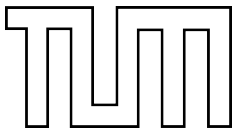
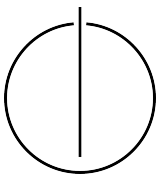


Technische Universität München
Lehrstuhl für Bildverstehen und
wissensbasierte Systeme



Knowledge Processing for Autonomous Robots

Moritz M. Tenorth

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende: Univ.-Prof. Gudrun J. Klinker, Ph.D.

Prüfer der Dissertation: 1. Univ.-Prof. Michael Beetz, Ph.D.

2. Prof. Dr. Masayuki Inaba, University of Tokyo, Japan

Die Dissertation wurde am 27.06.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08.11.2011 angenommen.

Contents

Abstract

Autonomous robots are becoming more and more skilled in performing human-scale manipulation tasks, and will soon become common co-workers in our homes. Compared to well-structured and deterministic factory environments, a human household is much more demanding regarding the knowledge a robot needs to have: The robot is to perform an open-ended set of tasks in this unstructured and dynamic environment, thereby interacting with a variety of objects of different kinds that all need to be handled in special ways. To understand instructions given by humans, it also needs a large amount of common-sense knowledge to translate the words into the correct actions and parametrizations.

In this work, we describe a framework for representing the knowledge that an autonomous robot needs for its tasks that also supports reasoning about this knowledge. We present novel representations for actions, objects, and the environment which enable a robot to reason about the effects of actions and processes and to describe changing world states, including the creation, destruction and transformation of objects which are common effects in human everyday tasks. The representations are grounded in the robot's internal data structures and integrated with the robot's control program. They allow to write more general and flexible control programs and help robots to execute incomplete and underspecified instructions.

We further present techniques for acquiring knowledge from Internet sources, namely task instructions and object models, and for translating them from natural language into formal representations in the knowledge base. In addition, we investigate how robots can make use of observations of human activities. We developed a system that integrates methods for observing human everyday activities, for splitting the continuous motions into meaningful segments, and for generating abstract task-level descriptions of actions and their properties from these observations. The models are represented in the same format that is also used to describe the robot' plans and are therefore directly usable by the robot in task execution.

Experiments on different robot platforms in different environments demonstrate how these techniques can help to improve the problem-solving performance of autonomous robots.

Zusammenfassung

Autonome Roboter sind zunehmend in der Lage, alltägliche Manipulationsaufgaben zu erledigen und werden bald den Weg in unsere Haushalte finden. Im Hinblick auf das Wissen, das ein Roboter benötigt, um seine Aufgaben zu erledigen, ist der menschliche Haushalt allerdings erheblich herausfordernder als stark strukturierte und recht vorhersagbare Fabrikumgebungen. Man erwartet von ihm, dass er ein großes Aufgabenspektrum kompetent beherrscht, flexibel auf Veränderungen reagiert, sich eigenständig neue Fähigkeiten aneignen und mit einer Vielzahl von Objekten sicher umgehen kann.

In dieser Arbeit beschreiben wir Methoden um das Wissen, das ein Roboter für seine Aufgaben benötigt, in einer Weise darzustellen, die es ermöglicht daraus automatisiert Schlussfolgerungen zu ziehen und neue Aussagen abzuleiten. Insbesondere stellen wir neue Repräsentationen für Aktionen, Objekte und die Umgebung des Roboters vor, die es erlauben, die Effekte von Aktionen vorherzusagen, sich verändernde Weltzustände zu beschreiben und die Entstehung, Zerstörung, und Veränderung von Objekten, etwa beim Kochen von Mahlzeiten, darzustellen. Diese Wissens-Repräsentationen sind in den Datenstrukturen des Roboters grundiert und in sein Kontrollprogramm integriert. Sie unterstützen den Programmierer dabei, generischere Roboter-Kontrollprogramme zu schreiben und helfen dem Roboter, sich flexibel auf geänderte Bedingungen einzustellen.

Um neues Wissen zu akquirieren, kann der Roboter zum einen auf Beschreibungen von Alltagsaufgaben und Objekt-Eigenschaften im Internet zugreifen. Hierzu präsentieren wir neue Methoden, mittels derer die natürlich-sprachlichen Erläuterungen in für den Roboter verständliche formale Darstellungen übersetzt werden können. Zum anderen können Roboter aus Beobachtung des Menschen lernen. Diese Arbeit beschreibt verschiedene Methoden, mit denen beobachtete Bewegungen zunächst segmentiert und in der Wissensbasis abgelegt werden können, bevor durch Anwendung von Hintergrundwissen über die Aktionen abstrakte Beschreibungen generiert werden, die der Roboter bei der Planung seiner Aktionen verwenden kann.

Mit Experimenten auf verschiedenen Robotern in unterschiedlichen Umgebungen zeigen wir, wie die entwickelten Techniken dazu beitragen, die Fähigkeiten autonomer Roboter zu verbessern.

Acknowledgments

First, I would like to thank my advisor Prof. Michael Beetz for giving me the opportunity to join the inspiring environment of the Intelligent Autonomous Systems Group. His vision, guidance, motivation and widespread support helped me a lot throughout my thesis. Thanks also to Prof. Fernando De la Torre for inviting me to the exciting Robotics Institute at CMU, for introducing me to a lot of people, and for giving me advice while being there. I would also like to thank Prof. Masayuki Inaba for serving on my thesis committee.

Much of the work presented here is the result of close collaboration and would not have been possible without the work of my colleagues who contributed perception modules, robot control software and inference tools. Thanks to Lorenz, Dominik, Uli, Dejan, Jan, Lars, Tom, Daniel, Séverin, Bernhard, Freek, Nico, Zoltan, Alexis, Federico, and the other members of the Intelligent Autonomous Systems group for the good collaboration and many inspiring discussions. The same holds for all the colleagues from the RoboEarth project. Special thanks to Bernhard, who developed the predecessor of the system presented here, for introducing me into the world of Prolog and for patiently answering all my questions. I would further like to thank Michael, Lorenz, Lars and Dominik for reading initial drafts and providing helpful comments and criticism.

Over the years, I have worked with several students and have also learned a lot from supervising them. Thanks to Daniel, Lars, Jakob, Martin, Cristina, Kai, Steffen and Tobias who have helped to implement the ideas presented here.

My parents and my brother Julian have excited my curiosity from early years on, encouraged me whenever I needed it and continuously supported all my plans over all the years, for which I am very grateful!

Finally, I would like to thank Ramona for supporting and encouraging me while I was writing this thesis and for reminding me that work is not everything in life.

This work received funding from the CoTeSys (Cognition for Technical Systems) cluster of excellence as part of the project "Knowledge4CoTeSys" and from the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement number 248942 RoboEarth.

Nomenclature

In the text, the names of classes, instances, properties, predicates and other identifiers are written in *italic*. For better readability, we use the Manchester Syntax [Horridge et al., 2006] instead of the RDF/XML syntax for OWL examples and omit the OWL name spaces that are usually described by an Internationalized Resource Identifier (IRI) like *http://ias.cs.tum.edu/kb/knowrob.owl#identifier*. Instead, we only use the *identifier* part of the class or instance behind the hash sign.

The syntax of the Prolog code examples has been slightly adapted for better readability. Variables are marked by a preceding question mark like *?A*. Identifiers starting with a lowercase letter or those inside single quotes are considered to be constants. The logical AND is expressed by a comma, the logical OR by a semicolon. The \+ operator in front of a predicate means "cannot be proven" and provides negation as failure. Whenever an example of a query is given, the code starts with the *?-* prompt. The percentage sign % starts a comment. For a detailed introduction to the Prolog syntax see [Sterling and Shapiro, 1994].

Chapter 1

Introduction

We are seeing more and more robots like the PR2 [Bohren et al., 2011], the Care-o-bot [Reiser et al., 2009], TUM-Rosie [Beetz et al., 2010b], and Herb [Srinivasa et al., 2010] that are able to perform mobile pick-and-place and even more sophisticated manipulation tasks like folding towels or preparing meals. With the hardware, perception and control routines maturing, such robots will become increasingly common in human everyday environments. However, performing human-scale manipulation tasks in dynamic environments like a household or a nursing home remains very challenging since robots often do not have the knowledge to perform them the right way.

Especially informal instructions given by a human user require a lot of knowledge to be understood correctly. Ideally, users should be able to explain a task to the robot in the same way as they would explain it to a human. The problem is that such descriptions are usually lacking much information that humans consider obvious. For example, a human would never describe to switch off an oven as part of a recipe for baking a cake, or explain that a bottle needs to be opened before the water can be poured into a glass. Humans can understand these incomplete instructions because of their remarkable ability to store an enormous amount of knowledge and to quickly retrieve everything they need to know in a given situation. Since people can assume other humans to have this kind of knowledge, they do not have to explain it when describing how to perform a task or how to handle an object. While this has the advantage of reducing redundancy in communication, it requires the recipient to be able to interpret the information correctly. Humans can assign meaning to abstract phrases in the instructions like “cup”, “inside of” or “picking up” by relating them to knowledge they already have about these things, like the fact that cups are a kind of container that is used for drinking liquids and should be carried upright when filled, that things are not visible if they are inside a closed container, or that the result of picking up an object is that this object is held in the hand.

Robots are still largely lacking such capabilities. Normally, they follow pre-programmed step-by-step plans to perform their tasks without having a notion of which effects the actions have, why they need to be done, and what the objects they are interacting with actually are. The knowledge about actions and objects is hidden in the program code and only implicitly described in terms of if-then statements. This makes it hard to re-use the knowledge and to apply it in different circumstances. The robot only knows *that* it should do something, but it does not know *why*, which other options exist to achieve the same goal, or how to adapt an action to different situations. A formal knowledge representation helps the robot to make these aspects explicit and to increase re-usability and flexibility. Having access to such a knowledge base and being able to quickly retrieve the right pieces of information is required in several situations:

1. Understanding instructions given by humans and translating them into effective task specifications requires substantial knowledge about the actions involved. Completing the abstract and incomplete instructions means extending them with additional actions and object descriptions that are not part of the original descriptions. The robot needs to determine which information is missing and decide how to obtain it, which requires deep knowledge about actions, their parameters and outcomes.
2. The robot needs to adapt actions in the correct way depending on the current context. This includes changing action parameters, adding constraints on the robot's motions or the order of actions, or changing which object is to be manipulated. To decide if and how actions should be adapted, the robot has to integrate information from the perception system with suitable background knowledge. If for example a cup is to be picked up and the robot later detects that it is filled with coffee, the system should add a constraint to the action that the robot should obey a maximum acceleration and should especially keep the cup upright in order not to spill anything. Detecting *that* such modifications are needed and deciding *which* changes are needed requires a lot of knowledge.
3. The robot is to be able to interpret general control programs in which control decisions are not hardcoded, but need to be inferred during execution. Often, these decisions cannot even be taken before the plan is being executed since they depend on the current context and the robot's belief state. For example, the decision where to stand to pick up an object or where to search for it in the environment depends on the robot's belief about obstacles and object positions.

If control decisions are inferred based on the robot's knowledge about the world, they can be based on all information the robot has accumulated so far. Missing pieces of information can

be detected, and based on the type of information that is missing, the robot can decide which method to use to acquire this information.

This separation of knowledge from program code helps to ensure composability and re-usability: Composability is facilitated since the robot's control program does not need to be changed to take additional knowledge into account. It still sends the same queries, but receives different results that are now based on the extended knowledge. Depending on which knowledge is added, there can be more solutions or additional constraints on the results. Re-usability is improved because knowledge that has been described once can now be used multiple times: The properties of an object may be relevant for recognizing it, for inferring where to search for it, or for parametrizing actions that interact with the object. Since the knowledge is separated from the program context, it can easily be used in all these contexts. If the knowledge is represented in a formal language, it can further be used for automated inference in order to derive new facts by combining different pieces of information.

Despite a large body of work on knowledge representation, these techniques have not yet found their way into today's robots. One reason is that robotic applications have some very specific demands that make the construction of knowledge bases for robots a challenging tasks and because of which much of the work on knowledge representation in general can only partially be applied.

Grounding Much of the information a robot needs is in some form already present in its control program. The robot already has a belief about its position, about its environment and about the actions it performs. This information can be used to automatically derive symbolic knowledge from the low-level data structures. However, this raises the problem of keeping the abstract, symbolic knowledge in the robot's knowledge base consistent with the low-level data and leads to the *grounding* problem [Harnad, 1990]. Grounding means establishing the link between sub-symbolic data and symbolic descriptions, and maintaining this link over time while ensuring consistency. It is specifically needed when applying symbolic reasoning techniques to systems acting in the physical world.

Action-centric representations A robots' main task is to act in the world. For this reason, many inference tasks are related to determining which actions to take, to decide which parameters to choose, or to predict which consequences an action will have. These reasoning schemes can be supported by modeling the knowledge in an action-centric way, i.e. to describe objects, locations and grasps in terms of their relation to the respective actions. Action-centric representations are

especially needed in systems that can deliberatively choose and parametrize their actions and therefore need to reason a lot about these topics.

Integration of sensed information A large part of a robot's knowledge is dynamically acquired by sensing and acting, in particular knowledge about objects, about the robot's environment, and about the consequences of actions. To reason about this information, the knowledge base needs methods to access sensor data and to formally represent the perception results.

Uncertainty Robotics is characterized by the inherent uncertainty: Action outcomes cannot be predicted with certainty, percepts are neither exact nor necessarily correct, human preferences change, objects get moved. A robot knowledge processing system therefore needs to represent uncertain knowledge, derive symbolic knowledge from uncertain sensor data, and properly propagate uncertainty by performing probabilistic inference where needed.

Time and Dynamics Human environments dynamically evolve over time: Objects are moved, split, destroyed, created, devices are switched on and off, containers are opened and closed. Representations describing these environments therefore need to explicitly take change into account and qualify descriptions with the time at which they are valid. This also involves the description of the relation between actions, processes, and their influence on objects.

Practicality In order to be useful to a robot, a knowledge representation has to be effective and embedded in the robot's feedback loops. That is, it has to generate answers that are not necessarily optimal, but good enough to be useful to the robot, and it has to generate them fast enough not to slow down the robot's operation. An "optimal" solution that takes too long to compute may well be outdated at the time it is available and is therefore not useful for a robot.

In this thesis, we present an integrated framework for acquiring, representing, and using knowledge to enable robots to acquire new tasks and perform them in a more flexible and general manner. The knowledge is grounded in the robot's data structures, which enables the robot to perform reasoning about its internal state, about estimates of the outer world, and about perceptions of objects. The system consists of four main components:

Knowledge representation and reasoning system: The robot's knowledge needs to be explicitly and unambiguously represented and encoded in a format that supports reasoning, i.e. drawing conclusions from it to derive novel facts. In addition, the underlying representation needs to

be scalable enough to cover the large range of knowledge required to competently solve everyday tasks. There are various approaches for representing knowledge in terms of graphs (Semantic Networks, Topic Maps), as probabilistic graphical models (Bayesian networks, Markov Networks), or in logical representations like first-order logic or description logics. We chose a representation based on descriptions logics that is formal, has clear semantics, is expressive enough for our application, but still supports efficient inference. It can describe general relations between classes of things that abstract away from concrete entities, like knowledge about classes of objects. Its clear semantics allow the robot to autonomously combine single pieces of knowledge in order to integrate different information sources. We extended the logical representation with methods for probabilistic reasoning and special-purpose inference methods for reasoning tasks that are especially important for autonomous robots, like spatio-temporal reasoning about objects and the creation of grounded symbols based on the robot's internal data structures.

Representations for robot-related knowledge: The aforementioned techniques provide basic methods for representing knowledge, which need to be combined with both general and domain-specific knowledge to be useful to a robot. In particular, robots need to be capable of reasoning about their actions, their parameters and effects, as well as about physical processes, events, and other temporal information. They need to describe objects and spatial information and predict how they are changed by actions and processes. Robots should have models of themselves and their own capabilities, and should be able to use these models to determine if they can perform an action. Such representations have been developed as part of this work and are available in the knowledge base.

Methods for acquiring knowledge from the Internet: The Internet has become the largest available resource of knowledge, and there is a trend towards using this information to improve the problem-solving skills of our robots. All the information is available in digital form and is therefore in principle machine-readable, though it often needs to be converted from a format that is convenient to read for humans into formal descriptions that can be used by robots. We present methods for acquiring task descriptions and object models from existing sources on the Internet. Once such conversion methods are available, robots have access to very large sources of information and can massively increase the spectrum of tasks they can perform and of objects they can recognize. Such methods will also help to automate the knowledge acquisition process by exploiting existing sources of knowledge and thereby alleviate the need for highly skilled human knowledge engineers.

Methods for observing and analyzing human activities: Not all kinds of information can reasonably be found on the Internet. Apart from environment-specific kinds of information, like

the locations of objects and pieces of furniture, this especially includes descriptions of how to perform certain motions. Such information can often be better derived by observing how humans perform similar activities and transferring the derived knowledge to the task at hand. This requires the robot to have methods for segmenting observed motions, for enriching them with additional semantic information, and for relating them to the remainder of its knowledge. In this thesis, we present novel methods for the different analysis tasks that are involved in the interpretation of human everyday activities.

1.1 The Assistive Kitchen project

The work presented in this thesis was conducted in the context of the assistive kitchen project whose goal is to develop assistive robots that are able to act in realistic human environments and to perform everyday tasks like setting a table, tidying up, or preparing simple meals. By assisting elderly people and taking over tasks they cannot perform any more, the robots are intended to help them remain independent and enable them to stay at home for a longer time.

Though most common for humans, the kitchen environment is a highly demanding environment for robots. In a common household, the kitchen is the place where the most challenging object manipulation actions are done. Competently performing meal preparation tasks includes very skilled dexterous manipulation and interaction with fragile and deformable objects. Robots need to recognize and localize the objects required for a task, pick up tools and calibrate them. But the challenges are not only in the areas of perception and manipulation: A typical kitchen is much less structured and much more dynamic than for example an industrial environment. It contains hundreds of objects of different types that all need to handled in very specific ways, also depending on the current task context. Objects are stored at different locations in the environment and need to be retrieved from there before they can be used in a task. Especially cooking tasks fundamentally change how objects look, behave, and need to be handled.

Acting in human environments also means to interact with humans and adapt to their habits both in terms of how actions are performed and how the environment is structured. These are all requirements of the kitchen scenario that make successful operation more challenging than in other domains.

