One example for this is transporting a grasped object and monitoring its correct position in the gripper. Heavy or difficult to handle objects can slip out of a robot's hand, resulting in

object loss. This either interrupts the task and requires recovery (e.g., picking up the object again), or renders the task impossible (the object is completely out of reach).

To monitor the object position in the robot's hand, a vision system can permanently watch it and notify the plan system about significant changes. This constantly allocates the vision system, making it unavailable for tasks like navigating in a dynamic environment or searching for further objects while moving. A better alternative is to monitor changes in the force profile of the grasp (as described by Kragic et al. [48]) and to then trigger a rudimentary check using the vision system.

4.3.5 Example: Autonomous Object Search Tasks

Object search is a very common task for autonomous robots. It is required for virtually any other task that involves objects, such as delivery, meal preparation, tablesetting, etc. The generalized plan for performing such a task, written as a CRAM function, has the following structure (BT version shown in Figure B.3):

```
(def-plan search-object (object)
  (let ((lazy-locations (possible-residence-locations object))
        (found nil))
    (while (and (not found) (has-solutions lazy-locations))
      (with-failure-handling
        ((location-unreachable (retry))) ;; On failure, try the next solution
        (when (has-solutions lazy-locations)
          (let ((current-location (next-solution lazy-locations))
                (approach current-location)
                (when (requires-articulation current-location)
                  (articulate current-location :open)) ;; Open container
                (let ((containing-volume (3d-volume current-location))
                      (look-at containing-volume :center)
                      (let* ((perceived-objects
                             (perceive-objects object containing-volume))
                           (matched-object (any-of perceived-objects
                                                    #'matches object)))
                                (unless matched-object ;; Leave open when object was found
                                  (when (requires-articulation current-location)
                                    (articulate current-location :close))) ;; Close container
                                (when matched-object ;; We are done
                                  (setf found matched-object)))))))
          found))) ;; Return either the found object, or 'nil'
```

A number of fundamental external functions take over the reasoning-part for this generalized plan function:

- (o) **possible-residence-locations**: Given the object type and the place the robot is currently located at (such as in a kitchen), there exists a number of possible places where the object could be. A cup might be on the table, in the cupboard, or in the dishwasher. While these are some of the most probable residence locations for cups in kitchens, it could be anywhere else: in the sink, the refrigerator, or the microwave. In order to reflect this in the generalized plan, these locations are identified by their characteristic of being a storage construct [94]: An orthogonal ontology in a knowledge base can define this property, allowing **possible-residence-locations** to infer that a microwave is possible, although less probable [79]. Recording EMs, the resulting probability distributions of which objects

were found where in what context can act as a statistical prior for these possible residence locations [103].

- (o) **approach**: In order to interact with any given object, the robot must first be brought in physical vicinity, i.e. it must approach the object in order to be able to reach it. Starting from a current location, arbitrary obstacles can hinder the robot from reaching its goal location. Path planning can partially help overcome this problem in both, convex and concave environments. Approaching an object therefore requires decisions on the path to take to reach an object (using e.g. waypoints), or signal a failure when the goal location cannot be reached.
- (o) **requires-articulation**: Objects that are stored in containers such as drawers, cupboards or cabinets require articulation in order to make them accessible. This can result in opening doors, or pulling open drawers.
- (o) **articulate**: While articulating a container holding an object, a robot must pay special attention to possibly blocking joints, obstacles hindering doors or drawers from opening, and must not get in its own way while articulating. Additionally, handle opening and closing trajectories must be calculated based on the global pose of the container, the joint axis of its opening, and the joint boundary limits.
- (o) **perceive-objects**: After the place where an object potentially resides has been made accessible, a perception routine capable of inspecting the containing space must judge its contents. Drawers have a different internal structure than e.g. refrigerators or cupboards, and require different approaches to meaningfully detect objects inside.
- (o) **matches**: All objects that are detected inside of a containing space must be checked for their fitness compared to the object in question. Lighting, obstruction, vague descriptions, and exogenous events can make the object differ from its expected image. Assuming plausible error margins can help identify these objects nevertheless, but open up the possibility for false positives.

The usage of external functions in the given **search-object** function explains them and their purpose pretty well. While some of them are very specific to the task at hand, such as **matches** and **possible-residence-locations**, the rest is rather common with other functions, too: **approach**, **requires-articulation**, **articulate**, and **perceive-objects** are valid functions for picking up or placing objects in other scenarios as well.

4.3.6 Example: Autonomous Fetch Tasks

Fetch tasks, as a common consequence of object search tasks, are the predominant activity primitive for autonomous robots. Fetching objects is at the core of tasks like table setting, tidying up rooms, or going shopping, but also feeding material into factory machines or supporting surgery. While there are examples of tasks without the need of fetching objects, it does act as a basic building block to enable everyday activity, for example fetching required tools to perform a completely different task.

A generalized plan that takes into account different object locations and grasp types is shown below. A lazy variant of the **search-object** plan is used to find all instances of objects that match the description of the **object** (BT version shown in Figure B.4).

```
(def-plan fetch-object (object)
  ;; Vague description is turned into lazy list real world references of object
  (let ((lazy-found (lazy-search-objects object)))
    (with-failure-handling
      ((manipulation-failed (retry))) ;; On failure, try the next solution
      (while (has-solutions lazy-found)
        (let ((current-object (next-solution lazy-found)))
```

```

;; Assumptions: Object approached, container (if any) is open
(let* ((valid-arms (or (value (context-constraints '(:valid-arms))
                                     :valid-arms) ;; Which fact to return
                       '(:left :right))) ;; Default value
      (lazy-grasps (lazy-calculate-grasps ;; Lazy list of grasps
                    current-object (handles current-object)
                    :valid-arms valid-arms))
      (constraints (context-constraints '(:max-tilt-angle
                                         :max-velocity))))
  (with-failure-handling
    ((manipulation-failed ;; Try the next grasp
      (when (has-solution lazy-grasps) (retry))))
    (let ((current-grasp (next-solution lazy-grasps))
          (perform-grasp current-grasp :constraints constraints))))))

```

Fundamental external functions common with the `search-object` plan are the ones handling lazy lists (`next-solution` and `has-solutions`). Moreover, `fetch-object` uses the following external functions for reasoning and actual manipulation:

- (o) `lazy-search-objects`: This is a slight variation of the `search-object` plan from Section 4.3.5 in which not only the first matching object is returned, but all objects detected matching successively. The function still handles searching for, and approaching objects, but switches to the next probable one when a new solution is requested.
- (o) `lazy-calculate-grasps`: A similar principle as in `lazy-search-objects` applies to this function: Based on the geometry of the object to grasp, its handles (i.e. its allowed grasp points) and the valid arms of the robot to use for grasping, grasps are calculated lazily such that this calculation is only ever performed when a new set of arm/grasp point relations is required. Calculating grasps can become (time-)expensive when the shortest arm travelling path or a lot of physical constraints are taken into account. Lazy lists of grasps — if properly implemented — never perform more calculations than actually necessary, with the drawback that no global optimum can be ensured this way. The latter would require all results to be available for the sake of comparison.
- (o) `context-constraints`: Contextual constraints can require specific modifications of otherwise generic actions. In this case the modifiers `:valid-arms`, `:max-tilt-angle`, and `:max-velocity` specify constraints on how to handle objects when holding them.
- (o) `perform-grasp`: This function executes the actual grasp. Grasping an object consists of multiple phases, starting with assuming a pregrasp pose (see Section 4.3.3), opening the gripper, assuming a grasp pose, closing the gripper, and finally retracting the arm under consideration of movement constraints. Any of these phases can fail due to for example physical constraints or controller failure, triggering a `manipulation-failed` failure that rewinds the performed actions, leading to trying the next possible grasp, if available.

When fetching objects and after having found a valid object, the main decision making matter is how to handle the object exactly. This includes deciding on what parts of the object to touch with which of the robot’s grippers, and taking into consideration contextual constraints. While the former is mostly geometric reasoning, the latter depends on both, the task itself, the object, and the current context.

4.4 Contextual Knowledge in Autonomous Robot Agents

Commonly, robot plans in cognitivist architectures such as CRAM are structured as hierarchical trees, allowing encapsulation and modular reuse. An underrepresented aspect in plan languages

is implicit context awareness. Tasks ought to perform differently based on their *explicit parametrization*, but also on *implicit circumstances*. These depend on dynamic factors, such as volatile perception knowledge, but also on static knowledge available through external knowledge bases.

Abstract generalized, modular plans are not supposed to cover all possible situations explicitly, but represent a generalist strategy that changes its structure and resulting behavior based on available knowledge. As briefly introduced in Section 4.2, I therefore introduce the `with-context` environment:

```
(with-context (c1 ... cn) code)
```

Within plans, a contextual parametrization for other plan building blocks they are using can be defined. Any layer can therefore either add new contextual parameters c_i , or alter old ones. The behaviour of single plan blocks in `code` can thus be influenced by a semantically higher hierarchy. One simple example that benefits from this approach is a context that describes the transport of liquid filled mugs. The result would produce different maximum tilting angles during motion planning than when transporting empty ones, given the assumption that nothing should be spilled.

In the following subsections, I will start with an example explaining how the `with-context` environment is used, going into detail about its implementation. I will then briefly discuss static and dynamic knowledge, and give an overview of where the different types of knowledge commonly originate from in robot systems.

4.4.1 Contextually Constraining Generalized Plans

This section motivates the usage of `with-context`, and its effect on lower-level plans. Here, the `fetch-object` task from Section 4.3.6 is constrained when transporting liquid-filled objects (e.g. coffee cups). In this plan, the non-lazy variant of the `search-object` plan is used. For the sake of clarity, if `search-object` is called with the description of an object already found, it just returns that instance.

```
(def-plan constrained-fetch-object (obj)
  (let* ((obj (search-object obj))
         (obj-state (perceive-state obj)) ;; Identifies facts of 'obj'
         (obj (or (when (has-fact obj-state :liquid-level)
                   (update-object
                     obj '(:liquid-level
                           ,(get-fact obj-state :liquid-level))))
                  obj)))
    (with-context '(:max-tilt-angle ,(kb-constraint :max-tilt-angle obj))
                  (:max-velocity ,(kb-constraint :max-velocity obj)))
      (fetch-object obj))))
```

The above plan is overly explicit. It does, however, show the implications of using contextual constraints: `fetch-object` is implicitly re-parametrized without mentioning those changes in any arguments to the function. `fetch-object` retrieves required constraints by calling `context-constraints` (again, see Section 4.3.6). In the above plan, `kb-constraint` returns the value of a specific constraint as reported by the knowledge base, based on a complete object description. A more general version of `constrained-fetch-object` incorporating all possible modifiers from an external knowledge base is shown below:

```
(def-plan constrained-fetch-object (obj)
  (let* ((obj (search-object obj))
         (obj-state (perceive-state obj))
```

```

(facts (mapcar (lambda (name) '(,name ,(get-fact obj-state name)))
              (fact-names obj-state)))
(obj (update-object obj facts))
(constraints (mapcar (lambda (name) '(,name ,(kb-constraint name obj)))
                    (fact-names obj-state))))
(with-context constraints
 (fetch-object obj)))

```

In this version, first all facts are extracted from the object state and inserted into the object description. Using this updated description, all constraints available from the knowledge base are collected. These constraints then parametrize `with-context`, and further implicitly constrain `fetch-object`. A possible implementation of `with-context` is seemingly simple:

```

(defvar *contextual-constraints* (make-hash-table))

(defun get-contextual-constraints ()
  *contextual-constraints*)

(defun set-contextual-constraints (constraints)
  (setf *contextual-constraints* constraints))

(defun context-constraint (constraint)
  (gethash constraint *contextual-constraints*))

(defun context-constraints (constraints)
  (mapcar (lambda (constraint) '(,constraint ,(context-constraint constraint)))
          constraints))

(defmacro with-context (constraints &body code)
  '(let* ((old-constraints (get-contextual-constraints))
         (new-constraints (make-hash-table)))
    ;; Make a copy of the existing constraints table
    (loop for h being the hash-keys in old-constraints
          do (setf (gethash h new-constraints) (gethash h old-constraints)))
    (dolist (constraint ,constraints) ;; Update inner scope
      (destructuring-bind (name value) constraint
        (setf (gethash name new-constraints) value)))
    (set-contextual-constraints new-constraints)
    (unwind-protect (progn ,@code)
      (set-contextual-constraints old-constraints)))) ;; Return to outer scope

```

4.4.2 Static and Dynamic Knowledge

I will briefly explain two types of knowledge a robot has at its disposal, what their origins are, and what they are used for: (1) Static and (2) dynamic (volatile) knowledge.

Static knowledge is all information known a priori that does not change during the course of a single robot task episode. Task episodes are enclosed sets of actions. If a piece of information, such as the opening degree of a drawer, changes in between two task episodes — opening the drawer and picking an object from it — the information about its collision model becomes quasi-static for the governing task, such as setting a table.

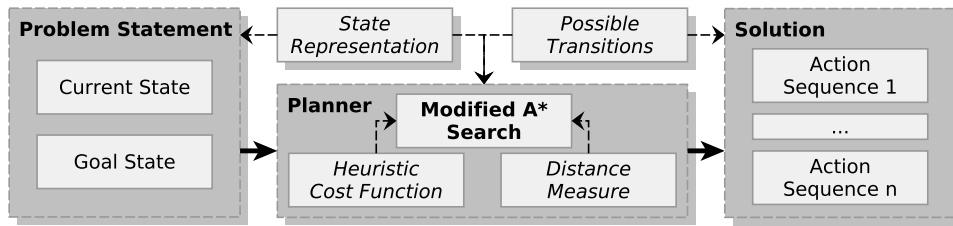


Figure 4.11: Architecture overview of the modified A* planner. Four components are domain specific (shown in *italic*): State representation, state transitions, heuristic cost function, and distance measure. The state representation is used in the problem statement and while planning, and the possible transitions are used during planning and when presenting the solution. The heuristic cost function and the distance measure are only used by the A* search algorithm itself.

robot reordering a shopping rack [100].

4.5.1 An A*-based Planner for Generalized Actions

Fetch and place actions are usually part of a larger task which builds on top of object transport. Common challenges include obstructed objects that need to be made accessible, and placing objects in a way such that they do not hinder placing other objects later on. To cover as many of such cases as possible, I designed the above mentioned planner such that it can:

- (o) Generate action sequences from a current scene state into a desired goal scene state,
- (o) Match generic goal states, with multiple states fitting the goal (multiple end-points), and
- (o) Produce multiple solutions, in descending quality order.

The resulting planner is very generic and can be applied to any graph search based problem. To apply it to a specific problem domain, four components need to be defined:

- (o) A state representation (e.g. shopping shelf state, meal table state),
- (o) Possible transitions between two states (e.g. pick, place, move base, lower/raise torso),
- (o) A heuristic cost function for state transitions, and
- (o) A distance measure between states.

The planner's general architecture is shown in Figure 4.11. Action sequences calculated with this planner consist of an ordered list of atomic transitions as defined for the domain, iteratively leading from the start state to the desired goal state. Ideally, a robot performing such a sequence can then linearly iterate over all entries, executing them one by one, ultimately arriving in the desired state. Due to the purely symbolic nature of the planner, it does not address reality's uncertainty in e.g. physical object repositioning tasks. Therefore, the action sequences can only act as prior strategies for physical robots. In-between steps a robot should always validate the perceived vs. the expected state of the scene. Upon divergence of the two, a new strategy can be planned based on the perceived state, recovering from object misplacement, misperception, and exogenous scene changes.

The fully implemented planner is available online¹ under the open source BSD license.

4.5.2 Example: Re-arrangement of Objects in Retail Shopping Racks

One application for a planner that generates action sequences for autonomous robots is the re-arrangement of arbitrary objects in a shopping rack. As mentioned in Section 4.5.1, four domain-specific components need to be defined in order to make use of the planner:

¹http://www.github.com/fairlight1337/shopping_scenario

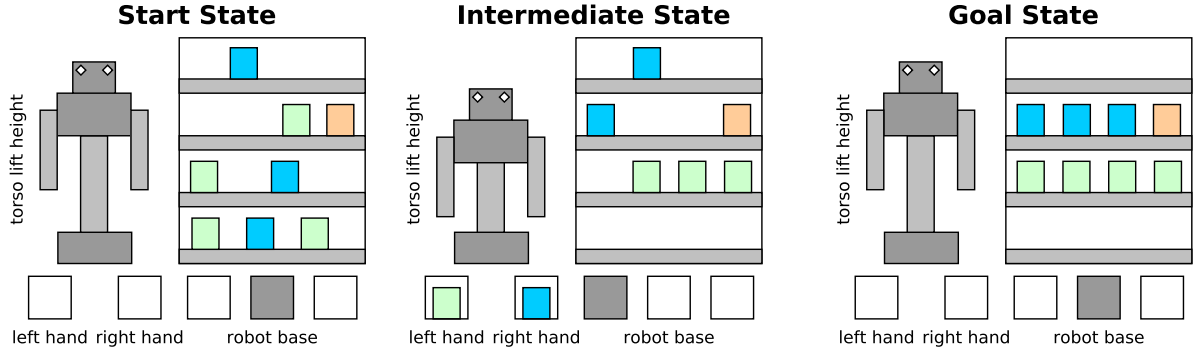


Figure 4.12: Internal planning representation of start, intermediate, and goal states of a shopping rack re-arrangement scenario. The performing robot can pick and place objects with both hands individually, hand over an object from one hand to the other, move its base to different fixed floor locations, and raise or lower its torso.

Action Primitive	Parameter(s)	Description
<i>pick</i>	<i>object, hand</i>	Picks up an <i>object</i> with the given <i>hand</i> .
<i>place</i>	<i>object, location</i>	Places the held <i>object</i> at the given <i>location</i> .
<i>handover</i>	—	Hands over a held object from one hand to the other.
<i>move-torso</i>	<i>height</i>	Moves the robot's torso to the given <i>height</i> .
<i>move-base</i>	<i>position</i>	Moves the robot's base to the given <i>position</i> .

Table 4.3: Action primitives available to a robot that autonomously re-arranges objects in a shopping rack.

- (o) **State Representation:** In principal, the shopping rack is divided in a number of shelf levels, offering space for a limited amount of objects on each level. A rack state represents the object occupancy in this rack at any given point, in addition to which objects are held in the robot's hands, where the robot's base is positioned, and at what height its movable torso currently is. Figure 4.12 depicts these internal representation states exemplarily.
- (o) **Possible Transitions:** To transform a rack from one state into another, a robot can perform a number of action primitives, as described in Table 4.3. Per transition, exactly one action is performed.
- (o) **Heuristic Cost Function:** The heuristic cost function of transforming a state S_0 into S_1 is given by $H(S_0, S_1)$ as shown in equation 4.1.

$$H(S_0, S_1) = \text{Number of misplaced objects in } S_0 \text{ relative to } S_1 \quad (4.1)$$

- (o) **Distance Measure:** The distance between two states S_0 and S_1 is the accumulated cost of the action sequence A_i , such that $A_i(S_0) \mapsto S_1$, under consideration of the atomic action weights w as shown in Table 4.4. Equation 4.2 represents the mathematical formulation of the distance measure.

$$D(S_0, S_1) = T_i^{S_0 \rightarrow S_1} = \left[\sum_{k=1}^n w(A_{i,k}) \right] = D(S_1, S_0) \quad (4.2)$$

$$D(S_0, S_1) \geq 0 \quad , \quad D(S_0, S_0) = 0 \quad , \quad D(S_0, S_1) \geq H(S_0, S_1)$$

Since $w \geq 1$, $H(S_0, S_1)$ never overestimates $D(S_0, S_1)$, as required by A*.

Action $A_{i,k}$	Pick	Place	Move Torso	Move Base	Handover
Weight w	1.2	1.2	2.0	1.0	1.5

Table 4.4: Atomic action weights for elements in an action sequence for re-arranging a shopping rack.

The parametrized planner produces action sequences that transform the shopping rack from its current state into any given goal state, if possible. The transitions as shown in Table 4.3 represent generalized plans — their exact manner of execution (and feasibility) depends on the context and the concrete task description. Instead of “*pick*” or “*place*”, any other generalized action such as “*fetch a glass*” or “*clean the living room*” are allowed in their respective scenarios as long as they fit the heuristic cost function and a state distance measure.

4.6 Context-dependent Failure Handling Strategy Selection

This section addresses strategies in generalized plans for selecting failure handling routines and parametrizations based on the failure and the current context. As already introduced in Section 6.1, failures can partially be addressed in the local context they occur in. The hierarchical character of generalized plans then allows each layer of the hierarchy to attempt to solve the problem, or escalate it.

The most simple strategy for dealing with failures is statically retrying: The task is re-attempted n times, a frustration limit, before it is escalated. Blindly retrying a task only has potential to work in non-deterministic settings, and can have three types of outcomes:

- (o) **Success:** The task is performed as intended
- (o) **No change:** The same failure occurs again, possibly with slightly different details
- (o) **Different failure:** A different failure occurs

Blind retries are a common fallback solution when no more elaborate approach is available. More intelligent — and knowledge enabled — strategies include the following:

- (o) **Reparametrization:** Domain knowledge about the task performed allows to deduce parameter ranges it expects. Together with additional contextual constraints (see Section 4.4), the allowed ranges can be narrowed down, and a possibly more feasible part of the search space can be explored. I explain how to do this in Section 5.5.1.
- (o) **Strategy change:** Several approaches can exist to alleviate any one failure. By trying different ones — keeping track of what did not work — a generalized plan raises the chances to successfully recover.
- (o) **Requesting external help:** An autonomous robot can request help (from humans or other robots) to transform the currently problematic situation into a manageable one. It must explain what the problem is and which parts it cannot solve itself.

The strategies suggested above can be selected based on rules stored in an external knowledge base. Three pieces of information are required for a proper strategy selection: (1) The failure itself, (2) the current context, and (3) the task description. In my work, I make failure handling strategies explicit when possible: A controller failure that was successfully detected in one of the robot arms always results in a restart of the controller once, and requests human intervention if this does not work. For more general situations, such as fetching an object resulting in no object being found, more intelligent strategies are required: Try a different cupboard, a different room, or try to substitute the object.

In the AI research field — and especially planning — failure detection and recovery is a whole subfield of its own. In my work, I deal with failures following the concepts presented above. For plan repair, partial and global replanning, and more specialized strategies I direct the interested reader to the respective literature [23, 25, 36, 67].

4.7 Summary

In this chapter, I motivated the current challenges for autonomous robots in human households, and subsequently sketched plan design principles for generalized, abstract robot action plans. I covered three main areas: (1) Mobile manipulation and its inherent challenges and pitfalls, (2) the role of contextual knowledge in autonomous robots, and (3) planning for generalized, abstract action descriptions.

I have described key concepts important for generalizing plans and for removing task and domain specific knowledge from action descriptions, most importantly being: (1) Role and handling of anticipated vs. unanticipated failures, (2) the difference between, and role of symbolic and subsymbolic knowledge, (3) local and global task recovery in uncertain situations, (4) common tasks and their structure for mobile manipulation, and (5) contextually constraining plans using static vs. dynamic knowledge. I implemented the presented concepts in a set of CRAM compatible macros and functions and presented code examples where appropriate.

The concepts described in this chapter lay the logical foundation for all chapters to follow. My work revolves about generalized plans, how they can be parametrized from autonomously collected robot experience, and how their connection to the real world is established without hindering their performance or versatility.

Data Collection and Experience-Based Learning

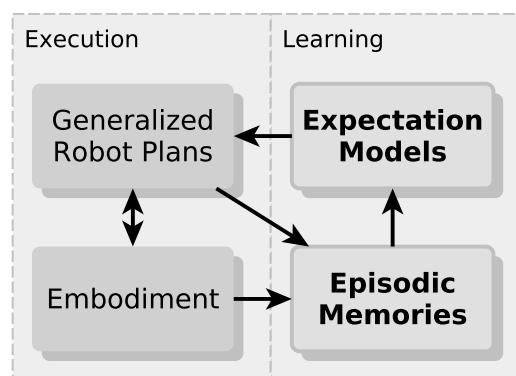
“Experience is simply the name we give our mistakes.”

— Oscar Wilde [98]

Even simple decision-making processes are based on mostly volatile information that is either derived from existing knowledge, or inferred from the situational context. While for robots the result of these processes is often visible in physical actions, their reasoning process stays opaque to outside observers. This has two drawbacks: (1) It is difficult to trace back the reasons that led to a decision, hindering the detection of behavioral anomalies, and (2) the reasoning model stays hidden, never allowing a robot to learn from its own decision making.

In this chapter, I will describe the benefits of experience collection for autonomous robots, and show the conceptual design and implementation of a framework for collecting Episodic Memories. These memories allow to answer high level semantic questions like *“Which steps did you take to achieve your goal?”* or *“Did unexpected failures happen?”*, but also more concrete questions like *“What are all positions you stood at when grasping succeeded?”* or *“In which cupboard are plates usually stored?”*. As a case in point, I will present my findings at the example of an autonomous robot setting and tidying up a meal table, what the resulting experiences are, and how they can be used to improve the robot’s behavior and performance. The framework I am presenting fully integrates with the concept of Generalized Plans from the previous chapter, and benefits from their design patterns. Using the collected experiences, I introduce action prediction models (Expectation Models) and how experiences can be generalized into a descriptive action model (Prototypical Experiences). Afterwards, I introduce a sophisticated framework for automatically collecting and archiving Episodic Memories from simulation, and automatically diversifying environment and task parameters for rich datasets.

Building on top of the Episodic Memories introduction in Section 3.2, two topics around robot experience are highlighted here: (1) The process of and tools for collecting experiences, and (2) machine learning techniques applicable to the resulting data. Based on the second topic, I will draw connections to how the performance of Generalized Plans is improved through experience. For all areas, I will introduce programming language constructs that enable and ease data logging where applicable.



5.1 Performance Enhancement through Experience

Acting autonomously in a dynamic environment requires making decisions under uncertainty. For robots, this uncertainty is grounded in two facts: Neither can a robot be 100% sure about

its knowledge of the world, and thus whether it is deciding based on correct information, nor can it be sure of the consequential success of its decisions. Humans make decisions based on a mixture of background knowledge and practical experience [71]. While we can supply today's robots with sufficient background knowledge about a task, they completely lack the ability to draw from experience or even form some sort of intuition from it. My claim is that robots increase their performance when having task experience at hand, being able to project the most probable outcome of their actions, and making decisions that have proven to work well. Gaining task experience basically means collecting information about one's doings, extracting rules and facts from it, and transforming these into a more general model of the task's characteristics, requirements, and consequences. In my research, I therefore collect all information utilized by a robot to make its decisions, the ever-changing world state as far as it is known, and the perceived consequences of the robot's actions, into EMs. I then generate experience models from these memories that influence the robot's decision making process. To achieve this, I developed a set of software tools that collect and process memory data, and have created an interface for CRAM robot plans [12] to make use of experience models.

The symbolic portion of all robot memories is encoded in the OWL format to make it semantically accessible. I use the KNOWROB framework [93] to access this data. This choice was made as KNOWROB offers a highly flexible reasoning ontology fit for the robotics domain and allows to formulate PROLOG queries based on symbolic as well as subsymbolic memory data when encoded in OWL. To date, no other framework that offers these capabilities in the domain of robotics or adjacent fields is known to me.

5.2 Machine Learning

Robot experience data is represented as task episodes that include all information collected while the robot performed a task. This data features a very broad content, from low level sensor information like laser scanners or fingertip pressure sensors to high level semantic data such as the robot's intentions, or the purpose of a trajectory it performed with its arms. Which parts of this information is important for optimizing ahead of performing a task is difficult if not impossible to deduce for human developers, given that those tasks are potentially novel, performed continuously, and are met with slight variations every time.

To identify relevant information from an episodic memory, I use different machine learning approaches. Which techniques are used depends on which layer of information to process: For low level, quantitatively abundant and subsymbolic data Reduced Error Pruning (REP) trees yield great results. High level symbolic data is processed using the C4.5 algorithm [73]. Both approaches result in decision trees that, based on statistical evidence, identify a causal relationship between recorded memory data and perceived effects in the robot's environment while performing a task. For data that structurally cannot yield a direct causal relationship, I developed a Mixed Multivariate Gaussian model that produces Gaussian distributions over an n -dimensional parameter space, learned from experiences, and able to give a good estimate of how probable the success of a parametrization is. Combined with knowledge about the task's general structure, which is also included in the episodic memories, I formulate Expectation Models (ExMods) [103] to optimize a plan's performance. These models are further extended and enhanced through reinforcement learning: When encountering new situations or failures when doing its duties, a robot gets more in-depth experience, resulting in more elaborate ExMods.