1.2 Example scenario

Throughout the thesis, we will use the scenario of making pancakes as an example to show how the different mechanisms contribute to enabling a robot to perform complex manipulation tasks. Though a very simple task for a human, it includes several aspects that make it interesting and challenging as a robot task: The robot needs to determine the sequence of actions to perform, it has to parametrize the actions in order to achieve the desired effect, and it must reason about the effects of actions on the involved objects which are changed in a substantial way during the cooking process.

Imagine that a household robot receives the command "make some pancakes". Given this command, it has to create a plan to achieve the given goal, which is usually solved by planning, i.e. by generating a plan from first principles by searching for a sequence of actions that leads to the desired goal state. However, doing this for human everyday tasks like cooking is still far beyond the capabilities of today's planning systems: The goal state is massively underspecified, and the robot needs to select the right actions from a large number of candidates and choose their parameters like the right objects, locations, motions, and the right timing. This makes it difficult to use classical planning systems in the context of everyday activities.

As an alternative, we propose to make use of existing descriptions that explain how to perform these activities that are available in their thousands on web sites like ehow.com which explain them to humans. Instead of trying to generate an action sequence from first principles, the robot could thus search the Web for suitable instructions, download them and convert the natural-language instructions like the one exemplarily shown in Figure 1.1 into an executable plan.

The instructions only describe the overall course of actions on a very abstract level, which is by far not sufficient to actually enable a robot to perform the actions. One important reason is that they were written for humans and therefore lack a lot of information that a robot has to supply. Since no single source provides all required pieces of knowledge, the robot needs to integrate several of them: Encyclopedic knowledge about the properties of actions and objects, common-sense knowledge, environment models, experience data it has collected like memories of the outcome of actions or the positions of objects. This knowledge needs to be complemented with information extracted from observations of human activities or information sources on the Web. Figure 1.2 visualizes which pieces of information are required to fill in the different gaps in the original instructions. In the following, we will discuss exemplarily how a robot could proceed to complete the instructions and where it can obtain the required information from.

In a first step, the robot uses its knowledge about actions and their effects on objects to predict which results the different actions in the plan will have and which intermediate objects will

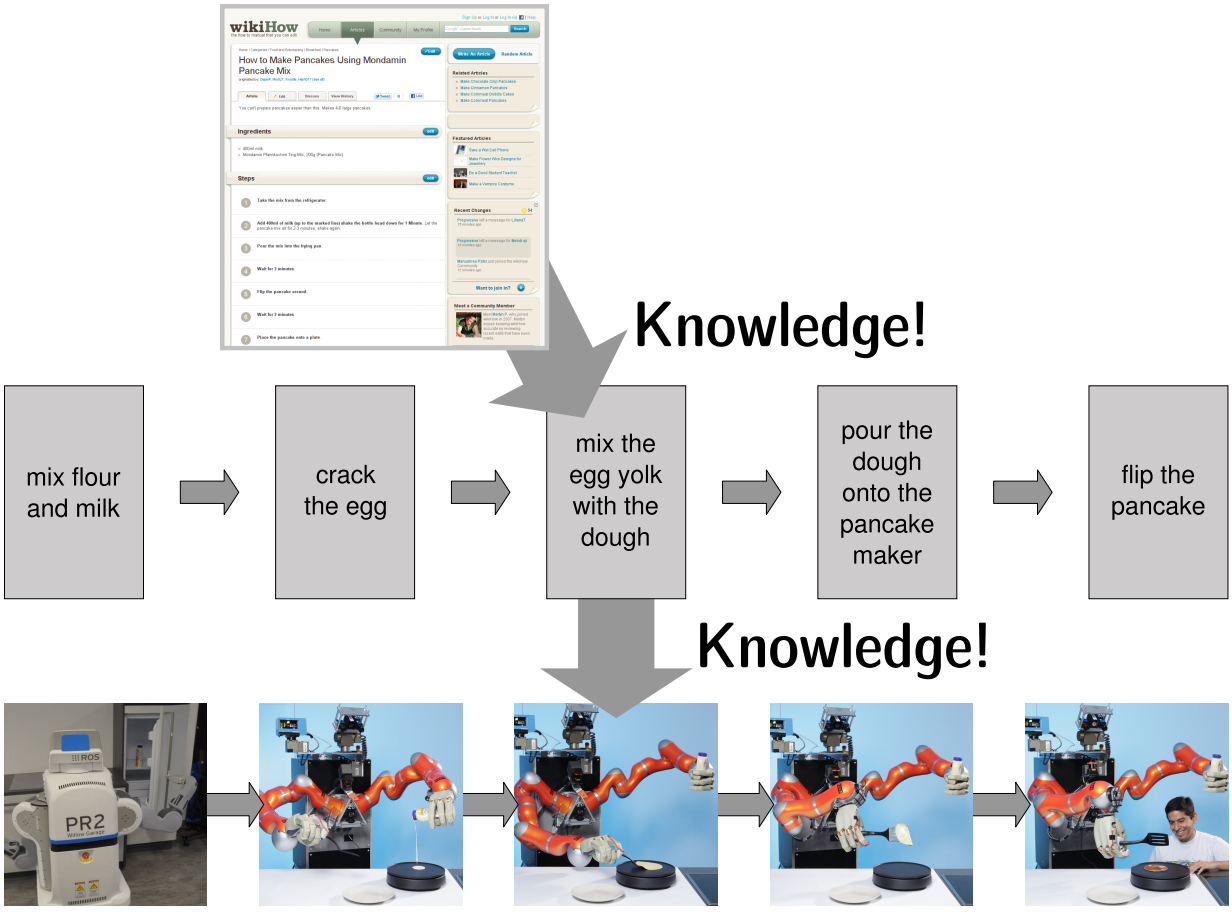


Figure 1.1 The translation of natural-language instructions like they can be found on web sites into a logical representation and into actually executable robot plans is a very knowledge-intensive process. Knowledge is needed to resolve ambiguities in the instructions and to fill in missing information.

appear. Especially in complex tasks like cooking and meal preparation, objects can be destroyed, created, mixed and transformed. For example, when making pancakes, the egg is destroyed, the egg white, egg yolk and the egg shells appear as new objects, the dough is created by mixing flour with milk and is later transformed into a pancake. To manipulate these objects, the robot needs methods for recognizing them and for computing their poses. The robot therefore needs to know which object recognition models it has, and should have methods to generate object models if none are available. This could for instance be done by using information from on-line sources, like product pictures from online shops, or by exchanging object models with other robots.

In many cases, the plan will refer to some kind of stuff, something that can be split into pieces which are still of the same type. Flour, milk and dough, for example, are stuff-like since a cup of milk is still milk. Stuff-like things are very common in the kitchen, and as they are often some

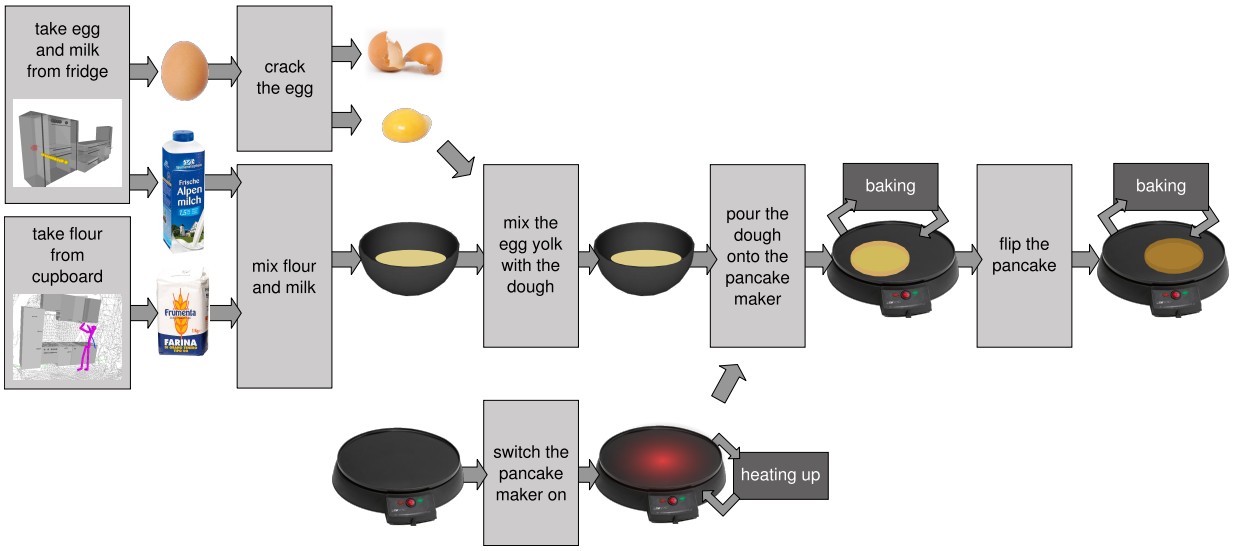


Figure 1.2 Completed task specification with intermediate objects created by the actions. The following chapters explain how the different kinds of knowledge can be integrated into the initial action sequence.

kind of liquid or powder, they are normally stored inside containers. Actions for locating stuff-like things or for picking them up therefore need to be performed on the surrounding container instead of the stuff itself. Based on its knowledge about actions and objects, a robot has to decide if an action needs to be adapted in this way.

After this first step, the robot should know all explicitly and implicitly referenced objects and has made sure it has all recognition models for each of them. Now it has to decide where these objects are likely to be found and add actions to search for them in the environment. Normally, objects like food, cooking utensils or tableware are stored inside cupboards and drawers, so they are not visible to the robot. To locate these objects in the environment, the robot needs information about the pieces of furniture as well as information about the different kinds of objects stored inside of them. Based on this knowledge, it can automatically add additional actions to the plan to retrieve all required objects from their storage locations, and parametrize these actions with the position of the respective container and information how to open it like the pose of the handle and the opening trajectory. This information can be read from a semantic environment map that combines spatial information about the locations of pieces of furniture with semantic information about their types and articulation models – the poses and types of hinges and, if the robot has already opened the respective container, also with the trajectory it recorded.

Now the robot knows all the objects, can recognize them and locate them in its environment, and, using its projection mechanisms, has created a first prediction of the effects of actions.

However, this projection only covers the immediate effects of actions, while indirect effects may often have even more significant impact. For example, when pouring pancake dough onto a pancake maker, the immediate effect is that some amount of dough is on top of the pancake maker, but the more important and desired effect is that the dough is to be baked to a pancake. This baking process is started when the dough gets in contact with something sufficiently hot and thereby establishes a heat path through which the dough can heat up and bake. Indirect side-effects of actions are often caused by processes that become active by the direct effects of an action. It is thus important for a robot to have qualitative knowledge about processes to predict the outcome of its actions, to diagnose failures, and to plan actions in order to achieve a given goal.

Many actions involve special movements that need to be performed, like the motion to push a spatula under a pancake, to lift and to turn it. These motions can hardly be described verbally, so this information is missing in the original descriptions and also needs to be inferred by the robot. Planning these motions is usually very difficult due to the very high-dimensional search space and the lack of well-defined success criteria. If the robot has the ability to observe how humans perform a similar task involving the same kinds of motions, it can, however, learn from these observations how its own movements should look like. This requires it to find the right motion segments in a large amount of motion tracking data, for which it needs semantic descriptions of what the human is doing at which time. If these descriptions are in the same language that is also used to plan the actions of the robot, they can be used to retrieve the motions to use for certain parts of the plan.

This example shows which different kinds of knowledge about actions, trajectories, objects, the environment, and processes from very different sources like engineered background knowledge, internal models, experience, observations of humans, and the Internet need to be integrated to enable a robot to competently perform an underspecified meal preparation task. Most of these knowledge sources use different representation and different vocabulary to describe similar things. Integrating them requires to transform them into a common representation with well-defined formal semantics and a joint vocabulary that ensures that the same kind of information is described in the same way in all parts of the system. Serving as such an *interlingua* is one of the main tasks of a formal knowledge representation system. In addition, it provides inference methods that can derive novel statements from the existing knowledge. Over the course of the following chapters, we will come back to this scenario and explain how the different kinds of knowledge can be acquired and integrated to complete the instructions. Section 7.3 will then sum up and explain in detail how the different methods contribute to the overall goal.

1.3 Reader's guide

Though this work can be read in a linear fashion, the interested reader may also decide to concentrate on single parts. The chapters are mainly self-contained, though a basic understanding of the knowledge representation concepts used is helpful.

Chapter 2 introduces the main concepts and techniques for knowledge representation and reasoning and describes the knowledge representation framework that has been developed as part of this thesis. This chapter is recommended for all readers, especially for those without strong background in knowledge representation techniques.

Chapter 3 describes in more detail how the different kinds of knowledge about events, actions, objects, spatial configurations, processes and robots are represented. It will help with a more in-depth understanding of the methods used in the following, more application-oriented chapters.

Chapter 4 presents methods for acquiring knowledge from Internet resources, namely task instructions and object descriptions, and their translation from natural language into a logical format.

Chapter 5 deals with techniques for analyzing and abstracting observations of human manipulation actions in order to make them available to the robot as a source of knowledge. Having read at least Section 3.3 is recommended since the action representations introduced there are used to describe observed human actions.

Chapter 6 finally describes how the knowledge processing techniques are integrated into the robot control system and how the knowledge is used to take decisions. Sections 3.2, 3.3 and 3.5 introduce the concepts used here.

Chapter 7 presents several integrated experiments in addition to the evaluation of the single components. Since the different chapters deal with rather diverse topics, there are separate sections for experiments, discussion and related work at the end of the respective chapters. The global evaluation section then puts the different aspects into their context and discusses the strengths and limitations of the overall system.

1.4 Contributions

We present an integrated framework for acquiring, representing, and using this knowledge to enable robots to acquire new tasks and perform them in a more flexible and general manner. The main contributions of this work are the following:

- We created a large-scale robotics ontology and an ontology of the household domain. The ontologies allow to describe in a coherent format events, temporal information, actions, complex tasks, action parameters, objects, spatio-temporal information, processes, and robot components and capabilities.
- KNOWROB, the state-of-the-art robot knowledge processing system developed as part of this work is a scalable knowledge base that is embedded into the robot's control program, combines various reasoning methods, and provides close links to the robot's perception and action system.
- We introduce the concept of on-demand computation of semantic information based on the robot's internal data structures. These procedural attachments allow to automatically derive grounded symbolic representations of different granularity from subsymbolic data and to compute different abstract views on the same original data. In our experience, this kind of computation is key to the application of knowledge representations to robotics.
- We created novel representations of change for object poses as an extension of the Fluent calculus [Thielscher, 1998] that additionally store the source from which a piece of information entered the knowledge base, like a certain perception or inference method. To store this information in KNOWROB, we developed a representation in pure description logic.
- A novel integrated representation of actions and processes that supports both projection and planning of the effects of actions and processes on the objects they interact with. The representation combines declarative representations of the inputs and effects with projection rules.
- Methods for translating task instructions from the WWW into a formal logic-based format, including semantic parsing, word sense disambiguation and ontology mapping.
- Techniques for the automated generation of an ontology of household products based on Web information from an on-line shopping web site.
- Integrated, knowledge-based models of human everyday activities that describe observations of human actions from the level of single motions up to the level of complete activities. To automatically construct the models from observations of humans, we developed methods for segmenting, abstracting and analyzing human motions and for learning the

partial order in human activities.

- To exchange knowledge between robots, we investigated representations to encode actions, object models, and environment maps in a way that robots can both communicate the information itself and autonomously verify if the data is usable by them given their capabilities.
- All developed software, the ontologies and models that have been created and the datasets that have been used to evaluate the approaches have been released to the public as open-source and are already being used at several other research institutions.

Chapter 2

The KnowRob knowledge processing system

A knowledge base that is being used by a robot during operation does not only need to answer queries fast enough not to slow down the execution of actions, it also needs to provide the basic techniques to realize representations for all pieces of knowledge the robot needs. It should provide suitable representation formalism that allow the robot to assign meaning to the content of its knowledge base and to draw conclusions based on this information. The representation has to be expressive enough to cover everything the robot needs to describe, but it still has to support efficient inference. We propose to extend a classical first-order logical knowledge base with probabilistic classifiers that translate between continuous observations and discrete symbols, and with statistical relational models that can describe complex uncertain relational knowledge. This combination allows to describe continuous and discrete, propositional and relational, deterministic and probabilistic knowledge, and ensures scalability by always using the most efficient description available.

Since the robot needs to perform reasoning about phenomena in the outer world, its representations need to be grounded in its perception and action systems. This requires mechanisms to construct abstract symbols from observations, to recognize them later, and to update the robot's internal belief about them. We introduce the paradigm of treating the"world as a virtual knowledge base" that allows the robot to forward queries "to the world" in terms of perception tasks and to use the answer as if it had already been in the knowledge base. This on-demand knowledge acquisition and on-demand computation of relations helps to ensure that symbols are grounded in the perception system and in the robot's internal data structures.

In this chapter, we present techniques to represent the required knowledge and to use it for inference. In particular, we describe KNOWROB, the knowledge processing system that has been

developed as part of this thesis. It combines knowledge representation and reasoning methods with techniques for acquiring knowledge and for grounding the knowledge in a physical system and can serve as a common semantic framework for integrating information from different sources. KNOWROB combines static encyclopedic knowledge, common-sense knowledge, task descriptions, environment models, object information and information about observed actions that has been acquired from various sources (manually axiomatized, derived from observations, or imported from the web). It supports different deterministic and probabilistic reasoning mechanisms, clustering, classification and segmentation methods, and includes query interfaces as well as visualization tools.

We start with a discussion of the main concepts and design decisions realized in KNOWROB (Section 2.1). We will then describe the components of the system and the different kinds of knowledge sources that are being used (Section 2.3), before we describe the main reasoning methods in more detail: deterministic logical inference (Section 2.4), probabilistic logical inference (Section 2.5), and procedural attachments to the semantic representations (Section 2.6).

2.1 General concepts

2.1.1 The world as a virtual knowledge base

An important aspect of the KNOWROB knowledge representation system is the tight integration with the robot's sensing and acting capabilities. Robots have to perform reasoning based on information that was perceived from the outer world, and need to take decisions based on the most current information and at the most accurate (or most appropriate) level of abstraction. This requires them to have access to sensor data, robot-internal data structures like the plan that is currently being executed, and to information like the current localization quality.

The common procedure in literature [Lemaignan et al., 2010; Daoutis et al., 2009] is to abstract the sensed information into symbolic concepts, to assert these concepts to the knowledge base, and to perform reasoning only on these (already abstracted) pieces of information. For example, a robot may detect some objects, compute that they are on top of a table, add these new object instances to its knowledge base and assert the 'on-top-of' relation with the table instance. While this method is straightforward, its drawback is that the link between the abstract descriptions and the original information, in this case the positions of the objects in space, is lost. The robot has only qualitative information about the object positions and cannot derive any new relations from the information in the knowledge base, for example to determine which objects are in front

of which other ones. All relations that could possibly be of interest have to be computed when adding the objects to the knowledge base.

Our approach is different: We do not pre-compute all knowledge that could potentially be needed and push this information into the knowledge base. Instead, we give the knowledge base the ability to compute knowledge on demand when it is actually needed, and to ask other components if the required information is not available. When seeking a solution to a query, the knowledge base can forward the query, for example to the perception system to generate an answer based on information perceived from the outer world. We thus regard the world as a "virtual knowledge base" to which we can send "queries" in terms of perception tasks that are answered based on the estimated world states.

2.1.2 On-demand computation

The virtual knowledge base paradigm requires the system to have the ability to load and compute information on demand. We thus need to provide it with descriptions how to obtain a certain piece of information when it is needed to answer a query. These descriptions are realized as procedural attachments to semantic relations. They can either be attached to classes of objects and describe how the system can compute instances of these classes, or to semantic relations and compute if the relation holds between two objects. We call these two kinds of procedural attachments "computable classes" and "computable properties".

One important application of computables is to load information into the knowledge base: Computable classes can for example generate object instances by asking the vision system for the objects it has detected. Another application is to compute qualitative relations between objects. If the object poses are known, qualitative spatial relations like "in", "on", or "next to" can easily be computed on demand.

These qualitative relations can be seen as different *views* on the original position data. The abstraction is not performed when the data is acquired, but immediately before the abstracted information is needed, giving the system greater flexibility. Using computables, the system can compute new relations at a later point in time, abstract information up to different levels, and also generate a less abstracted version of the stored information. Only those relations which are actually needed are computed, and are computed when they are needed, which helps to avoid inconsistencies due to outdated information. The implementation of computables is explained in more detail in Section 2.6.

2.1.3 Logic-based representation

We chose first-order logic as representational formalism because of its expressiveness, its capability to describe relational knowledge, and because of its formal semantics that allow to draw conclusions from the available knowledge. There are various logical languages for describing first-order relations in logics, ranging from the generic predicate logics [Frege, 1879] over logical programming languages like Prolog [Sterling and Shapiro, 1994] to languages that combine logical with probabilistic representations [Getoor and Taskar, 2007].

All these logic dialects differ in what they can describe and how elegantly different kinds of facts can be expressed. Choosing the right representation with the right expressiveness is an important choice for the design of a knowledge representation and reasoning system. In general, simpler and less expressive representations (e.g. RDF [Beckett, 2004] or OWL-lite [Motik et al., 2009]) allow more efficient reasoning, often guaranteeing desired properties like decidability. Their drawback is that important relations may not be expressible in these languages, or cannot be expressed in an elegant way. On the other hand, very expressive representations (like CycL [Matuszek et al., 2006], Scone [Fahlman, 2006], or Topic Maps [ISO/IEC 13250:2000, 1999]) are able to model almost everything that can be expressed in natural language, but often have poor support for efficient reasoning.

In KNOWROB, we chose Description Logics (DL) as formalism to represent the robot's knowledge. Description logics are a family of logical languages for knowledge representation, consisting of several dialects with different expressiveness, most of which are a decidable subset of first-order logic. In particular, we use the Web Ontology Language (OWL [Motik et al., 2009]) for storing Description Logic formulas in an XML-based file format. OWL was originally developed for representing knowledge in the Semantic Web [Lee et al., 2001], but has since become a commonly used knowledge representation format. In the remainder of this section, we will briefly summarize the main concepts of description logics. An extensive overview can be found in [Baader et al., 2007], a shorter introduction in [Baader et al., 2008]. Table 2.1 gives an overview of the DL syntax.

Description Logics distinguish between terminological knowledge, the so-called TBOX, and assertional knowledge, the ABOX. The TBOX contains definitions of concepts, for example the concepts *Action*, *SpatialThing*, *PickingUpAnObject* or *TableKnife*. These concepts are arranged in a hierarchy, a so-called taxonomy, using subclass definitions that describe for instance that a *TableKnife* is a specialization of *SilverwarePiece*. The ABOX contains individuals that are instantiations of these concepts, e.g. a concrete knife *knife1* as an instantiation of the concept *TableKnife*. When modeling knowledge in robotics, the ABOX usually describes detected object

instances, observed actions or perceived events. The TBOX, in contrast, describes classes of objects or actions. Note that the differences between ABOX and TBOX are not domain-specificness or environment-dependency: Individuals in the ABOX can be very general, like an instance of *SpatialThing*, and classes in the TBOX be very specific, like the class of action "Grasping an egg from the refrigerator with the right hand using a pinch grasp".

Roles can describe the properties of an individual, describe the relation between two individuals, and can also be used in concept definitions to restrict the extent of a class to individuals having certain properties. For example, the concept *OpeningABottle* can be described as a subclass of *OpeningSomething* with the restriction that the *objectActedOn* has to be some instance of a *Bottle*.

$$OpeningABottle \sqsubseteq OpeningSomething \sqcap \exists objectActedOn.Bottle$$

A knowledge representation that consists of a taxonomy of concepts and relations between them is called an "ontology". The knowledge is formally represented and allows to draw conclusions using logical inference. OWL is a file format for storing and exchanging description logic formulas. The nomenclature in OWL differs slightly from DL: "concepts" are usually called "classes" in OWL, "roles" are called "properties", and "individuals" are called "objects" or "instances". Table 2.1 compares the language constructs in DL and their correspondences in OWL (taken from [Baader et al., 2008]).

While OWL has several advantages including the structured descriptions, the decidability, and the fact that it is standardized and widely used, there are also some limitations with respect to the representation of the knowledge of an autonomous robot: As a description logic dialect, it is limited to binary predicates and stores all knowledge in terms of Subject-Predicate-Object triples. If more complex n-ary relations are to be expressed, one has to resort to reification, i.e. to creating an intermediate instance that represents the relation to be expressed. Though this is less elegant than native support for such relations in the language, it is no general limitation, and we will see later how KNOWROB uses this approach to express for example complex spatio-temporal relations (Section 3.2.5.1). With the triple structure, one can only express that an object *is* at a location, but not that it *was* at a position at some point in time with a certain probability. By reifying this relation, such information can be stored.

Reification can create rather complex structures, but using the computables described in the previous section, these structures can be made transparent to the user, still allowing simple queries for default cases: If, for example, the user asks for the pose of a bottle without specifying the time, the system by default returns the current pose. In this case, the query is the same

OWL Syntax	**DL Syntax**	**Example**
Thing	$\top$	
Nothing	$\bot$	
intersectionOf	$C_1 \sqcap \cdots \sqcap C_n$	$Human \sqcap Male$
unionOf	$C_1 \sqcup \cdots \sqcup C_n$	$Doctor \sqcup Lawyer$
complementOf	$\neg C$	$\neg Male$
oneOf	$\{x_1 \ldots x_n\}$	$\{john, mary\}$
allValuesFrom	$\forall r.C$	$\forall hasChild.Doctor$
someValuesFrom	$\exists r.C$	$\exists hasChild.Lawyer$
hasValue	$\exists r.x$	$\exists citizenOf.USA$
minCardinality	$(\leqslant\ n\, r)$	$(\leqslant\ 2\, hasChild)$
maxCardinality	$(\geqslant\ n\, r)$	$(\geqslant\ 1\, hasChild)$
inverseOf	r^-	$hasChild^-$
subClassOf	$C_1 \sqsubseteq C_2$	$Human \sqsubseteq Animal \sqcap Biped$
equivalentClass	$C_1 \equiv C_2$	$Man \equiv Human \sqcap Male$
subPropertyOf	$P_1 \sqsubseteq P_2$	$hasDaughter \sqsubseteq hasChild$
equivalentProperty	$P_1 \equiv P_2$	$cost \equiv price$
disjointWith	$C_1 \sqsubseteq \neg C_2$	$Male \sqsubseteq \neg Female$
sameAs	$\{x_1\} \equiv \{x_2\}$	$\{Pres\_Bush\} \equiv \{GW\_Bush\}$
differentFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	$\{john\} \sqsubseteq \neg\{peter\}$
TransitiveProperty	$P\, transitive\, role$	$hasAncestor$
FunctionalProperty	$\top \sqsubseteq (\leqslant\ 1\, P)$	$\top \sqsubseteq (\leqslant\ 1\, hasMother)$
InverseFunctionalProperty	$\top \sqsubseteq (\leqslant\ 1\, P^-)$	$\top \sqsubseteq (\leqslant\ 1\, isMotherOf^-)$
SymmetricProperty	$P \equiv P^-$	$isSiblingOf \equiv isSiblingOf^-$

Table 2.1 Language elements in OWL and DL syntax. C_i are concepts, x_i are individuals, r,P_i are roles, and n is a positive integer. Examples taken from [Baader et al., 2008].

as in a simple system without reification. But, if the user explicitly asks for the pose of this bottle at another point in time, thus using a more complex query, that position is determined based on the last observation of that bottle before the respective point in time.

2.1.4 Prolog-based inference

The inference process has to be adapted to include the additional solutions that are generated by computables into the result set. In a classical system, all inferences are drawn based on only the knowledge that is asserted to the knowledge base. It is thus assumed that all knowledge is known to the system at the beginning of an inference procedure which can thus be optimized for this case. With computables, the knowledge base grows while answering a query: New instances are created by computable classes, and new relations between instances are computed by computable properties.

On the one hand, the inference algorithm has to provide hooks for the computable predicates to plug in and to provide these additional pieces of knowledge, and on the other hand, it has to be flexible enough to take these pieces of knowledge into account. We chose Prolog to implement our system: Inference in Prolog is mainly a search procedure, and it is very easy to add additional alternatives to each step by just defining an additional predicate. For example, when exploring all instances of a class, the search first returns all asserted instances, but additionally provides a hook for computables to provide further solutions. The results of these computables are then included into the reasoning process in the same way as the normal, asserted instances. Details about the inference mechanisms can be found in Section 2.4.

2.1.5 Modular design

KNOWROB is designed to be usable on a wide range of robot platforms with different capabilities (and even on non-robotic systems). To flexibly add, remove, or exchange parts of its functionality, it is implemented in a very modular way. Each module can provide two kinds of extensions: First, it can contain additional knowledge as an extension of the KNOWROB ontology, and second also additional reasoning or computation capabilities, realized as new computable classes or properties. Dependencies between modules are resolved automatically: Each module initializes its direct dependencies, which then initialize their dependencies and so on.

When a module is loaded, it loads its local ontology fragment, which makes the contained knowledge available to the rest of the system, and further announces which computables it provides for which classes or properties. From that time on, these computables are automatically included for answering future queries. This allows to easily extend the system's capabilities by just loading a module.

2.2 Ontology layout

The layout of the upper levels of the ontology, including many classes, their hierarchy and properties, has been adopted from the OpenCyc ontology [Lenat, 1995]. Adopting the ontological structure also means to adopt a certain way of thinking since the vocabulary means that a language provides shape the way how things can be described. The modeling of events and processes in KNOWROB is similar to the representations in OpenCyc, whereas other parts like the description of object poses and the models of change have been developed specifically for KNOWROB.

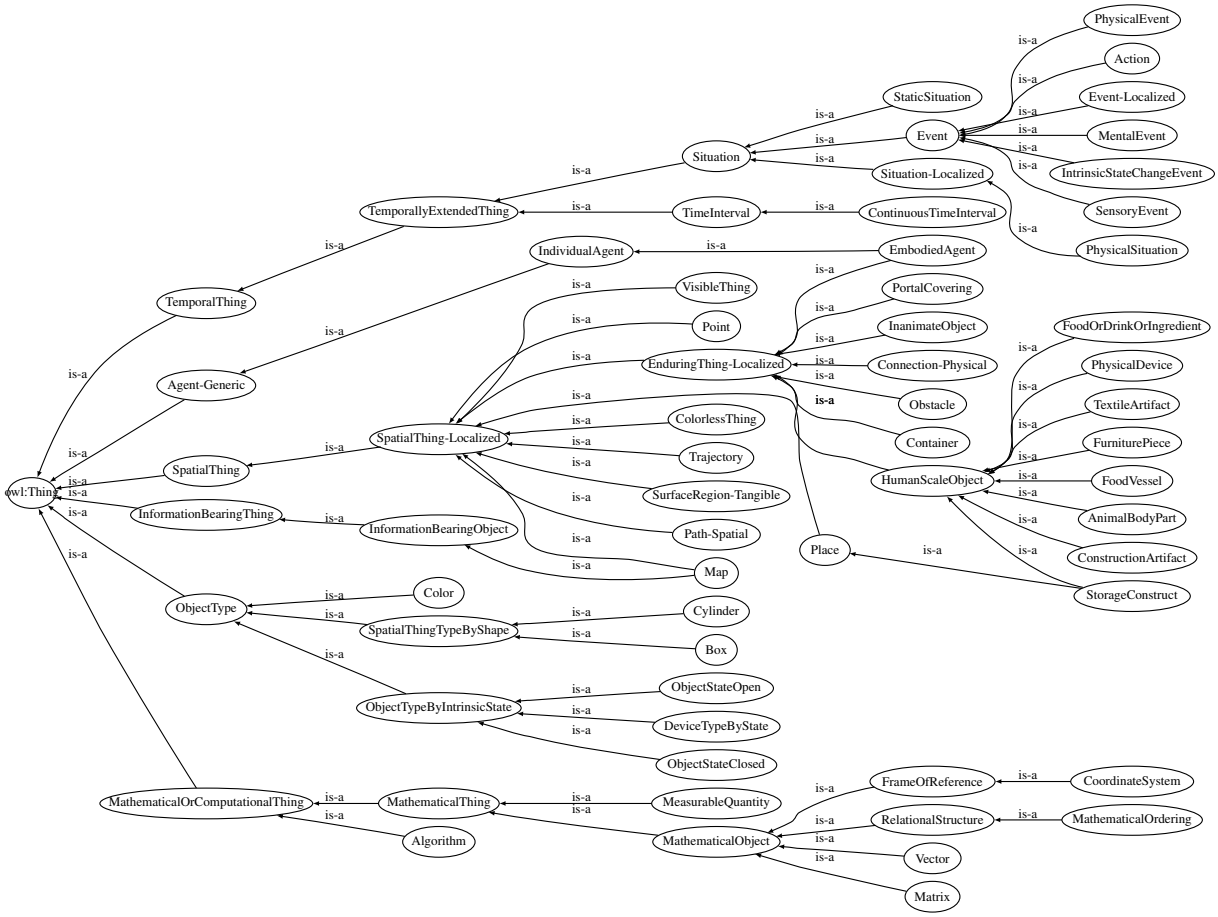


Figure 2.1 Layout of the KNOWROB upper ontology, including the main classes for describing spatial, temporal and mathematical things.

We decided to adopt the OpenCyc ontology for several reasons: OpenCyc has emerged as quasi-standard for robot knowledge bases, and we would like to remain compatible in order to facilitate the exchange of knowledge with other systems. There are also many tools and

links between OpenCyc and other knowledge bases (e.g. the links to WordNet that are used in Section 4.1) that we can profit from. Another advantage is that the Cyc ontology has been created by experienced experts with the intention of building a general ontology. We hope that this will help in case we need to extend the ontology from household robots to a larger domain.

However, since OpenCyc was developed as a general upper ontology with the intention of covering the whole range of human knowledge, the ontology is broad and extensive, but not always as detailed as needed for robots. Much domain-specific knowledge is missing, for example in the areas of mobile manipulation and human everyday activities. We therefore extended Cyc with more detailed descriptions of e.g. everyday tasks, household objects and robot parts. In addition, we developed special representations to reason about change, like the changing positions of objects (Section 3.2.5.1) or objects that change over time.

Figure 2.1 visualizes the uppermost levels of the KNOWROB ontology. The most important branches are the *TemporalThings*, containing descriptions of *Events*, *Actions* and *TimeIntervals*, the *SpatialThings*, describing abstract spatial concepts like *Points* or *Trajectories* as well as all the different object classes. Most objects in the robot's environment as well as pieces of furniture or body parts are subsumed under the *HumanScaleObject* class. Other notable branches are *MathematicalThings* like a *Vector* or a *CoordinateSystem* and the *InformationBearingObjects* describing data objects like maps.

2.3 System architecture

This section is to give an overview over the interplay between the different components in KNOWROB in order to better understand their role in the system. Figure 2.2 groups the components by their functionality. Figure 2.3 will later show the flow of knowledge and explain which kind of knowledge can be acquired from which sources.

KNOWROB consists of several functional modules that can largely be grouped into five categories. The central component is the *knowledge representation* that provides the mechanisms to store and retrieve all the different kinds of information in the system. Robots need very powerful representations that are expressive enough to describe all aspects of actions, objects, processes, temporal events, their properties and relations. A detailed description of the different representational mechanisms can be found in Chapter 3.