5.3 Methods in AI based Robotics

For my research, I borrow several concepts from classical Artificial Intelligence (AI) and research in AI based Robotics, the most prominent ones being:

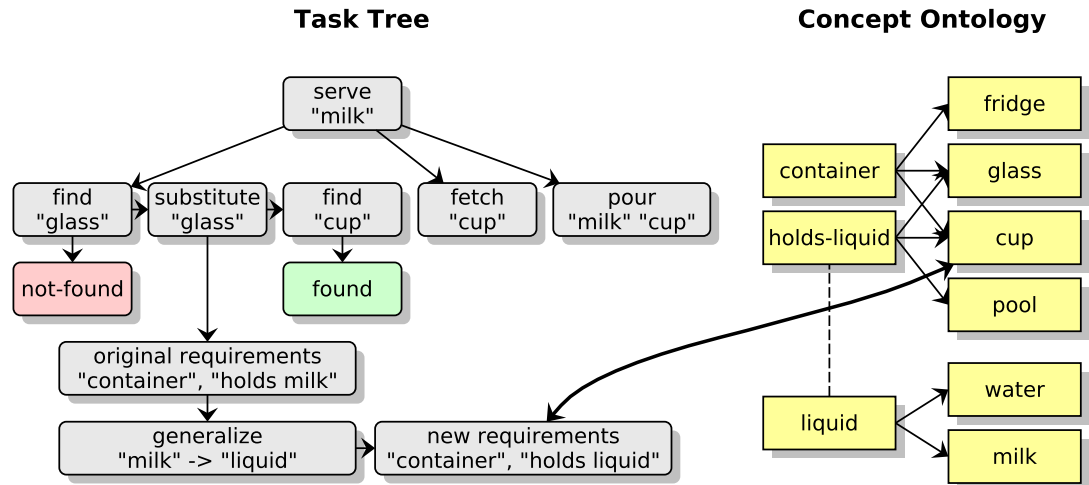


Figure 5.1: Example task tree for the task “serve milk” and the fitting concept ontology for household robots. In the task tree, the object concept “glass” was substituted by “cup” because it is — in the context of the task — a suitable replacement based on the properties “container” and “holds liquid”.

- (o) **The Hybrid Deliberative/Reactive Paradigm:** The actions to take are “planned” in advance, represented by a robot plan engineer manually designing elaborate generalized plans. These plans are then executed, reactively changing strategies and reparameterizing on the go as the situation requires it.
- (o) **Backtracking during Task Performance:** When one approach does not work, its effects are unwound until a state is reached in which an alternative approach becomes available. This lets a robot explore promising solution candidates more easily, while only performing the actually required real world actions to assess its options before performing an expensive switch to a different strategy.
- (o) **Unsupervised and Reinforcement Learning:** By collecting experiences and equipping a robot with the ability to judge whether it reached its end goal, it can supervise itself, removing the human operator from the reinforcement learning loop.

These concepts — being very abstract in their nature — are the underlying ideas of my generalized robot plans. Architecture-wise, a human designer gives the robot a head-start into how to perform a task on a very general level. It then explores its environment while following its duties, getting better over time at tasks it performs.

5.4 Anatomy of Episodic Memories

An Episodic Memory consists of a symbolic and a subsymbolic part. The symbolic part, mainly representing a hierarchical task tree, is strongly linked to an ontology describing all action classes, entities, and properties used in an EM. This ontology is orthogonal to the task tree, allowing reasoning in two dimensions: Based on task structure, and based on an entity-conceptual level. Figure 5.1 depicts these structures and gives an example of how a reasoning process incorporates these dimensions. The task tree is encoded as strongly-structured OWL data, holding all symbolic annotations relevant during execution. An example of how the task tree is stored is shown here:

```
<owl:NamedIndividual rdf:about="&log;CRAMAction_xSG6XCCFq3ptV06o">
  <rdf:type rdf:resource="&knowrob;CRAMAction"/>
```

```

<knowrob:taskContext rdf:datatype="&xsd:string">GRASP</knowrob:taskContext>
<knowrob:taskSuccess rdf:datatype="&xsd:boolean">>true</knowrob:taskSuccess>
<knowrob:startTime rdf:resource="&log;timepoint_1461715200.813"/>
<knowrob:endTime rdf:resource="&log;timepoint_1461715232.819"/>
<knowrob:objectActedOn rdf:resource="&log;object_Xo2qWwmXaJ1xVH"/>
<knowrob:arm>right</knowrob:arm>
<knowrob:graspDetails rdf:resource="&log;action_tAQ9QHrudOPee6"/>
</owl:NamedIndividual>

```

The above excerpt describes a node in the task tree of type **GRASP** — grasping an object. Here, the most important data is:

- (o) **taskContext**: String identifier, describing the type of node
- (o) **taskSuccess**: Boolean flag, signifies whether the task was successful
- (o) **startTime**: Time at which this node was entered (when the task started)
- (o) **endTime**: Time at which this node concluded (when the task ended)
- (o) **objectActedOn**: Which object was grasped/manipulated
- (o) **arm**: Which arm was used for grasping
- (o) **graspDetails**: Additional information (relative pregrasp poses, force applied, etc.), stored as a hierarchical tree of key/value pairs

Generalized plans can annotate the currently active node with any type of information: Atomic data (such as **arm**), larger, hierarchical records (such as **graspDetails**) or time points and time periods for marking portions of subsymbolic sensor data.

The subsymbolic part of an EM represents all continuous and binary data that cannot be represented symbolically. Its semantic meaning is defined in the symbolic data part, and the link between the two is established by filenames, time points, time periods, and kinematic link names. By its nature, this data comes in a much higher volume — in manners of magnitude — and requires special handling when storing it. The two most complex objects in terms of space consumption are camera image streams and still images, and kinematic robot link transformations. The former is addressed by using compressed versions of the image streams, at the cost of quality. This is passable as long as image processing algorithms can still properly analyze the data. The latter is stored by employing a throttling mechanism: Only significant changes in any one relative cartesian transformation are stored [105]. If the robot is not moving, virtually no data is stored. “Virtually” because — as the algorithms using this data interpolate in between data points — the transformations get stored every other second even if they did not change in order to have up to date data between interpolation points. Besides binary image data, the remaining subsymbolic portion of EM data is stored as JavaScript Object Notation (JSON) files. Every EM JSON file contains a distinct part of recorded data. In its default configuration, my experience collection mechanism records the following two parts:

- (o) **tf.json**: All kinematic configurations of involved robots and known articulatable furniture (drawer joints, fridge doors, etc.); in their purest form (an identity transformation between the coordinate systems “*world*” and “*robot_base*” at time 0), these assets have the following format:

```

{header: {frame_id: "world", stamp: {date: 0}},
  child_frame_id: "robot_base",
  transform: {translation: {x: 0, y: 0, z: 0},
             rotation: {x: 0, y: 0, z: 0, w: 1}}}

```

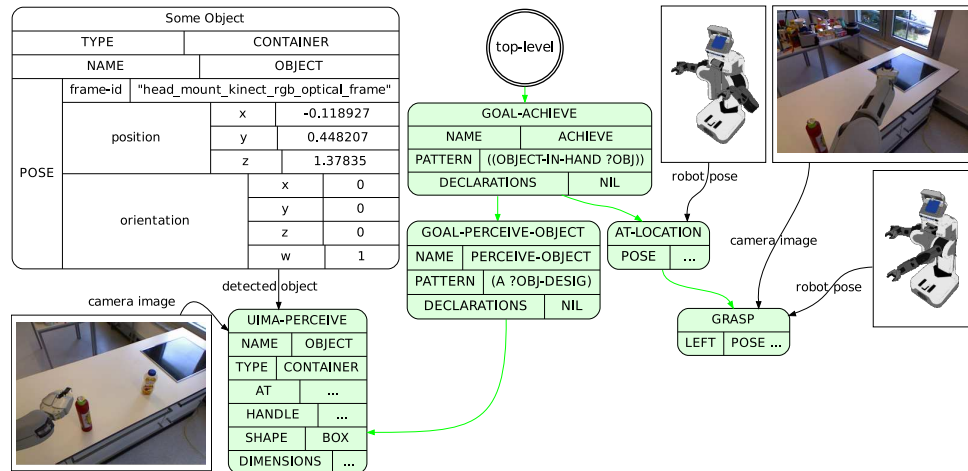


Figure 5.2: Example task tree visualization from the Graphviz *DOT* file generated for every recorded experience. It includes the task hierarchy, camera images, robot poses, and object details as applicable.

- (o) **logged_designators.json**: Hierarchical parameter configurations (trees of key/value pairs) for task parametrizations, descriptions of perceived objects, request/response pairs of communication with external components (e.g. perception system), etc. The stored format is similar to the above, but describes different entities.

To give a first impression of any recorded experience, the task tree is additionally stored as a Graphviz *DOT* file, which in turn can be used to create *PDF* files showing the hierarchical tree. Figure 5.2 gives an impression of that visualization. For key moments, such as before and after grasping, or when requesting information from the perception system, camera images and robot poses are annotated as applicable. Whenever available, object annotations are added for tasks supplying them.

5.4.1 How Episodic Memories are Recorded

To efficiently and reproducibly generate robot experiences in a standard format, I designed and implemented the Semantic Hierarchy Recorder (SEMREC)¹ software, and added vital extensions to the `mongodb_log`² software.

SEMREC offers input interfaces for cognition-enabled architectures to report the actions that are performed, collects and organizes this data in a hierarchical tree, and is able to generate a number of output formats. Using its plugin system, SEMREC can be extended with any input, output, or processing components necessary. A number of plugins is already available³, covering all needs of household robot experiments. The symbolic part of any of my robot experiences is constructed using this software. For a cognition-enabled architecture to make use of SEMREC, it needs to notify the logging system about tasks it begun, finished, their parametrizations, and any other data the final EMs should contain. Using an Remote Procedure Calling (RPC) interface, any client can add information to the recorded log. The most prominent and most used calls are shown in Table 5.1. Its asynchronous nature allows SEMREC to process all data from an architecture in the correct order without slowing down plan performance (no wait time for the caller).

¹<https://github.com/fairlight1337/semrec>

²https://github.com/fairlight1337/ros-mongodb_log

³https://github.com/fairlight1337/semrec_plugins

Signal Type	Description
<code>begin-context</code>	Starts a new task context, with start time and parameters
<code>end-context</code>	Ends a running task context, with end time, result, ending all children
<code>add-designator</code>	Adds a designator reference to the current task, optionally with a purpose (e.g. “ <i>perceptionRequest</i> ”, “ <i>perceptionResult</i> ”)
<code>equate-designators</code>	Equates two designators, linking them together in a linear list
<code>annotate-parameter</code>	Adds a symbolic, named parameter value to the current context
<code>add-failure</code>	Adds a failure instance to the currently active task context
<code>catch-failure</code>	The current context caught a failure and tries to handle it
<code>rethrow-failure</code>	The current context was not able to handle a caught failure, rethrowing it
<code>add-object</code>	Adds an object to the current context, optionally with a purpose (e.g. “ <i>objectActedOn</i> ”, “ <i>perceptionRequest</i> ”)
<code>add-image</code>	Captures an image from the robot’s camera, adds it to the context
<code>start-new-experiment</code>	Clears all of SEMREC’s buffers, and creates a new experiment space
<code>export-planlog</code>	Triggers all output plugins to create files based on the recorded data; this command creates resulting OWL and DOT files
<code>set-meta-data</code>	Sets optional meta data fields that are included in output files

Table 5.1: The most prominent RPC calls offered by SEMREC for recording symbolic data. This interface is called by a robot’s executing architecture.

The very idea of the Generalized Plans from Chapter 4 is that as little information as possible should be supplied to achieve a maximum of autonomous behavior. The same principle applies to automatic experience collection from executing these Generalized Plans: Beginning and ending contexts, annotating them with arguments, and adding objects created within are not part of the robot’s plan tasks. I designed and implemented a transparent interface for the CRAM architecture that seamlessly integrates with any active SEMREC instance. This interface is implemented as the native CRAM package `cram_beliefstate`⁴. When this package is added to any CRAM project, all relevant CRAM functionality is automatically equipped with the necessary calls and data transformations. A developer designing generalized robot behavior has no notion of its presence beyond including it in the project. Algorithm 5.1 shows a simplified example implementation of a logging supported plan definition macro. The logging related functions therein are marked in red, and begin a new context, add their respective information to the currently active context, and end the context again. An example use-case of this macro is given below the macro. All nested plans (`fetch`, `search`, `approach`, and `grasp`) make up a hierarchical context structure — the symbolic task tree.

`mongodb_log` was originally presented by Niemüller *et al.* [61]. It is used for recording ROS topic data streams into a MongoDB database and features both, generic logging mechanisms for arbitrary data as well as fast, specialized loggers for individual topic types. I extended⁵ this software by means of fast, structured designator logging (resulting in the content of the `logged_designators.json` files) and data throttling mechanisms when recording kinematic transformation data. As in practice I use ROS for communication between robot components, this software is a natural fit. The same experience data can be recorded for other systems too, though. The low level JSON data format contained in EMs is easily reproducible from any source with the same information content.

An example of the data included in EMs is shown in Figure 5.3. Shown therein is the hierarchical task tree, annotated Relevant Task Parameters (RTPs), meta data, and task outcomes. Especially important is the information about failure handling: Which task failed due to what

⁴https://github.com/fairlight1337/cram_beliefstate

⁵https://github.com/fairlight1337/ros-mongodb_log

Algorithm 5.1 Example implementation of a plan definition macro that automatically adds logging capabilities. A simplified example for fetching an object is given below the macro.

```
(defmacro def-plan (name arguments &body code)
  `(defun ,name (&rest argument-values)
    (let ((log-id (begin-context ,name))) ;; Begin the logging context
      (annotate-argument-list ',arguments) ;; Annotate the original argument list
      (mapcar (lambda (argument value)
                (annotate-call-parameter argument value) ;; Annotate argument values
              ',arguments argument-values))
      (unwind-protect ;; Make sure 'end-context' is called even when a failure occurs
        (let ((result (progn ,@code)))
          (annotate-result result) ;; Annotate result
          result)
        (end-context log-id)))))) ;; End the logging context

(def-plan fetch (object) ;; Plan for fetching an object using sub-plans
  (let ((found-object (search object)))
    (cond (found-object (approach found-object)
                       (grasp found-object))
          (t (throw-failure :object-not-found)))))

(def-plan search (object) [...]) ;; Plan for searching an object
(def-plan approach (object) [...]) ;; Plan for approaching an object for interaction
(def-plan grasp (object) [...]) ;; Plan for grasping an object
```

failure, what was the reparametrization, and which task handled the problem? In the example, a *navigate* task for manipulation failed, was retried, and succeeded. This information helps generating Experience Models — mainly for action effect prediction.

5.5 Generating Experience Models from Episodic Memories

Experience Models are the results of any algorithm processing EMs for the sake of improving a robot’s behavior. While any type of processing algorithm could achieve this, I concentrate on machine learning techniques that create models correlating a robot’s behavior and the observed action effects. These models serve the main purpose of making predictions about a task’s course of action, and extracting generalized knowledge about causal relationships between a task’s context, parametrization, and outcome. In general, the more experiences used for creating a model, the more generalistic and versatile the model becomes.

In this section, I present three concepts for (1) increasing a task’s success rate and overall performance, (2) generalizing a task’s course of action, thus allowing a robot to explore alternative task approaches in an informed way, and (3) learning new, complex PDDL actions based on experience and primitive actions. Additionally, I give an overview of common pitfalls when generating experience models, and discuss how to avoid them.

5.5.1 Expectation Models: Task Outcome Prediction

Expectation Models (ExMods) are generalized representations of robot experiences that allow prediction of task success [103]. They also allow suggestion of parameter ranges suitable for reaching a desired task outcome — be it success or a specific type of failure. The more experiences a robot collects and the more corner cases and regions of the parameter search space it explores, the more precise and elaborate ExMods become. They form an “*intuition*” for robots, giving them informed priors when parametrizing their actions. The overarching prediction framework in which ExMods play the central role is shown in Figure 5.4. It closes an information feedback

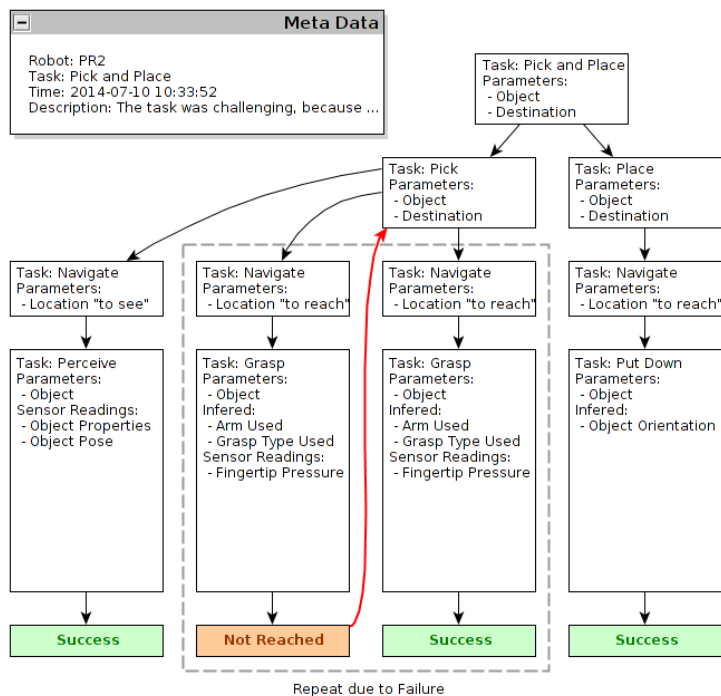


Figure 5.3: Structure and Content of a single Episodic Memory instance, consisting of meta data, semantic task tree, thrown and caught failures, task parameterizations, and outcomes.

loop for improving plan performance:

1. Robot plans are annotated with RTPs, the parameters considered (tracked and suggested) by ExMods.
2. EMs recorded from performing these plans are used as the basis for creating ExMods.
3. ExMods allow a performing robot to do two things: (1) Track its current progress of the task at hand, and (2) make informed choices for parameter ranges that lead to a desired task result.
4. The original plans are executed again with improved — more informed — plan performance.

Multiple EM records of the same plan, or even structurally different plans, can be successively added to an existing model. In the remainder of this section, I describe the anatomy of an ExMod, how it is created from EMs, how it is applied to generalized robot plans, and what predictions it allows. I offer code examples where possible and draw links to the preceding chapters where I see fit.

Anatomy of an Expectation Model

Expectation Models consist of two elements: A task tree reflecting the combined structure of all source EM's task trees, and a collection of decision trees — one per task type encountered, describing the observed effects of experienced parametrizations. Figure 5.5 shows a schematic example of a generalized task tree and what it is composed of: Two EMs of different tasks (setting a table and loading a dishwasher) are represented by structurally different, although partially similar task trees. Based on their similarity in paths from the root node, they are combined into a more general, compound model. The resulting task tree, if traced from the root node, can now be used to *track* both tasks, recalling specific experience details for each — e.g.

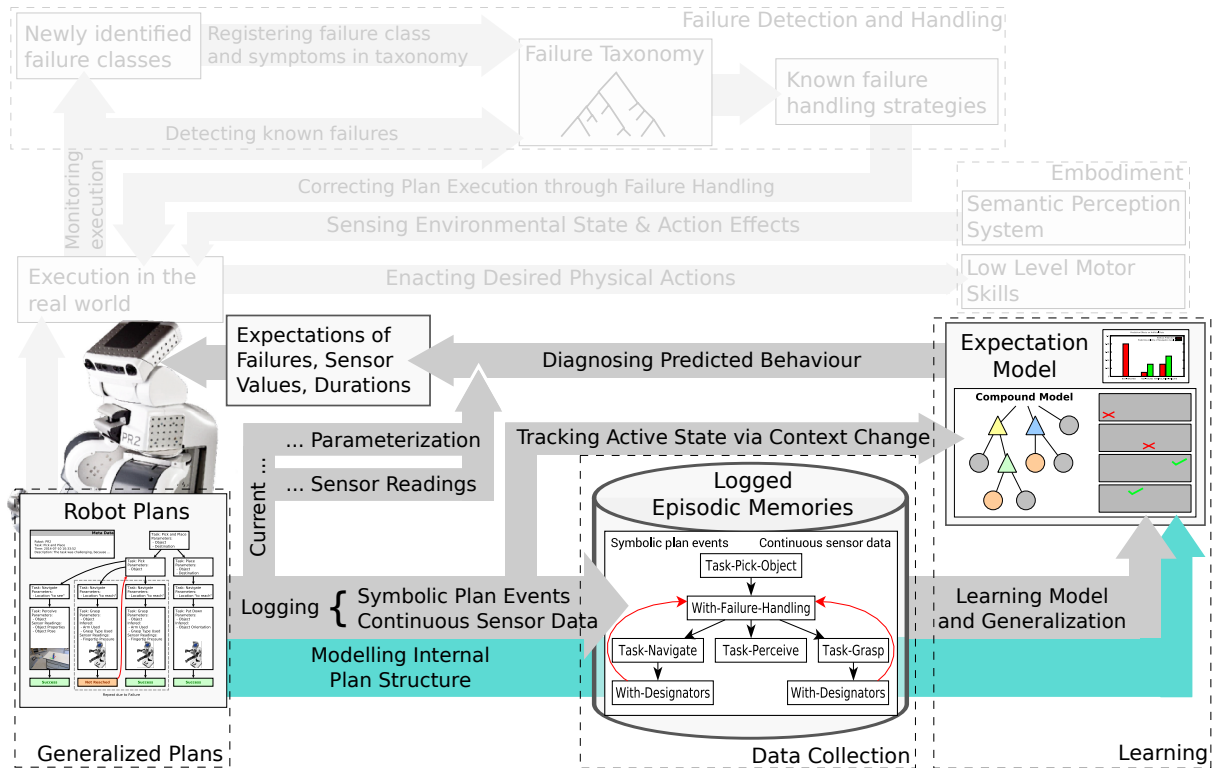


Figure 5.4: The architecture of a robot’s prediction capabilities, based on ExMods. This figure highlights their distinct role in the overall architecture.

possible next steps, prior parametrizations, past outcomes, etc. Using these stored correlations, decision trees are calculated that identify the most prominent relationships.

Figure 5.4 shows the principle role of ExMods in their prediction framework. Their input is a set of logged EMs. After generating the models, they offer two types of functionality: (1) Tracking the current task execution, such that the performing robot knows *where* in the task’s course of action it currently is, and (2) what the outcome of a given parametrization is or which ranges of values to choose every parameter from in order to reach any desired outcome. All of these will be addressed in detail later in this section.

Creating Expectation Models from Episodic Memories

ExMods are created from (and for the purpose of supporting) decision making processes based on volatile information (see also Section 4.4.2). In order to create ExMods for robot plans, the following steps are required:

- (o) **Identifying a plan’s RTPs:** Relevant Task Parameters are significant factors for decision making when performing a plan. As every sub-plan in a hierarchical task tree has a semantically well-defined purpose, these modularized plans can annotate any parameter or assumption that is required for their task and is not a pure flow-control operator (loop variables, while conditions, etc.). Compare also Section 4.1.2 for parameters influencing strategy selection in generalized plans. Examples of meaningful parameters are which hand to use for grasping, the current distance between a robot and an object during visual examination or manipulation, or the order in which objects are placed on a table. Plans need to mark parameters as relevant when recording EMs. Beyond this, no additional annotation is required. Algorithm 5.2 shows a simplified example of this process — note here that the annotated RTPs are not correlated beyond the fact that they belong to the same task.

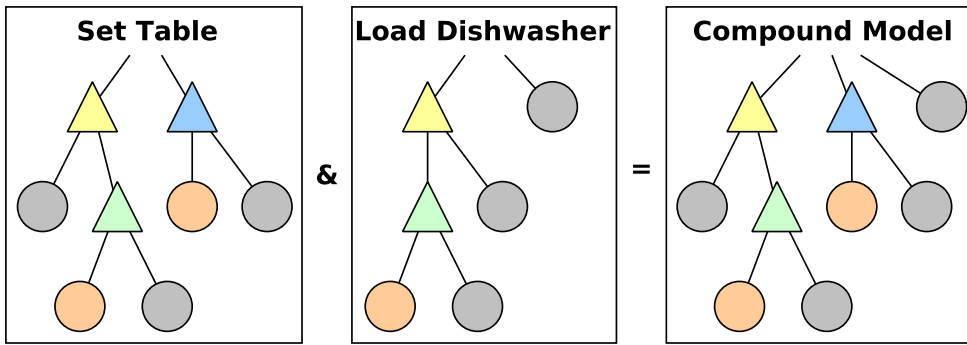


Figure 5.5: Generalized task tree from Episodic Memories, combining two structurally different tasks: Setting a table and loading a dishwasher.

Algorithm 5.2 Simple plan for annotating RTPs during a decision making process for object search. The exemplified ‘def-plan’ macro from Section 5.4.1 ensures that all information is properly recorded into EMs.

```
(def-plan probable-object-residence-location (object-type)
  ;; To be more versatile, the actual locations should be read from a knowledge base
  (let ((location (cond ((string= object-type "Milk") "fridge")
                       ((string= object-type "Bowl") "cupboard")
                       ((string= object-type "Fork") "upper-drawer")
                       (t "table"))))
    (annotate-rtp :object-type object-type) ;; Annotate 'object-type' RTP
    (annotate-rtp :location location)      ;; Annotate 'location' RTP
    location))
```

- (o) **Performing a plan under parameter variance:** After equipping a plan with RTP annotations, it must be performed a number of times to explore the available parameter search space. The more variance the recorded EMs contain, the more precise and versatile the resulting ExMod becomes. *Parameter variance* marks changes in the plan’s RTPs, which may initially be random to ensure an even exploration of their possible value ranges.
- (o) **Generalizing task trees of all EMs:** Two or more EMs may be condensed into a generalized task tree. As introduced in Section 5.5.1, structural similarity is of minor importance, and any two EMs independent from their task can be combined. Algorithm 5.3 shows a Pseudo-code representation of the generalization algorithm. A common root node is created from which any start nodes of the task trees to combine branch off. Already existing node paths are reused when adding a new EM task tree, and new nodes are added where necessary. Every node additionally contains a combination of past input/outcome pairs together with the “active” RTPs handed down from parent nodes — the context in which any pair was recorded.
- (o) **Generating decision trees for outcome predictions:** The plan conglomeration from Algorithm 5.1, when equipped with the respective RTPs for current global robot x and y position and its relative distance to the object of interest, produces EM data similar to what is shown in Table 5.2. This data is extracted from stored RTP annotations within any node in the generalized task tree. Each of the three plans shown here can either be successful or yield a distinct failure. Additionally, the *obj-dist* parameter is only available in the `grasp` plan. The ExMod decision trees are computed using the J48 classifier implemented in Weka [42]. J48 is an open source implementation of the C4.5 algorithm [73], which supports pruning of learned decision trees to a set of majority classes. The

Algorithm 5.3 Task Tree Generalization

```

1: function COMBINETREES(treeList)                                ▷ Common root node structure
2:   return {ctx : Toplevel, trees : treeList}
3: end function

4: function UNIFYTREE(tree)                                       ▷ Unify into a single tree
5:   for sub-tree in tree do
6:     unified-sub-tree ← UnifyTree(sub-tree)
7:     sub-tree ← CombineTree(unified-sub-tree)
8:   end for
9:   return tree                                                 ▷ tree is mutable
10: end function

11: function COMBINE TREE(tree)                                     ▷ Collect Information from task tree instance tree
12:   ctxTree ← {}                                               ▷ Dictionary (key: The current task type)
13:   for sub-tree in tree do
14:     for ctx in sub-tree do                                   ▷ ctx is the task type, or "context"
15:       gParams[ctx] ← ctx[param]                             ▷ Collect RTPs
16:       gTrmnls[ctx] ← ctx[trms]                               ▷ Collect terminal states (no child nodes)
17:       ctxTree[ctx]  $\stackrel{\pm}{\leftarrow}$  sub-tree                       ▷ Append this subtree to dictionary by task type
18:     end for
19:   end for
20:   return ctxTree
21: end function

```

Task	<i>x</i>	<i>y</i>	<i>obj-dist</i>	Result
Search	7	5	?	Success
Search	8	9	?	ObjectNotFound
Approach	4	0	?	Success
Approach	8	5	?	LocationNotReached
Grasp	5	4	2	Success
Grasp	2	3	3	ManipulationFailure

Table 5.2: Example training data for decision trees, fitting the plan structures in Algorithm 5.1.

training features I use herein are: (o) The task type, (o) all RTPs, and (o) the task result (success vs. failure class). The latter becomes the set of majority classes in the decision tree. Missing parameters, such as *obj-dist* for non-**grasp** tasks are replaced by J48 using appropriate values from other classes to minimize their influence and thus the resulting error.