There are four groups of components interacting with the representation: First, there are knowledge acquisition methods to populate the knowledge base with newly acquired information. Parts of the robot's knowledge have been manually encoded, other parts can be acquired

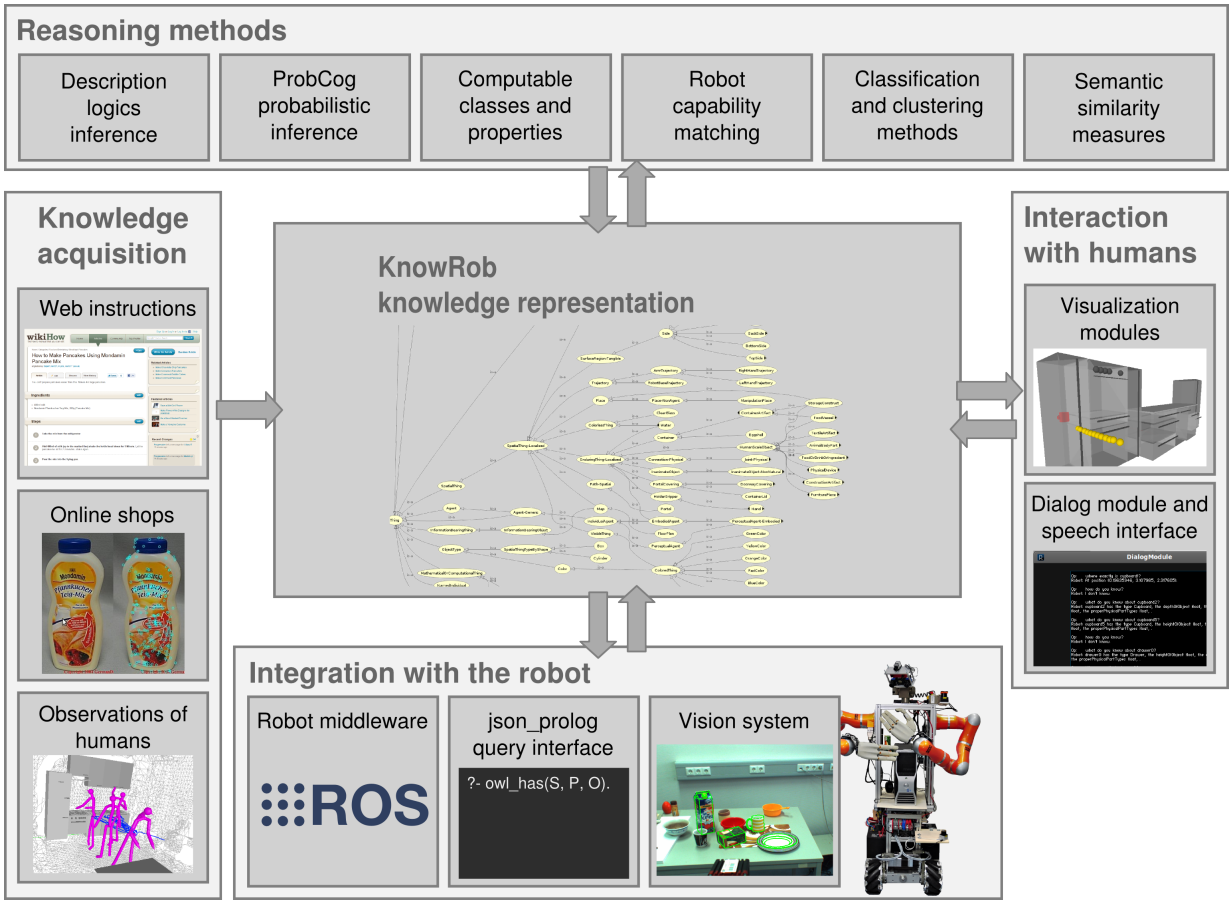


Figure 2.2 Functional overview of the KNOWROB knowledge processing system.

automatically from sources like web sites on the Internet, or from observations of humans. The different knowledge sources are described in more detail later in Figure 2.3.

A second important type of component are the reasoning methods which enable the robot to derive new statements from the existing knowledge. KNOWROB integrates multiple general- and special-purpose inference methods: Description logic inference is the basic mechanism, and mainly deals with the types of things and the automatic classification of things based on their properties. This method is described in more detail in Section 2.4. While pure description logics inference is completely deterministic, it is often desirable to represent uncertain information. The Markov Logic Networks and Bayesian Logic Networks in the PROBCOG toolkit allow to draw *probabilistic* inferences that combine the expressiveness of first-order logics with the representation of uncertainty (see Section 2.5 for details). Not all inferences can elegantly be performed using pure logical reasoning, some require computations or even more complex data manipulation. For these cases, computable classes and properties allow to specify procedural attachments to the semantic representations. They can be used to generate individuals of a class, to determine the type of an individual, and to compute relations between individuals (Section 2.6).

The previous methods are general inference methods, while others are more specific for certain applications. For example, a robot may need to verify whether it is able to execute a novel plan, or if the actions would require additional capabilities. This decision can be taken based on a self-model of the robot, in conjunction with methods for matching the robot's capabilities against the requirements of the actions in that task. These methods are described in Section 3.5.

KNOWROB further integrates the Weka [Witten and Frank, 2005] and Mallet [McCallum, 2002] libraries for learning and applying statistical classification and clustering techniques. These methods can be used to integrate continuous sensor data into the symbolic knowledge base. An example use case is the segmentation and classification of human motion tracking data (Section 5.4). Another module computes semantic similarity measures that describe how close two concepts are in the knowledge base. This similarity value can for instance be used to determine appropriate storage locations for an object, namely where similar objects are stored (Section 6.2).

The third kind of component are interfaces to other parts of the robot's control system. On the one hand, they update the belief state inside the knowledge base based on external information, on the other hand, they offer knowledge and reasoning services to other components. KNOWROB can access information via the ROS communication middleware and include this information into the reasoning process. An example are the object poses determined by an object recognition system, which are obtained as described in Section 6.1. KNOWROB also provides a query interface that allows other components to send queries via a ROS service.

Finally, there are tools for visualizing knowledge and communicating with humans. Visualizations are very useful for inspecting and debugging the content of the knowledge base, but also to showcase the robot's belief state. KNOWROB's visualization canvas accepts arbitrary object instances and draws them at the position where they are believed to be. The graphics generated by the visualization tool are used throughout the following sections. For communication with humans, there is a simple dialog system that receives questions in natural-language (either via a chat client or via spoken language), translates them into Prolog queries, and converts their result back into natural-language sentences.

Figure 2.3 gives a different view on the system by showing the different kinds of knowledge that are represented and the sources from which they are acquired. The basic KNOWROB ontology has been created manually and provides the robot with encyclopedic knowledge about the types of things and their properties as well as common-sense knowledge about the household domain. This knowledge serves as the basic vocabulary the robot can use to describe the world it operates in, and is thus very important to integrate the other sources of information.

Since the manual creation of large-scale knowledge bases is very complex and time-consuming, we created methods to automate the acquisition of knowledge. One important source of knowledge is the Internet: Chapter 4 describes our approaches to using task instructions and object information from Web sources. Most of the information on the Web has been written for humans and is thus available in terms of natural-language descriptions. In order to use this information, a robot has to convert it into a language that is compatible with its knowledge base. Our system performs this conversion by a combination of semantic parsing, word sense disambiguation, and mappings of natural-language words to concepts in the ontology.

Once a robot has finished this conversion process, it can share the acquired knowledge with other robots via the RoboEarth platform. This on-line database is intended to be a "Wikipedia for robots" that is filled and used by robots and contains information about tasks, objects, and environments. The representations described in this thesis are used to encode the exchanged information, and KNOWROB serves as inference engine for different reasoning tasks. A description of this concept can be found in Section 6.3.

While textual information, like task instructions, and visual information, like pictures of objects, can easily be found in Internet sources, information about the motions that are needed to perform a task is more difficult to obtain. One possible source of such information is the observation of humans performing similar tasks. To make use of this information, the robot needs methods for observing human motions and for selecting the segments it is interested in (e.g. the motion for opening a certain cupboard door; see Section 5.4). If higher-level information

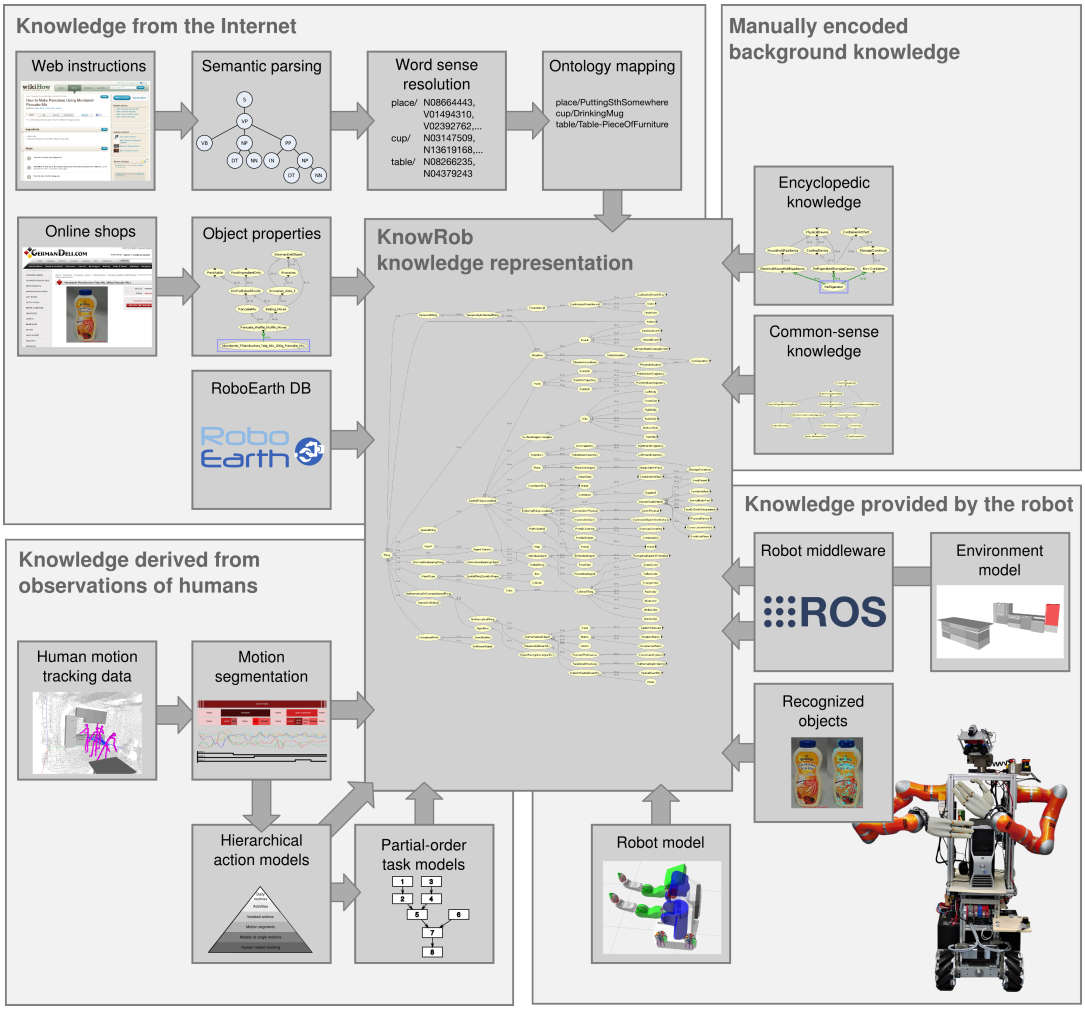


Figure 2.3 Sources of information integrated into the KNOWROB knowledge base.

about the complete task context is desired, the robot should also be able to abstract from the motion level up to more meaningful actions (Section 5.6), and to learn the structure of tasks from multiple observations (Section 5.7).

The previous kinds of knowledge were largely independent of the specific robot and its concrete working environment, but a robot also needs information about the structure of the environment it operates in (Section 6.2), the objects detected by the vision system (Section 6.1), its own capabilities (Section 3.5), and the current state of its control program. These kinds of information can only be provided during run-time, and since they strongly depend on the current situation the robot is in, they are computed on demand based on the robot's internal data structures. In KNOWROB, there are computable classes and properties that interface different sources of knowledge, like the vision system or the robot's communication middleware, and compute symbolic views on this often sub-symbolic data.

2.4 Description logic inference

A knowledge processing system needs methods to store knowledge, to query for it, and to combine pieces of knowledge to perform logical inferences. In this section, we discuss how these aspects are realized in the KNOWROB knowledge base. Many inferences in description logics can be reduced to the problems of class subsumption (i.e. to determine if one class is a subclass of another one) and classification (i.e. to determine if an individual belongs to a certain class). These tasks are usually performed by a description logics reasoner. There are several existing reasoners, including Racer [Haarslev and Müller, 2001], Pellet [Sirin et al., 2007], and HermiT [Shearer et al., 2008] which are highly optimized for these inference tasks, but unfortunately not well-suited to robotics applications: First, they always keep a classified version of the knowledge base in memory, and whenever the knowledge base changes, everything needs to be re-classified. For small knowledge bases, this classification can be done in few milliseconds, but for larger and more complex ones, it can take significant time. This makes updates to the knowledge base rather costly, though a robot needs to perform them whenever new sensor data is acquired. Re-classifying the complete knowledge base every time would be too expensive. Robots thus need reasoning methods that can handle continuous updates of the knowledge base.

Moreover, the robot's knowledge is not static, i.e. not all knowledge is known at the beginning of the inference process. New pieces of knowledge can be generated by on-demand computation methods and need to be included into the running inference process. The inference methods have to be modified in order to take these results into account in addition to the knowledge that is statically asserted to the system. The system should further be able to load and unload computable definitions during run-time, for instance to allow the robot to compute object poses from vision information only when the respective vision system is running.

Integrating these techniques into a classical reasoner would have been very difficult, so we chose a solution based on Prolog: The knowledge is internally represented in terms of Prolog predicates to which the common Prolog inference methods can be applied. The methods for storing and querying knowledge thus become very similar, using the same predicates with different combinations of bound and unbound variables. Introducing additional knowledge sources, like loading computable definitions, can easily be done simply by adding a new predicate definition to the Prolog database, which is then automatically included in the reasoning process.

An important difference to common DL reasoners is that KNOWROB uses Prolog's closed-world assumption: Everything that is not known to be true is assumed to be false, whereas the usual DL semantics make the open-world assumption that everything that is not explicitly known to be false is true. While this conflicts with the usual DL semantics, it proved to be useful for the

implementation of robot programs in which the non-existence of knowledge is by itself an important piece of information that can for example trigger actions for knowledge acquisition. With closed world semantics, representations are more compact since the non-existence of something does not have to be extensively described but can be concluded from the fact that its existence cannot be proven. For example, if the robot has to decide whether it can perform a task or if some component is missing, it can simply check whether all required components are known to be available and decline otherwise.

Due to the conceptual closeness of the logical representations in description logics and the Prolog language, the implementation of the inference predicates for reasoning about OWL classes, properties, restrictions and individuals is rather simple. Most predicates are realized as a rather shallow mapping from the OWL properties onto first-order expressions in Prolog describing e.g. in which cases one class is assumed to be a sub-class of another one. Based on the low-level predicates described in the following sections, KNOWROB can apply standard Prolog inference methods to find solutions to more complex problems.

2.4.1 Parsing and storing OWL triples

The static pieces of knowledge in KNOWROB are stored in OWL files using the RDF/XML syntax, a standardized format that is supported by many established tools and by import and export routines of several other knowledge bases. In order to load the file, it has to be parsed and represented in the knowledge base. The implementation in KNOWROB is based on the SWI Prolog Semantic Web Library [Wielemaker et al., 2003] for loading and storing RDF triples, and the Thea OWL parser library [Vassiliadis et al., 2009] that provides OWL reasoning on top of these representations. The general procedure for loading files is to first parse the RDF/XML file and then assert all triples to the Prolog database using the *rdf(?Subj, ?Pred, ?Obj)* predicate. Internally, the Semantic Web Library stores these triples in an efficient database implementation that combines several indexing schemes (by subject, predicate, object and combinations thereof; the predicate index also takes sub-properties into account). For this reason, the database scales very well up to large knowledge bases with more than 10 million triples[1]. We extended these libraries to include the results generated by computable classes and properties in the reasoning process.

[1] http://www.swi-prolog.org/pldoc/package/semweb.html

2.4.2 Individuals

The efficient internal database representation is very important to ensure scalability of the overall system, but does by itself only support queries for knowledge that is already available in exactly the same form. In terms of expressiveness, this representation does not go beyond the capabilities of a database. To enable the more advanced features of OWL, like hierarchies of classes and properties, transitivity of predicates and the computable properties for calculating relations on demand, this representation needs to be extended with predicates that compute complex relations by combining these single pieces of knowledge. KNOWROB supports a set of increasingly complex query predicates that add functionality with each layer. The advantage of having this stepwise increase in complexity is that one can select how much inference is to be included by choosing the appropriate predicate. Especially computables that forward queries to other components like the perception system can slow down the inference process, though it may often be sufficient to answer a query based on the already available knowledge.

The following hierarchy of predicates allows to query for individuals and their properties. Note that in OWL classes and properties themselves are also represented as *NamedIndividuals*, which means that the following predicates can also be used for inspecting descriptions of classes and properties.

- *rdf(?S, ?P, ?O)* returns only exactly matching triples from the internal triple database.
- *rdf_has(?S, ?P, ?O)* also takes the *subPropertyOf* relation into account, returning matches for all specializations of *?P*
- *rdf_reachable(?S, ?P, ?O)* further considers transitivity of the predicate *?P*
- *rdf_triple(?P, ?S, ?O)* additionally includes results generated by computable classes and properties
- *owl_has(?S, ?P, ?O)* is the most general query predicate, also returning results of the OWL inference process (e.g. inferred class membership of an individual)

2.4.3 Classes

While a class hierarchy can in principle be explored using the predicates listed in the previous section, there are specialized predicates like *owl_subclass_of(?Sub, ?Super)* that are more convenient to use. Depending on the combination of bound and unbound variables, this predicate reads all classes derived from *?Super* or computes super-classes of *?Sub*.

The classification of individuals links information in the ABOX and TBOX by determining if an individual is part of a certain class. In the simplest case, this relation has been asserted to

the knowledge base (*A is of type B*) and can simply be retrieved. However, classes in OWL can also be described implicitly by a set of properties all individuals of this class need to have. For example, one could define that all individuals must be of a certain type and must be related to a minimum number of other individuals of a specific type. These specifications are called "restrictions" – they restrict the set of potential class members to those having the required properties.

Figure 2.4 explains the concept of restrictions on the example of the class *CoffeeCup*. In the left part of the figure, the cup in the picture is asserted to belong to the class *CoffeeCup*. The system has no further knowledge about a *CoffeeCup* beyond the subclass relation, so the only way to determine if an object belongs to this class is to check if the user asserted it. Restrictions describe the properties all members of a class need to have and thus add knowledge that can be used to determine if a novel object belongs to the respective object class. In the right part of the figure, the class *CoffeeCup* is described using a restriction saying that a coffee cup is a *subClassOf Cup* that *contains Coffee*. Based on this knowledge about a *CoffeeCup*, the system can now automatically assign all detected cups that contain coffee to this class.