One possible resulting decision tree from the training data in Table 5.2 is shown in Algorithm 5.4. Parameter ranges are identified for all task types that lead to their specific failure classes. If none of these are met, the task is implicitly marked as successful. One major advantage of this model element is its interpretability: The generated rules can easily be judged by humans, identifying anomalies or just for understanding the intrinsics of the system.

The generated ExMod is then stored in JSON format for easy reuse in the prediction framework, and for manual inspection. ExMods can either be created (1) per task to specialize on its distinct intrinsics, or (2) in a compound task model, generalizing over all different tasks experiences. The former has an advantage in only including parameter ranges that actually fit that task at hand, while the latter can make use of synergies in structurally semi-equal task

Algorithm 5.4 ExMod decision tree generated from EMs, partially based on the training data in Table 5.2.

```

1: Result ← Success                                ▷ Implicit Success
2: if task = Search and ( $x > 7$  or  $y > 7$ ) then
3:   Result ← ObjectNotFound                        ▷ Failure
4: else if task = Approach and ( $x > 7$  or  $y > 5$ ) then
5:   Result ← LocationNotReached                    ▷ Failure
6: else if task = Grasp and obj-dist > 3 then
7:   Result ← ManipulationFailure                  ▷ Failure
8: end if

```

trees. Depending on the use-case, a plan designer has to choose which strategy suits her needs the most.

Applying Expectation Models to Robot Plans

Given a current parameterization and an ExMod’s decision tree, the prediction framework can make educated guesses about the most probable outcome of the current situation. A robot then knows which failures, sensor values, task durations, and results to expect, and can decide to reparameterize its plans if necessary. To make use of ExMods for predicting action effects in generalized plans, I introduce four robot plan language constructs:

- (o) **with-expectation-model**: Overarching, nestable environment that initializes a new, or loads an existing ExMod. The subordinate language constructs use this model for their respective functionalities. Example for loading (or creating) an ExMod by the name *tablesetting*:

```
(with-expectation-model tablesetting
 [...])
```

Different tasks can be parametrized with separate ExMods:

```
(with-expectation-model picking      (with-expectation-model placing
 (achieve '(object-in-hand ?o)))    (achieve '(object-placed-at ?o ?d)))
```

- (o) **with-tag**: Helper-environment giving code segments in plans semantic names. These names are directly reference-able during effect prediction (e.g. *"What would be the outcome of subtask grasp with this particular parametrization?"*). Example:

```
(with-tag fetch
 (with-tag search [...])
 (with-tag approach [...])
 (with-tag grasp [...]))
```

- (o) **predict-behavior**: Given the current position in the known, tracked task tree and the currently active parametrization, a list of probable outcomes of the current task is generated; they are returned along with, and ordered by their relative probability. Example for predicting the current task’s outcome:

```
(predict-behavior)
```

Or, when referencing a particular subtask *"grasp"*:

```
(predict-behavior grasp)
```

Embedded in a larger plan, it can be used to signal a failure when failures are predicted:

Algorithm 5.5 Example plan using the `choose` operator for selecting parameters from ranges proven suitable by experience. The chosen parameters are the arm to use for grasping an object, and the grasp type to apply.

```
(def-plan grasp (object)
  (with-tag grasp-object ;; Allow targetted predictions for this task
   ;; Annotate all relevant RTPs, allowing the ExMod framework to generate
   ;; correlations between them and the chosen parameters
   (annotate-rtp :object-type (get-value object :object-type))
   (annotate-rtp :object-orientation (robot-object-orientation object))
   (annotate-rtp :object-distance (robot-object-distance object))
   (choose ((arm (any-of-random '(:left :right)))
            (grasp-type (any-of-random '(:push-grasp :top-grasp
                                         :power-grasp :pinch-grasp))))
            :satisfying ((manipulation-failed 0.2) ;; 20% manipulation failure OK
                        (pose-unreachable 0.15)) ;; 15% pose unreachable failure OK
            :attempts 5 ;; Try it 5 times before giving up
            (with-context ([:grasp-with-arm arm] [:grasp-type grasp-type])
                          (execute-grasp object)))))) ;; Starting the physical grasp
```

```
(def-plan fetch-and-place (?o ?d)
  (unless (success? (predict-behavior))
    (fail 'failure-predicted))
  (achieve '(object-in-hand ?o))
  (achieve '(object-placed-at ?o ?d)))
```

- (o) `choose`: Variable-binding environment that consults the active ExMod for suitable parameter ranges to successfully perform the current task. Example:

```
(choose (( $p_1$   $func_1$ ) ( $p_2$   $func_2$ ), ...)
  :satisfying (( $f_1$   $t_1$ ) ( $f_2$   $t_2$ ), ...)
  :attempts  $n$ 
  code)
```

This operator generates task parameter values for the variables p_i using the functions $func_i$. After all p_i have been generated, this parameterization is used for predicting the current task's outcome. If the outcome satisfies the failure tolerances (maximum relative occurrences of failures) specified as (f_i (failure) t_i (tolerance)), `code` is executed with these parameters. If tolerances are violated, new parameters are generated and checked. This process happens at most n times to avoid deadlocks in impossible situations, signalling a failure when exhausted. Once the task code gets executed, `choose` automatically annotates all p_i as RTPs in the Episodic Memory of the current execution.

An example plan describing how to use the `choose` operator when grasping an object is shown in Algorithm 5.5. Therein, the robot/object relative orientation and distance as well as the object type are annotated as RTPs, while the `choose` operator decides on which arm to use for grasping and which grasp type to apply. The chosen parameters are then included in the grasp context using the `with-context` environment, as described in Section 4.4.1.

Tracking and Prediction in Unknown Tasks

An ExMod serves as a blueprint showing what the course of action of a known task should look like. During execution, the plan language structures that record task hierarchies into EMs are the same ones that track the position of the current execution in the compound task tree. The mechanism descends into the hierarchical model tree when new tasks are entered and ascends

Algorithm 5.6 Computing Task Result Probabilities

```

1: function RECURSESUBTREE(node, params)
2:   probs  $\leftarrow$  DTreeProbabilities(node, params)
3:   for subnode in node[subnodes] do
4:     subprobs  $\leftarrow$  RecurseSubTree(subnode, params)
5:     for p in subprobs do
6:       probs[p]  $\stackrel{\pm}{\leftarrow}$  subprobs[p]  $\cdot$  probs[success]
7:     end for
8:   end for
9:   return probs
10: end function

11: function DTREEPROBABILITIES(node, params)
12:   probabilities  $\leftarrow$  {}
13:   numIndivs  $\leftarrow$  count(node[individuals])
14:   for individual in node[individuals] do
15:     dTreeResult  $\leftarrow$  EvaluateDTree(params)
16:     probabilities[dTreeResult]  $\stackrel{\pm}{\leftarrow}$   $1/\textit{numIndivs}$ 
17:   end for
18:   return probabilities
19: end function

```

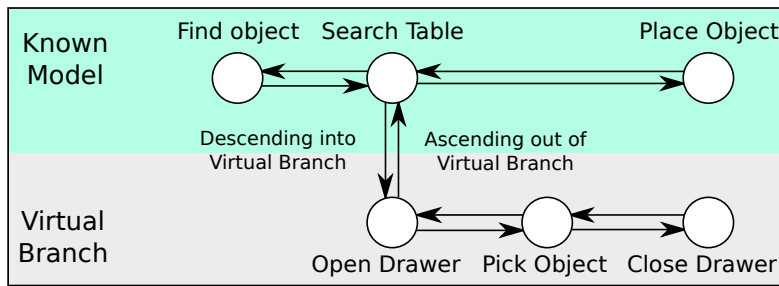


Figure 5.6: When unknown task types are encountered, a virtual branch is opened and the known ExMod is left. No predictions are possible until the known model is re-entered.

out of subtrees when the tasks finish. As generalized models can include multiple structurally different task types, their common tree diverges at some point. The tracking mechanism will descend into the subtree fitting the type of the task currently being performed, allowing models of diverse task hierarchies, without intermixing their distinct structural features.

When new tasks are faced, the tracking mechanism might encounter tasks not present in the known model, as they were not part of prior executions. In this case, it keeps track of where it left the known model and descends into a virtual branch, denoting task types along the way. In practice, the tracker creates a new sub-branch every time a new context is entered when in a virtual branch, accepting any kind of task (see Figure 5.6). When ascending out of the unknown subtree, the virtual branch is resolved until the known model is re-entered. While outside of the known task model, prediction only yields that the current task is not known.

Predictions Based on Relevant Task Parameters

To predict the outcome of a task, its sub-tree in the ExMod is selected and all of its branches are traversed. For each node, the model’s decision tree is used to compute the potential outcomes based on the given parameters. The result is a table of potential failures, their joint probability over all subtasks, and the overall assumed chance for success. Algorithm 5.6 shows a pseudo code version of this. `RECURSESUB` is called with the task node *node* to start predicting for, and

Algorithm 5.7 Inverted decision tree for choosing parameters using the `choose` operator

```

1: Rule ← {}
2: if Result = ObjectNotFound then
3:   if task = Search then
4:     Rule ← (x > 7 or y > 7)
5:   end if
6: else if Result = LocationNotReached then
7:   if task = Approach then
8:     Rule ← (x > 7 or y > 5)
9:   end if
10: else if Result = ManipulationFailure then
11:   if task = Grasp then
12:     Rule ← (obj-dist > 3)
13:   end if
14: else if Result = Success then
15:   if task = Search then
16:     Rule ← (x ≤ 7 and y ≤ 7)
17:   else if task = Approach then
18:     Rule ← (x ≤ 7 and y ≤ 5)
19:   else if task = Grasp then
20:     Rule ← (obj-dist ≤ 3)
21:   end if
22: end if

```

the current parameters *params*. The function `EVALUATEDTREE` evaluates the model’s decision tree using these parameters.

An explicit parameter to outcome relation model cannot only be used for value validation (such as in `choose`). By inverting the decision tree in Table 5.4, we get a decision tree for value ranges to pick parameter values from, as shown in Algorithm 5.7. The *task* variable is not used as a relevant task parameter during the inversion, as the robot cannot alter it during plan execution. By inverting the tree, parameters can be chosen much more efficiently from a range that has proven useful, as opposed to randomly choosing a parametrization and validating it.

To predict task characteristics other than task results, decision trees can be computed and evaluated in the exact same way, just changing the target class. This can be for example the expected duration of a task based on parameters. J48 only computes decision trees for nominal classes. For numeric classes, such as the duration, we use the `REPTree` classifier, also implemented in Weka. Virtually any parameter represented by a nominal or numeric class can be predicted this way, including sensor readings, e.g. robot gripper forces, as long as it is annotated as an RTP.

5.5.2 Prototypical Experiences: Informed Strategy Exploration

Prototypical Experiences (yellow in Figure 5.7) are generalized representations of all memorized situations of performing a task. Figure 5.8 shows the Prototypical Experience (PE) of a Fetch and Place activity in which a robot was to find and pick up objects, take them out of drawers, fridges, and from tables, and place them at similar places. In principle, these tasks and their experiences all have the same structure: Find an object (possibly opening containers to look into), pick the object, go to where it should be placed, potentially open a container to put it into, and place it there. Whether a container needs to be opened solely depends on the context, the object, and the task description (“put the milk into the fridge”). Given enough memories of different situations while performing this task, the depicted Prototypical Experience is obtained: All tasks share a common “root” of going to a place at which the object is expected, sometimes opening a container (denoted by “optional” steps in the PE), picking the object, possibly closing

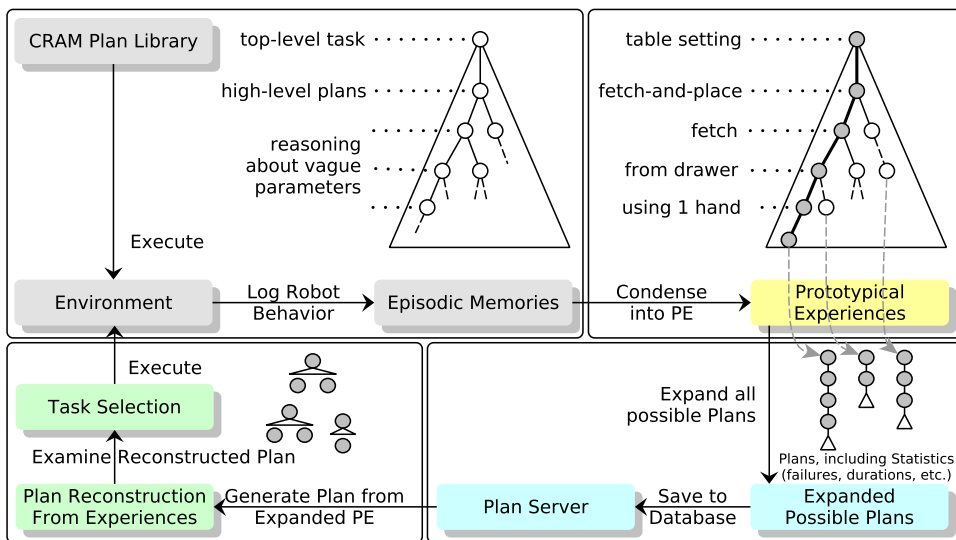


Figure 5.7: Architecture embedding Prototypical Experiences as a generalized representation for performed tasks. PEs are shown in yellow

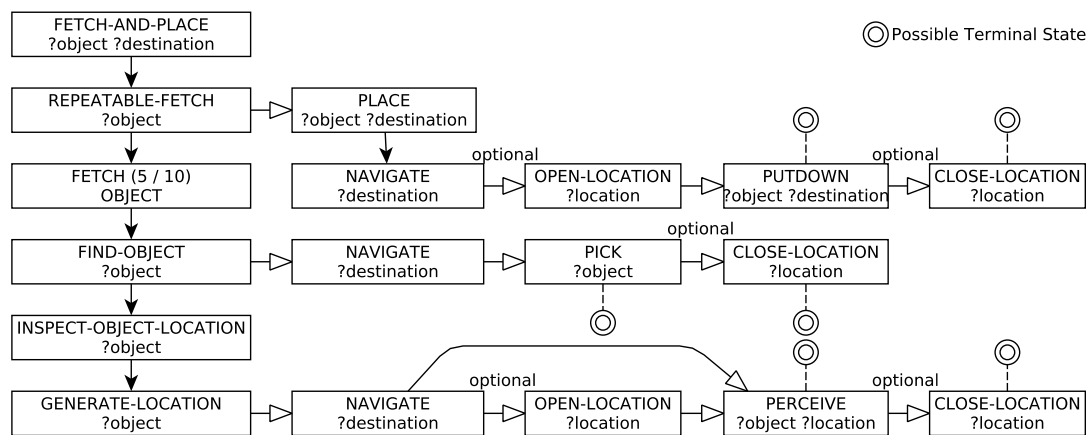


Figure 5.8: PE showing the structure of a fetch and place task. The PE is the result of multiple episodes under task variance and combines multiple execution paths and terminal states.

the container, and continuing to place the object.

The purpose of PEs is to serve as a blueprint describing all memorized variants of performing a task. It therefore can reconstruct all source memories in a loss-free manner, effectively storing a compressed version of all relevant memories. New memories are compared to the PE, resulting in either an extension of the model, by adding new nodes and links if the new memory is not yet represented by the PE, or in using the new memory simply to update the collected statistics about its corresponding path through the PE. A special property of PEs is that they can produce new strategies for achieving a task that were not part of the original source memories, based on optional steps. These combinations do not always make sense, but serve as a semantically intact search space for an autonomous robot that tries new, non-predefined strategies. In the given example, leaving containers open after picking objects from them is a legit strategy, although the original plans always closed them when an object was stored in a container. Based on the cost function to optimize (“do it as fast as possible”), this can result in more performant plans.

Algorithm 5.8 Recursively creating Prototypical Experiences from raw, unprocessed experience sets as in openEASE [13]

```

1: function CREATEPROTOEXP(Raw-Exps)
2:   PE  $\leftarrow$  {} ▷ Initialize Empty PE
3:   for Exp in Raw-Exps do
4:     InjectExperienceNode(PE, Exp)
5:   end for
6:   return PE
7: end function

8: function INJECTEXPERIENCENODE(PE, Node)
9:   T  $\leftarrow$  Node.taskType
10:  if not T in PE then
11:    PE[T].optional  $\leftarrow$  No
12:    PE[T].termState  $\leftarrow$  No
13:    PE[T].startState  $\leftarrow$  not Node.previousAction
14:  end if
15:  PE[T].invocations  $\stackrel{+}{\leftarrow}$  Node.parameters
16:  PE[T].outcomeData  $\stackrel{\cup}{\leftarrow}$  Node.outcomeData
17:  PE[T].updateOutcomeStatistics()
18:  if not (Node.nextAction or Node.subNodes) then
19:    PE[T].termState  $\leftarrow$  Yes
20:  else
21:    for Sub-Node in Node.subNodes do ▷ Depth
22:      InjectExperienceNode(PE[T], Sub-Node)
23:    end for
24:    Next-Node  $\leftarrow$  Node.nextAction
25:    while Next-Node do ▷ Breadth
26:      PE[T].nextActions  $\stackrel{+}{\leftarrow}$  Next-Node
27:      p  $\leftarrow$  Next-Node.parameters
28:      PE[T].nextActions[Next-Node].invocations  $\leftarrow$  p
29:      Next-Node  $\leftarrow$  Next-Node.nextAction
30:    end while
31:  end if
32: end function

```

Creating Prototypical Experiences from Episodic Memories

Algorithm 5.8 depicts a pseudo-code version of how to generate PEs from raw, unprocessed memories. The algorithm is implemented and available as open source software⁶. It recursively inserts nodes it finds in the hierarchical task tree of the EMs into the PE, and distinguishes between them by their task type, creating an incomplete n -ary tree. Task types can be as general as *grasp*, *navigate*, *perceive*, etc. Every occurrence of a task is appended to its respective task type node as an *invocation*, denoting the parameters that were used to start this task, and the node trace that lead to this invocation, preserving the original execution path. Nodes have hierarchical children, and can have horizontal previous and next actions. When no previous action is present, a node is a start node on this level. When no children and no next actions are present, it is a terminal node. Terminal nodes finish the current parent-task without evaluating further siblings. When a node is present in only some source memories but not all, it is optional. Optional nodes are branching points for experience reconstruction (discussed further below). Nodes can therefore have multiple terminal child nodes. Refer to Figure 5.8 for the final form of a PE, using Fetch and Place as an example scenario. Note that some non-relevant intermediate states were left out for the sake of clarity and brevity.

⁶https://github.com/fairlight1337/planning_graph_node

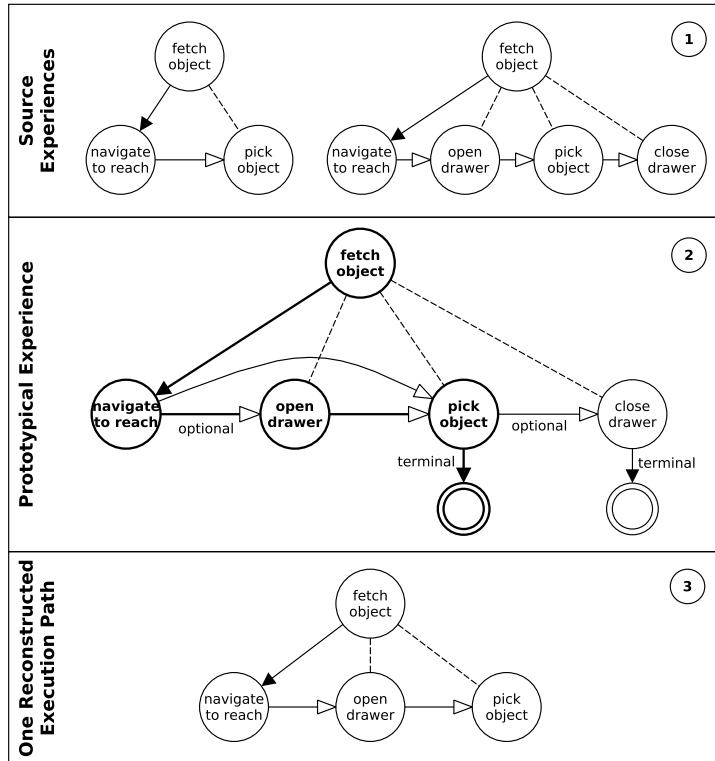


Figure 5.9: Source Experiences (1), generated Prototypical Experience (2), and one of the possible reconstructed plans from the PE (3).

During construction of a PE one also annotates characteristics of the experiences such as the amount of time needed for a task, or whether the run was successful, and associates such records with each node in the PE. A branch of the PE will in fact contain information about more experiences, because different plan runs may have pursued the same sequence of tasks. In that case, one can aggregate statistics such as the number of failures vs. the number of attempts (an estimate for the probability of failure), or average and standard deviation values for the time taken for execution.

Reconstructing Original Experiences

By performing a depth-first search on a PE and considering optional nodes (that expand into multiple paths), a list of possible execution paths through the PE graph is constructed. A step in a path is to be understood as a request to achieve some goal (for example, fetching an object, opening a container, or picking an object). Each step also contains information about which is the parent goal for this step (for example, an open-container step would know it is a subgoal of a fetch-object goal).

Since every node in a PE has a notion of the individual traces and parameterizations leading towards it, each individual experience can also supply possible parameterizations valid for its individual strategy for the represented task. The more optional nodes present, the more possible explanations for how to resolve a task are generated. The number of possible paths reconstructed from a PE is:

$$\text{Number of Paths} = 2^{\text{Number of optional nodes}}$$

With five optional nodes, the PE in Figure 5.8 can reconstruct $2^5 = 32$ execution paths from its 10 source experiences, resulting in 22 hypothetical, structurally unique strategies to perform the PE's task. This way, it generates new — partially physically valid — solutions to the task's

Metric	Feature	Aggregator
Optimistic Shortest Projected Time (OSPT)	Time	SUM
Pessimistic Shortest Projected Time (PSPT)		
Least Projected Failure Occurrences (LPFO)	Failures	MAX

Table 5.3: Implemented Plan Scoring Methods; For each plan request, one or more metrics for scoring can be chosen. Default is OSPT+LPFO. The metrics each make use of a feature of the plan steps, and either use their sum, or the max value.

$$\begin{aligned}
CI_\alpha &= \left[\bar{X} - z_{1-\frac{\alpha}{2}} \hat{\sigma}_{\bar{X}}, \bar{X} + z_{1-\frac{\alpha}{2}} \hat{\sigma}_{\bar{X}} \right] \\
\hat{\sigma}_{\bar{X}} &= \sqrt{\frac{S^2}{n} \cdot \left(1 - \frac{n}{N}\right)} \\
S^2 &= \frac{1}{n-1} \cdot \sum_{i=1}^n (X_i - \bar{X})^2
\end{aligned}$$

Equation 5.10: Calculation of confidence intervals for expected task time consumption. α is the error probability, X are the known time values for all tasks, \bar{X} their mean value, n is the number of samples considered (usually > 30 , so I heuristically assume a normal distribution), and N is the absolute number of known task instances. $z_{1-\frac{\alpha}{2}}$ is the normal distribution value.

problem that a robot can use to explore the solution search space.

Generating New Strategies from Known Task Structures

As mentioned in Section 5.5.2, PEs can combine known task strategies to form novel and possibly more performant approaches to solve a task, because the path enumeration procedure through a PE can generate paths that do not correspond to a previously seen experience. In the example from Figure 5.8, the PE can generate a plan that fetches an object from a container after opening it, but leaves the container open. The same happens for placing objects: Opening but not closing the container saves time and, if no other constraints forbid such shortcuts (like not leaving fridge doors open), they result in more performant plans.

The whole process of transforming source experiences into hypothetical strategies is shown schematically in Figure 5.9. Two different episodes of a *fetch object* task are combined into a PE. Then, one possible strategy — leaving a drawer open after picking an object from it — is generated.

Ranking Solutions Generated from PEs

How "*good*" a solution generated from an existing PE is depends on the criteria used. Table 5.3 shows three common metrics I defined for measuring a PE's quality: The projected time required when (1) being optimistic, or when (2) being pessimistic, and (3) the projected number of failures. For example to determine how long a task usually takes to complete, the confidence interval of all known times of this task type is calculated as shown in Equation 5.10. As the PE is a hierarchical tree and only leaf nodes actually consume time, non-leaf nodes' duration intervals consist of the sum of their child nodes' intervals. The result for the top level node is an interval denoting the lower and upper boundary of probably necessary time to perform the task, according to the confidence interval formula. For this, I use $\alpha = 5\%$, resulting in $z_{1-\frac{\alpha}{2}} = 1.96$, as a default setting.

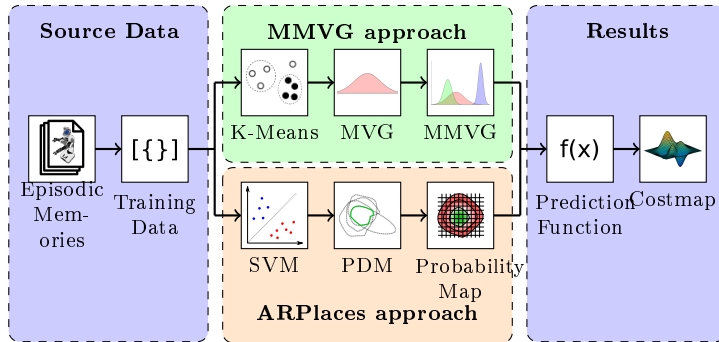


Figure 5.11: Processing pipeline to generate prediction functions from EMs using multi-modal analysis techniques. Green shows my MMVG approach, compared to the ARPlaces approach in orange.

5.6 Multi-modal Analysis of Robot Experiences

As extensively described in Chapter 4, the main point of generalized plan design is to abstract and generalize two elements: Robot action plans, and the information to parametrize them. While the former results in highly abstract static plan constructs, the latter results in a lack of information actually required to perform a task. To ground the abstract task description in the current situation and context — and to make it executable — these semantic, vague descriptions need to be formulated in a language low level robot components understand: Poses, velocities, distances, etc. Manually specializing every possible situation based on known pairs of task description/context information would ultimately undo the plan generalization (and again produce non-scaling, major effort for human plan designers). Instead, I propose a multi-modal analysis of Episodic Memories to automatically determine which parameters fit any particular situation well, including inter- and extrapolation of arguments if necessary.

Finding the correct correlations from these EMs and transforming them into actionable parameters for an autonomous robot is hard and effort-prone in itself due to the sheer amount of data a robot produces, and the often non-obvious relations between intentions and effects. To ease this process, I propose a novel approach for multi-modal data analysis on the basis of Gaussian distributions [104]. In particular, I concentrate on non-deterministic environments and deduce multivariate, mixed Gaussian distributions for parameter ranges to help an autonomous robot in making informed decisions.

In the remainder of this section, I will give an overview of the approach together with a comparison to a strongly related technique by Stulp *et al.* [90], will explain in detail how correlations are determined, and discuss their use-case in generalized robot action plans.

5.6.1 Approach and Comparison

I demonstrate the multi-modal, experience-backed learning process at the example of mobile manipulation, and more specifically object fetching tasks in a kitchen environment. The processing pipeline of the approach is shown in Figure 5.11. More concretely, a mobile robot parametrizes its own plans of where to position itself for picking up objects based on its own experience data, processed using a Gaussian regression technique. In the following, I give an overview of the steps taken to present a distribution of possible parametrizations to a learning robot:

First, recorded Episodic Memory data is transformed into a suitable format for machine learning. This is the decision point for the characteristics a plan designer wants to take into account when learning correlations. Any data extractable from the EMs can be used, be it either nominal or numerical values. The n -dimensional training data is then clustered using a K-Means algorithm, and for each of the resulting clusters an n -dimensional Multivariate Gaussian Distri-

bution (MVG) is calculated. Using all resulting MVGs, an equally weighted, normalized Mixed Multivariate Gaussian Distribution (MMVG) is calculated. The MMVG now acts as a non-linear multivariate interpolation mechanism between all the prior experiences' parametrizations and can be queried for a probability value at any point in the n -dimensional parameter space. From this distribution, costmaps are generated that a robot control program can sample from. Examples for these costmaps are robot base locations most suitable for grasping an object, or for opening a drawer.

A similar goal, but with a completely different approach, was pursued by Stulp *et al.* [90]. They used a Support Vector Machine (SVM) based learning approach to generate Point Distribution Models (PDMs), and finally generate a probability map of the regions well-suited for grasping using a Monte Carlo Simulation. I compare my approach to theirs, and show how my approach extends the type and dimensionality of source data that can be used, at the cost of precision. I also show that, given the use-case of producing costmaps for location selection, the loss in precision is negligible. Their processing steps are parallel to mine, and are shown in Figure 5.11 as well.

5.6.2 Multi-modal Data Analysis

One of the main advancements of this work over previous approaches is the increase in search space dimensions. While previously the only features used to determine whether a position to perform, say, a grasp action was well-chosen were the numerical relative distances in x and y direction, I introduce MVGs over an arbitrary number of task parameters. My multi-modal data analysis covers both, real and nominal values: Real values are measured based on their actual numerical value, while nominal values are assigned an index number in their category. This approach is well-formed, as nominal values are not interpolated while querying for probabilities, but their exact indices are used.

Clustering of Source Data using K-Means

Robot EMs contain local areas of interest in a global context. In this concrete case, these are collections of poses around an object where the robot tried to place itself for grasping. Given that these local areas are scattered throughout a larger area, they need to be clustered according to their n -dimensional feature vector.

To arrive at a sensible number of clusters, K-Means clustering is performed up to a given maximum number of clusters. Their average silhouette value [76] is calculated, and the cluster amount with the lowest average is used. This results in an optimal point distribution over all clusters. I use the Euclidean distance measure for n -dimensional data.

Multivariate Gaussian Distributions

Based on clustered source data, an MVG is calculated for each cluster. To this end, for cluster i , the covariance matrix \mathbf{C}_i is calculated:

$$\mathbf{C}_i = \sum_{k=1}^{n_i} (\mathbf{X}_{i,k} - \bar{\mathbf{X}}_i)^T (\mathbf{X}_{i,k} - \bar{\mathbf{X}}_i) \quad (5.1)$$

While any number $p \leq n$ of aspects can be included from the source data, the amount of data actually stored in the MVGs is minimal: For each MVG (i.e. per data cluster) only its covariance matrix $\mathbf{C}_i \in \mathbb{R}^{p \times p}$ and the source data's mean value $\bar{\mathbf{X}}_i \in \mathbb{R}^p$ are stored, resulting in a space complexity of $\mathcal{O}(p^2)$ per cluster. The density function $f_i(\mathbf{X})$ allows to evaluate the distribution at any one particular point $\mathbf{X} \in \mathbb{R}^p$ for the MVG around cluster i :

$$f_i(\mathbf{X}) = \exp\left(-\frac{1}{2}(\mathbf{X} - \bar{\mathbf{X}}_i)^T \mathbf{C}_i (\mathbf{X} - \bar{\mathbf{X}}_i)\right) \quad (5.2)$$

Gaussian Mixture Models from Overlaid MVGs

Given the MVGs from the source data, I form an MMVG by applying the same weight $\frac{1}{m}$ to every one of the m clusters involved. To sample from the overall distribution, the mixture’s density function $F(\mathbf{X})$ needs to be evaluated:

$$F(\mathbf{X}) = \sum_{i=1}^m w_i f_i(\mathbf{X}) \quad w_i = \frac{1}{m} \quad (5.3)$$

The weight w_i could be chosen differently, for example using the relative amount of points involved in the cluster i . Since I assume that the experience data can include isolated clusters with only a few occurrences that still play an important, legitimate role (compare case 1 in Figure 5.12), I decided to use equal weights.

5.6.3 Evaluation

I have implemented the presented parametrization learning framework⁷ and applied it to a scenario in which a mobile robot performs object fetch tasks in a kitchen environment. It picked up objects from three tables, multiple times. A manually defined heuristic based on Euclidean robot/object distance and orientation was used to collect the training data.

In total, around 40 pick trials were recorded. Figure 5.12 shows the extracted feature points (green) as well as the resulting MMVGs, shown as gradient heatmaps. The distributions reflect the probability of success in the given task based on the relative position of the robot w.r.t. the object, as well as their relative orientation to one another. It is important to note here that the learned model does not include characteristics about the environment itself; the coordinates used for training and results retrieved afterwards are purely relative. From top to bottom, the relative orientations between robot and object in the Figure’s plots are -230° , -90° and $+300^\circ$. Except for the last plot, the resulting distributions reflect the source data very well. My assumption is that the number of data points (given three independent variables, x , y , and θ) in that region is too low (and scattered too much) to generate a properly aligned distribution. A statistically significant amount of source data would mitigate this problem. Besides this, the distributions give a very good prior of where to stand in order to grasp an object, before falling back to the manually designed heuristic, even for the suboptimal last case.

The EMs used in this evaluation were recorded using the robot memory system SemRec [105] and include both, high-volume low level sensor data and low-volume high level semantic plan data. Therein included are object descriptions, exact robot motions, grasp details, and kinematic poses at all times. These are the source for the training data used.

The maximum expected cluster count depends on the task performed, the size of the environment, and the number of experiences involved. It is safe to say that — for this example case — the number of involved objects gives a hint towards that maximum. I decided to use ten clusters at maximum, while having five objects involved in the pick and place task. Most of the time, this would result in two to three clusters, but leaves room for situations in which objects are cluttered in the same space.

⁷<https://github.com/fairlight1337/MultiVarGauss>

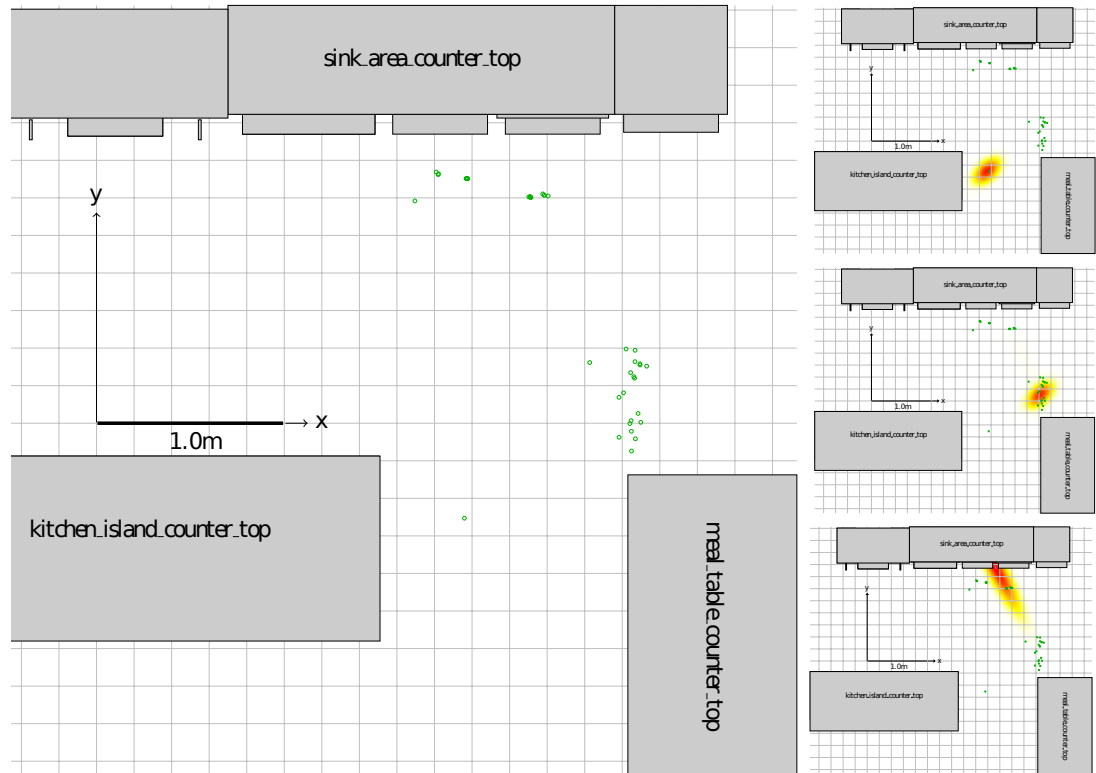


Figure 5.12: Learned probability distributions for successful grasping, depending on relative distance and orientation between robot and object. The right three plots show regions of high success probability for grasping the objects on the table near them, as a density function $f := f(\text{relative-position}, \text{relative-orientation})$; chosen relative orientations from top to bottom: -230° , -90° , and $+300^\circ$. The pure experience data points are shown in the left plot.

5.7 Learning PDDL Domain Knowledge from Experience

Service robots are becoming an ubiquitous sight in different, often labor intensive areas of everyday life. From vacuuming living rooms [34] and mowing the lawn [78] to weeding fields [72], relatively simple robots perform specialized tasks in very defined and quasi-static environments.

A current development is the introduction of service robots into human households [87]. Initially, their tasks are limited to predefined activities such as cooking [16], cleaning windows [55], or acting as a watchdog [56]. Each of these activities in itself requires different robot skills and action descriptions, and can fail for various different reasons. Fluent behavior is achieved by the reactive nature of the robotic systems: They select one of a few known responses, as appropriate to the situation at hand, and fail for all domain unrelated tasks. Often, to simplify the tasks performed, they make explicit use of the Closed World Assumption.

With a growing number of robot tasks, defining and maintaining a library of known responses manually has two major drawbacks: (1) When tasks are interleaved, the effort to handle all situations in all contexts grows exponentially, and (2) there are no synergies between already known and new tasks. On the other hand, generalizing these activities into one large hand-designed robot action plan that can perform all of them still lacks scalability for both new activities and slight changes to existing ones. With new robot platforms and chores surfacing rapidly, today's robot architectures still lack an important skill: Learning new behaviors and applying them according to the current situation and their best knowledge about the background information and already known activities.

Traditional AI planning offers a set of tools for combining known primitive actions into solutions to new problems. Classical planning, such as formalized by the STRIPS, or more

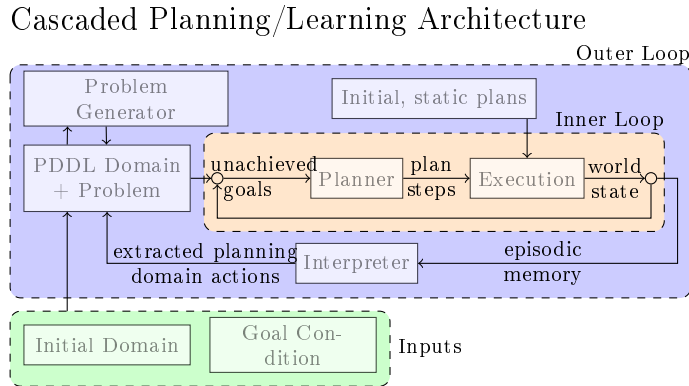


Figure 5.13: Architecture overview for cascaded planning (inner loop) and learning from experience (outer loop). Problems are generated for an evolving PDDL planning domain, based on robot EMs. Goal conditions can be “meal table set”, “pizza cut into pieces” and the like. The initial domain only contains robot primitives such as “perceive”, “pick”, “place”, and “navigate”.

generally as a PDDL domain, selects actions based on their preconditions and effects coupled with a global start and goal state. However, PDDL action atoms have to be defined manually in the respective planning domain, which suffers from the same limitations as mentioned above. Also, classical task planning has difficulties when handling incomplete information about the world state and actions with continuous parameters, which — due to sheer planning problem intractability — has resulted in its limited adoption in robots working in realistic environments.

To make the management of different action primitives tractable, allow automated identification of new primitives, and enable planning for real world domains accordingly, I introduce a knowledge acquisition and activity identification framework for robot tasks. Herein, I use EMs to identify actions a robot performed using initial, static plans. These are then transform into sets of PDDL actions along with their perceived preconditions and effects. Given enough robot experience from manually designed plans, the resulting action library can ultimately solve the tasks of the original plans while sharing action primitives with all other known activities. This makes the robot’s behavior more versatile and takes advantage of synergies while at the same time addressing the problems traditionally associated with classical planning in real world environments (see above). As an additional plus, the resulting PDDL domain allows human-readable inspection of the learned actions.

5.7.1 Identifying PDDL Actions from Experience

The overall architecture for learning PDDL actions from EMs is shown in Figure 5.13. It consists of two cascaded control loops: An inner loop that uses a parametrized PDDL planner to generate plan steps for reaching a given goal state, and an outer loop that — through means of Episodic Memories — interprets the executed actions’ structure and characteristics, formulates them as new actions in the PDDL planning domain, re-parametrizes the inner planner and generates a new problem for execution. The inner loop runs at a much higher frequency than the outer loop (n plan steps per generated problem). At its core, this requires four components to be closely integrated:

- (o) **Sensors:** Perceiving the world state before and after performing actions, measuring relevant aspects of their effects
- (o) **Planner:** Generating action steps to move towards a given goal, based on the current world state
- (o) **Episodic Memories:** Recorded assets of executed tasks, action dependencies and parameters, perceived effects, and RTPs

- (o) **Memory Interpreter**: Extracting actions from EMs and formulating them as new candidates for planning

In its implementation⁸, the **Planner** component relies on an initial, simple PDDL domain description requiring a robot platform to implement atomic actions such as navigation, perceiving the environment, and picking and placing objects. This is accompanied by sensor-backed monitoring of the actions' effects. The planner component then provides the problem solving ability of the system — generating the next plan step for the robot to execute, and record into an EM. It is based on STRIPS and after executing each plan step updates its internally believed world state if necessary. This measure alleviates traditional constraints from the Closed World Assumption. The EMs generated from these task executions precisely reflect what actions the robot performed, what their parametrizations were, and what the hierarchical structure of parent and child tasks was.

Finally, this memory data is processed by the **Memory Interpreter** component, extracting tasks and formulating their PDDL action equivalents, composing sets of known (primitive) actions into new action complexes. This enables the planner to use not only its primitive atomic actions, but also more complex, constructed actions composed of primitive and other constructed actions.

Static Planning Domain Knowledge

To generalize over and plan for multiple hierarchical tasks, they must ultimately consist of basic, static building blocks. In the case of mobile manipulation robots, these need to be defined per robot platform and include

- (o) 2D navigation (given a 2D goal pose for the robot's base)
- (o) Perception (localization and identification of objects)
- (o) Grasping objects (including constraint-aware motion planning)
- (o) Putting down objects (including free space checking)

for covering tasks from the household domain. Their definition consists of two parts: A description of the symbolic action in PDDL (necessary once per domain) and the actual implementation attached to this action (necessary once per platform). Each action is associated with a set of preconditions and effects. (1) Navigation requires the goal to be reachable, resulting in the robot being at the new goal pose. (2) Perception yields new object instances. (3) Grasping needs enough hands to be free and the object to be reachable, afterwards having the grasped object in hand. (4) Putting down the object reverts this process.

Learning new Actions

To identify new actions in an experience's task tree, it is traversed recursively. New (constructed) actions consist of only known primitive and constructed actions. Their structure and signatures are extracted from the tree, and the backtracking algorithm continues on the parent level until all levels are covered. It was necessary for me to write PDDL descriptions for the basic actions of the plan execution architecture I use, because it was not originally designed to work with PDDL planning.

Actions within the task tree take on the form shown in Figure 5.14. The tree yields three new constructed actions: **fetch**, **place**, and the overarching **fetch-and-place**. Exemplarily, (**fetch** ?obj) consists of the primitive actions to **find**, **approach**, and **grasp** an object. These all share the same parameter ?obj, which thus is the only parameter identified for **fetch**. Primitive actions have well-defined preconditions and effects, which ultimately make up the overall preconditions and effects of the newly constructed action **fetch**.

⁸<https://github.com/fairlight1337/GDAPlanner>

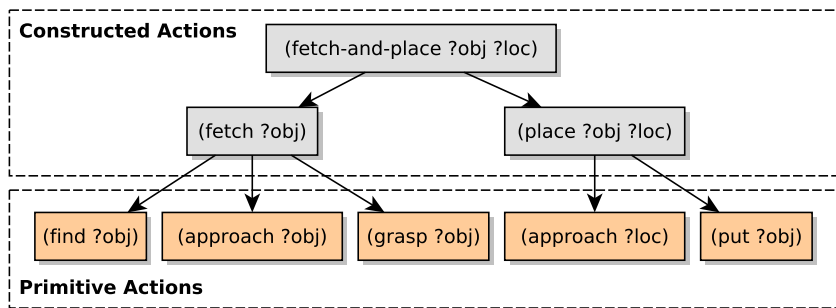


Figure 5.14: Task tree of a robot experience. Primitive actions are encoded per robot platform and must be properly annotated. Constructed actions consist of primitive and other constructed actions. Ignored actions are omitted here for the sake of clarity.

To keep the constructed actions semantically meaningful, any control flow structures generated by the plan execution architecture that are present in the hierarchical task tree are black-listed. According to any EM, its recorded control flow was identified to lead to the perceived and annotated result, so this concrete instantiation of the intermediate control flow operators is directly associated with the input parameters and outcome. An overarching control flow thus becomes superfluous when using this constructed action with the currently active preconditions. This, again, is a prior for planning — upon failure, the new situation requires a new plan based on new preconditions.

Theoretically, a complete snapshot of the plan execution architecture’s characteristics would allow code inspection and analysis to make constructed actions more complete and possibly more fine-grained. Since robot experience can be generated by a multitude of architectures different in strategies and programming languages, individual code inspection becomes too effortuous and error-prone in comparison. The above architecture is applicable to any platform that follows the simple rules stated above: (1) Implement the basic actions and (2) annotate their use and parametrization in EMs. This way, it becomes very easy for the Memory Interpreter to decide whether an experience is valid (and interpretable) or not.

5.8 Automated Experiments for Data Collection

Due to the sheer amount of data required for statistically significant machine learning scenarios, the process of generating EMs is very mundane, time-consuming, and prone to user error during data collection and processing. The obvious solution is to automate at least the former of the two, given that the experiments considered are just variations of one prototypical scenario. This, of course, requires a very good understanding of the experiment to automate: A trained operator spots undesired anomalies early and in general can judge corner cases, and decide on required next steps.

To ease the especially time-consuming process of simulated, physics-enabled longterm robot experiments, I developed and implemented an experimentation system that satisfies the above requirements:

- (o) Complete, automatic bootstrapping of experiment environment
- (o) Monitoring for correct startup, approving individual steps based on their output before continuing
- (o) Consideration of output-based failure conditions and timeouts
- (o) Automated parameter variance to randomly customize every experiment episode in given bounds

I will outline the requirements for automating this process, introduce a new set of tools for conducting automated experiments, and describe the flow of execution of the final system.

5.8.1 Requirements for Experiment Scenarios

The most basic requirement for automating the execution of an experiment scenario is that it is non-interactive. This means that at no point user decisions are required to perform the actual experiment, its setup, or its teardown. There are three program states in which user interaction is usually required:

- (o) **Configuration:** The program’s work schedule must be set up and execution variance parameters must be chosen.
- (o) **Handling Exceptions:** The program’s failure handling capabilities are exhausted and user intervention is required to return to a normal state of execution.
- (o) **Post Processing:** If any, the resulting output data must be post processed, stored, labelled, etc.

All three can be addressed by automation, but require extra effort and thought. For statistical data analysis, an experiment’s configuration and variance usually does not change qualitatively, but quantitatively. This means that an external configuration script that has knowledge about parameter bounds and special cases can set up the experiment while exploring the whole allowed parameter space. When the program’s failure handling capabilities run out during execution, meaning an exception comes up that cannot be handled, this is usually escalated to the user. When automating such a process, a wrapper script or popular “`try { ... } catch(...) { ... }`” construct needs to detect this case, and perform e.g. a shutdown and restart of the whole experiment. After successful execution, data processing often involves external applications that are not part of the actual experiment. A sequential execution of applications is therefore required, going forward in the sequence only if the prior steps concluded successfully.

When these requirements are met, and all residual processes are torn down completely after finishing, an experiment can be repeated any given number of times. In the following section, I will present a tool that implements those requirements and performs experiments arbitrarily often: The AutoExperimenter.

5.8.2 AutoExperimenter: A tool for generating meaningful data

To automatically conduct experiments and collect significant amounts of EM data, I designed and implemented the AutoExperimenter tool⁹. It accepts an experiment description and offers all required functionality to fully start, parametrize, conduct, and tear down any experiment performable using software. For my work, I developed a simulation environment for a PR2 robot that performs household tasks in a kitchen setting, mainly setting a meal table with varying requirements.

To define an experiment, I designed a very abstract description language that controls the AutoExperimenter instance’s behavior. The definition is written in Yet Another Markup Language (YAML) format, and thus both easily readable by humans and interpretable by program code. In the following, I highlight the most important features.

Workers are the main element for any AutoExperimenter instance. They represent any program started to perform an experiment. This ranges from making instantly returning calls to external components down to processes that run throughout the full lifetime of the experiment. Their syntax is as follows:

⁹https://github.com/fairlight1337/auto_experimenter

```

workers: ;; Includes all workers
- command: roslaunch ;; Starts worker with the executable 'roslaunch'
  parameters: [ltnfp_executive, ltnfp_simulated.launch] ;; Arguments
  checklist: ;; Must match all for successful startup
  - name: moveit ;; Internal identification name
    matchmode: contains ;; How to match 'template'
    template: "All is well!"
    message: "MoveIt! launched successfully" ;; Print this upon success
  - name: reasoning
    matchmode: contains
    template: "ltnfp_reasoning/prolog/init.pl compiled"
    message: "Reasoning started successfully"
  quithooks: ;; If any of these match, tear down experiment
  - name: failed_to_load
    matchmode: contains
    template: "failed to load"
    message: "A component failed to load"
  timeout: 120 ;; Optional timeout; 0 for infinite wait
  append-variance: true ;; Optionally append task variance to process

```

The `command` signifies which executable to start a new process for, followed by its `parameter` list as command line arguments. From then on, `AutoExperimenter` checks for the presence of all `checklist` items. If all items were detected, the worker was started successfully. If one or more were not met within `timeout` seconds, the whole experiment is torn down. A `checklist` item is matched/found when any of the worker's output lines coincides with its `template`. The matching methods available are:

- (o) `match`: Full match of both strings
- (o) `contains`: The output line contains the template string
- (o) `beginswith`: The output line begins with the template string

The `quithook` entries are the exact negative of `checklist` items: Whenever one or more match, the process is deemed unsuccessful and the experiment is torn down. Syntactically, they function exactly like their `checklist` counterpart.

Optionally, a worker can have the current task variance appended to the `parameter` list when the `append-variance` switch is present and set to `true`. How task variance is defined is described further below. All items in `workers` are started sequentially.

Cleaners are clearing up any leftover resources or intermediate changes the workers did. Syntactically, they are equivalent with workers:

```

cleaners: ;; Includes all cleaners
- command: kill_processes.sh ;; Start an external script
  parameters: [] ;; No arguments given
  checklist:
  - name: killed
    matchmode: match
    template: "Finished killing processes"
    message: "Processes killed"

```

Other than workers, cleaners have no notion of `quithook` elements, `timeout`, or `append-variance`. Like workers, cleaners are executed in sequence.

Task variance allows the AutoExperimenter to automatically re-parametrize an experiment in-between runs. To this end, different types of parameters can be defined:

```
task-variances: ;; Includes all variance parameters
  attendants: ;; Parameter name: 'attendants'
    label: "Meal attendants" ;; A descriptive label
    value-type: [multiple-choice] ;; Choose a random number of items
    items: [{"Mary", mary}, {"Tim", tim}]
    allow-empty: false ;; Allow empty list?
    default: [tim] ;; Default value if not sampling
  object-availability-factor:
    label: "Percentage of objects spawned based on how many are required"
    value-type: [percentage, range] ;; Return a percentage
    value-range: [0, 2]
    default: 1
    distribution: normal ;; Distribution type
  mealtime:
    label: "Meal time"
    value-type: [choice] ;; Choose one random item
    items: [{"Breakfast", breakfast}, {"Lunch", lunch}, {"Dinner", dinner}]
    default: [breakfast]
```

Each variance parameter consists of a variable name, a descriptive label, a default value, and a set of variable type dependent fields. The following variable types are allowed:

- (o) `multiple-choice`: Choose a random number of items from the `items` field. The switch `allow-empty` marks whether an empty list is valid. Every item is a pair of a descriptive label and an interpretable symbol.
- (o) `choice`: Choose a single element from the `items` field (similar to `multiple-choice`).
- (o) `[percentage, range]`: Return a floating point value from within the boundaries given in the `value-range` field. The `distribution` field defines the distribution type used.
- (o) `[integer, range]`: Similar to `[percentage, range]`, but works with integer values rather than floating point values.

Should parameter variation be deactivated in the AutoExperimenter instance, the respective `default` value will be assumed for every task variance parameter.

Meta Information describes who is responsible for the experiment definition, and what version of the experiment is described in the respective YAML file. Valid fields are:

```
meta-information:
  author: <firstname lastname>
  email: <email@domain.com>
  website: <url>
  version: x.y.z
```

Changelog data shows who changed which element when. Every item denoted here includes an author, their email address, a date, and a changeset describing the changes:

```
changelog:
  author: <firstname lastname>
  email: <email@domain.com>
```

```

date: YYYY-MM-DD
changeset:
- Moved to new YAML format for experiment description
- Added meta information and task variances

```

Running an experiment for a fixed maximum number of times follows a strict order of events:

1. Load experiment description.
2. Generate task variance parameters.
3. Run all workers in order. Should any fail, run cleaners, tear down experiment and go back to step 2 until maximum number of experiments reached; quit when exhausted.
4. If all workers started successfully (all checklists complete) mark the experiment as successful.
5. Tear down experiment, running all cleaners in order.
6. Go back to step 2 until exhausted.

To actually record EMs for every experiment, SEMREC needs to be run as a worker before the actual experiment starts. This is achieved with this worker definition:

```

- command: rosrun
  parameters: [semrec, semrec, "-q", "-s"]
  checklist:
  - name: initialize
    matchmode: contains
    template: "Signify: semrec init complete"
    message: "Semrec Initialized"
  timeout: 15

```

A live example with all required steps for the experiments described in my work is available online¹⁰.

5.8.3 Adaptation to other Scenarios

While I present example experiment setups in the context of the fetch and place scenario, the concepts described here are by no means fixed to this scenario. Any experimental setup that can be conducted unsupervised and can automatically be restarted, such as in simulation, can be performed using my approach. Additionally, the task variance parameters are — albeit being completely optional and can be left out — fully customizable to fit other experiments' requirements.

5.9 Summary

In this chapter I introduced robot EMs, a persistent storage for all relevant data available during task execution. Subsequently, I presented three areas that make use of EMs: (1) Generating experience models that allow prediction of task action effects, (2) multi-modal analysis and extrapolation of subsymbolic parameters from experience, and (3) learning of PDDL domain knowledge from experienced tasks.

Finally, to enable the methods above, I presented the requirements of, and a working and tested implementation for experiment automation for simulated robot activities. These produce statistically significant amounts of EM data supporting the many probabilistic methods used in my work. I show how generalized plan scenarios are automated, and how the automation

¹⁰https://github.com/fairlight1337/longterm_fetch_and_place/tree/master/ltnp_executive/assets

approach automatically varies parameters to produce both, qualitatively and quantitatively different EMs.

The present chapter is an explorative conglomeration of possible uses for EMs in order to parametrize generalized plans, and to improve their performance. The amount of untapped information in robot EM data allows for more use-cases which I did not pursue here.