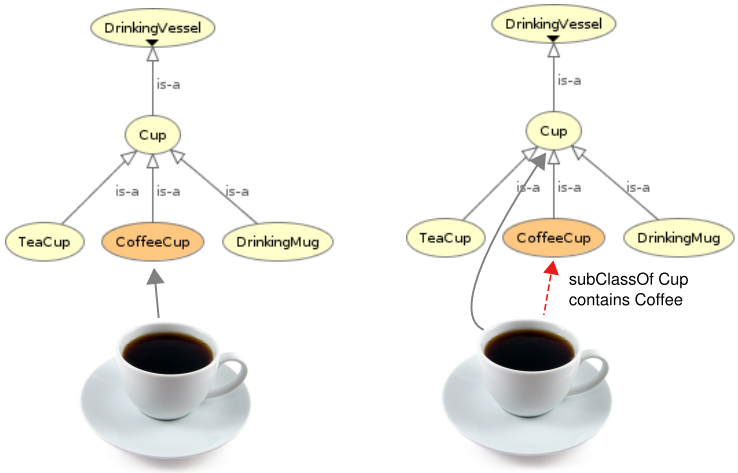


Figure 2.4 Example of an OWL restriction. Left: The cup is asserted to be a *CoffeeCup*. Right: If the properties of *CoffeeCups* are described by restrictions, the system can infer the class membership based on the cup's properties.

There are different kinds of restrictions to describe the properties of the class members in more detail: Existential restrictions, described using the keywords *someValuesFrom*, specify that each member needs to have at least one relation of the respective kind. For example, the restriction $\exists hasChild.Lawyer$ specifies that each member of the class needs to have at least one child of type $Lawyer$. In contrast, universal restrictions like $\forall hasChild.Doctor$ specify that all individuals related to any member of the class by the $hasChild$ relation have to be of type $Doctor$. They

however do not imply that there are any such individuals. Universal restrictions are described in OWL using the *allValuesFrom* keyword. While existential and universal restrictions describe that class members need to be related to an arbitrary individual matching the specification, value restrictions, specified by *hasValue*, require that each member is related to a specific individual. Furthermore, one can define cardinality restrictions like $(\leqslant 2\, hasChild)$ or $(\geqslant 1\, hasChild)$ to describe that each member of the class needs to have at most two or at least one child.

If a class is described by a set of restrictions, the system can infer that an instance belongs to this class if it fulfills all of them. This can be queried using the *owl_individual_of(?Ind, ?Class)* predicate that either returns instances *?Ind* of a class *?Class* or returns all classes that can be inferred for an individual based on its properties and the class restrictions. The expression *owl_individual_of(?Restr, owl:'Restriction'), owl_subclass_of(?Class, ?Restr).* reads all restrictions that are defined for *?Class*.

2.4.4 Properties

Similar to the class hierarchy, there is also a taxonomy of sub-properties and super-properties. The advantage of this hierarchical structure is that knowledge can be described as concise as possible, using very specific properties, while queries can use more generic relations and will return all relations asserted for any of the sub-properties. The property hierarchy is exploited by all of the above query predicates except the very basic *rdf(...)* predicate. To inspect the property hierarchy, one can use the *rdfs_subproperty_of(?SubProp, ?Prop)* predicate that lists all sub-properties of *?Prop*.

2.5 Probabilistic inference

Not all knowledge a robot encounters can reasonably be described in a deterministic knowledge base. Many facts are either not certainly known, for instance due to uncertain sensor information, or only true to a certain degree, like fuzzy human preferences. Statistical models have become an indispensable tool in robotics for representing such uncertain information [Thrun et al., 2005], for example to solve the localization and mapping problem by representing the probability distribution over the robot's location using particle filters. While these models are well-suited to describe probabilistic information about concrete entities, the representation of generic principles and knowledge about types of or relations between objects requires the greater expressiveness of first-order representations. Statistical relational models [Getoor and Taskar, 2007] combine the expressiveness of first-order logical representations with the ability of statistical models to represent uncertain information. The extension towards relational models is similar to the transition from propositional logical descriptions to first-order logics: These models are able to abstract away from concrete entities in the domain and to represent relational knowledge that can also be applied in different circumstances. In order to efficiently perform inference, probabilistic models need to compactly represent the probability distributions; a problem for which graphical models are an established solution. Therefore, many statistical relational models are realized as extensions of undirected (Markov Logic Networks [Richardson and Domingos, 2006]) or directed graphical models (Multi-Entity Bayesian Networks [Laskey, 2008], Bayesian Logic networks [Jain et al., 2009]) and can therefore make use of inference methods that have been developed for these underlying representations.

On the one hand, the combination of high expressiveness with the ability to represent uncertainty makes statistical relational models well-suited for the uncertain, partially observable and dynamic environments autonomous robots are acting in. On the other hand, inference in such models becomes very hard, often even unfeasible, especially when the models are large and cover many relations between a lot of instances. Completely representing every piece of information in terms of statistical relational models is therefore usually unfeasible. In KNOWROB, these models are thus only used for those kinds of information that require this combination of expressiveness and uncertainty, which are often related to human behavior. In the remaining cases, we resort to approximating relations with deterministic models or use classical (non-relational) statistical models to translate uncertain, continuous data into the most likely symbol in the knowledge base. An interesting aspect of the models is that they can derive logical knowledge from observations, which is for example being used to describe which utensils humans use and which food they consume in a meal context and to bring the right items to the table (Section 7.6).

2.5.1 Bayesian Logic Networks

In KNOWROB, Bayesian Logic Networks (BLNs) [Jain et al., 2009] are used to represent uncertain relational knowledge. BLNs are expressive enough to describe the complex interactions between actions, parameters of these actions, and their relative ordering, can at the same time handle the inherent uncertainty, and still have reasonable learning and inference complexity. We will only briefly describe BLNs and refer to [Jain et al., 2009] for details.

A BLN is formally described as a tuple $\mathcal{B} = (\mathcal{D}, \mathcal{F}, \mathcal{L})$ consisting of the declarations of types and functions $\mathcal{D}$, a set of fragments of conditional probability distributions $\mathcal{F}$, and a set of hard logical constraints $\mathcal{L}$ as formulas in first-order logic. The fragments $\mathcal{F}$ describe dependencies of abstract random variables. An example of such a fragment is shown in the left part of Figure 2.5, in which the oval nodes denote random variables and the rectangular nodes contain preconditions for the respective fragments to be applicable. Each random variable node contains a table of the conditional probabilities of its value given the values of the surrounding nodes.

For a given set of entities, the BLN gets instantiated to a ground mixed network to which Bayesian network inference techniques can be applied. The abstract description of a BLN can thus be thought of as a template for the construction of the ground mixed network including probabilistic and deterministic dependencies. Such a ground network can become reasonably large as can be seen in Figure 2.5 (right), in which hundreds of nodes are arranged in a circular shape. Practically, the declarations $\mathcal{D}$, the fragments $\mathcal{F}$ and logical constraints $\mathcal{L}$ are defined manually, while the probabilities are learned from data. For learning BLNs, the conditional probability tables in the fragments in $\mathcal{F}$ need to be determined, which reduces to simply counting the relative frequencies of the relations.

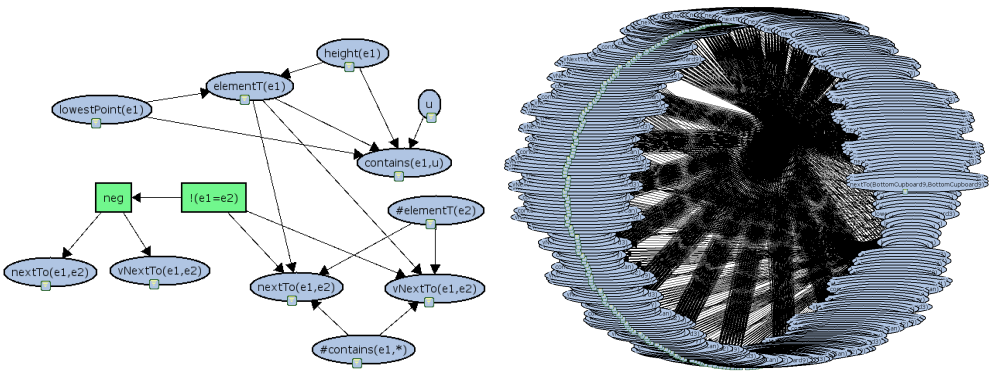


Figure 2.5 Example fragment of a Bayesian Logic Network (left) and the very large ground Bayesian network consisting of hundreds of nodes that is generated from this template given an actual domain (Pictures courtesy of Dominik Jain).

2.5.2 Integration with KNOWROB

The implementation of the statistical relational learning methods was realized by interfacing the PROBCOG system by Dominik Jain with the KNOWROB knowledge base. PROBCOG provides a variety of learning and inference algorithms for both Markov Logic Networks and Bayesian Logic Networks, all integrated in a common framework.

The integration of the PROBCOG inference engine with KNOWROB is realized in a bi-directional way: On the one hand, there are methods for sending queries to the PROBCOG inference engine from within KNOWROB and to retrieve the results. On the other hand, the PROBCOG engine can read evidence that is needed for performing the inference from KNOWROB. Reading this information from the KNOWROB knowledge base gives the probabilistic inference methods access to all the knowledge that is available in the system or which can be derived using any of KNOWROB's inference methods, including logical inference or computables. Since all evidence is read from the same original representation, inconsistencies in the knowledge are avoided.

Figure 2.6 describes how the integration is realized technically. The statistical relational models in PROBCOG form a kind of statistical knowledge base that uses a different set of predicates with different semantics than the KNOWROB system. A generic interface module (depicted on the left side) provides predicates to send queries to the PROBCOG inference engine. In addition, this module has to decide which PROBCOG model to load for an inference task: There are different models specialized for different use cases, describing for instance the table setting context or the partial order in human activities. Predicates can be part of multiple models in which they play different roles: The *objectActedOn*, for example, may be the main outcome of a model, in which case that model is the one to be selected, but can also be a peripheral piece

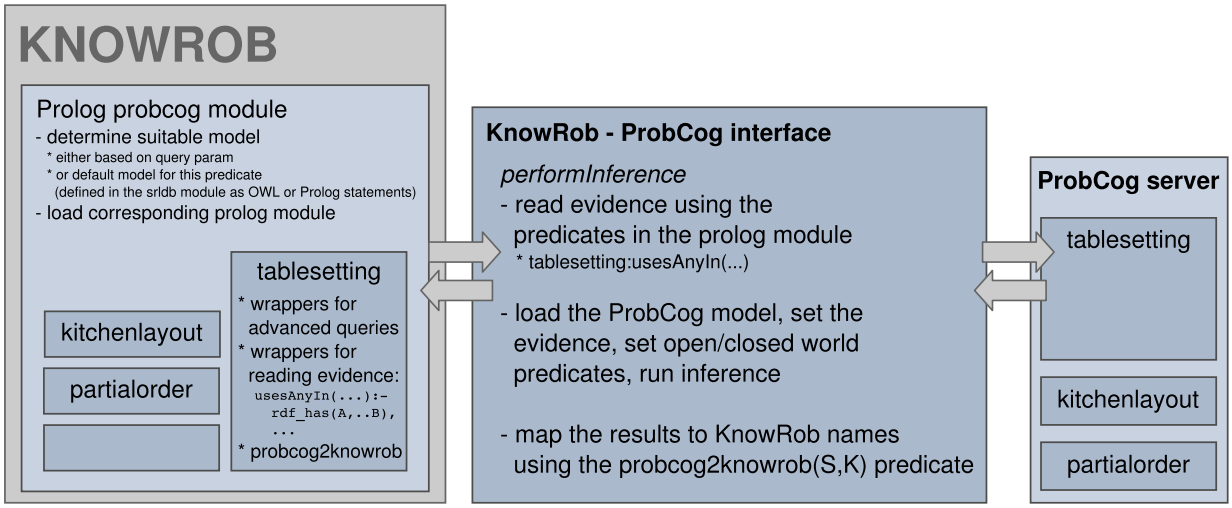


Figure 2.6 Integration of the PROBCOG inference system into the KNOWROB knowledge base.

of information that is rather considered as input. The decision for a model is taken using the *probcog_model(?QueryPredicates, ?Model)* predicate that selects the appropriate model for a given set of query predicates. The implementation of this predicate can be as complex as necessary, also taking the current context into account, but can also simply select a default model.

Once the right model is determined, a Prolog wrapper module is loaded that contains all model-specific interfaces (in Figure 2.6 exemplarily described for the models *kitchenlayout*, *partialorder* and *tablesetting*). Each of these modules contains four main components: Mappings between the predicates in PROBCOG and KNOWROB, mappings between identifiers in PROBCOG and KNOWROB, meta-query predicates for advanced queries, and methods to set the open/closed world property on a per-predicate basis.

The mapping between the predicates in both knowledge bases needs to translate both the predicate names and their semantics. This is realized by one wrapper module per PROBCOG model that implements the PROBCOG predicates using the KNOWROB methods. These implementations map the respective predicate semantics and allow PROBCOG to read evidence values from KNOWROB. The mapping of evidence values is performed by the *probcog2knowrob()* predicate. Each module can further provide convenience predicates that combine several queries and compute more advanced concepts. One example is the post-processing of the PROBCOG response to select the answer with highest probability, to combine the inference results with other information from KNOWROB, or to combine the results of different PROBCOG queries

2.6 Computable classes and properties

Computables are a kind of procedural attachment to OWL classes or properties. They effectively extend the reasoning capabilities beyond pure description logics inference, e.g. to compute more complex relations or to load information from external sources. There are two kinds of computables: Computable classes, which create instances of their target class, and computable properties, which compute relations between instances.

2.6.1 Realization

Figure 2.7 describes how computables are attached to OWL relations: The OWL relation *objectActedOn* is modeled as usual including specifications of its domain and range (here: *ActionOnObject* and *SpatialThing*). An instance *computeObjActOn* of a *ComputableProperty* that has this property as its *target* specifies a binary Prolog predicate that can be used to compute this relation, in this case *objectActedOn*.

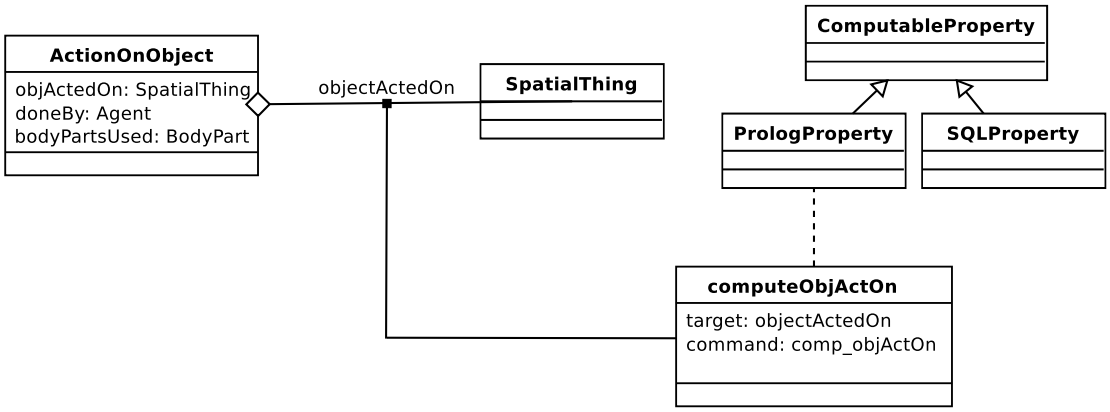


Figure 2.7 The computable property *computeObjActOn* is attached to the OWL property *objectActedOn* and describes procedurally how this relation can be computed.

Since computables can trigger reasonably complex calculations, one does not always want to include them in the reasoning process. Therefore, there is a special predicate *rdf_triple(?P,?S,?O)* that returns the union of results from the asserted knowledge (via *owl_has(?S,?P,?O)*) and those obtained via computables (*rdfs_computable_triple(?S,?P,?O)*), and further provides a hook for custom predicates. Results are calculated for all sub-properties of the property that is being queried.

```
rdf_triple(?Prop, ?Subj, ?Obj) :-
  subproperty_of(?SubProp, ?Prop),
  ( owl_has(?Subj, ?SubProp, ?Obj) ;
    rdfs_computable_triple(?Subj, ?SubProp, ?Obj) ;
    rdf_triple_hook(?Subj, ?SubProp, ?Obj) ).
```

The predicate *rdfs_computable_triple* combines the results of all computables of all kinds. In addition to the Prolog computables described here, there are also SQL computables that calculate results based on SQL database queries instead of calling a Prolog predicate.

```
rdfs_computable_triple(?Subj, ?Prop, ?Obj) :-
  rdfs_computable_prolog_triple(?Subj, ?Prop, ?Obj) ;
  rdfs_computable_sql_triple(?Subj, ?Prop, ?Obj).
```

The attachment of computables via the *target* property has several important advantages: First, it separates the semantic modeling (the OWL relations to be computed) from implementation aspects. Second, it allows to easily load and unload computables for a relation, and to attach multiple computables to the same semantic relation. This helps to ensure modularity of the whole system.

2.6.2 Computable properties

There are two ways how computable properties can be evaluated, either by calling a Prolog predicate that computes the relation, or by sending an SQL query to a database.

2.6.2.1 Prolog properties

Let us consider the *after* relation between two points in time as an example which is defined as:

```
ObjectProperty: after

    SubPropertyOf: temporallyRelated
    Domain: TimePoint
    Range:  TimePoint
```

If this relation is to be computed, we can attach a computable property and specify that the relation can be computed by the *comp_after(..)* predicate:

```
Individual: computeAfter

    Types: PrologProperty

    Facts: target  after
           command comp_after
```

If the *after* relation is queried using the *rdf_triple(..)* predicate, the system thus calls the *comp_after(..)* predicate with the respective values for the subject and object. The implementation of the predicate should be able to handle the different combinations of bound/unbound variables (read all objects for a subject, all subjects for a given object, or all valid combinations of subjects and objects related by the *after* relation). The implementation of the *comp_after(..)* predicate is given below. This predicate first checks if the subject and property have the correct types, then transforms the time points into numerical values using *term_to_atom(..)*, and finally compares them to check if *?Pre* is really earlier than *?After*.

```
comp_after(?Pre, ?After) :-
  owl_has(?Pre,    type, 'TimePoint'),
  owl_has(?After,  type, 'TimePoint'),
  term_to_atom(?P, ?Pre),
  term_to_atom(?A, ?After),
  ?P<?A.
```

Prolog computables can also be used for more advanced tasks than just comparing numerical values: The import of instructions from the Web (Chapter 4) or interfaces to perception systems (Section 2.6.4) can transparently be integrated. Computables can also be used to integrate rule knowledge – the computable implementation becomes the body of the rule, the *target* forms the head of the rule.