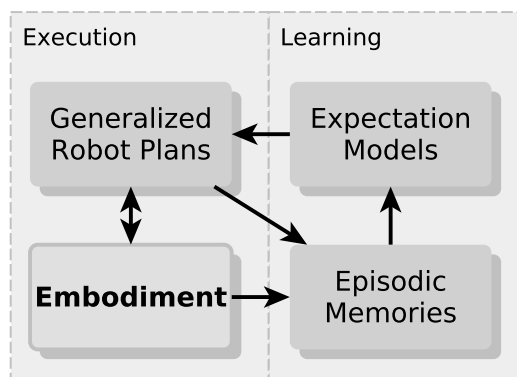
Embodiment of Autonomous Robot Control Programs

“We categorize as we do because we have the brains and bodies we have and because we interact in the world as we do.”

— George Lakoff [50]

Embodiment can be seen as the act of manifesting virtual systems in a real world environment. In the case of AI, and more particularly robots, this means establishing an information flow from the environment towards the AI system via its sensors, and at the same time allowing the system to enact actions using its actuators, changing the environment’s state. All of this must be done while accounting for the differences between an internal world model and the real world — as any assumption about what will happen next can lead to failures in enacting a plan’s intentions. According to Vernon *et al.* [96], cognitivist architectures “do not need to be embodied, in principle”, as the physical form is independent from the functional one. For controlling a real world robot though, embodiment is essential.

In this chapter, I present a failure taxonomy approach for robot plans, and the most important sensing and acting interfaces for a mobile manipulator in everyday human environments. More particularly, I will go into detail about how semantic perception is used within my high level behavior control programs, and how static and dynamic knowledge is used to appropriately parameterize a robot’s manipulators. I showcase particularly difficult areas I worked on in object manipulation and interpreting a sensed world state, such as dual arm manipulation, and object identity resolution. To counter any failures, I go into failure handling and recovery strategies. Finally, I will explain the different semantic layers that need to be bridged between physically manifested sensor and actuator systems, and abstract control entities such as generalized plans.



6.1 Failure Handling and Recovery in the Real World

While an autonomous robot performs tasks, two classes of failures can occur: First, **anticipated** failures that the robot has explicit knowledge about encoded in its plans. These failures include being unable to reach a cartesian pose with a kinematic arm, or not finding an object in the field of view. Anticipated failures are expected to happen and have definite recovery strategies, as they are part of the normal flow of operation during the task execution. This class also includes checks for whether all preconditions hold when executing a task (e.g. having a free hand before grasping an object) — while these “failures” are not always recoverable, they belong to the class of expected problems. The second class covers all **unanticipated** failures: Problems

Algorithm 6.9 Example CRAM grasping code: Failure checking (red), purely task-related code (black), and comments (brown). A BT version of the plan is depicted in Figure B.1.

```
(def-plan get-object (arm object)
  (when (is-arm-free? arm)
    (when (is-object-reachable? object)
      (let ((traj (calculate-trajectory arm object)))
        (when (is-trajectory-valid? traj)
          (execute-trajectory traj)
            (when (grasp-location-reached? object)
              (grasp arm object) ;; Open hand, enclose object, close hand
                (when (object-in-gripper? object)
                  t)))))) ;;; Success
```

during task execution that the robot can neither properly detect, nor (commonly) recover from appropriately — even in case of a proper detection.

The former class of failures is straight-forward, although mostly labor-intensive to implement. The base (non-generalized) plan for the task is implemented (or planned symbolically), and then all possible failures a programmer can think of are explicitly checked for. These checks ensure the executability of the program, as far as predictions are possible, before executing it and running into failures.

The latter class of unanticipated failures requires heuristic checks on the results of performed actions, and possibly concurrent monitoring of variables. The difference between these and the former failure class is that when no failures are anticipated, the program is expected to run fine. Unanticipated failures can still disrupt a robot’s plans without being noticed. Their detection commonly indicates that *something* went wrong, but not necessarily why and what exactly did not work as expected. Heuristic detection mostly detects secondary failure effects rather than the actual problem. Cognition-enabled architectures executing Generalized Plans are expected to handle both failure classes to some sensible degree. As Vernon *et. al* [96] elegantly put it, being able to react to these unanticipated, unexpected issues is a major part of cognitive systems:

“The hallmark of a cognitive system is that it can function effectively in circumstances that were not planned for explicitly when the system was designed. That is, it has some degree of plasticity and is resilient in the face of the unexpected.”

An example for the difference between the two failure classes is the execution of a trajectory: A valid motion trajectory might have been calculated (although reasons for an invalid result could have been anticipated), but the execution fails due to an unknown reason. All the information the control program has at its disposal is the failure itself and the intended motion; possible recovery strategies include either retrying the action with different parameters, or heuristically trying to isolate the problem source and work around it. At the same time, unanticipated failures can have lasting effects on the surrounding environment, such as pushing objects off tables or destroying a piece of furniture (in case of a wrongly executed trajectory, say due to a controller failure, wrong collision model of the environment, or imprecise perception).

The most important difference is therefore: For anticipated failures, a set of reasons is known that can be checked for. Trajectory calculation fails due to an unreachable goal pose, either because it is too far away, or no path could be found among the collision environment. These can be validated by the control program a priori. Trajectory execution on the other hand can fail due to a multitude of exogenous reasons, such as someone holding the robot’s arm, motor failure, or sensor drift (and subsequent controller failure during movement). The control program does not have access to the necessary information for validating these reasons, and can only try to recover from them heuristically, if at all.

Failure handling can make up most of a program’s code, as shown in Algorithm 6.9. Task-

Algorithm 6.10 Concurrently monitored object transport task: Moving towards destination until arrival or interruption, the latter resulting in re-grasping. A BT version of this plan is shown in Figure B.2.

```
(def-plan transport-object (object destination threshold)
  (let ((initial-object-pose (get-object-pose-in-hand object))
        (interrupted nil))
    (par ;; Execute in parallel
      (while (not (at destination))
        (when (not interrupted)
          (pursue ;; Until one finishes
            (progn (move-to destination)
                  (return-from par) ;; Reached destination
                  (until interrupted))))
        (whenever (> (distance initial-object-pose (get-object-pose-in-hand object))
                    threshold)
          (setf interrupted t) ;; Interrupts navigation
          (regrasp-object object)
          (setf interrupted nil)))))) ;; Restarts navigation
```

related code is separated from failure handling-related code. For the sake of simplicity, the example signifies a detected failure by not returning the success flag “*t*”. More elaborate failure handling and recovery strategies would be retracting the arm, checking whether the object was moved during the motion, or calculating a different trajectory and relaxing constraints when no trajectory could be calculated. These are plan repair measures, being part of failure recovery strategies, and in general try to leave the scene in an expected, clean, and safe state after finishing execution.

For tasks that are active for longer periods of time, such as holding an object and transporting it across the room, the classes of anticipated and unanticipated failures interleave: The object might slip from the hand, but the time window for this to happen is the whole task of transportation. So it is an expected failure that needs checking, but its not clear at which point in the given time interval it will appear. In order to properly detect and possibly circumvent this problem, a concurrent monitoring process needs to regularly check the state of the object in the robot’s hand. If irregularities are detected, this monitor is supposed to interrupt the main transportation task and take countermeasures, such as signalling a re-grasping action. Algorithm 6.10 depicts an example of this strategy in code. Of course this example does not take into account that re-grasping can fail, a very real and legit situation, which would possibly require the robot to stop its current task altogether.

When defining robot plan failures more generally, they can be formulated as:

“Circumstances that prevent the robot in its current state and configuration from achieving some or all of its currently pursued goals.”

Given this definition, robot plan failures can be categorized in a second dimension: They are either platform-dependent and can be addressed locally (such as restarting a controller, switching perception routines, or trying a different arm for grasping), or are task-related and require specialized methods based on domain knowledge (there is no glass in the cupboard, so look into the dishwasher). Table 6.1 shows how failures are categorized using both dimensions, and all four cases are explained in more detail in the following.

- (a) **Anticipated, Platform Dependent:** Action effects significantly differ from the expected results. This can be failed grasps, objects not being seen in the field of view, and missing space for putting down an object. These failures can be explicitly checked for at appro-

	Platform Dependent	Task Related
Anticipated	Outcome differs from Expectation	Planned Uncertainty triggered
Unanticipated	Robot Component failed	Task Preconditions violated

Table 6.1: Two dimensions for categorizing robot plan failures: Anticipated vs. Unanticipated, and Platform Dependent vs. Task Related failures

priate times (check grasp after grasping, scan table for space before putting down object) and are in general recoverable.

- (b) **Anticipated, Task Related:** The “recipe” for performing a task cannot be completed without extra steps. This can include required tools not being mentioned in the action description, or no glass being in the cupboard when looking for one. In general, preconditions for task steps are not met. These failures can be taken care of by planning under uncertainty, and by postponing resolution of symbolic action, object, and location descriptions.
- (c) **Unanticipated, Platform Dependent:** Preconditions are retracted abruptly and with no prior evidence. This can include controller failures, a breakdown of the plan execution system, or very sudden changes in lighting conditions (thus hindering perception). In general, these failures have crucial effects on the task execution and would normally require human intervention to overcome.
- (d) **Unanticipated, Task Related:** Preconditions for task steps are permanently not met and cannot be substituted. Starting with the assumption that in some way the task is performable, a robot can only fail in these cases. Failures of this kind include no glass at all being found in a kitchen when it is required (and not substitutable) for the task, or all paths to a vital room being blocked.

To act robustly, generalized plans need to address detection and (possible heuristical) recovery of issues (a) and (b). They also need to be able to detect occurrences of issue (d), and act accordingly (e.g. gracefully cancelling the task, notifying a human operator). Issue (c) is beyond the capabilities of abstract generalized plans: Detection of these situations falls under the jurisdiction of the underlying middleware, and must be communicated to a human operator independent from the currently active task.

6.1.1 Exhaustingly Repeating Actions as most Naive Handling Strategy

Definitions of expected failures make a common assumption: Once their failure condition is known, strategies can be applied to solve the underlying problem. While failure handling strategies usually do not guarantee a 100% success rate (often also due to their heuristic character) they are — in theory — commonly assumed to be the best action to take. In purely deterministic systems — assuming that they allow failures to happen — this is always true. In reality, which is non-deterministic in at least the area of mobile manipulation for robots, this simple assumption does not hold — see Section 6.1.3 about uncertainty in the real world. This uncertainty can lead to false positives: Objects not found on a tabletop can be caused by lighting conditions, and a control failure can happen due to a modeling error in object data (object mass in the description is different from its actual mass, changing the executed motion due to inertia).

The simplest and most applied naive approach to a failure is retrying the same action again, with exactly the same parameters. For all sensors that suffer from drift, noise, and interferences (vision, odometry, fingerprint force, etc.), retrying the action some n times yields a good chance of succeeding if the problem was related to a misdetected failure. To apply this to

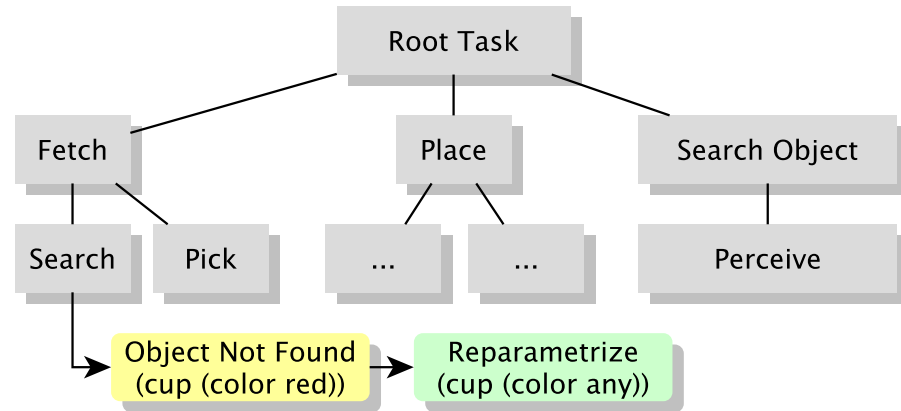


Figure 6.1: Global Failure Taxonomy: Failures are organized in a hierarchy of tasks in which they occur, and each task can fail due to a multitude of failures. If a new failure is not found in the taxonomy, a leaf is added. If one is found, its associated failure handling strategy is tried; and upon failure, the parent’s strategy is selected.

my generalized plans, I make use of CRAM’s specialized facilities for exactly this purpose: `with-retry-counters` and `do-retry`. A minimal example is shown below (BT version in Figure B.5):

```

(def-plan grasp-object (object)
  (with-retry-counters ((grasp-retry 2)) ;; Initialize retry counters
    (with-failure-handling ((manipulation-failed ;; Manipulation failed
                            (do-retry grasp-retry))) ;; Retry until exhausted
      (perform-grasp-object object))) ;; Perform some failure-prone function
  
```

For every `do-retry` that is met, `grasp-retry` is decreased by one. Once it reaches zero and `do-retry` is called, the activity is not retried and the failure is escalated to the parent plan. How often an individual activity is retried depends on the average relative occurrence of false positives when detecting failures. This information comes from either (1) a human plan designer’s experience, or (2) the robot’s EMs. To keep generalized plans abstract and free of explicit task and domain knowledge, the latter is preferable. In the present work, I did not yet use EMs to decide on these counter values, although I think extracting that kind of knowledge from the EMs and generating retry amounts from it should be straight forward.

6.1.2 Global Failure Taxonomy

In order to not fixate specific failures in plans where they are expected, I define a failure taxonomy. In this taxonomy, failures are identified by the situations in which they occur, and what good handling practices are to deal with them. If a known failure occurs (or is predicted), more information about why it happens and how it can be prevented or handled can be read from the taxonomy. If a new failure (one that does not match any in the taxonomy) is identified, it is added alongside all information known about it. An example failure taxonomy is shown in Figure 6.1.

The starting point of identifying failures is a divergence between an expected path of action and an actual one. If at any point a generalized plan detects a failed sub task, this means the aspired action failed. Based on its context, and the implicit and explicit parametrization this plan knows the circumstances in which this failure appeared.

The taxonomy itself starts at a very crude level: Any failed task matches the root node. From there on, child nodes are matched until either (1) a node is a perfect match or (2) a new leaf is created. Nodes are identified by: A parent task, a failed sub task, a primitive failure symbol,

and any information valid for that failure symbol (reaching motion failed: which arm was used; grasping object failed: used arm, object type; location unreachable: symbolic description of the location; etc.).

When a failure occurs and a failure handling strategy for a specific node in the taxonomy is required, the following situations can lead to one:

- (o) A strategy is mapped to the node: Use that strategy
- (o) No strategy is mapped to the node: Move up one parent and try their strategy

Ultimately, when no parent is left to try, the plan control is escalated to the next higher plan. This plan now has a broader view on the task that is performed and uses the same approach as the sub plan, but with a different failure: Now the originally failed plan is the failed sub plan. If all plans along an execution depth failed and the control is handed up beyond the root plan, user intervention is required. In the example in Figure 6.1 the “Search Object” plan does not have any failure handling associated for failures occurring in the “Perceive” sub plan. It escalates into the more general “Fetch” plan it is part of. The original failure — “Object not found” — is now present in the node, and has a handling strategy available: Reparametrizing the object to look for. The sub plan is then re-executed with the new parameter.

Initially, nodes have no failure handling strategies mapped to themselves. As described for Episodic Memories above, failures and successful retries of actions after reparametrizing them can be recorded and extracted from robot experience data. These reparametrizations make up the valid failure handling strategies for failed plans — and if they fail, the second of the above rules applies.

6.1.3 Uncertainty for Mobile Robots in the Real World

When operating in the real world, uncertainty is an omnipresent constraint for robots. In this section, after discussing the principle types of uncertainty that robot plans commonly encounter, and briefly commenting on other approaches, I will show how my generalized plans resolve, or effectively work around different types of uncertainty.

Regan *et. al* [74] separated uncertainty into seven distinct classes, which were later described in more detail and compared to one another by Uusitalo *et. al* [95]: (1) Inherent randomness, (2) Measurement error, (3) Systematic error, (4) Natural variation, (5) Model uncertainty, (6) Subjective judgement, and (7) Linguistic uncertainty. Of these, I deem the following three relevant for generalized plans:

Measurement error Sensor systems as a prime source of continuous data from the environment are prone to noise, drift, interference, and discretization errors (Nyquist-Shannon sampling frequency dilemma [80]). Especially perception (2D, 3D) systems suffer from perturbations in data (lighting, motion blur) and require specialized, probabilistic algorithms to even them out.

Model uncertainty All models derived from realistic environments inherently abstract from irrelevant details, let alone to overcome the Frame Problem’s intractability issues. These abstractions can eliminate not yet identified vital process variables. If gone unnoticed, this results in *cause and effect relationship* errors that are very difficult to spot or even quantify.

Subjective judgement Judgemental algorithms rely on known structures in data, and are easily misled by data containing new characteristics, producing wrong results. The scope, range, and distribution of models and raw input data are especially difficult to predict and judge for scarce and error prone data.

For symbolic, ambiguous parametrization of generalized plans, I add a fourth class:

Uncertainty Type	Symbolic	Subsymbolic
Measurement error		✓
Model uncertainty	✓	✓
Subjective judgement	✓	✓
Description vagueness	✓	

Table 6.2: Different types of uncertainty and their categorization by applicability to symbolic and subsymbolic aspects of a robot’s cognitive architecture

Description vagueness Symbolic entity descriptions for objects, locations, and actions are specified in varying degree of detail. For descriptions referencing real world entities, vague descriptions must be disambiguated before resulting in actionable items usable by robot plans. The quality of these results strongly depends on the knowledge available to the robot. I enlist this category as uncertainty since a description’s meaning — and thus a plan’s course of action — can change drastically based on its content and resolution, of which both are not necessarily known at design time.

These four classes can further be distinguished by whether they apply to symbolic, subsymbolic, or both aspects of a robot’s cognitive architecture, as shown in Table 6.2. The *Description vagueness* uncertainty per se only applies to symbolic task aspects. It should — rather than being treated as a problem — be seen as a degree of freedom, enabling “opportunities” for how a robot solves its task. Before being actionable, it ultimately still requires full resolution.

To alleviate uncertainty, most planning approaches, learning methods, and perception algorithms avoid the open world paradigm and implicitly assume a closed set or narrow range of possible solutions [26, 29, 106]. Talamadupula *et al.* [92] present an approach of how to apply a closed world planner to an open world environment by dynamically reconfiguring the planner based on observations made in the open world. Their approach is tailored precisely to the requirements of their scenario, and planners in general lack this functionality completely.

Generalized plans use two strategies for dealing with uncertainty: (1) Exhaustible backtracking over the search space, and (2) Experience-based search space constraintment.

- (1) **Exhaustible backtracking:** Probabilistically sampling from a linear search space or traversing over a set of discrete solution candidates until either a sufficient amount of valid solutions is found, or a frustration limit is reached and the search is cancelled due to exhaustion.
- (2) **Search space constraintment:** Explicitly valid and invalid parameter ranges are identified from Episodic Memories, resulting in Expectation Models. Inverting these models allows identification of viable parameter ranges for desired task outcomes, constraining the search space to proven to work priors. This is elaborated on more in Section 5.5.1.

Initially, without prior experience from EMs, only (1) applies to any generalized plans. Static rules may be present in an external knowledge base, but the executing robot explores the full (accessible) search space. For any uncertainty that is encountered, generalized plans feature heuristic fallback solutions as per Section 4.1.2. The solutions attempted are ranked based on their reciprocal generality. For the symbolic case, an example PROLOG implementation using the CRAM architecture for — if unspecified — determining which hand to use for grasping looks like this:

```
;; Prolog rules determining hand to use for grasping
(<- (use-hand ?task ?object ?hand)
  (desig-prop ?task (hand ?hand))) ;; Is the task defining a hand?
```

```

(<- (use-hand ?task ?object ?hand)
     (desig-prop ?object (hand ?hand))) ;; Is the object defining a hand?

(<- (use-hand ?task ?object ?hand)
     ;; Is a rule defined in the general knowledge base for ..
     (or (kb-triple ?task ?object ?hand) ;; .. the task/object combination?
          (kb-triple ?task ?_ ?hand)      ;; .. just the task?
          (kb-triple ?_ ?object ?hand))) ;; .. just the object?

(<- (use-hand ?task ?object ?hand)
     ;; Fallback solution: Use whatever hands are free, or fail if none available
     (free-hand ?hand))

```

The rules stated herein are processed top-down: Hands defined in the task description take precedence over hands defined in the object description; after that, rules defined in a knowledge base are considered (from specific to general), and finally, if no other rules apply, whatever hand is free is considered. Should none of these rules yield results, there is no valid solution for the required information and the calling task should reconsider its requirements, change its strategy, or fail gracefully and hand control back to its parent task.

In CRAM, a function that calls upon these PROLOG rules can have the following form:

```

(defun available-hands-iterator (task object)
  (lazy-mapcar ;; Create lazy 'just-in-time' list iterator over all solutions
               (lambda (bindings) (with-vars-bound (?hand) bindings ?hand))
               (prolog '(use-hands ,task ,object ?hand)))) ;; Call Prolog

```

This function iterates over all possible solutions if any previously returned solution turns out to be infeasible. The `prolog` call returns a lazy list of solutions: Each solution contains one binding for the `?hand` parameter.

Once significant amounts of EMs are collected, ExMods as described in Section 5.5.1 can be generated. This enables (2), allowing the robot to learn how to handle the intrinsic uncertainties of any task it performs. This strategy allows synergies between different (sub)tasks with similar structure. The constraintment does not rule out regions of the search space indefinitely though. The same rules as for (1) apply: If the learned, more specialized solution yields no feasible results, the more general solution fallbacks and heuristics are used. This allows to explore new tasks that have different characteristics than previous ones, creating new specializations.

Based on the abstract and underspecified nature of generalized plans themselves, the notion of unknown solutions and insufficient descriptions of tasks, objects, or locations as in *Description vagueness* is a vital component: It identifies where situational or inferred knowledge needs to be assumed, or (in the case of still too little information to perform a task) when to query a human operator for more information. This is a major strength of plans designed with this principle in mind — they can be defined very generically and can then be applied to a large variety of situations. Fetching a glass from the kitchen (which possibly resides in a cupboard) is — speaking in general terms — not very different from fetching the newspaper from the lawn.

6.2 Raw Interfaces to the External World

While pure cognitivist systems do not need to concern themselves with an outer, external world, robots require means for perceiving their environment and acting upon it. Many interfaces exist

for exactly these purposes — for my work, I chose two distinct frameworks on which I build the embodiment of generalized plans: (1) ROBOSHERLOCK as a semantic perception system, and (2) MoveIt! as an abstraction framework for motion planning under physical constraints. I consider both components raw in terms of the semantic connection between them and generalized plans as they per se have no means of understanding the abstract, vague set of descriptions that my approach uses. After introducing the components themselves, I will explain which transformations from a generalized plan’s knowledge content are necessary to parametrize them, and how to interpret the results they return.

6.2.1 ROBOSHERLOCK: Semantic Perception

ROBOSHERLOCK is a UIMA [30] based ensemble of Computer Vision (CV) expert algorithms that, based on a majority voting mechanism, identifies features in captured image data [7]. Examples for available CV experts include cluster segmentation in 3D point clouds, shape detection (2D, 3D) using RANSAC, color segmentation, template matching, SIFT feature matching, and CAD model fitting. ROBOSHERLOCK accepts object descriptions based on a number of criteria, and tries to find a match between currently visible objects and the supplied description. If a match is found, all known characteristics from all experts (whether asked for or not) are returned. Common queries have the following form:

```
(an object ((color red) (shape cylinder) (size small)))
```

Based on this query, the perception system would return all object instances that are predominantly red, cylindrical, and relatively small. The last criterion is highly subjective, but since ROBOSHERLOCK is also developed in, and specialized on the household domain, its understanding of object sizes coincides with that of my own generalized plans.

The expected return value when querying a robot perception system depends on the purpose of the result data. High level control programs — and cognitive architectures at large — mostly operate on a symbolic and conceptual level, and therefore expect a perception system to return symbolic information. Depending on the task, this can involve either exact 6-DOF poses of objects in vicinity, their qualitative characteristics (shape, size, primary color, etc.), or only whether they are located on a particular table. Besides the granularity of the contained information, semantic perception systems should carry intelligence on their own. Instead of just returning whatever information they have at their disposal, a high level control program might query for particular types of objects, of a certain color, all objects in a drawer but not around it, or objects in the robot’s hand. To answer these queries, that system requires knowledge itself, elaborate filtering mechanisms, and ideally a set of algorithms that help reduce uncertainty about the results by — as in ROBOSHERLOCK— applying e.g. a majority vote over different experts for the same purpose in case of ambiguity.

To interface ROBOSHERLOCK with the CRAM plan language, I implemented a set of modules that abstracts away from ROBOSHERLOCK’s expected input and output format and automatically translates all data into CRAM compatible high level information: `cram_uima` and the `robosherlock_process_module`¹.

6.2.2 MoveIt!: Abstract, Constrained Motion Planning

MoveIt! is a framework that combines the motion planning algorithms from The Open Motion Planning Library (OMPL) with a managed, persistent planning scene, sensor integration for reactive perception, and interfaces to common programming languages. It ties all components together into a streamlined pipeline that receives motion (planning) requests and can return a multitude of outcome codes. Of the included components, I mostly make use of the first

¹https://github.com/fairlight1337/cram_perception

two: I use OMPL's motion planning capabilities while populating MoveIt!'s planning scene with information about the current collision environment.

For a seamless integration with high level control programs, both the motion requests and the final outcome need to be translated between symbolic and subsymbolic representations. Requests for motion planning or execution are either (1) 3-DOF or 6-DOF poses relative to the manipulator's root link, or (2) trajectories to be executed by the joint controllers of the controlled kinematic manipulator. In the case of MoveIt!, the former is able to generate the latter. Optionally, a planned trajectory can be executed right away, but if post-processing is necessary — as in dual arm motion described below — the generated trajectory can be returned, altered, and executed at a later stage. Likewise, the result codes returned by a motion planning framework can only consist of the knowledge the framework actually has: Planning succeeded, planning failed (e.g. due to collision, no IK solution found, time limit exhausted, or out of memory), motion execution succeeded, motion execution failed (e.g. controller failure), or a multitude of internal problems (no planner loaded, framework not initialized, etc.). Both, the input and output data need to be translated, which I discuss in more detail in Section 6.3.

I implemented a comprehensive interface module for the CRAM plan language to control MoveIt!'s motion planning and collision environment management functionalities, as well as to interpret its result codes appropriately: `cram_moveit`².

Dual-Arm Motion Planning

Wielding two or more arms simultaneously is necessary for manipulation tasks in which objects are either too heavy for one manipulator, or the object dimensions are too bulky for single-handed manipulation. Examples are trays, and heavy pots or plates. Another area that benefits from simultaneous (or even asynchronous) motion planning is motion blending — beginning the next motion while the current one is not yet finished. While this can save time and resources, it is beyond the scope of my work and I leave it open for further research.

OMPL uses randomized motion planners. This — and lots of practical experience while using it — nurtures the assumption that heuristics such as *"Try a straight line in cartesian space first"* are not used. Thus, planning motions for multiple arms for manipulation poses a number of challenges:

- (o) **High search space complexity:** Since both arms are independent from one another, every solution for one arm must be cross-validated against every other arm. When a complexity for n links per arm is assumed as $\mathcal{O}(n)$ and k arms are involved, the overall complexity rises to $\mathcal{O}(n^k)$. This results in a very long planning time which possibly violates any given timeouts.
- (o) **Non-optimal solutions:** Due to the fact that generating a single solution is very complex, the planning algorithm would not have much time left to find a more optimal one. Additionally, since the initial motion found (if any) has a high chance of being strongly entangled, path simplification would not be able to deskew it. The resulting motion when executed on a robot would look rather odd and use a large part of the available workspace.
- (o) **Long execution time for entangled trajectories:** Based on the points above, the resulting trajectories take very long to execute compared to their separated individual trajectories, prolonging the overall task time.