2.6.2.2 SQL properties

The definition of a computable SQL property consists of the target property, some authentication information to access the database, and most importantly three SQL queries. Depending on which variables in the query are bound (subject, object, or none of them), the system automatically chooses one of them: If the subject is bound and the object unbound, it uses *valueSelect*, if the object is bound and the subject unbound, it uses the *frameSelect* query, and if both are unbound, it uses the *frameValueSelect* query.

```
Individual: computeX

    Types: SqlProperty

    Facts: target           xCoord
           valueSelect      select robX from positions where time = ~frame~
           frameSelect      select time from positions where robX = ~value~
           frameValueSelect select time, robX from positions
           user             db_user
           password         db_pwd
           database         db_db
```

2.6.3 Computable classes

The definition of computable classes is very similar to the specification of computable properties. Again, there are two kinds of computables evaluated by a Prolog predicate or an SQL query.

2.6.3.1 Prolog classes

The *target* property describes the class for which instances can be computed, the *command* speficies the predicate that is to be called. Again, different combinations of bound/unbound parameters are possible in order to generate instances of a class or determine the class of a given instance.

```
Individual: computeObjectsOnTable

    Types: PrologClass

    Facts: target   HumanScaleObject
           command  comp_objectsOnTable
```

2.6.3.2 SQL classes

An important use of computable SQL classes is to read human motions from a database, for example all poses labeled as "Reaching". Here, we need to give one command for creating instances (command) and one for checking if an instance belongs to a class (testCommand).

```
Individual: computeReaching

    Types: SqlClass

    Facts: target       Reaching
           command      select id from tab_arm where type=Reaching
           testCommand  select type from tab_arm where id=~instance~
           user         db_user
           password     db_pwd
           database     db_db
```

2.6.4 Computables for interfacing external data sources

Computables can also be used to load data from external sources of information. For example, the import of instructions from the Web (Chapter 4) is integrated by computables that return a plan for a command given in natural language. From a user point of view, the retrieval of instructions and the translation into a logical plan representation appears to be a simple query to the knowledge base, and only the slightly longer response time indicates if the plan already existed in the knowledge base or had to be generated first.

Another important use of computables is the interface to perception components. The internal representation of objects is introduced in Section 3.2.3. In this context, computables are used to build up these data structures based on information from the perception. To generate object instances of objects from perception, we define a computable class for a reasonably generic target class, e.g. the *HumanScaleObject*, which is a class containing practically all objects the robot encounters in its environment.

```
Individual: computeTabletopObject

    Types: PrologClass

    Facts: target HumanScaleObject
           command tabletop_object
```

With this definition, the system now checks for new object detections whenever a query asks for instances of *HumanScaleObject*. The generated object instances thereby have the more specific types that have been returned by the object recognition system. The implementation of the Prolog computable is sketched below: First the predicate calls the perception service to get the list of detected objects. For each object, it determines the type and pose and builds up the internal object representation shown in Figure 3.5.

```
tabletop_object(?Obj, ?Type) :-

    % call perception service to get object detections
    perception_client(?ObjList),
    member(?Match, ?ObjList),

    % split into object ID and pose
    ?Match = [?ID, ?Pose],

    % create object instance
    id_to_type(?ID, ?Type),
    create_object_instance([?Type], ?ID, ?Obj),

    % create perception instance
    create_perception_instance(?Perception),

    % set pose
    set_perception_pose(?Perception, ?Pose),

    % link object and perception
    set_object_perception(?Obj, ?Perception).
```

Using computables to generate object instances works best if the perception component detects objects on demand. For passive perception modules, which continuously detect objects in e.g. video images, a different interface that listens to these detections in a parallel thread and creates the object representation for each detected object. These interfaces are further described in Section 6.1.

2.7 Discussion

In this chapter, we introduced KNOWROB as a practical knowledge processing system for autonomous robots. Its system design follows the "world as a virtual knowledge base" paradigm we introduced. Instead of adding all available information to the knowledge base and pre-processing it in way that all possible queries are covered, we rather use on-demand computation of the required information. If the knowledge base needs to answer a query about something that is not yet known, it can forward the query to "the world", for example in terms of perception tasks, to get the desired information. The on-demand computation of information can also be used to compute a different, often more abstract "view" on information that is already present in the knowledge base, for example to compute qualitative spatial relations based on absolute object positions. It proved to be an extremely useful tool in robotics, where most information is already present in some form in the robot control program.

We explain the logical formalism, description logics, that is used as the basis for representing knowledge in the system. Description logics allow to represent knowledge in a very structured way and are a good compromise between sufficient expressiveness and still good support for automated reasoning. The knowledge about classes of objects and actions is represented in form of an ontology, a taxonomy of classes that are inter-related by properties. The layout of the upper ontology was described in Section 2.2 and explained how the information about actions, events, objects, spatial and temporal information is organized in the ontology.

The overall system architecture and the main components for knowledge acquisition, for automated reasoning, for visualization and for querying for information that are integrated in the KNOWROB system are explained in Section 2.3, as well as the different sources of knowledge that can be used.

There are three main inference techniques in KNOWROB which are each suited for different kinds of knowledge: Description logics inference forms the backbone of the overall system and is used for all deterministic information, including the large class ontologies. For reasoning about probabilistic information, we integrated the PROBCOG toolkit that provides various inference methods for reasoning in statistical relational models which combine the expressiveness of first-order logics with the ability to represent uncertainty. The "computables" described in the last section of this chapter allow to perform a procedural computation of semantic information on demand during the inference procedure. They are the main tool for loading information from the perception system and for computing relations between entities based on the information in the knowledge base.

Chapter 3

Knowledge representation for robots

A knowledge processing systems is to equip a robot with all knowledge it needs to take informed control decisions, to parametrize its actions in the right way, and to understand incomplete and ambiguous human instructions. This requires formal descriptions of many different aspects of the actions the robot is to perform, the objects it interacts with, the environment it operates in, and the components and capabilities of the robots itself. All these aspects need to be described in a common formal language that allows the robot to understand their meaning, to combine them and to draw conclusions.

In the pancake scenario, for example, the robot needs to infer which pieces of information are missing in the instructions, i.e. to first notice that something is missing at all, and then to decide how to fill in this missing information and where to obtain it from. Drawing these inferences requires the robot to have detailed descriptions of the properties of actions and objects, to have methods for predicting the effects of actions, and to reason about its own capabilities to determine if they suffice to perform the task at hand.

The problem when developing the representation language is how to abstract away from the complexity of the world, how to appropriately structure the information and how to develop compact, abstract representations that cover all important aspects while still allowing efficient inference. While the area of knowledge representation has a long tradition in artificial intelligence research, the application of these methods to robotics still creates its very own challenges. Some of them are caused by the large number and the complexity of the topics that need to be described, which requires expressive representations that integrate time and space, discrete and continuous information, and goal-directed actions as well as physical processes. Other challenges are the need for representing change and to account for the dynamics in the world caused by actions and processes, with a focus on spatio-temporal reasoning. For pick-and-place tasks, a representation of the changes in object positions over time is sufficient, but more sophisti-

cated manipulation actions, as they occur for example in cooking tasks, require more expressive descriptions of how objects are changed, created or destroyed by actions. These explicit representations of actions and their effects should further support planning, projection and diagnosis. Moreover, the representations should be able to deal with inconsistencies which can hardly be avoided if information is acquired from inaccurate sensors. Therefore, they should be handled in a way that they do not render the whole knowledge base unusable. Often, inconsistent beliefs about object positions are resolved once the robot detects the respective objects again.

While the previous chapter introduced the basic concepts for the technical implementation, we now describe the language that has been developed to represent the complex information required by robotic systems and the inferences that can be drawn from it. The first section deals with temporal information, the second one with spatial information about objects, their poses, and spatial relations between them. Actions and processes are the topic of the following sections. We conclude with a description of robot self-models and methods that use them to decide if the robot has all required capabilities to perform an action.

3.1 Events and temporal information

Robots are acting in dynamic environments: Actions are performed, objects change their positions and properties over time. Therefore, the robot's knowledge representation should be capable of describing and reasoning about temporal information like the start time of an event, the duration of an action, or relations like contemporaneity.

The main specializations of a *TemporalThing* are *Situation*, *Event*, *TimeSpan* and *TimePoint*. An overview of the upper ontology of temporal things is shown in Figure 3.1. We use the term *Event* as defined in OpenCyc:

"Each instance of Event is a dynamic situation in which the state of the world changes; each instance is something one would say 'happens'. [...] Events should not be confused with TimeIntervals. The temporal bounds of events are delineated by time intervals, but in contrast to many events time intervals have no spatial location or extent." [1]

Events can be instantaneous (like the moment when a perception is made) or temporally extended (like the execution of a trajectory for reaching towards an object). Note that actions are described as specializations of events, namely those events that are caused by an agent acting in the world. This allows to describe actions using the same vocabulary as events like the *startTime* or temporal relations like *after*. Other important sub-classes are *MentalEvents*, events that (at

[1] Definition from OpenCyc http://sw.opencyc.org/concept/Mx4rvViADZwpEbGdrcN5Y29ycA

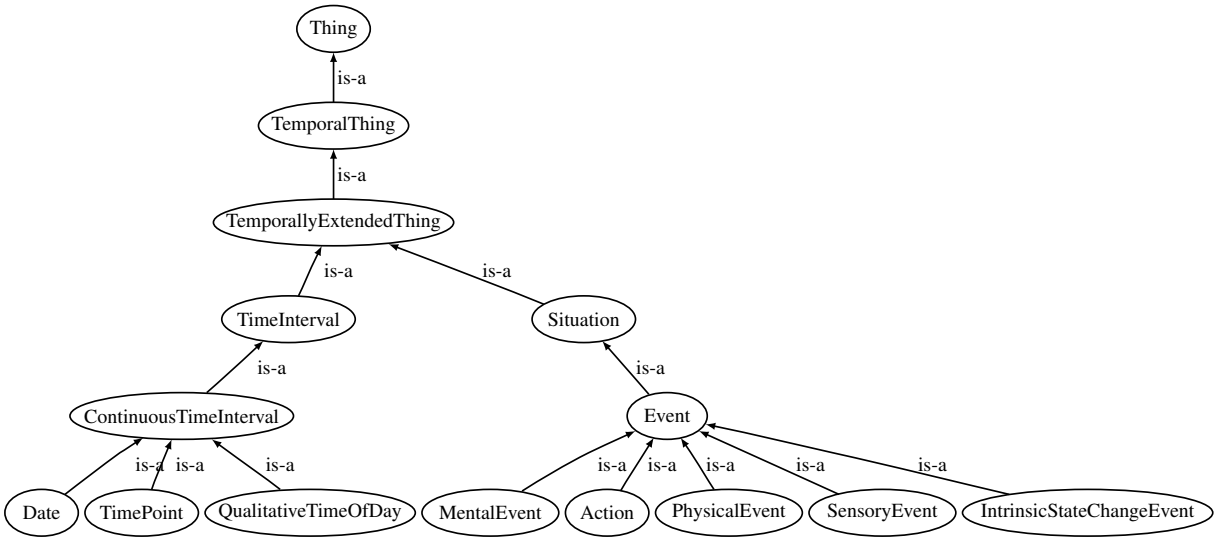


Figure 3.1 Excerpt of the ontology of temporal things, describing time points, time intervals, and events.

least partially) happen in a person's (or robot's) mind like a reasoning process or a decision that is being made. In KNOWROB, this class of events is used to describe how the robot obtained a certain belief, for example the belief that an object is (or was, or should be, or could be) at a certain location. They are described in more detail in Section 3.2.5.1. If the belief was the result of some kind of perception, the event is at the same time a *SensoryEvent*.

Objects can change over time: they can be created, destroyed, combined into new objects, and can change their intrinsic state (like the temperature or aggregate state). The events causing such changes are described as specializations of *PhysicalEvent* and *IntrinsicStateChangeEvent*. In Section 3.4, we will discuss how KNOWROB uses these event descriptions to reason about the consequences of actions and processes.

Each event has a *startTime* and, if its duration is finite, also an *endTime*. Both relations link an event to a *TimePoint*. A *TimeInterval* is the time between two *TimePoints*. Instances of time points contain the time as UNIX timestamps (i.e. the seconds since 1.1.1970). For performance reasons, time points are internally stored as a concatenation of the prefix 'timepoint_' and the numeric timestamp value.

This event representation allows temporal reasoning using computable properties operating on the start- and end times. They can compute the *duration* of a *TemporallyExtendedThing*, and can compute qualitative temporal relations. Already two relations suffice for describing most possible relations between two time intervals: *after(TimePoint, TimePoint)* and *temporallySub-*

$X{<}Y$ $Y{>}X$	after before	$\exists Ex, Sy.endTime(X, Ex) \wedge startTime(Y, Sy) \wedge (Ex < Sy)$
$X\mathrm{m}Y$ $Y\mathrm{mi}X$	meets meetsInv	$\exists T.\ endTime(X, T) \wedge startTime(Y, T)$
$X\mathrm{o}Y$ $Y\mathrm{oi}X$	overlaps overlapsInv	$\exists Sx, Sy, Ex, Ey.\ startTime(X, Sx) \wedge startTime(Y, Sy) \wedge endTime(X, Ex)$ $\wedge\, endTime(Y, Ey) \wedge (Sx < Sy) \wedge (Sy < Ex) \wedge (Ex < Ey)$
$X\mathrm{s}Y$ $Y\mathrm{si}X$	starts startsInv	$\exists T, Ex, Ey.\ startTime(X, T) \wedge startTime(Y, T) \wedge endTime(X, Ex)$ $\wedge\, endTime(Y, Ey) \wedge (Ex < Ey)$
$X\mathrm{d}Y$ $Y\mathrm{di}X$	during duringInv	$\exists Sx, Sy.\ startTime(X, Sx) \wedge startTime(Y, Sy) \wedge endTime(X, Ex)$ $\wedge\, endTime(Y, Ey) \wedge (Sx < Sy) \wedge (Ey < Ex)$
$X\mathrm{f}Y$ $Y\mathrm{fi}X$	finishes finishesInv	$\exists T, Sx, Sy.\ endTime(X, T) \wedge endTime(Y, T) \wedge startTime(X, Sx)$ $\wedge\, startTime(Y, Sy) \wedge (Sy < Sx)$
$X{=}Y$	equal	$\exists S, E.\ startTime(X, S) \wedge startTime(Y, S) \wedge endTime(X, E) \wedge endTime(Y, E)$

Table 3.1 Allen's interval algebra [Allen, 1983] and its implementation in our system.

sumes(TemporalThing, TemporalThing). Note that both predicates can also compute their inverse ('*before*' and '*temporallyContained*') if the arguments are swapped.

By combining these two relations with the *startTime* and *endTime* properties, all of Allen's 13 temporal relations [Allen, 1983] between time intervals can be computed. All of them can be defined using only the start time and end time of the two time intervals (Table 3.1) and can easily be evaluated using computables. This on-demand computation allows to easily perform reasoning about observations made over time.

3.2 Objects, stuff and the environment

Acting in human environments means interacting with objects of many different kinds. In order to competently perform everyday tasks, robots have to represent knowledge about object classes and their properties as well as information about concrete object instances which have been detected in the environment. Multiple object instances can form a semantic environment map which, in combination with maps the robot can use for self-localization, allows the robot to locate objects in the environment. Robots further need to describe the relations between object classes and models that allow them to recognize objects of this kind.

We are distinguishing objects from stuff-like things. Stuff, in contrast to objects, is something that can be divided into parts over and over again while still maintaining its type. Positive examples are water, flour, or sand, negative ones, which are object-like rather than stuff-like, are cars, apples, or knives.

3.2.1 Classes of objects and stuff-like things

All classes are organized in a taxonomic structure, from very general classes like *SpatialThing* to specific ones like *Refrigerator-Freezer*. Figure 3.2 gives an overview over the most important classes describing objects and stuff in KNOWROB. Like in other parts of the ontology, we use multiple inheritance to reflect the different aspects of an object: A *MicrowaveOven*, for example, is a *FoodOrDrinkPreparationDevice* as well as a kind of *Oven* and an *ElectricalHouseholdAppliance*. It therefore inherits all properties that are specified for any of these super-classes. This multi-faceted modeling is very important to capture the complexity of real-world environments.

The classes are further described by properties, e.g. that the primary function of an *Oven* is *HeatingFood*, and that it has a *Handle* as *properPhysicalPart*. As described in Section 2.4, these restrictions can also be used to classify objects: If something has all properties required for a certain kind of object, it can automatically be classified as being an instance of the respective object class. The taxonomy has the advantage that knowledge can be represented at different levels of granularity: For instance, the fact that a *Container* can *contain SpatialThing*s can be described on this abstract level and gets inherited by all specialized classes.

Figure 3.2 Ontology of objects and stuff-like things.

3.2.1.1 Relation between stuff and objects

Things that can be split in pieces while maintaining their type are commonly described as being *stuff-like*, rather than *object-like*. Examples are water, flour, dough, or sugar. A portion of sugar can be split into two parts, still being a portion of sugar. An apple or a cup, in contrast, are object-like in that splitting them into pieces results in qualitatively different objects.

While having many things in common, stuff needs to be handled differently than objects, especially with respect to actions. First, the amount of stuff needs to be specified, in terms of weight, volume, etc. Then, we often have to translate between the stuff itself and the container it is stored in. In many cases, these two are used interchangeably: If milk is required as part of a recipe, for example, the plan may refer to the milk, while the pick-up action has to be performed on the bottle containing the milk.

Pieces of stuff are modeled as instances of the respective class, e.g. as an instance of *Water*, with the amount specified using the properties *volume*, *weight*, etc. Some default rules allow to convert between stuff and the containing entities. To search and locate stuff, a recognition model for a container that contains this kind of stuff can be used instead of a model for the stuff itself. Actions like picking up or putting down something stuff-like are performed on the container containing the stuff instead of the stuff itself.