Motion planning — especially when considering multiple end effectors and separate kinematic chains — is a research field on its own. To still be able to manipulate with both arms on a household robot, I circumvent the above problems by separating the available work space into

²https://github.com/fairlight1337/cram_moveit

a left and a right part, relative to the robot's base. This way, each arm is only allowed to stay on one side of the robot's coordinate system, implicitly preventing entangled or colliding trajectories. The complexity rises by a mere constant factor, $\mathcal{O}(2 \cdot n)$ which, again, is $\mathcal{O}(n)$. This heuristical solution makes some — very rarely used — motion trajectories impossible, but ultimately enables dual-handed manipulation without paying for it in complexity.

Another important remark to make here is that when dual-wielding objects, a closed kinematic chain is created. Motions of any link must be synchronized with all other links in the chain, as otherwise either part of the kinematic chain gets damaged, or the transported object is lost. Again, this is beyond the scope of my work and is left here as a side note.

6.3 Translating Symbolic and Subsymbolic Information

Both, object perception and object manipulation must be parametrized using object information available to the cognitive architecture. In my work, I use the knowledge processing system KNOWROB to encode that information in OWL. A sample encoding for an object "Fork" is shown in Figure 6.2. At the example of this object, I will explain the translation process for each interface.

6.3.1 Perception Queries and Results

Based on the object definition in Figure 6.2, a query to ROBOSHERLOCK via my CRAM interface (the `robosherlock_process_module`) takes on the following form:

```
(an action ((:to :perceive)
            (:obj (an object ((:shape :box) (:color "d3d3d3") (:type "Fork"))))))
```

The details `:shape`, `:color`, and `:type` are read directly from KNOWROB's knowledge base interface to CRAM. To translate between the available and the expected information is part of the process module's tasks. The return value of this query would be similar in structure and content to the following example:

```
(an object ((:pose <6-DOF pose>) (:colors ((:red 0.82) (:green 0.82) (:blue 0.82)))
           (:shape :box) (:dimensions ((:x 0.05) (:y 0.03) (:z 0.12)))
           (:detection ((:clusterid 0) (:objectid 3))) (:timestamp 1421539200)))
```

Parts of the results can be interpreted as they are, others need to be transformed: Is the `:pose` related to any particular piece of furniture — on a table, inside a drawer, on the ground? The cognitive architecture takes the burden of checking all furniture of interest to see whether this object is above its supporting surface (plus some threshold margin to account for perception errors).

6.3.2 Motion Planning Queries and Results

To manipulate an object like the one described in Figure 6.2, the appropriate manipulation poses relative to the kinematic chain's root link of the end effector in use need to be determined. In terms of e.g. grasping the object, this means calculating the correct end effector pose relative to the global object pose, taking into account all applicable offsets.

Figure 6.3 depicts this situation. A cup-like object with a handle on its side must be grasped. The end effector controllable (in world coordinates) through motion planning is the robot's wrist. The origin of the object in world coordinates is known. The object's handle has a pose relative to the object's origin. The robot's Tool Center Point (TCP) has a pose relative to the wrist. I formulate a homogeneous transformation between two coordinate systems as T_A^B where A is the reference coordinate system and B the coordinate system whose origin is described by T ,

```

<owl:Class rdf:about="&knowrob;Fork">
  <rdfs:subClassOf rdf:resource="&knowrob;LTFnPObject"/>

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&knowrob;pathToURDFModel"/>
      <owl:hasValue rdf:datatype="&xsd:string">
        package://ltnp_models/models/fork/fork.urdf
      </owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&knowrob;boundingBoxSize"/>
      <owl:hasValue rdf:datatype="&xsd:string">0.09 0.07 0.19</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&knowrob;primitiveShape"/>
      <owl:hasValue rdf:datatype="&xsd:string">box</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&knowrob;color"/>
      <owl:hasValue rdf:datatype="&xsd:string">d3d3d3</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&knowrob;semanticHandle"/>
      <owl:hasValue rdf:resource="&knowrob;Fork_Handle_laok8b26"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:NamedIndividual rdf:about="&knowrob;Fork_Handle_laok8b26">
  <rdf:type rdf:resource="&knowrob;SemanticHandle"/>
  <knowrob:handlePose rdf:resource="&knowrob;Fork_Handle_laok8b26_pose"/>
  <knowrob:graspType rdf:datatype="&xsd:string">push</knowrob:graspType>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="&knowrob;Fork_Handle_laok8b26_pose">
  <rdf:type rdf:resource="&knowrob;Transformation"/>
  <knowrob:quaternion rdf:datatype="&xsd:string">0.5 0.5 -0.5 0.5</knowrob:quaternion>
  <knowrob:translation rdf:datatype="&xsd:string">0.0 0.0 0.03</knowrob:translation>
</owl:NamedIndividual>

```

Figure 6.2: Excerpt of object information about an object “Fork” encoded in KNOWROB. The full file is available here: https://github.com/fairlight1337/longterm_fetch_and_place/blob/master/ltnp_models/owl/objects.owl

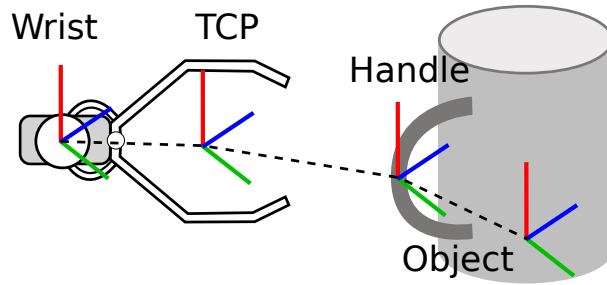


Figure 6.3: Relevant coordinate systems for grasping an object by its handle: Robot wrist, tool center point, object origin, and relative object handle

relative to A 's origin. In general, $(T_A^B)^{-1} = T_B^A$. Two transformations T_A^B and T_B^C are appended by multiplying them: $T_A^C = T_A^B \cdot T_B^C$. The transformations used in this example thus are:

$$\begin{array}{ll}
 T_w^W & \text{world} \rightarrow \mathbf{Wrist} \\
 T_W^T & \mathbf{Wrist} \rightarrow \mathbf{TCP} \\
 T_w^O & \text{world} \rightarrow \mathbf{Object} \\
 T_O^H & \mathbf{Object} \rightarrow \mathbf{Handle} \\
 T_T^H & \mathbf{TCP} \rightarrow \mathbf{Handle}
 \end{array}$$

The transformation in question is T_w^W : How to place the wrist such that the TCP correctly grasps the object's handle. Any required offset between the TCP and the handle is defined by T_T^H — commonly this is the identity transformation for grasp poses, and an offset for pregrasp poses. According to Figure 6.3, the straight forward solution (starting from the wrist) is:

$$T_w^W = T_W^T \cdot T_T^H \cdot T_H^O \cdot T_w^O$$

Taking into account that only T_W^T , T_T^H , T_O^H , and T_w^O are known, the solution changes to:

$$T_w^W = T_W^T \cdot T_T^H \cdot (T_O^H)^{-1} \cdot (T_w^O)^{-1}$$

The transformations T_W^T and T_O^H are static. The final formulation for where to place the robot's wrist after detecting and then trying to grasp an object by its handle thus becomes a function of the object pose and the desired grasp offset:

$$T_w^W := T_w^W(T_w^O, T_T^H)$$

The former is the result of a successful perception query, the latter is available from the task being executed. The static transformations are supplied by the knowledge base — the robot's knowledge about itself for T_W^T , and the object description for T_O^H . The parametrization of the motion framework therefore becomes a product of other components' output, properly directed by the plan execution architecture.

After issuing a motion planning or execution request, the motion planning framework returns a result code depending on its success — see Section 6.2.2. This result needs to be translated into failure classes the overarching plan execution architecture can understand and handle. From all possible result codes MoveIt! can return³, I reduced all cases to the following four that convey semantic meaning for the controlling robot plans:

- (o) **:success:** The request was fulfilled successfully.
- (o) **:planning-failed:** The motion planning request failed (no collision free path exists be-

³https://github.com/fairlight1337/cram_moveit/blob/master/cram_moveit/src/failures.lisp

tween start and goal pose).

- (o) `:no-ik-solution`: No valid IK solution could be found (the goal pose is unreachable or in collision)
- (o) `:control-failed`: During motion execution, the motor controllers did not move as expected.

With these result types, a generalized plan can distinguish between failures it can handle (`:planning-failed`, `:no-ik-solution`) or needs to escalate eventually (`:control-failed`); see also Section 6.1 for handling failures of both classes. Also, every plan can decide which measures to take: Upon `:no-ik-solution`, a grasping plan can try a different handle on the object if available; a putdown plan needs to reposition the robot's base or find a new place to put the object down at. For `:planning-failed`, a heuristical approach works well when using randomized motion planners: Retrying n times, then transforming the failure to a different class for alternative handling (such as `:no-ik-solution`).

6.4 Maintaining a Dynamic Planning Scene for Manipulation

The planning scene is the motion planning framework's central data structure for maintaining information about the collision environment. Whenever a robot changes the state of the world around it, or detects such changes, the planning scene must be updated to reflect this accordingly. An intact planning scene is the prime requirement for a properly functioning motion planner in constrained environments. If this is violated, plans start off on wrong assumptions on why a problem during manipulation comes up, which actions they have available (based on reachability of objects and furniture), and produce behavior totally out of sync with what a plan developer expects to happen. The events after which the planning scene must be updated include, but are not limited to:

- (o) **Detecting an object**: Add a collision model for the detected object to the environment
- (o) **Picking up an object**: Attach the object to the robot's end effector(s), remove it from the environment
- (o) **Putting down an object**: Detach the object from the robot's end effector(s), add it to the environment at the destined putdown pose
- (o) **Opening or closing a container (drawer, fridge, dishwasher, etc.)**: Reposition the door's/drawer's collision model accordingly

To manipulate collision objects in CARAM, I developed a number of functions that interface with MoveIt!'s planning scene management functions:

- (o) `register-collision-object`: Takes an object designator as its parameter. The designator should include the object's name, its (bounding box) dimensions, and its shape. A respective collision model is created, but not yet added to the planning scene. Example:

```
(register-collision-object
  (an object ((:name "cup0") (:shape :box)
              (:dimensions ((:w 0.03) (:d 0.05) (:h 0.21))))))
```

- (o) `add-collision-object`: Takes an object name and a pose as its parameters. The collision object identified by the name is added to the planning scene at the given pose. Example:

```
(add-collision-object "cup0" <6-DOF pose>)
```

- (o) `remove-collision-object`: Takes an object name as its parameter. Removes the named collision object from the planning scene. Example:

```
(remove-collision-object "cup0")
```

- (o) `without-collision-object`: Environment that temporarily removes a named collision object from the planning scene. Example:

```
(without-collision-object "cup0"  
  ;; Force grasps penetrate objects; collision checking would forbid that  
  (perform-force-grasp "cup0"))
```

For removing multiple objects simultaneously, `without-collision-objects` exists that uses a list of object names.

- (o) `clear-collision-environment`: Removes all known collision objects from the planning scene and purges the registered collision objects. Example:

```
(clear-collision-environment)
```

- (o) `attach-collision-object-to-link`: Takes an object name and a target link as its parameters. The named collision object is removed from the planning scene and is attached to the given link, extending the link's collision model. Example:

```
(attach-collision-object-to-link "cup0" "left_hand")
```

- (o) `detach-collision-object-from-link`: Takes an object name and a target link as its parameters. The named collision object is added to the planning scene at its current pose and is detached from the given link. Example:

```
(detach-collision-object-from-link "cup0" "left_hand")
```

All functions can be called by any plan at any abstraction layer. Reasons for managing the collision environment can have their origin in high level plans that decide which objects to ignore for planning reasons (simplifying a too complex collision scene), or in lower level plans that for example enable force or push grasps that would penetrate the object's collision model.

6.5 Navigation in Semantically Known Environments

Besides primary sensing and acting interfaces, a mobile manipulation robot needs navigational capabilities. The basic mode of operation for robot bases is movement in two-dimensional cartesian space in relative coordinates. In my scenario, I rely on a static ground map that never changes during operation of the robot, and that is known at all times (see Figure 1.4). This map introduces a reference coordinate system for the robot's base. To make use of global (map relative) coordinates ("*In front of fridge*", "*Close to meal table*"), the navigating robot requires two prerequisites: (1) Localization, and (2) a coordinate system transformation framework.

- (o) **Localization**: A robot knowing its own pose in a reference coordinate system is the result of a localization process. While a large variety of localization algorithms and approaches exists [69], I rely on the Adaptive Monte Carlo Localization (AMCL), a probabilistic localization system for moving a robot in 2D [35]. It uses a particle filter for tracking a robot's pose against a known map. For all robot platforms I use, a properly configured AMCL implementation exists.

In ambiguous environments, localization can resort to a wrong local feature match maximum, resulting in the Kidnapped Robot Problem [27]. The kitchen environment in my scenario does not suffer from this effect.

- (o) **Coordinate System Transformation Framework:** Often, target coordinates for navigation tasks are specified in coordinate systems other than global (map) coordinates. Examples are: *"One meter in front of the fridge, facing the fridge door"* or *"Left of the table"*. With every piece of furniture being grounded in global map coordinates, the kinematic transformations from such relative coordinates to the global frame are known. To efficiently transform poses between these representations, I use Tully Foote's The Transform Library (TF) [33]. I use the same library for coordinate transformations during manipulation activities, especially for grasp planning — see Section 6.3.2.

Resolving symbolic descriptions of locations (e.g. *"In front of fridge"*) is done by the plan execution architecture. The main means for describing a location are location designators. These can contain either fully qualified 6-DOF poses, or symbolic descriptions. Common location designators are:

```
;; Global 6-DOF pose
(a location ((:pose <6-DOF pose>)))

;; Relative pose in gripper
(a location ((:in :gripper) (:side :left) (:offset <6-DOF pose>)))

;; Relative pose on a counter top
(a location ((:on :countertop) (:name "meal_table") (:offset <6-DOF pose>)))

;; Floor pose relative to a piece of furniture
(a location ((:relative-to "meal_table") (:offset <6-DOF pose>)))
```

Transforming these descriptions into actual numerical values for positioning the robot base (x, y, θ), I make use of CRAM's costmap feature. Costmaps are areas of discretely distributed probability from which a random sample is chosen. Each region can have a different probability weight, resulting in maps of interesting points with adjacent, less interesting areas around them. Typical probability distributions used in costmaps are Gaussians. Costmaps are either heuristically defined by describing the area in question with mathematical formulations, or are learned as described in Section 5.6.1.

6.6 Summary

In the current chapter, I discussed the vital interfaces necessary between a robot and its surrounding world, and how a robot's architecture uses them for embodiment in the real world. I presented the following three most important interface types: (1) Visual perception for interpreting the state of the world, objects, and furniture around the robot, (2) physical, constrained manipulation using one or more robot arms, and (3) navigation around a relative or a global coordinate system. Additionally, I went into detail about occurring failures in the real world, and how they can differ from the theoretical foundations in earlier chapters.

For all interface types, I described how symbolic high level knowledge is translated into sub-symbolic information used by low level systems, and how their result values are interpreted. Overall, this chapter enables the implementation and operation of all previously presented topics on actual robots, be it simulated or on actual hardware. It thus poses a vital component in my overall reasoning as implementing and executing puts theoretical concepts to test, validating or falsifying their purpose.

Evaluation

“There are three principal means of acquiring knowledge: Observation of nature, reflection, and experimentation. Observation collects facts; reflection combines them; experimentation verifies the result of that combination.”

— Denis Diderot [24]

For the majority of the presented topics, I have shown comprehensive evaluations in my previous publications. These include A* based planning for generalized plans [100], knowledge-enabled plan parametrization [101], design principles of generalized plans [102], action effect prediction and task models [103], and the recording, detailed characteristics, and general use of Episodic Memories [105]. Additionally, where I saw fit, I added an evaluation section to the respective topic in this thesis in form of concrete examples. This includes:

- (o) *Plan Design for Reactive Task Monitoring in Mobile Manipulators* (Section 4.3.4),
- (o) *An A*-based Planner for Generalized Actions* (Section 4.5.1),
- (o) *Expectation Models: Task Outcome Prediction* (Section 5.5.1),
- (o) *Prototypical Experiences: Informed Strategy Exploration* (Section 5.5.2), and
- (o) *Multi-modal Analysis of Robot Experiences* (Section 5.6)

In this evaluation chapter, I will concentrate on the remainder of the topics from the previous chapters: (1) Contextual parametrization of abstract generalized plans with a strong dependency on external knowledge, (2) embodiment of an autonomous robot in a household environment with specialized as well as heuristic failure handling, and (3) automated experiment conduct with a strong focus on EM generation and variation of parameters to create diverse robot experience data.

The structure of this chapter is as follows. First, I apply the experiment scenario motivated in Section 1.2 to an actual problem setting, describing the importance of contextual reasoning and external knowledge in an actual working environment. To this end, I use a simulated environment featuring a PR2 robot that performs meal table setting tasks in a human household kitchen. I will then show the role of generalized plans in this scenario, and what their pros and cons are, in relation to Chapter 4. Finally, I go into generation of variation-rich, automated experiments using my AutoExperimenter (AE) tool as introduced in Section 5.8.2. The product of these experiments are Episodic Memories that serve as a source for all concepts described in Chapter 5.

The whole experiment environment as used during the development of the derived concepts in this thesis is available online¹.

7.1 Experiment Scenario

The scenario on which I will enlarge is based on the experiment context from Section 1.2. More precisely, I generate concrete problem instances of the table setting scenario along three dimensions: (1) The number and identity of meal participants, (2) the time of the day — and thus the type of meal prepared, and (3) the day of the week.

¹https://github.com/fairlight1337/longterm_fetch_and_place

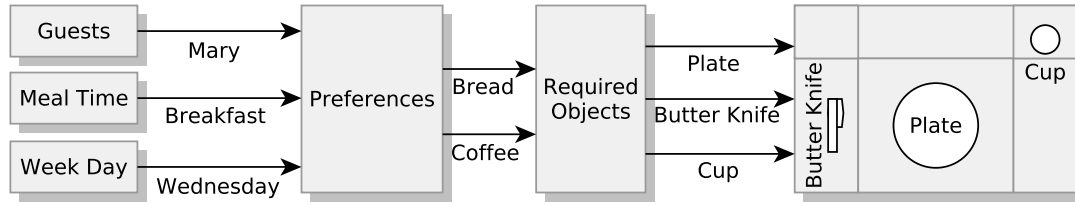


Figure 7.1: Problem inference for Mary as the only guest on a Wednesday breakfast. It results in having a plate in the center of the seat area, a butter knife to the left of it, and a cup to the back right.

Algorithm 7.11 Problem generator for table setting tasks. Guests, week day, and time of day (meal time) are chosen. The generator follows a simple set of rules to give the tasks more semantic meaning.

```

1 (defun generate-table-setting-problem ()
2   (let* ((pool-guests '(:mary :tim))
3         (pool-meal-times '(:breakfast :lunch :dinner))
4         (pool-week-days
5          '(:monday :tuesday :wednesday :thursday :friday :saturday :sunday))
6         (week-day (nth (random (length pool-week-days)) pool-week-days))
7         (meal-time (nth (random (length pool-meal-times)) pool-meal-times))
8         (guests (if (and (not (weekend-p week-day))
9                       (eql meal-time :lunch))
10                    '(:,(nth (random (length pool-guests)) pool-guests))
11                    (let ((temp-guests
12                          (loop for guest in pool-guests
13                             when (>= (random 10)
14                                       (if (weekend-p week-day) 2 5))
15                               collect guest))))
16                    (if (= (length temp-guests) 0)
17                        '(:,(nth (random (length pool-guests)) pool-guests))
18                        temp-guests))))))
19   '(:,guests ,meal-time ,week-day)))

```

An example problem inference is shown in Figure 7.1. Therein, a guest named Mary has breakfast on a Wednesday. Through the use of external knowledge about Mary’s preferences (learned, dynamic knowledge), the types of tableware and consumables required for her meal (static knowledge) and where each type goes on the table (situationally dependent knowledge), a complete set of fetch and place activities is defined. To cover all dimensions of the problem’s search space when generating instances, Algorithm 7.11 shows an example implementation of a problem generator. It follows a set of very simple rules, applies the reasoning steps described in the experiment scenario, and produces a list of required objects — and where they should be placed — for each problem instance.

The rules used in the generator are:

- (o) The potential guests `tim` and `mary` never have `lunch` together on workdays.
- (o) On workdays, `tim` and `mary` have a 20% chance of meeting during `breakfast` or `dinner`, and 50% on the weekend.
- (o) At least one guest is present during every meal.

In Mary’s example given above, she likes to have bread and coffee for breakfast during the

Object	Destination
(an object (:type "Plate"))	(a location (:centerof :seat))
(an object (:type "Knife"))	(a location (:leftof :seat))
(an object (:type "Cup"))	(a location (:rightof :seat) (:behind :seat))

Table 7.1: Resulting set of fetch and place tasks from the example problem instance in Figure 7.1.

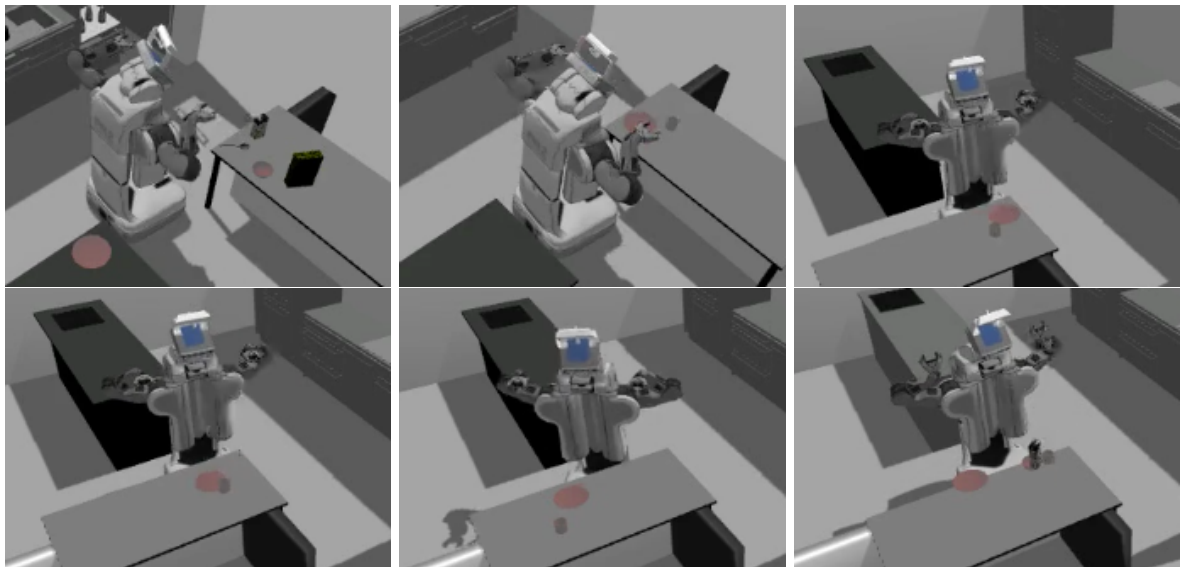


Figure 7.2: Selection of final experiment states for table setting

week (according to Table 1.1a). Based on the data shown in Table 1.1b, bread requires a plate and a (butter) knife while coffee just needs a cup. The resulting objects then need to be placed according to the knowledge from Table 1.2. The meal objects need to be placed on the meal table shown in the lower right of Figure 1.4. The surrounding kitchen — on counters, in drawers, in the fridge — contains the objects required. The operating robot now has a set of tasks to achieve, as shown in Table 7.1.

Two principle types of knowledge are used here, as already described in Section 4.1.2 and 4.4: Static knowledge (object places on the table; also quasi-static knowledge: Mary’s preferences), and situational context (where does everyone sit, what needs to be placed). Both types may not be part of the robot’s internally encoded plans, but must be fed from an external knowledge base or must be dynamically inferred. The reason is simple: Static knowledge changes in-between tasks, and where which piece of tableware goes absolutely depends on the current situation.

I performed a number of experiments represented by roughly 800 EMs of table setting instances. Figure 7.2 shows a selection of their final states. Two different characteristics become apparent here: There are qualitative differences (completely different meal set) and quantitative ones (same meal type — plate and cup — but different locations). This is directly in line with the concepts presented in Chapter 5. All objects are collected from the surrounding kitchen, and multiple objects of the same type can be present — the robot chooses the first matching object instance it finds.

7.1.1 Experimental Data

To underline the importance of data collection from Generalized Plans, I collected a set of 425 experiments that have shown representative characteristics of the tasks I examine:

- (o) They give an idea of how central each task is, based on its number of occurrences.

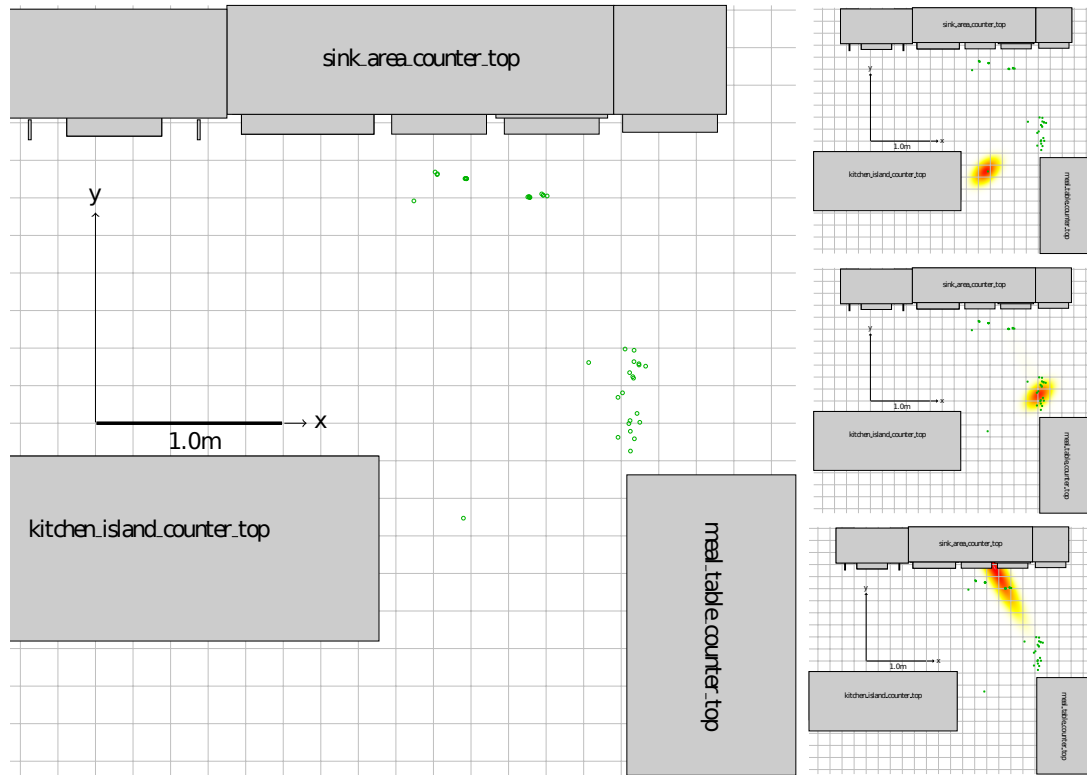


Figure 7.3: Reiteration of Figure 5.12 to make the point at hand clear.

- (o) They display the time required for individual tasks, confirming intuition.
- (o) They show the sensitivity of certain tasks to failures.

Table 7.2 shows the numerical analysis of these 425 experiments. For each task, its total number of occurrences is shown, alongside a time histogram and the number of failures that were recorded for this task type. The histogram takes on the format of minimum value in seconds, distribution in 10 steps between minimum and maximum value, and the maximum value in seconds. Each “bar” shows the fraction of tasks that took an amount of time between two discrete steps of the histogram:

$$i \cdot \text{stepsize} \leq x - \min < (i + 1) \cdot \text{stepsize} \quad \text{stepsize} = (\max - \min)/10 \quad (7.1)$$

From the table, it becomes apparent that failure handling plays a major role even on a global scale over so many experiments: 164296 individual occurrences of `WithFailureHandling` outnumber any other task type by far. It is to be noted here that the actual failure was mapped to the higher level task using the `WithFailureHandling` construct to make their reason more clear. As described above, this kind of data is the source to construct prediction models of a task’s probability to fail due to a certain failure, or even how long a task will probably take, using confidence intervals.

This data and the task’s individual parametrizations are also the source for the gaussian distribution shown in Figure 5.12 in which base positions to stand at were evaluated when grasping an object. Here, I reiterate Figure 7.3 to make the point clear.

7.2 The Role of Generalized Plans

Generalized plans are meant to be as abstract as possible, but must still lead an autonomous robot to successful actions in the real, very non-abstract world. For example, a robot that needs

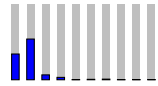
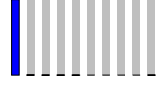

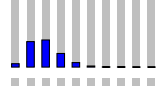
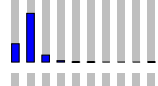
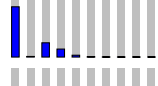
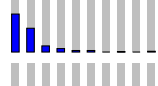
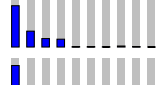



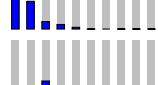
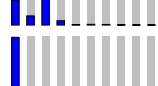

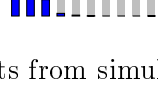
Task	Occurrences	Time Histogram (sec)	Manipulation Pose Unreachable	Location Not Reached	
LiftingAnObject	728	1.44 	15.70	0	0
WithFailureHandling	164296	0.00 	613.08	0	0
MotionPlanning	8831	0.07 	18.08	200	0
PerceivingObjects	3074	0.16 	0.77	0	0
CarryingAnObject	933	0.03 	38.20	0	0
PickingUpAnObject	2205	0.07 	37.25	1526	0
ClosingAGripper	197	0.00 	0.10	0	0
ArmMovement	8831	0.08 	18.11	4	0
FindingObjects	1821	0.02 	34.91	0	0
PuttingDownAnObject	2073	0.95 	568.71	12743	0
Navigate	9188	0.00 	35.11	0	2708
HeadMovement	9392	0.00 	3.10	0	0
MotionExecution	2818	1.40 	13.09	0	0
OpeningAGripper	948	0.00 	0.57	2	0
ParkingArms	1661	0.00 	28.01	96	0

Table 7.2: This numerical analysis of 425 experiments from simulation and from the real robot show the distribution of single task occurrences, their time consumption on a global scale, and their susceptibility to failure of two kinds: Unreachable manipulation poses and unreachable navigation goals.

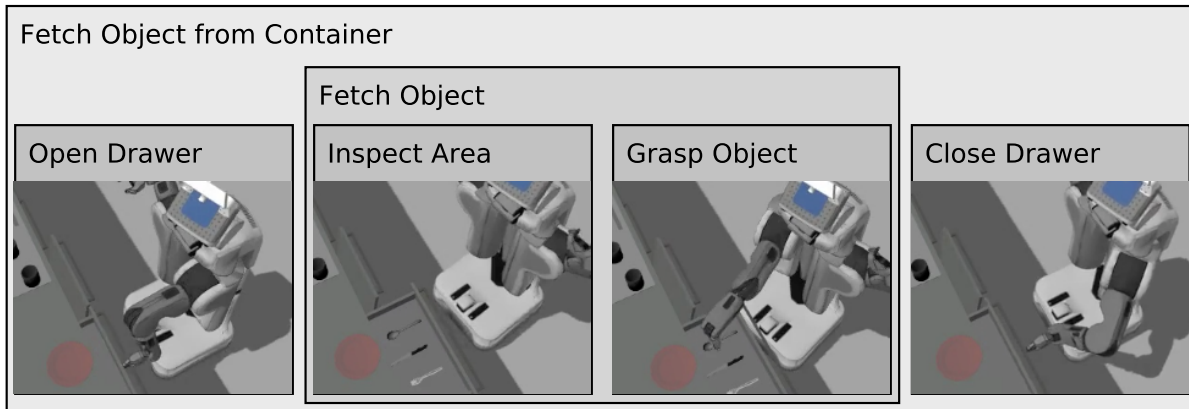


Figure 7.4: Composition of hierarchical generalized plans for fetching objects from containers. Here, a piece of cutlery is grasped from a drawer that was previously opened.

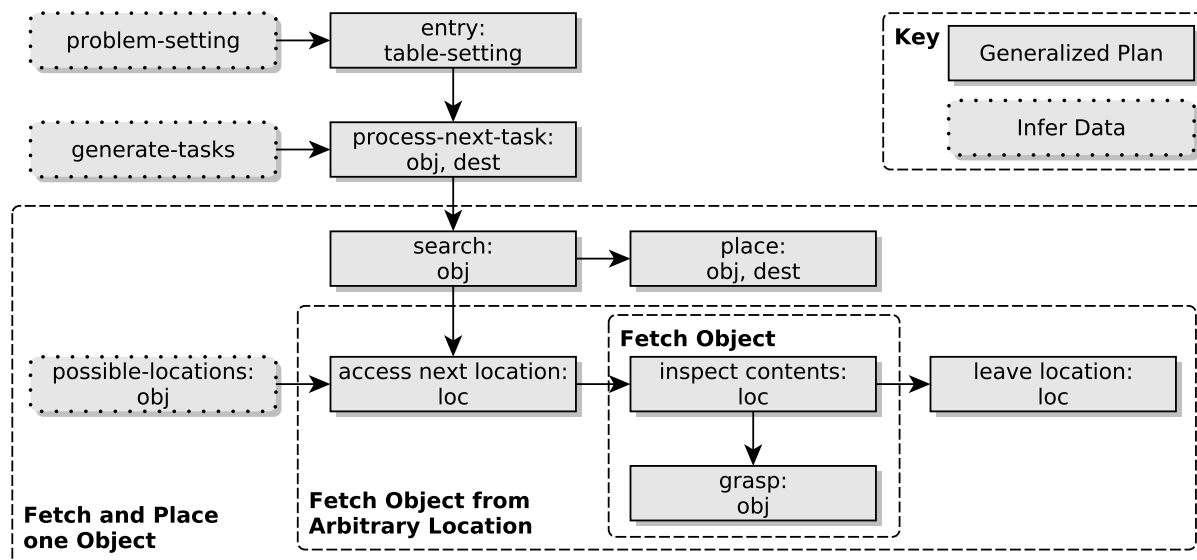


Figure 7.5: Typical phases visited during table setting using my generalized plans. This hierarchy puts the process shown in Figure 7.4 in a larger context.

to set a table according to the objects and destinations shown in Table 7.1 needs to achieve the following high level task for every pair of object *obj* and destination *loc*:

```
(achieve '(object-placed-at ,obj ,loc))
```

This task description is highly underspecified. For example, no notion is given of where the object is found, or at what precise coordinates it needs to be placed on the meal table. To enable a robot to perform the tasks, generalized plans make use of (1) their abstract structure, and (2) a fair amount of external knowledge.

- (1) My generalized plans for table setting largely make use of generalized sub plans for fetching and placing objects. Figure 7.4 shows the hierarchical composition of a task fetching objects from a drawer. The top level plan fetches objects from containers. It first opens the container, then calls the sub plan for fetching objects, and closes the container afterwards. These plans strictly follow the principles introduced in Chapter 4: Plans solve local problems and are unaware of the parent plan's semantics, but can be contextually constrained using the `with-context` environment. A (simplified) example based on the

actually executed plans is the handling of an open drawer's altered collision environment during manipulation:

```
(with-context ((:with-collision-objects (walls-of "drawer_top")))
  (fetch-object obj))
```

Figure 7.5 shows typically visited phases during a table setting instance.

- (2) The generalized plans make use of external knowledge to realize a concrete parametrization based on vague descriptions. A common task is to open a container: Drawers open prismatically, fridge doors have revolute joints. To cover both cases (and thus almost all cases a robot might encounter in human environments) I developed a mechanism for generating trajectories based on the knowledge available about the joint to actuate:

```
(let* ((joint-to-actuate "drawer_joint")
      (current-opening-degree (current-degree joint-to-actuate))
      (opening-trajectory (create-opening-trajectory joint-to-actuate
                                                    :from current-opening-degree :to 1.0)))
  (when opening-trajectory
    (execute-trajectory opening-trajectory)))
```

The "degree" of a joint is the normalized amount to which it is opened between its lower and upper actuation limits. The function `create-opening-trajectory` generates a trajectory that reaches from the current degree up to the desired degree. It is based on information about the joint, stored in the knowledge base: Its rotation axis \vec{R} , the position and orientation of the rotation axis M relative to the parent object, the pose of the grasped handle H relative to the rotation axis, and the type of the joint (prismatic, revolute). This results in a function that defines the exact pose (position and orientation) of a robot gripper for every opening degree of a joint. With a given granularity, this function is then sampled in equidistant steps, the resulting poses validated for existing Inverse Kinematics (IK) solutions, and executed on the robot.

A second important example of knowledge necessary for generalized plans are possible residence locations of an object. The naive fallback solution is to search all available locations (counter tops, drawers, fridge, dishwasher, oven, etc.). My plans follow a more informed approach before falling back:

- (o) Was the object found in the past? \rightarrow Search at that location again.
- (o) Does the object have a default place associated with it? \rightarrow Search at that location.
- (o) Does the object category belong at a specific place (or class of places)? \rightarrow Search at those locations.

This is aligned with the knowledge priority list discussed in Section 4.1.2.

As a default strategy in this scenario, if any unhandleable failure occurs in a generalized plan, it is escalated to its parent.

7.3 Insights about what did not work

While my experiments validate my claims about generalized behavior in robots that self-improve over time and through experience, I encountered a series of crucial learning steps that required me to develop new strategies and refine my approach. I present the most important ones here:

- (o) **Learning high level flow control operators in plans fosters overfitting:** It is tempting to let a learning algorithm make its decision based on *all* encountered plan

```

(def-plan set-table (required-objects table)
  (loop for object in required-objects
    as fetched-object = (fetch-object object)
    do (progn
      (if (object-found? fetched-object)
        (place-object fetched-object table)
        (progn
          (find-substitute fetched-object)
          [...])))))

(def-plan load-washing-machine (clothes-pile washing-machine)
  (loop for piece in clothes-pile
    as picked-piece = (fetch-object piece)
    do (progn
      (when (cloth-dirty? picked-piece)
        (place-object picked-piece washing-machine))))))

```

Table 7.3: Example of two simple plans where resulting experience models from both help benefit both plans by ignoring flow control operators such as `if` and `when`.

language elements. While flow control is important (and is correlated to the context’s parameters), ignoring it and only learning the higher level course of plans yields much better results. When considering every `if`, `unless`, or `case`, the resulting decision trees are only valid for behavior generated by action plans that actually feature these operators. I decided to carefully ignore them — and found that now plans *similar in structure but different in intention* can benefit from each other. For a simple example, see Table 7.3. Here, simplified plans of setting a table and loading a washing machine are similar in structure but differ in flow control operators. Ignoring them lets the experience models concentrate on the actual contextual and task parameters and the functional structure of the plans.

- (o) **Generality and Efficiency require a tradeoff consideration:** The untold expectation when letting robots perform tasks is that they should find the best (or near best) possible solution and apply it efficiently. This is not generally easy or possible: Finding the best solution requires comparison of all (or at least a significant amount of all) solutions. When describing the action itself symbolically, actually producing a solution that is actionable in the real world can be computationally expensive. An example: Grasping an object with one hand should be done as fast as possible. If the robot has two hands, the one that has the shortest travel path (considering collision environment) should be used. Calculating this solution can easily take seconds, and if all solutions are generated (let’s say the object can be grasped from the front, left, right, and top), the comparison between all costs takes longer than just executing the first possible, which might not be the most optimal. Equations 7.2 and 7.3 show the amount of grasps possible when using one grasp point, and when using multiple grasps simultaneously.

$$n_{\text{single-grasp}} = n_{\text{possible-grasp-points}} \cdot n_{\text{hands}} \Rightarrow \mathcal{O}(n) \quad (7.2)$$

$$n_{\text{multiple-grasps}} = \frac{n_{\text{possible-grasp-points}}!}{(n_{\text{possible-grasp-points}} - n_{\text{hands-to-use}})!} \Rightarrow \mathcal{O}(n!) \quad (7.3)$$

Grasping an object with four different grasp points with one hand results in four different

solutions; using two hands already produces 12 solutions.

- (o) **Reality and Simulation have their own failures:** In both scenarios, the real world and simulation, a robot can encounter failures that do not exist in the other scenario. In simulation, physics can act differently than in reality, and for me in some cases caused the PR2 robot to stick to its environment upon collision. This is a difficult to detect situation, and basically destroys the task execution. I added a specialized detector to my simulator that restarted the scenario when this situation came up. On the other hand, in the real world a wrong belief model fundamentally destroyed a robot’s understanding of the current situation. Wrong (due to drift, etc.) sensor measurements caused a grasp detector to believe a grasp went OK for thin objects, and undetected in-hand-slippage of objects repositioned them without updating their collision model — upon putting down the object onto the table, this destroyed the whole scene. These difficult border cases of failures a robot is not able to detect itself required special handling to make sure the overall task actually worked.

7.4 Automated Experiments

As mentioned above, I use the AutoExperimenter as introduced in Section 5.8.2 to run my experiments and collect EM data. For my scenario, I developed an AE experiment description², complete with task variance parameters:

```

attendants:
  label: 'Meal attendants'
  value-type: [multiple-choice]
  items: [['Mary', mary], ['Tim', tim]]
  allow-empty: false
  default: [tim]

dayoftheweek:
  label: 'Day of the week'
  value-type: [choice]
  items: [['Monday', monday], ['Tuesday', tuesday],
          ['Wednesday', wednesday], ['Thursday', thursday],
          ['Friday', friday], ['Saturday', saturday], ['Sunday', sunday]]
  default: [monday]

mealtime:
  label: 'Meal time'
  value-type: [choice]
  items: [['Breakfast', breakfast], ['Lunch', lunch], ['Dinner', dinner]]
  default: [breakfast]

```

For every instance of the n experiments run, the AE thus produces a new experiment parametrization, ultimately resulting in a different set of manipulation tasks for the controlled robot — see Section 7.1. Running multiple — sometimes time consuming and feature rich — instances (1) shows the robustness of the approach, and (2) helps collecting data for analysis and machine learning.

Using my setup, I collected around 800 episodes that reflect the robot’s activities during table setting in different situations from varying states of the environment’s development. Their use

²<http://bit.ly/2iuvj3m>

and validity were evaluated previously in [103, 105]. When run continuously, with the current scope of the simulated environment, up to 100 EMs can be generated in one day.

7.4.1 Simulated vs. Real Experiments

Obviously, in real experiments, the AE cannot (and should not, due to security concerns) be used on an actual, physical robot. I conducted a number of experiments manually, as part of the evaluations of said prior publications, and to test whether the respective current version of the scenario works in both, simulation and reality. The plans use an abstraction layer towards the low level components: Process Modules (PMs). Every plan needs to be executed in the context of a set of process modules that are "*active*", supplying the necessary low level functionality for perception, manipulation, and navigation. When replaced by different ones with the same interfaces, PMs can control for example a real robot rather than a simulated one. These topics are covered in detail in [101].

I use two macros to distinguish between the two cases:

```
(with-process-modules ;; Interface to real robot
  (tablesetting-scenario [...]))
```

```
(with-process-modules-simulated ;; Interface to simulated robot
  (tablesetting-scenario [...]))
```

In both cases, the executed plan stays absolutely identical (with some minor additional checkups and context additions for object spawning and memory cleanup for simulation).

Given that the simulated PR2 robot I use in the Gazebo simulator is modeled to closely resemble the real PR2 robot I use, the Episodic Memories resulting from the experiments are identical, with the exception of the following details:

- (o) Recorded camera images show a real, not a simulated environment
- (o) Perceived object poses in Gazebo are exact, and are subject to uncertainty in reality
- (o) Collision bodies in the simulation can differ from the visual models, not so much in reality

Besides these points, all other vital details are identical: Task structure, poses, robot posture over time, etc. I therefore make the point that my simulated experiments (due to being modeled very realistically and based on a very elaborate physics engine) reflect effects observed in reality as well — at least very closely.

Conclusion

I conclude my work with a summary of the presented topics and a discussion of how my generalized plans in conjunction with robot experiences help solving the challenges stated in the *Introduction*. Afterwards, I elaborate on possible future research.

8.1 The Need for Abstract Activity Descriptions

With a growing need to control very different robot platforms and architectures, formulating very generic, abstract, and strongly underspecified activity descriptions is a logical next step after identifying the required actions for an activity. This leads to two necessities for making these activity descriptions perform their task well: (1) A suitable set of plan language constructs with the required expressiveness, and (2) a way to implicitly fill in the left-out parametrization once it is needed. Both pose significant challenges:

(1) must be the result of encapsulation and modularization of robot control elements, and requires step-by-step formalization and subsequent analysis of tasks a robot should be able to perform. Domain and task related knowledge must be removed (or made vague) from these steps, and the resulting actions are synthesized into the new abstract activity description.

(2) draws its requirements from the results of (1): The knowledge removed from every step needs to be accessible to the performing robot once needed. It must either be encoded in a generic knowledge base, or be inferrable from already available information. Thus, a knowledge formalization and fitting access functions from the plan language must be developed, and finally the knowledge must be encoded in a way as generic as possible to prevent overfitting.

8.1.1 Human Household Scenarios

Robots that perform tasks in human household scenarios — and, as in the case of this thesis, a kitchen — face a variety of tasks very easy for humans: Searching for objects in drawers, in the fridge, on tables, and implicitly knowing how to access these locations, and what the odds are that an object resides there. Also, objects — cups, plates, muesli boxes, cutlery, etc. — are handled much differently when grasped. Finally, when setting a table for a meal, humans know from experience and cultural background where on the table each piece should be.

Besides the domain-level intrinsics from above, a robot needs to have at least very basic (at least for humans) manipulation skills: Avoiding collisions when driving around a kitchen and when reaching for objects, only placing objects where there is free space, and properly position itself to be able to both see and reach objects.

Finally, such a robot needs to be able to deal with failures. Objects being unavailable or unreachable, the robot being unable to drive to a goal pose, cluttered objects that are not graspable — these constitute very common situations for humans, but need to be explicitly addressed by an autonomous robot.

8.2 Summary of the Approach

I propose the concept of vague generalized plans drawing from experience to approach the challenges described above. In my generalized plans, all explicit parametrizations are vague and underspecified, leaving leeway in decision-making and interpretation, and allowing a robot to improve itself over time.

For generalizing plans, I propose a number of concepts and language constructs introduced in this thesis, most notably being the `with-context` environment, plan extensions to record

comprehensive Episodic Memories, and the Expectation Model functions. These constructs allow the implicit parametrization of abstract plans without explicitly forwarding parameters into them. They are used to create robot experience models, and allow success prediction and informed exploration of the parameter space for any task involved. Additionally, the recorded EMs enable a level of plan introspection for a robot that code level analysis does not allow: Direct access to the *semantic meaning and relevant hierarchical structure* of plans rather than their syntactical meat and bones.

Generalized plans draw every parameter that is vague or unknown from either the current context, extrapolated experience of a similar, past situation, or an external knowledge base. If none of these yield usable information, a heuristic fallback is used (in the worst case, choosing a parameter at random). After each plan execution, a new experience model of which parameters in what context lead to which outcome is stored and becomes available during later runs. Any parameter that is either a *designator*, an explicit plan parameter, part of the context, or manually annotated inside a generalized plan is included into the model, available to arbitrary reasoning methods, and automatically used for action effect prediction.

Recording EMs without slowing down the actual plan performance requires an efficient, concurrently running system that captures all plan events and annotated data, forms a hierarchical tree, and adds auxiliary information: At what time did a plan start, end, what were its parent and children, did it throw, catch, or rethrow failures? To fulfill these requirements, I introduced SEMREC, a universal recorder architecture for plan events that produces OWL based Episodic Memory data.

To validate the plans' and experience models' functionality with an actual robot platform, I developed interfaces to all necessary low level robot components for embodiment and plan execution: Perception, manipulation, and navigation. This came along with the challenge of translating between the mostly symbolic, vague parametrization in generalized plans, and the almost exclusively subsymbolic (numeric, nominal) nature of the low level components. I have shown solutions for all embodiment components and tested my work on both, a simulated and a real PR2 robot.

Finally, to relieve human operators from the burden of manually performing experiments and the need to regularly check the consistency of the simulated experimentation system, I presented a tool for automating the whole chain: Setting up an experiment, educatedly parametrizing it, monitoring its conduct, recording an EM, cleanly shutting down the experiment, and archiving the EM. This opens up the possibility to generate vast amounts of distinctively different data as a valuable source for machine learning applications.

8.2.1 Discussion

I have evaluated my approach in a simulated as well as a real kitchen environment using a Willow Garage PR2 robot. The robot was supposed to set a meal table under different circumstances (number and identity of participants, time of the day, day of the week). For the majority of experiments I used the simulated environment due to the sheer complexity of real world experimentation. I could show the feasibility of my approaches and underline the necessity of self-improvement in autonomous robots.

My prior evaluation of action effect prediction has shown that an experience-backed robot can not only avoid problematic situations, but make more informed decisions when choosing parameters according to what it experienced earlier. While a manually encoded plan for an individual situation would have shown a much higher performance from the first run already, an experience-based generalized plan for the same task is able to scale beyond manual efficiency tuning and can self-adapt to the task's hidden relationships and nuances. Using my EMs, I developed techniques for extrapolating between multi-modal numerical and nominal parameter spaces from experiences, allowing a robot to *guess* a good parametrization from a continuous

value range, derived from discrete experience data points. This eases knowledge transfer for the robot between vaguely similar, but still different tasks or task parametrizations.

8.3 Future Research

I have presented a complete framework for defining generalized plans and running them on an actual robot, recording Episodic Memories, and building experience models from them. Although all components are implemented completely already, a number of improvements, prospective uses for EMs, and promising additions stand out to be pursued.

8.3.1 Significant Amounts of Episodic Memories on Real Robots

I have presented the results of my experiments, which were for the largest part conducted in a comprehensive simulation environment. Although modeled very carefully and using state of the art robot simulation technology, simulation can only ever be an abstraction of reality. While I verified at every development stage that my developed plans and interfaces work as well on real hardware, I collected most of the data used to train my experience models in simulation.

Therefore, a great enhancement would be to perform a major amount of experiments on the actual PR2 robot (or comparable). An actual formal verification that e.g. the resulting Expectation Models between both experience sources are identical is still missing. This would greatly help to identify key differences between both, but also what they have in common and what knowledge can be transferred between reality and simulation.

8.3.2 Plan Verification for Generalized Plans

Generalized plans, although vague in their parametrization, have some structure that describes an abstract task that has to obey causality. Based on statically available knowledge, the structure of the plan, and possibly known parameter ranges that will be used as inputs, a verification mechanism could be derived for verifying the plan's fundamental validity. Right now, no such verification mechanism exists for my plans, but since they are more difficult to debug than straight forward action sequences, such a mechanism would (1) relieve the development cycle of lengthy iterations, and (2) make assurances about which parameter ranges actually make sense for which plan. While (1) speeds up writing the plans, (2) prevents exploration of invalid parameter space regions during runtime, ultimately lowering the amount of experiments run to improve a robot's performance from experience.

8.3.3 Generalizing over Multiple Domains

In my work, I focussed on a well-understood household scenario, with a statically defined kitchen environment. All developed plans and generated Episodic Memories were validated against this setting.

Extending the experimentation setting to more, different domains would show new requirements for robot action plans and produce new problematic situations a robot would need to deal with. For starters, this would mean a different kitchen with different furniture and room setup. Afterwards, new rooms (living room, bedroom) pose even more tasks a robot could perform (switching off the TV, watering the flowers, making the bed) that require new knowledge and generalized plans. This would lead to (1) more generality in plans that can act e.g. in many different types of kitchens, (2) a larger number of generalized plans for structurally different tasks, and (3) an extensive collection of knowledge about objects encountered, tasks performed, and problems seen, from which *all* generalized plans would benefit.

8.3.4 Experience Transfer between Different Robots

During my development and experiments, I focussed solely on the PR2 robot. All generated EMs and configurations of low level components (e.g. kinematic model of the robot for motion planning) are aligned with that robot's characteristics.

A valuable addition to the experience system would be making sure that experiences can be transferred between robot architectures. A robot that has only one arm cannot make use of experiences from robots that grasped objects with two arms, and a robot with a larger base might not be able to navigate to all places while a smaller one can. Thus, the validity of parts of EMs needs to be checked for a robot making use of experiences for action effect prediction. At the same time, experiences need to store all these identifying intrinsics of the performing robot platform.

Enabling robots to transfer experience between each other would allow them to pool their insights, ultimately increasing the speed of performance improvement.

8.3.5 New Machine Learning Methods

For all experience models, I used one or more machine learning algorithms on collected Episodic Memory data. These include calculation of decision trees, computation of gaussian probability models, and graph generalization.

There exists a plethora of other algorithms fit for analyzing this data, generating qualitatively different models. A possible improvement of the presented learning techniques could be achieved by using different classification (e.g. K-Nearest Neighbors, Random Forests, Naive Bayes), or reinforcement learning algorithms (e.g. Q learning). Additionally, since the data stored tends to grow with more features in the environment and tasks performed, there is a strong need for dimensionality reduction, identification of significant variables, and statistical tests for parameter correlation. These would make identification of causal relations in the data as well as anomalies much easier to conduct.

Research Platforms

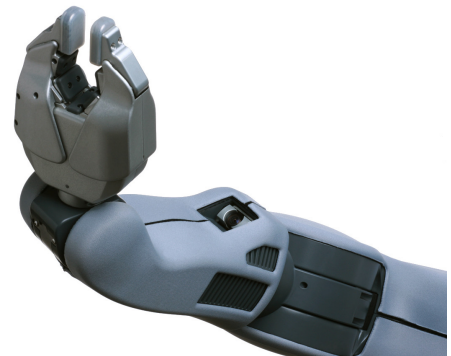
To develop and test my generalized plans and experience models, I used two platforms for embodiment of the defined behavior: A real Willow Garage PR2 in an experimentation kitchen environment, and a simulated replica of both using the Gazebo robot simulator. In this chapter, I describe both in sufficient detail for this thesis.

A.1 The Willow Garage PR2: A Capable Mobile Manipulator

For development, testing, and verification of my concepts and robot plan, I use the Willow Garage PR2 (Personal Robot 2) robot platform. The PR2 is a mobile manipulator on an omnidirectional base, featuring two compliant 7-DOF arms and a PTU head on which a Kinect depth sensor resides. Furthermore, it has a number of laser scanners helping in localization and collision avoidance, and a series of cameras which I did not use. The arms are each equipped with a jaw gripper capable of firm grasps with diameters of 0–8cm and parameterizable force. A brief depiction of the PR2 and its grippers is shown in Figure A.1.



(a) The Willow Garage PR2 Robot.



(b) Gripper of the PR2 Robot, featuring a wrist camera and fingertip pressure sensors.

Figure A.1: The Willow Garage PR2 Robot and one of its grippers. Photo by Luke Fisher Photography.

I chose the PR2 platform for my work as it unites many typical architectural details found in today's robots, and resembles humans in terms of manipulation ranges and general figure. It is well supported and component drivers are widely available and well integrated. Besides this, a precise physics simulation model with an identical driver interface is available for the PR2 (see Section A.2), making initial testing possible without having to deal with all of reality's quirks in early development stages, and allowing automated experiment conduct later on. For my research, I used an actual PR2 as well as a simulated one for different scenarios and continuous integration.

Since I am not concerned with low-level perception systems, I use ROBOSHERLOCK [7] for

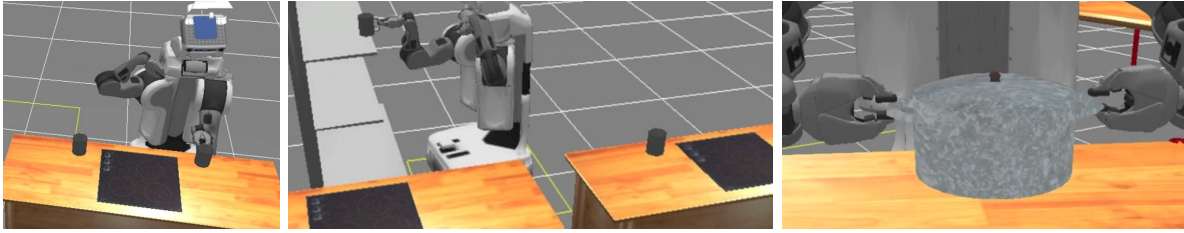


Figure A.2: Different manipulation tasks as performed by a PR2 in the Gazebo simulator. From left to right: Picking objects with one hand from a table; placing objects in a shelf; picking objects with two hands

estimating poses and visual characteristics of objects in the environment when using the real PR2. When using the simulated PR2, I use a simplified perception system I developed that directly interprets ground truth data from Gazebo, but offers the exact same interface as on the real robot.