3.2.1.2 Object recognition models

In order to recognize an object, a robot typically needs a detailed model that describes the object's shape, texture, appearance, or salient feature points. Such a model can be used by an object recognition system to detect, recognize and localize the object. Recognition models usually provide recognition services for objects of a certain type, they are thus linked to the respective object classes. Recognition models can be of various different kinds, so they need to specify which recognition system can load and use them. Since the recognition models can be quite large and complex, and since the knowledge base does not need to perform reasoning on the details described inside the model (for instance, some image feature descriptors that do not serve any other purpose than recognition), we do not represent the content of the model itself in OWL, but store only meta-data describing its properties and link it to the classes of objects it can recognize. The model itself is still stored in the object recognition system's binary format. Figure 3.3 exemplarily shows the description of a model that refers to a description of the object *IkeaExpeditShelf2x2*.

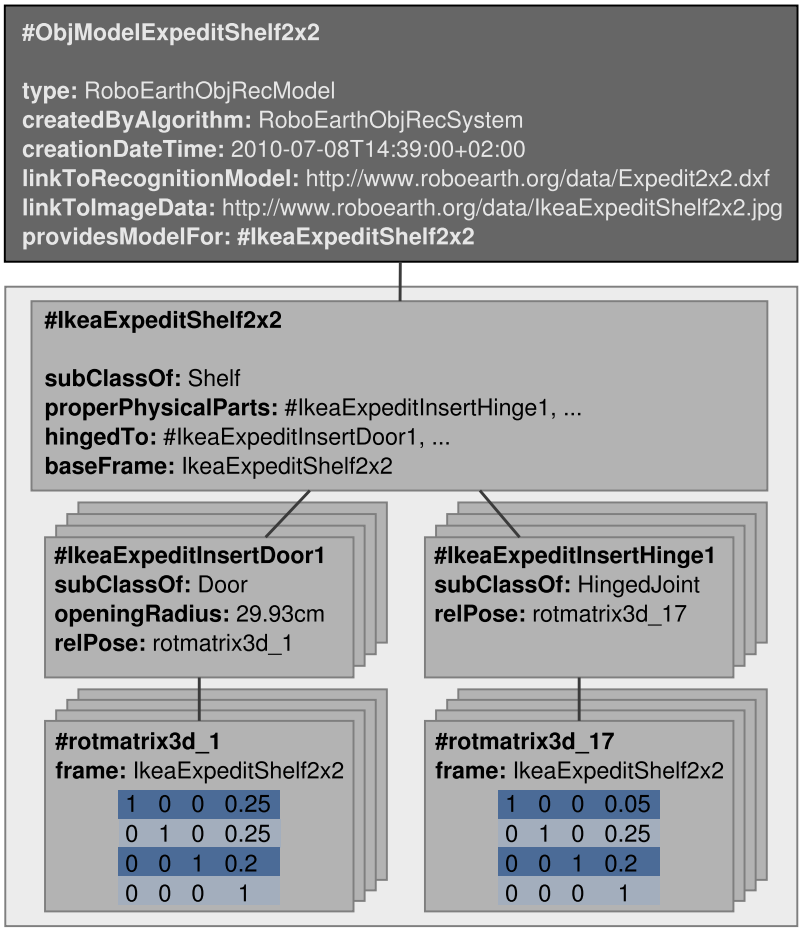


Figure 3.3 Description of an object recognition model (upper block) and the detection of an object using that model (lower blocks).

3.2.2 Object instances

Object instances in KNOWROB are interpreted as designators [McDermott, 1992], symbolic descriptions of objects in the real world. It is important to draw this distinction between the physical object itself and its representation in the knowledge base. An object instance is only the internal representation describing the object, but is not interpreted as denoting the object itself.

Distinguishing between the physical object and a description of it becomes especially important when describing processes in which objects are created or destroyed. The designator is usually created at the time of the first detection of an object, though the object itself may have been created earlier. Also, when objects are destroyed, one still wants to keep the designator describing that the object had existed and ceased to exist at some point in time in order to describe what happened.

We therefore have two kinds of creation and destruction times: The time of the creation and destruction of the object itself, and the time at which the internal representation was created (usually after the first perception of the object) or destroyed.

3.2.3 Positions, orientations and dimensions

Information about the poses and dimensions of objects is crucial for finding and manipulating them. In KNOWROB, object dimensions are described as simple bounding boxes or cylinders (specifying the height, and either width and depth or the radius). While this is clearly not sufficient for grasping, we chose this description as a compromise in order not to put too many details like point clouds or meshes into the knowledge base. Such information is rather linked and stored in specialized file formats.

Object poses are described via homography matrices. Per default, the system assumes all poses to be in the same global coordinate system. Pose matrices can, however, be qualified with a coordinate frame identifier. The robot can then transform these local poses into the global coordinate system, for example using the *tf* library[2].

Since robots act in dynamic environments, they need to be able to represent both the current world state and past beliefs. A naive approach for describing the pose of an object would be to add a property *location* that links the object instance to a point in space or, more general, a homography pose matrix. However, this approach is limited to describing the current state of the world – one can express neither changes in the object locations over time nor differences between the perceived and an intended world state. This is a strong limitation: Robots would not be able to describe past and (predicted) future states, nor could they reason about the effects of actions. Memory, prediction, and planning, however, are central components of intelligent systems.

The reason why the naive approach does not support such *qualified statements* is the limitation of OWL to binary relations that link exactly two entities. These relations can only express if something is related or not, but cannot *qualify* these statements by saying that a relation held an hour ago, or is supposed to hold with a certain probability. For this purpose, we need an additional instance in between that links e.g. the object, the location, the time, and the probability.

In KNOWROB, these elements are linked by the event that created the respective belief: the perception of an object, an inference process, or the prediction of future states based on projection or simulation. The relation is thus *reified*, that is, transformed into a first-class object. These reified perceptions or inference results are described as instances of subclasses of *MentalEvent*

[2]http://www.ros.org/wiki/tf

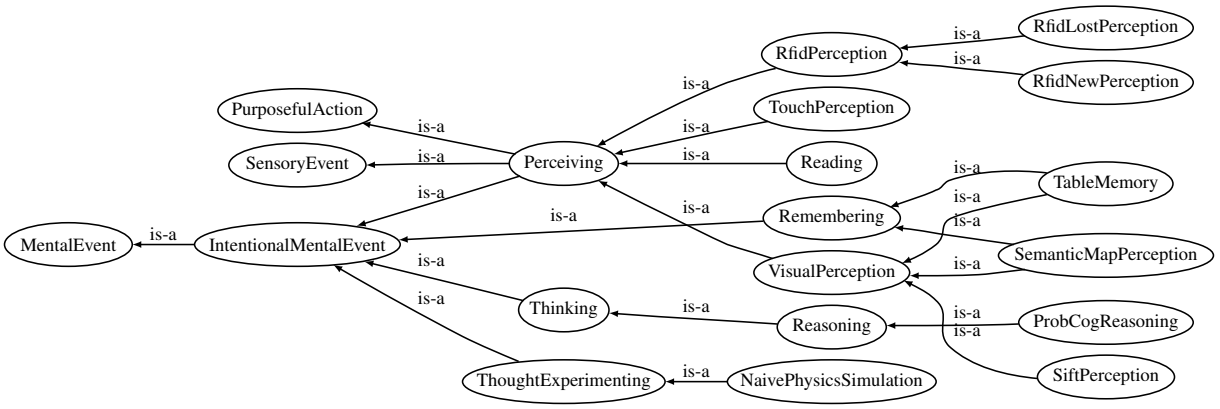


Figure 3.4 Ontology of mental events. Each of these events can result in changes in the robot's belief state about object poses.

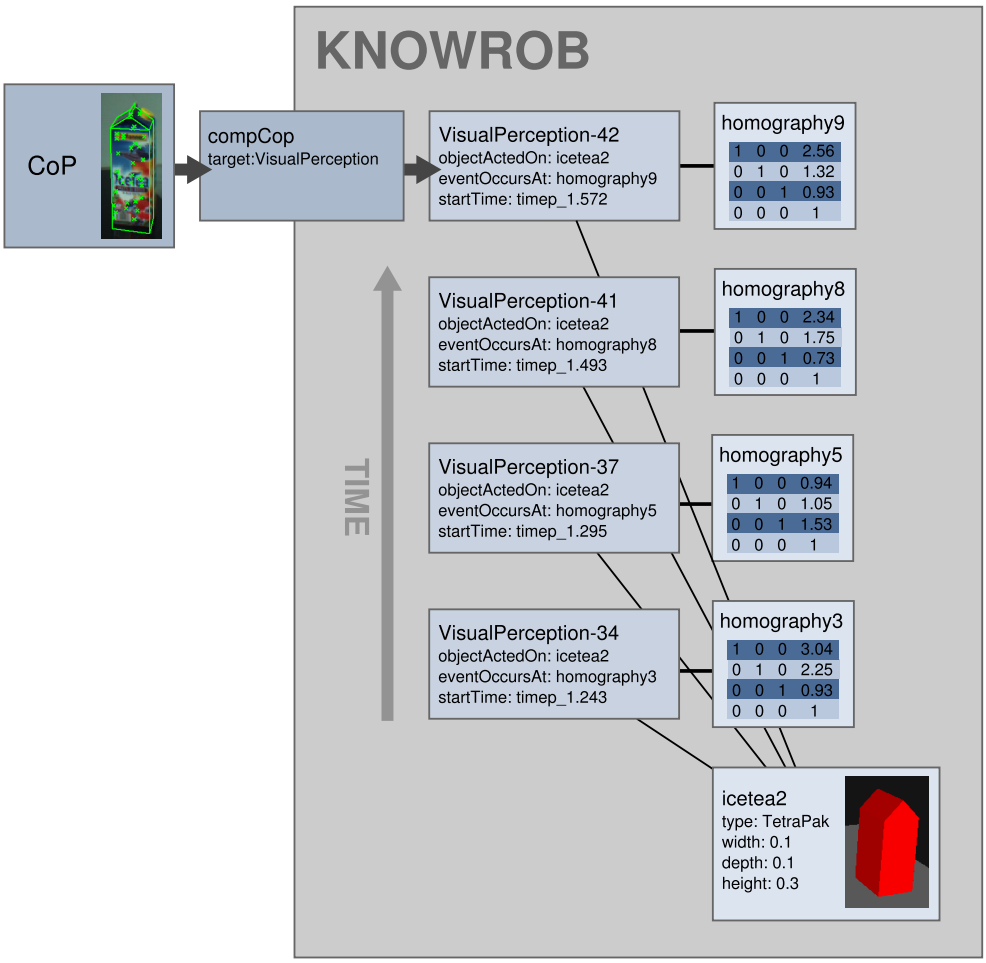


Figure 3.5 Visualization of the internal object representation. Based on information from the vision system, KnowRob generates *VisualPerception* instances that link the object instance *icetea2* to the different locations where it is detected over time.

(Figure 3.4), for instance *VisualPerception* or *Reasoning*. Object recognition algorithms, for instance, are described as sub-classes in the *VisualPerception* tree. Multiple events can be assigned to one object, describing different detections over time or differences between the current world state and the state to be achieved (Figure 3.5).

This representation is similar to the fluent calculus [Thielscher, 1998], in which fluents are objects that represent the change of values over time. In our case, however, the reified objects contain more information than just a changing value: the current and all past states of the relation, including the times at which state changes were detected, and the type of event that established the relation. Using our representation, we can describe multiple "possible worlds", for example the perceived world, a description of how the world is supposed to look like, and the world state a robot predicts as the result of some actions it performs. Since all states are represented in the same system, it becomes possible to compare them, to check for inconsistencies or to derive the required actions, which would be difficult if separate knowledge bases would be used for perceived and inferred world states.

3.2.4 Environment maps

Environment information has been described in various ways in robotics, just to name a few:

- Occupancy grid maps describe obstacles and free space in a grid-based structure.
- Topological maps describe the environment as a graph in which the vertices correspond to points of interest and the edges mean that one vertex can be reached from an adjacent one.
- Point cloud maps describe the surface of the environment by a set of points in 3D space.

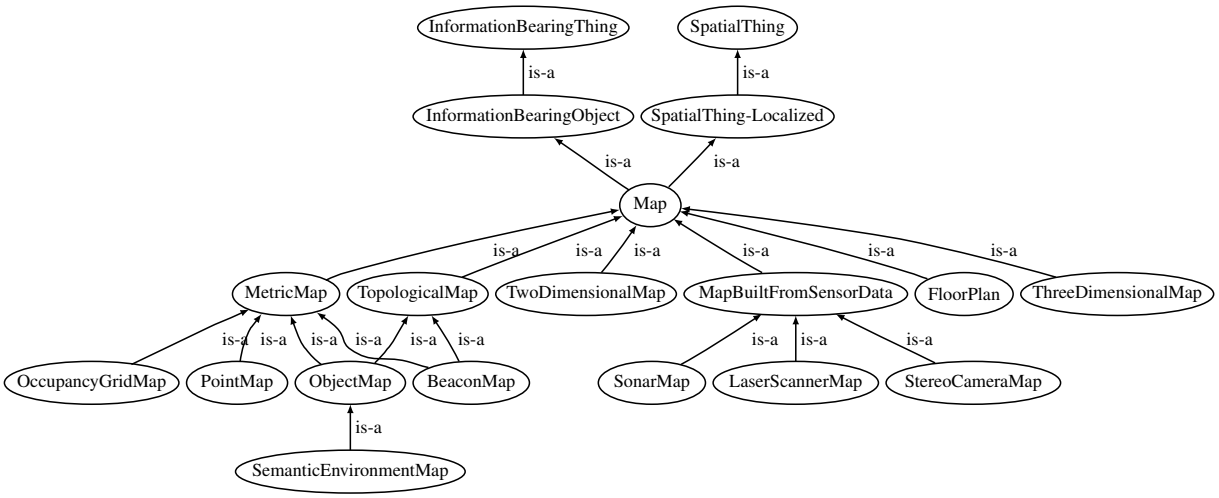


Figure 3.6 Part of the ontology of different kinds of environment maps.

- Maps of visual landmarks contain (often only visually distinctive, but otherwise meaningless) points in space that can serve for localization.
- Object maps consist of localized, typed objects that have been recognized in the environment.

The KNOWROB ontology contains a section describing different kinds of environment maps – both as a spatial description of the environment and as an information-bearing object (Figure 3.6). Specific maps can be described as a combination of these concepts, for example as a three-dimensional semantic environment map.

Some of these maps can easily be described in the knowledge base, especially object maps and maps consisting of meaningful landmarks. In these cases, a map is effectively a collection of object instances. In other cases, such an explicit representation does not make much sense, either because of the large amount of data in the map or due to the lack of structure that can be semantically interpreted. For these maps, KNOWROB allows to link a shallow description of the map to an external (binary) file. In both cases, there is an OWL description that specifies the type of the map and its properties so that the robot is aware of having this map. Both approaches can be combined: For example, an occupancy grid map that serves for describing free space, localization and path planning, can be combined with a set of objects and their positions in the environment, as illustrated in Figure 3.7.

3.2.5 Qualitative spatial relations

Often, objects are not described by their metric positions, but using qualitative spatial relations to other objects, like *inside*, *on top of*, or *underneath*. Such relations are commonly used for communicating with humans, but can also help to generalize spatial knowledge: If the system knows that an object is inside a cupboard or drawer, it can infer that the robot first has to open the respective container before it is able to see and manipulate these objects. Qualitative relations are also useful for describing plans since they leave the robot more freedom to select appropriate positions when actually executing the plan.

In KNOWROB, these qualitative relations are usually not asserted, but rather computed on demand using computables. As described earlier, the system stores the quantitative object poses and dimensions, which is sufficient to compute qualitative descriptions as a more abstract view on the data. We intentionally chose to store numeric poses and compute the qualitative descriptions on demand for several reasons: First, the numeric information has to be stored only once and allows to compute multiple qualitative relations, whereas storing all pairwise relations between

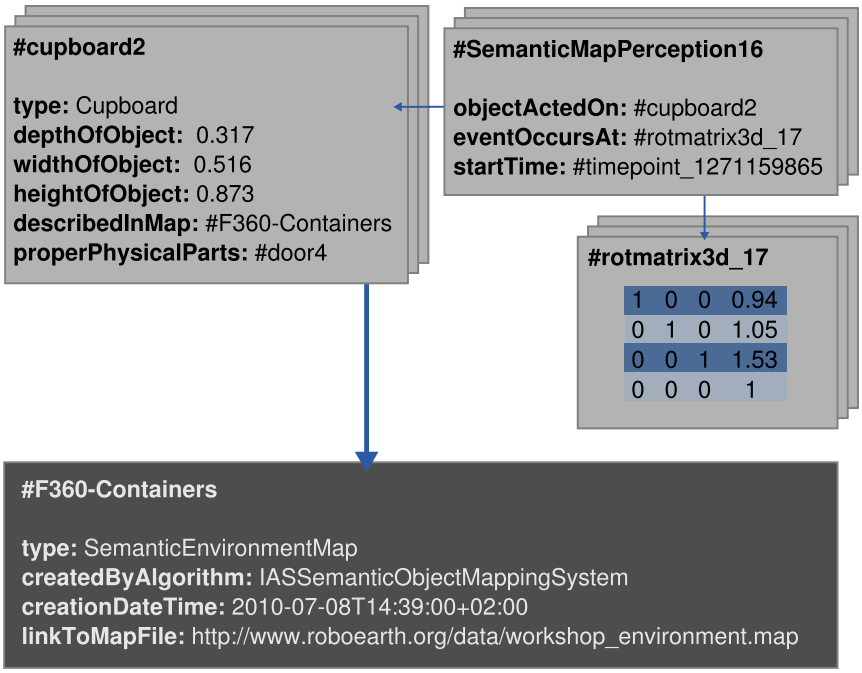


Figure 3.7 Encoding of an environment map that combines a binary file (linked using the *linkToMapFile* property) with an object that was recognized in the respective environment.

a larger number of objects would not be very efficient. Furthermore, maintaining consistency for asserted qualitative relations can become difficult: An external module needs to compute all relations that hold between two objects, check if already asserted relations are still valid and retract outdated ones. This computation has to be done whenever a change in the world occurs and can become quite costly – though many of these relations will never be needed. Storing the object poses instead reduces redundancy and helps ensure consistency by only taking those relations into account that hold for the object configuration at query time.