Throughout my research, the PR2 has proven to be well suited for mobile manipulation tasks. Grasping objects with one or two hands, opening and closing drawers, cabinet and fridge doors, and performing a number of manipulation activities beyond fetch and place are well within its physical capabilities. In my research, I focussed primarily on the fetch and place part: Fetching objects from arbitrary places (surfaces, storage places, containers) and placing them at similar locations again. In teleoperated demonstrations it was shown that both, the PR1 (the PR2’s predecessor, [85]) and the PR2 [99] robot are capable of rather agile manipulation, solidifying my claim that one of their largest constraint is their behavior generating plans.

A.2 Gazebo: Simulation-based Manipulation

For development, testing, and automated data collection, I use the realistic robot simulator Gazebo [47]. Gazebo offers a very concise physics simulation using the ODE physics engine, based on multi-body collisions and physical properties such as friction or damping.

As mentioned above, the existence of a precise PR2 simulation model, added the fact that a version of the research kitchen I perform real experiments in exists in Gazebo as well, made the simulator a prime choice. Figure A.2 shows some examples of object picking and placing tasks in a simplified environment. Here, I show the robot handling multiple objects in the same scene, placing an object onto a shelf, and handling objects with multiple hands.

Gazebo is well intergated with ROS, offering interfaces for spawning, manipulating, and removing objects. For my purposes, I had to extend Gazebo’s capabilities by writing a plugin¹ for controlling and inspecting the research kitchen’s container joints (drawers, fridge door, etc.). I also use this plugin for fixating grasped objects to the simulated robot gripper — otherwise, grasping an object results in unexpected behavior in both the robot and the grasped object due to numerical fluctuations. Gazebo plays a central role in performing automated experiments, as explained in Section 5.8.

¹https://github.com/fairlight1337/gazebo_attache_plugin

Plans

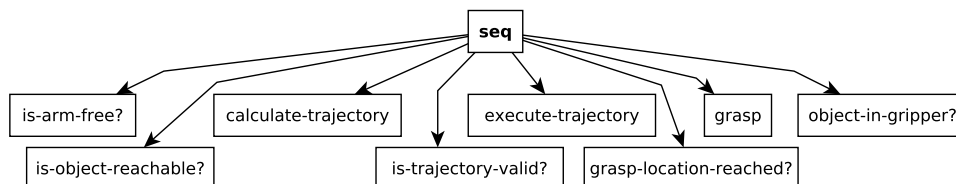


Figure B.1: Behavior Tree version of a simple CRAM grasping plan. The source code variant is shown in Algorithm 6.9.

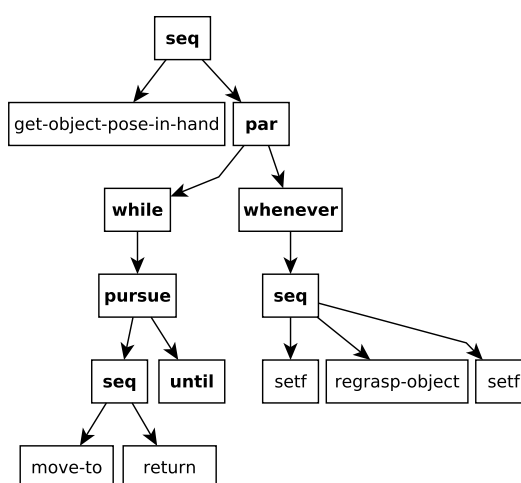


Figure B.2: Behavior Tree version of a monitoring plan. The source code variant is shown in Algorithm 6.10.

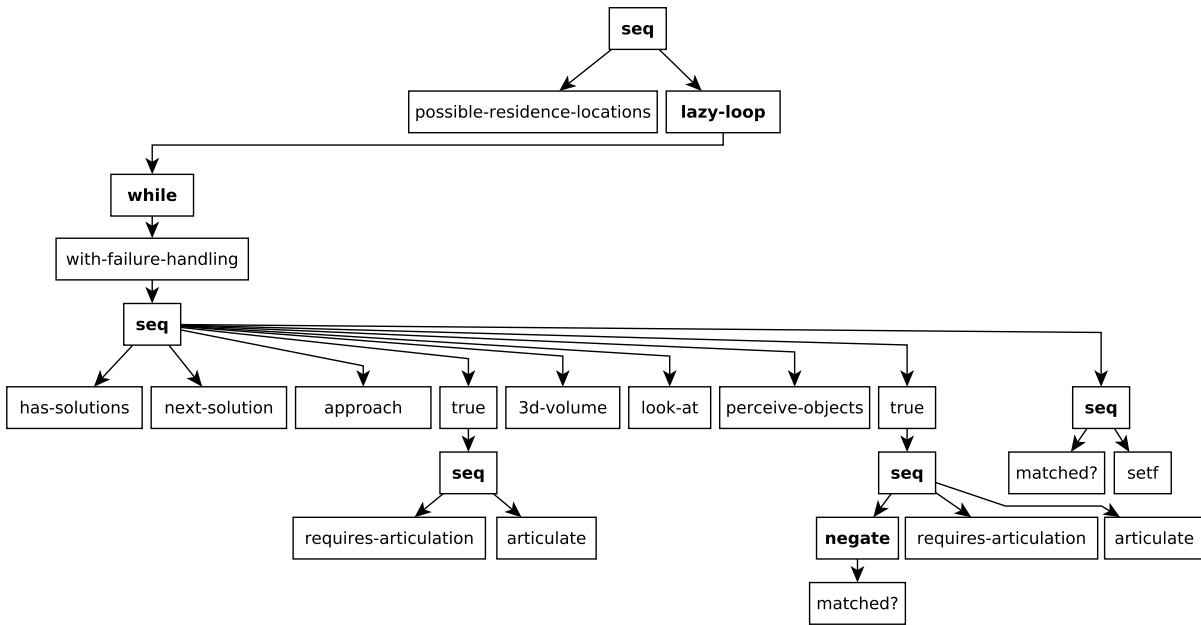


Figure B.3: Behavior Tree version of a search object plan. The source code variant is shown in Section 4.3.5.

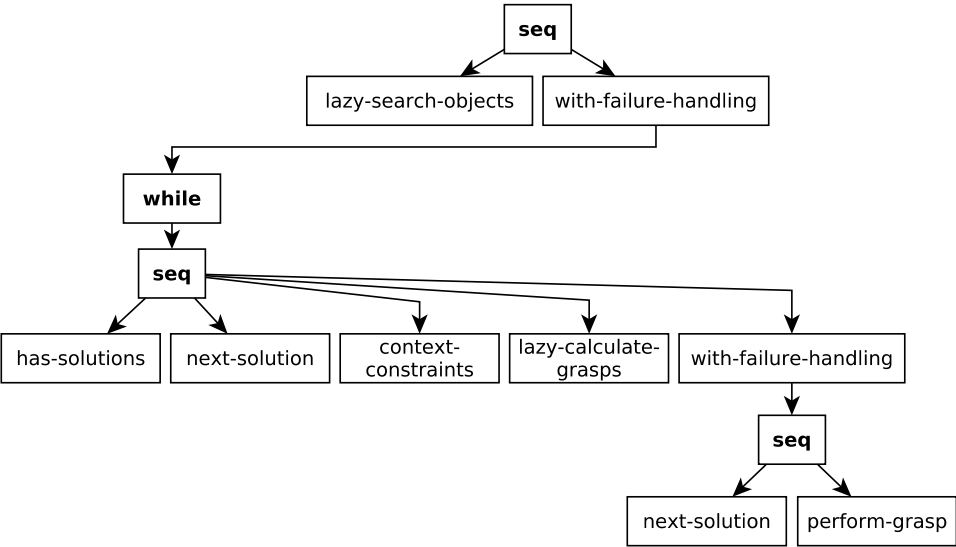


Figure B.4: Behavior Tree version of a fetch object plan. The source code variant is shown in Section 4.3.6.

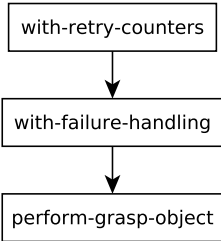


Figure B.5: Behavior Tree version of a simple object grasping plan with naive failure handling capabilities. The source code variant is shown in Section 6.1.

Keyword Index

- Action Effect Prediction, *64*
- add-collision-object, *96*
- AMCL, *see* Navigation
- attach-collision-object-to-link, *97*
- AutoExperimenter, *77*
- Automated Experiments, *76*
- Behaviour Trees, *33*
- choose, *63*
- clear-collision-environment, *97*
- Closed kinematic chain, *93*
- Constrained Manipulation, *38*
- Costmaps, *98*
- cram_uima, *91*
- detach-collision-object-from-link, *97*
- do-retry, *87*
- Dual-Arm Motion, *92*
- Embodiment, *83*
- Episodic Memory, *20, 25*
- Expectation Model, *20, 57*
- Experience Models, *57*
- Failures
 - Anticipated, *83*
 - Categories, *85*
 - Concurrent Monitoring, *85*
 - Recovery Strategies, *48, 83*
 - Unanticipated, *83*
- Gazebo, *114*
- K-Means Clustering, *71*
- Knowledge
 - Contextual, *42*
 - Dynamic, *45*
 - Static, *44*
- KnowRob, *93*
- Localization, *see* Navigation
- Mixed Multivariate Gaussians, *72*
- mongolog_db, *56*
- MoveIt, *91*
- Multi-modal Analysis, *70*
- Multivariate Gaussians, *71*
- Navigation, *97*
- OMPL, *91*
- Planning
 - A*, *45*
 - PDDL, *73*
 - STRIPS, *73*
- Planning Scene, *96*
- PR2, *113*
- predict-behavior, *62*
- Problem Generator, *100*
- Prototypical Experience, *65*
- Reactive Task Monitoring, *39*
- register-collision-object, *96*
- remove-collision-object, *96*
- RoboSherlock, *91*
- robosherlock_process_module, *91*
- Semantic Hierarchy Recorder, *55*
- SemRec, *see* Semantic Hierarchy Recorder
- TF, *see* Navigation
- Uncertainty, *88*
- Virtual Branch, *64*
- with-context, *31, 43*
- with-expectation-model, *62*
- with-retry-counters, *87*
- with-tag, *62*
- without-collision-object, *97*

References

- [1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 313–321. IEEE, 1996.
- [2] Omar Ahmad, James Cremer, Joseph Kearney, Peter Willemsen, and Stuart Hansen. Hierarchical, concurrent state machines for behavior modeling and scenario control. In *AI, Simulation, and Planning in High Autonomy Systems, 1994. Distributed Interactive Simulation Environments., Proceedings of the Fifth Annual Conference on*, pages 36–42. IEEE, 1994.
- [3] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4), 1998.
- [4] R. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.
- [5] Michael Beetz. Structured reactive controllers. *Autonomous Agents and Multi-Agent Systems*, 4(1):25–55, 2001.
- [6] Michael Beetz. Plan representation for robotic agents. In *AIPS*, pages 223–232, 2002.
- [7] Michael Beetz, Ferenc Bálint-Benczédi, Nico Blodow, Daniel Nyga, Thiemo Wiedemeyer, and Zoltán-Csaba Márton. Robosherlock: unstructured information processing for robot perception. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1549–1556. IEEE, 2015.
- [8] Michael Beetz, Dominik Jain, Lorenz Mösenlechner, Lars Kunze, Moritz Tenorth, Nico Blodow, and Dejan Pangercic. Cognition-Enabled Autonomous Robot Control for the Realization of Home Chore Task Intelligence. *Proceedings of the IEEE, Special Issue on Quality of Life Technology*, 2012. To appear.
- [9] Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth. Robotic Roommates Making Pancakes. In *11th IEEE-RAS International Conference on Humanoid Robots*, Bled, Slovenia, October, 26–28 2011.
- [10] Michael Beetz and Drew McDermott. Declarative goals in reactive plans. In *First International Conference on AI Planning Systems*, pages 3–12, 1992.
- [11] Michael Beetz and Drew McDermott. Executing structured reactive plans. In *Proc. AAAI Fall Symposium: Issues in Plan Execution, AAAI Technical Report FS-96-01*, 1996.
- [12] Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. CRAM – a cognitive robot abstract machine for everyday manipulation in human environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1012–1017, Taipei, Taiwan, October 18-22 2010.

-
- [13] Michael Beetz, Moritz Tenorth, and Jan Winkler. Open-EASE – a knowledge processing service for robots and robotics/ai researchers. In *IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, Washington, USA, 2015. Finalist for the Best Cognitive Robotics Paper Award.
- [14] Ralf Berger and Hans-Dieter Burkhard. Modeling complex behavior of autonomous agents in dynamic environments. *Proc. Concurrency, Specification and Programming (CS&P)*, 2006.
- [15] Jonathan Bohren and Steve Cousins. The SMACH High-Level Executive. *IEEE Robotics and Automation Magazine*, 17:18–20, 2010.
- [16] Mario Bollini, Stefanie Tellex, Tyler Thompson, Nicholas Roy, and Daniela Rus. Interpreting and executing recipes with a cooking robot. In *Experimental Robotics*, pages 481–495. Springer, 2013.
- [17] Peter Bonasso, James Firby, Erann Gat, David Kortenkamp, David Miller, and Marc Slack. Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1), 1997.
- [18] Rodney Brooks. New approaches to robotics. *Science*, 253(5025):1227–1232, 1991.
- [19] Randy C Brost. Planning robot grasping motions in the presence of uncertainty. 1985.
- [20] Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton, and Manuela Veloso. Prodigy: An integrated architecture for planning and learning. *ACM SIGART Bulletin*, 2(4):51–55, 1991.
- [21] Michele Colledanchise and Petter Ogren. How behavior trees modularize robustness and safety in hybrid systems. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1482–1488. IEEE, 2014.
- [22] Mark R Cutkosky. On grasp choice, grasp models, and the design of hands for manufacturing tasks. *Robotics and Automation, IEEE Transactions on*, 5(3):269–279, 1989.
- [23] Keith S Decker and Victor R Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(02):319–346, 1992.
- [24] Denis Diderot. On the interpretation of nature, no. 15, 1753.
- [25] Edmund H Durfee, Charles L Ortiz Jr, Michael J Wolverton, et al. A survey of research in distributed, continual planning. *Ai magazine*, 20(4):13, 1999.
- [26] Mahmoud El Shaikh, Andreas Koch, Bernd Eckstein, Kai Häussermann, Oliver Zweigle, and Paul Levi. Advanced perception for robots in a closed world environment. In *Intelligent Autonomous Systems 12*, pages 111–122. Springer, 2013.
- [27] Sean P Engelson and Drew V McDermott. Error correction in mobile robot map learning. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 2555–2560. IEEE, 1992.
- [28] Kutluhan Erol, James Hendler, and Dana S Nau. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.
- [29] Oren Etzioni, Keith Golden, and Daniel S Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89(1):113–148, 1997.

- [30] David Ferrucci and Adam Lally. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348, 2004.
- [31] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.
- [32] R James Firby. Task networks for controlling continuous processes. In *AIPS*, pages 49–54, 1994.
- [33] Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pages 1–6. IEEE, 2013.
- [34] Jodi Forlizzi and Carl DiSalvo. Service robots in the domestic environment: a study of the roomba vacuum in the home. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, pages 258–265. ACM, 2006.
- [35] Dieter Fox and Cody Kwok. A sample-based approach to landmark-based robot localization.
- [36] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In *ICAPS*, volume 6, pages 212–221, 2006.
- [37] Erann Gat. Esl: A language for supporting robust plan execution in embedded autonomous agents. In *In Working notes of the AAAI Fall Symposium on Plan Execution*. AAAI, 1996.
- [38] Erann Gat. On Three-Layer Architectures. In P. Bonasso, D. Kortenkamp, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots*, pages 195–210. MIT Press, Cambridge, MA, 1998.
- [39] Malcolm Gladwell. *Blink: The power of thinking without thinking*. Back Bay Books, January 2005.
- [40] Stuart Hart, Paul Dinh, John D Yamokoski, Brian Wightman, and Nicolaus Radford. Robot task commander: A framework and ide for robot application development. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1547–1554. IEEE, 2014.
- [41] Stephen Hawking. British theoretical physicist, cosmologist, and author (1941–).
- [42] Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka: A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361. IEEE, 1994.
- [43] M. Ingham, R. Ragno, and B. C. Williams. A reactive model-based programming language for robotic space explorers. In *International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*, Montreal, Canada, 2001.
- [44] Francois Fe’lix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 43–49. IEEE, 1996.
- [45] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *IEEE International Conference on Robotics and Automation*, pages 43–49, Minneapolis, 1996.

-
- [46] Charles C Kemp, Aaron Edsinger, and Eduardo Torres-Jara. Challenges for robot manipulation in human environments. *IEEE Robotics and Automation Magazine*, 14(1):20, 2007.
- [47] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154, 2004.
- [48] Danica Kragic and Markus Vincze. Vision for robotics. *Found. Trends Robot*, 1(1):1–78, 2009.
- [49] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, September 1987.
- [50] George Lakoff. *Philosophy in the Flesh: The Embodied Mind and its Challenge to Western Thought*. Basic Books, New York, 1999.
- [51] Karthik Lakshmanan, Apoorva Sachdev, Ziang Xie, Dmitry Berenson, Ken Goldberg, and Pieter Abbeel. A constraint-aware motion planning algorithm for robotic folding of clothes. In *Proceedings of the 13th International Symposium on Experimental Robotics (ISER)*, 2012.
- [52] David M Lane, Francesco Maurelli, Petar Kormushev, Marc Carreras, Maria Fox, and Konstantinos Kyriakopoulos. Persistent autonomy: the challenges of the pandora project. *Proceedings of IFAC MCMC*, 2012.
- [53] David M Lane, Francesco Maurelli, Tom Larkworthy, Darwin Caldwell, Joaquim Salvi, Maria Fox, and Konstantinos Kyriakopoulos. Pandora: Persistent autonomy through learning, adaptation, observation and re-planning. *IFAC Proceedings Volumes*, 45(5):367–372, 2012.
- [54] Pat Langley, Kathleen B McKusick, John A Allen, Wayne F Iba, and Kevin Thompson. A design for the icarus architecture. *ACM SIGART Bulletin*, 2(4):104–109, 1991.
- [55] Daniel Leidner, Alexander Dietrich, Florian Schmidt, Christoph Borst, and Alin Albu-Schäffer. Object-centered hybrid reasoning for whole-body mobile manipulation. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1828–1835. IEEE, 2014.
- [56] James NK Liu, Meng Wang, and Bo Feng. ibotguard: an internet-based intelligent robot security system using invariant face recognition against intruder. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(1):97–105, 2005.
- [57] Drew McDermott. Robot planning. *AI Magazine*, 13(2):55–79, 1992.
- [58] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – the planning domain definition language. 1998.
- [59] Robin R. Murphy. *Introduction to AI Robotics*. MIT Press, Cambridge, MA, USA, 1st edition, 2000.
- [60] Isaac Newton. Letter from Sir Isaac Newton to Robert Hooke. Historical Society of Pennsylvania. 1675.

- [61] Tim Niemueller, Gerhard Lakemeyer, and Siddhartha S. Srinivasa. A Generic Robot Database and its Application in Fault Analysis and Performance Evaluation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems 2012*, Vilamoura, Algarve, Portugal, 2012. IEEE/RAS.
- [62] Tim Niemüller, Alexander Ferrein, and Gerhard Lakemeyer. A lua-based behavior engine for controlling the humanoid robot nao. In *RoboCup 2009: Robot Soccer World Cup XIII*, pages 240–251. Springer, 2010.
- [63] Andrew Nuxoll and John E Laird. A cognitive model of episodic memory integrated with a general cognitive architecture. In *ICCM*, pages 220–225. Citeseer, 2004.
- [64] Andrew M Nuxoll and John E Laird. Extending cognitive architecture with episodic memory. *Ann Arbor*, 1001:48109–2121, 2007.
- [65] Petter Ogren. Increasing modularity of uav control systems using computer game behavior trees. In *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.
- [66] Kei Okada, Mitsuharu Kojima, Yuichi Sagawa, Toshiyuki Ichino, Kenji Sato, and Masayuki Inaba. Vision based behavior verification system of humanoid robot for daily environment tasks. In *Proceedings of the 6th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 7–12, 2006.
- [67] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. 1984.
- [68] Edwin PD Pednault. Formulating multiagent, dynamic-world problems in the classical planning framework. *Reasoning about actions and plans*, pages 47–82, 1987.
- [69] Samuel Thomas Pfister. *Algorithms for mobile robot localization and mapping, incorporating detailed noise modeling and multi-scale feature extraction*. PhD thesis, California Institute of Technology, 2006.
- [70] Joelle Pineau and Sebastian Thrun. High-level robot behavior control using pomdps. In *AAAI-02 Workshop on Cognitive Robotics*, volume 107, 2002.
- [71] Scott Plous. *The psychology of judgment and decision making*. Mcgraw-Hill Book Company, 1993.
- [72] C. Potena, D. Nardi, and A. Pretto. Fast and accurate crop and weed identification with summarized train sets for precision agriculture. In *Proc. of 14th International Conference on Intelligent Autonomous Systems (IAS-14)*, 2016.
- [73] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Mateo, California, 1993.
- [74] Helen M Regan, Mark Colyvan, and Mark A Burgman. A taxonomy and treatment of uncertainty for ecology and conservation biology. *Ecological applications*, 12(2):618–628, 2002.
- [75] Max Risler. *Behavior control for single and multiple autonomous agents based on hierarchical finite state machines*. PhD thesis, tprints, 2010.
- [76] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

-
- [77] Thomas Rühr, Jürgen Sturm, Dejan Pangercic, Michael Beetz, and Daniel Cremers. A generalized framework for opening doors and drawers in kitchen environments. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3852–3858. IEEE, 2012.
- [78] Haydar Sahin and Levent Guvenc. Household robotics: autonomous devices for vacuuming and lawn mowing [applications of control]. *IEEE Control Systems*, 27(2):20–96, 2007.
- [79] Martin Schuster, Dominik Jain, Moritz Tenorth, and Michael Beetz. Learning Organizational Principles in Human Environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3867–3874, St. Paul, MN, USA, May 14–18 2012.
- [80] Claude E Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.
- [81] Yasuhiro Shiraki, Kazuyuki Nagata, Natsuki Yamanobe, Akira Nakamura, Kanako Harada, Daisuke Sato, and Dragomir N Nenchev. Modeling of everyday objects for semantic grasp. In *Robot and Human Interactive Communication, 2014 RO-MAN: The 23rd IEEE International Symposium on*, pages 750–755. IEEE, 2014.
- [82] Reid Simmons. An architecture for coordinating planning, sensing, and action. In *Procs. DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 292–297, 1990.
- [83] Reid G Simmons. Structured control for autonomous robots. *IEEE transactions on robotics and automation*, 10(1):34–43, 1994.
- [84] Siddhartha Srinivasa, David Ferguson, Casey Helfrich, Dmitry Berenson, Alvaro Collet Romea, Rosen Diankov, Garratt Gallagher, Geoffrey Hollinger, James Kuffner, and J Michael Vandeweghe. HERB: A Home Exploring Robotic Butler. *Autonomous Robots*, 28(1):5–20, 2010.
- [85] Stanford University. Stanford personal robotics program. <http://personalrobotics.stanford.edu/>, 2008. Accessed August 16, 2012.
- [86] Andreas Steck and Christian Schlegel. Smarttcl: An execution language for conditional reactive task execution in a three layer architecture for service robots. In *Int. Workshop on Dynamic languages for RObotic and Sensors systems (DYROS/SIMPAR)*, pages 274–277, 2010.
- [87] Sarah Strohkorb, Chien-Ming Huang, Aditi Ramachandran, and Brian Scassellati. Establishing sustained, supportive human-robot relationships: Building blocks and open challenges. In *2016 AAAI Spring Symposium Series*, 2016.
- [88] Jörg Stückler and Sven Behnke. Benchmarking mobile manipulation in everyday environments. In *Proc. of the IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO)*, 2012.
- [89] Jörg Stückler, Dirk Holz, and Sven Behnke. Robocup@home: Demonstrating everyday manipulation skills in robocup@home. *IEEE Robotics and Automation Magazine*, 19(2):34–42, June 2012.
- [90] Freek Stulp, Andreas Fedrizzi, Lorenz Mösenlechner, and Michael Beetz. Learning and Reasoning with Action-Related Places for Robust Mobile Manipulation. *Journal of Artificial Intelligence Research (JAIR)*, 43:1–42, 2012.

- [91] Tapio Taipalus and Kazuhiro Kosuge. Development of service robot for fetching objects in home environment. In *Computational Intelligence in Robotics and Automation, 2005. CIRA 2005. Proceedings. 2005 IEEE International Symposium on*, pages 451–456. IEEE, 2005.
- [92] Kartik Talamadupula, J Benton, Paul W Schermerhorn, Subbarao Kambhampati, and Matthias Scheutz. Integrating a closed world planner with an open world robot: A case study. In *AAAI*, 2010.
- [93] Moritz Tenorth and Michael Beetz. Knowrob – a knowledge processing infrastructure for cognition-enabled robots. *International Journal of Robotics Research (IJRR)*, 32(5):566 – 590, April 2013.
- [94] Moritz Tenorth, Lars Kunze, Dominik Jain, and Michael Beetz. Knowrob-map – knowledge-linked semantic object maps. In *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*, pages 430–435. IEEE, 2010.
- [95] Laura Uusitalo, Annukka Lehtikoinen, Inari Helle, and Kai Myrberg. An overview of methods to evaluate uncertainty of deterministic models in decision support. *Environmental Modelling & Software*, 63:24–31, 2015.
- [96] David Vernon, Giorgio Metta, and Giulio Sandini. 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, 2007.
- [97] Rob Walker. The guts of a new machine. <http://www.nytimes.com/2003/11/30/magazine/the-guts-of-a-new-machine.html>, 2003. *The New York Times Magazine*. Accessed: 2015-11-26.
- [98] Oscar Wilde. Irish playwright, novelist, essayist, and poet (1854–1900).
- [99] Willow Garage, OSRF. Pr2 surrogate - immersive telepresence with the oculus rift. <https://www.youtube.com/watch?v=H0EoEywTmiY>, 2013. Accessed January 20, 2017.
- [100] Jan Winkler, Ferenc Balint-Benczedi, Thiemo Wiedemeyer, Michael Beetz, Narunas Vaskevicius, Christian A. Mueller, Tobias Fromm, and Andreas Birk. Knowledge-enabled robotic agents for shelf replenishment in cluttered retail environments: (extended abstract). In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, AAMAS '16*, pages 1421–1422, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.
- [101] Jan Winkler, Georg Bartels, Lorenz Mösenlechner, and Michael Beetz. Knowledge Enabled High-Level Task Abstraction and Execution. *First Annual Conference on Advances in Cognitive Systems*, 2(1):131–148, December 2012.
- [102] Jan Winkler and Michael Beetz. Generalized plan design for autonomous mobile manipulation in open environments. In *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, Istanbul, Turkey, 2015.
- [103] Jan Winkler and Michael Beetz. Robot action plans that form and maintain expectations. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Hamburg, Germany, 2015.
- [104] Jan Winkler, Asil Kaan Bozcuoğlu, Mihai Pomarlan, and Michael Beetz. Task parametrization through multi-modal analysis of robot experiences. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 1754–1756. International Foundation for Autonomous Agents and Multiagent Systems, 2017.

- [105] Jan Winkler, Moritz Tenorth, Asil Kaan Bozcuoglu, and Michael Beetz. CRAMm – memories for robots performing everyday manipulation activities. *Advances in Cognitive Systems*, 3:47–66, 2014.
- [106] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*, page 69. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.