Qualitative spatial relations can largely be split into two groups: topological and directional relations. Directional relations, like *left of* or *behind*, are relative to the position of an agent.

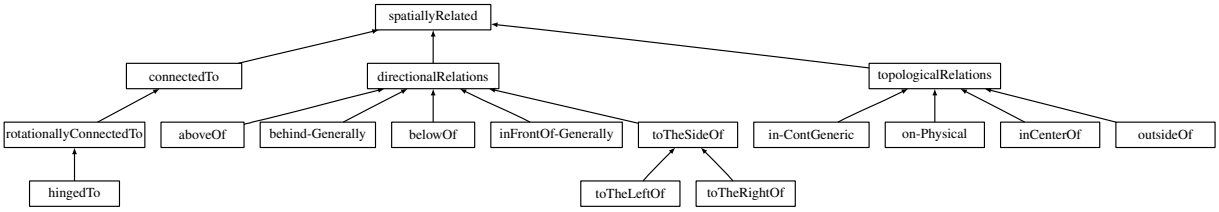


Figure 3.8 Taxonomy of qualitative spatial relations including directional and topological relations.

Topological relations like *in* or *on*, in contrast, can be computed based on only the configuration of objects. In addition, the *connectedTo* relation can describe articulated objects and hinges. All relations are arranged in a hierarchy (Figure 3.8) which enables queries like "Where is object A?" using the *spatiallyRelated* relation to obtain all spatial relations of this object.

In the current implementation, the computables for qualitative spatial relations are purely diagnostic: They can check whether a relation holds, but cannot generate appropriate locations. While there are straightforward solutions for simplified problems, a solution to generate positions that are actually usable by a robot is quite complex and has to take much information into account: The shapes and poses of the bottom and top object, obstacles, the purpose of the location, reachability etc. This is done in a parallel research project [Lorenz Mösenlechner and Michael Beetz, 2011].

3.2.5.1 Relations between objects at different points in time

The aforementioned representation of object poses using *MentalEvents* forms the basis to evaluate how qualitative spatial relations between objects change over time. For example, if a robot is to recall where it has seen an object before or which objects have been detected on the table five minutes ago, it has to qualify the spatial relations with the time at which they held.

We use the *holds(rel(?A, ?B), ?T)* predicate to express that a relation *rel* between *?A* and *?B* is true at time *?T*. Such a temporally qualified relation requires the description of the relation *rel* between the objects *?A* and *?B* and the time *?T*, which cannot be expressed in pure description logics. We thus have to resort to reification, for which we use the mental events described in Section 3.2.3. Based on these detections of an object, the system can compute which relations hold at which points in time.

Since objects are usually perceived only occasionally, the robot does not have a continuous update on their positions. Even when an object has been detected just an instance before its position is needed, this time point is already in the past, so the robot always has to interpolate between the poses. There are several different options, for example using the persistence assumption that objects do not move unless detected otherwise, performing linear interpolation between two poses, or using learned models that take the actions that are performed into account and can better predict how objects move. The current implementation uses the persistence assumption and assumes that the last perceived pose of an object is still valid at the time the relation is evaluated. However, this simple method can easily be replaced by a more sophisticated solution.

Let us consider the computation of the "inside" relation as an example (see the following listing for the implementation). The code computes the relation in a simplified way (not taking the

rotation of the objects into account) by comparing the axis-aligned bounding boxes of the inner and outer object to check whether one contains the other. First, the latest perception of each object before time *?T* is determined using the *object_detection* predicate. The poses where objects have been perceived in these events are read using the *eventOccursAt* relation. Then, the system reads the objects' positions and dimensions, and compares the bounding boxes. Figure 3.9 exemplarily shows how the *holds* predicate operates on a set of *VisualPerception* instances.

```
holds(in_ContGeneric(?InnerObj, ?OuterObj), ?T) :-

    object_detection(?InnerObj, ?T, ?VPI),
    object_detection(?OuterObj, ?T, ?VPO),

    rdf_triple(eventOccursAt, ?VPI, ?InnerObjMatrix),
    rdf_triple(eventOccursAt, ?VPO, ?OuterObjMatrix),

    % read the center coordinates of the left entity
    rdf_triple(m03, ?InnerObjMatrix, ?IX),
    rdf_triple(m13, ?InnerObjMatrix, ?IY),
    rdf_triple(m23, ?InnerObjMatrix, ?IZ),

    % read the center coordinates of the right entity
    rdf_triple(m03, ?OuterObjMatrix, ?OX),
    rdf_triple(m13, ?OuterObjMatrix, ?OY),
    rdf_triple(m23, ?OuterObjMatrix, ?OZ),

    % read the dimensions of the outer entity
    rdf_has(?OuterObj, widthOfObject, ?OW),
    rdf_has(?OuterObj, heightOfObject,?OH),
    rdf_has(?OuterObj, depthOfObject, ?OD),

    % read the dimensions of the inner entity
    rdf_has(?InnerObj, widthOfObject, ?IW),
    rdf_has(?InnerObj, heightOfObject,?IH),
    rdf_has(?InnerObj, depthOfObject, ?ID),

    % compare bounding boxes
    >=((?IX - 0.5*?ID), (?OX - 0.5*?OD)+0.05), =<((?IX + 0.5*?ID), (?OX + 0.5*?OD)-0.05),
    >=((?IY - 0.5*?IW), (?OY - 0.5*?OW)+0.05), =<((?IY + 0.5*?IW), (?OY + 0.5*?OW)-0.05),
    >=((?IZ - 0.5*?IH), (?OZ - 0.5*?OH)+0.05), =<((?IZ + 0.5*?IH), (?OZ + 0.5*?OH)-0.05),
    ?InnerObj \= ?OuterObj.
```

3.2.5.2 Default: current point in time

The *holds(rel(?A, ?B), ?T)* predicate is not part of the common OWL reasoning procedure, so spatial relations that have been computed using this predicate, for example *holds(in_ContGeneric(?InnerObj, ?OuterObj), ?T)*, are not linked to the respective OWL relations like *in-ContGeneric*. In particular, the mechanism is not backwards compatible, i.e. the manual assertion of a spatial relation and the result of the *holds* computation are not equivalent. To integrate the two inference schemes, we implemented a set of computables that calculate the OWL relations, listed in Figure 3.8, by evaluating *holds(in_ContGeneric(?InnerObj, ?OuterObj), ?T)* for the current time:

```
comp_in_ContGeneric(?InnerObj, ?OuterObj) :-
  get_timepoint(?NOW),
  holds(in_ContGeneric(?InnerObj, ?OuterObj), ?NOW).
```

Queries often deal with the current state of the world, so using it as the default has proven to be a sensible assumption. This way, users do not always have to specify the time when querying for current relations. This also shows one of the strengths of computables: On the one hand, the expressive and complex representation of object poses remains in the background, but users can ask simple queries that are translated by the computables.

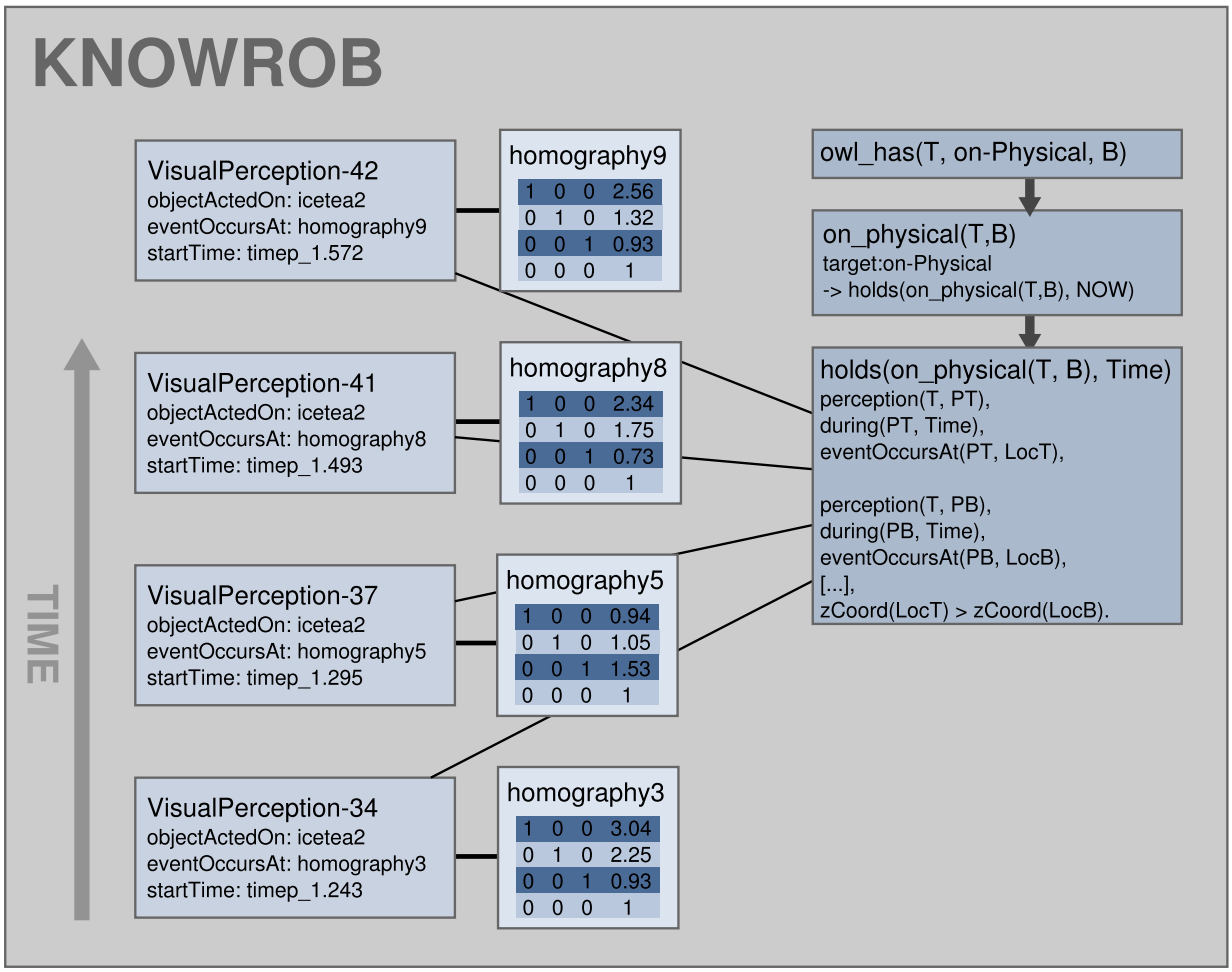


Figure 3.9 Computables operating on the KNOWROB object representations. The *holds* predicate compute the *on_physical* relation for a given point in time. For queries about the current state of the world, a simplified query scheme has been realized that evaluates the relation at the current point in time and maps the *holds(on_physical(?A, ?B), ?T)* predicate to the *on-Physical* OWL property using computables.

3.2.5.3 Relations between objects over time spans

In some cases, one may be interested in verifying if a relation holds over an extended time interval, e.g. if an object has been on the table continuously over the last hour. This can be computed by evaluating the *holds()* predicate at the start and end of the interval and for all detections of the respective objects during the interval. Obviously, this does not guarantee that the object has not been moved in between two detections, but, based on the robot's knowledge, it is the best answer a robot can give.

We thus define the *holds_tt* predicate (for "holds throughout") as follows: *holds_tt(?Goal, [?Start, ?End])* is true for an interval *[?Start, ?End]* iff *holds(?Goal, ?Start)*, *holds(?Goal, ?End)*, and *holds(?Goal, ?T)* for each detection of each of the involved objects in between *?Start* and *?End*. The following code example first temporarily saves the time interval the user has queried for in order to be able to use the computables to reason about temporal relations like *temporallySubsumes*. Then the system reads the arguments of the *?Goal* and reads all detections of either of these objects. The *forall* statement makes sure that for all time points when either of these objects is detected between the *?Start* and *?End* time, the relation *?Goal* still holds.

```
holds_tt(?Goal, [?Start, ?End]) :-

    rdf_assert(holds_tt, type, 'TimeInterval'),
    rdf_assert(holds_tt, startTime, ?Start),
    rdf_assert(holds_tt, endTime,   ?End),

    holds(?Goal, ?Start),
    holds(?Goal, ?End),

    % find all detections of the objects at hand
    arg(1, ?Goal, ?Arg1), arg(2, ?Goal, ?Arg2),
    findall([?D_i,?Arg1], ( (rdf_has(?D_i, objectActedOn, ?Arg1);
                             rdf_has(?D_i, objectActedOn, ?Arg2)),
                             rdfs_individual_of(?D_i, 'MentalEvent')), ?Detections),

      forall( ( member(?D_O, ?Detections), nth0(0, ?D_O, ?Detection),
                rdf_triple(startTime, ?Detection, ?DStT),
                rdf_triple(temporallySubsumes, holds_tt, ?DStT) ),
              holds(?Goal, ?DStT) ),

    rdf_retractall(knowrob:'holds_tt', _, _).
```

3.3 Actions and tasks

A very special kind of events – probably the most important ones for a mobile robot – are actions. Actions are intentionally performed by an agent to achieve a certain effect. This effect can either be to achieve physical changes of the outer world, for example to move an object, or to change the internal belief state, for example by actively perceiving an object.

There are two different ways how actions can be described: First, there are plans, descriptions of an action schema, which can be executed multiple times. By being executed, they create a grounded instantiation, the second kind of action description. While plans describe the abstract structure of an action, including references to the objects that are to be manipulated or the locations that shall be used, these instantiations describe concrete executions that happened at a specific point in time.

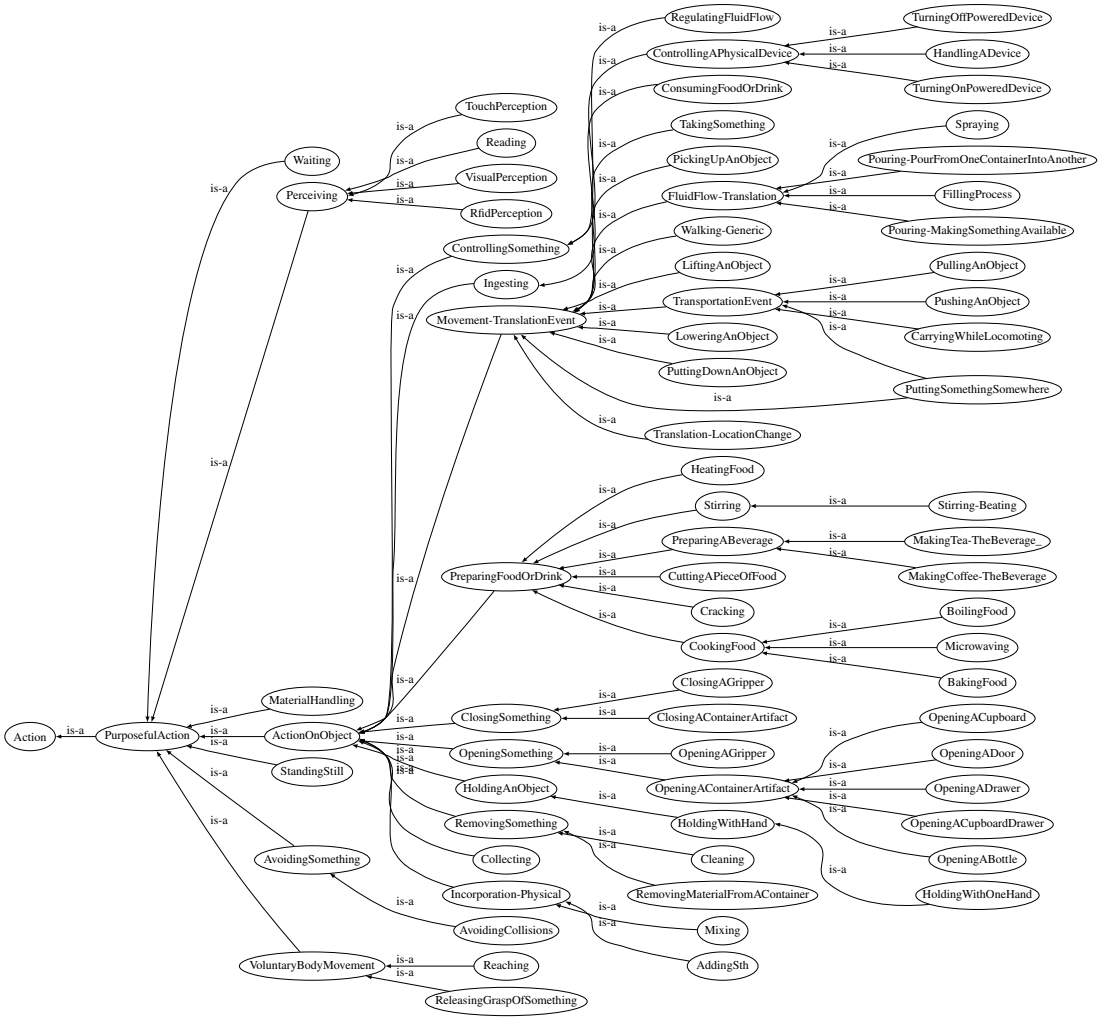


Figure 3.10 Part of the ontology of action classes.

In Description Logic, plans are described as terminological descriptions (in the TBOX), whereas instantiations are described as assertional knowledge (ABOX). Such instances of actions can describe actually observed or performed actions (something that happened at a certain time), planned actions (something the robot intends to do), inferred actions (something the robot imagines that happens or happened), or asserted actions (some action someone told the robot has happened). We will start with the description of actions on the class level and continue with the representation of action instances.

3.3.1 Action classes

The KNOWROB ontology provides a taxonomy of more than 130 actions that are commonly observed in everyday activities. Figure 3.10 shows an excerpt; we omitted several classes for the sake of clarity. Note that, while the classes in Figure 3.10 are all rather general, class definitions can also be very specific and describe, for instance, the action *PuttingDinnerPlateInCenterOfPlacemat*. Usually, the generic descriptions are part of the main ontology, while the speficic classes are often defined as part of a concrete plan definition. In addition to the specialization hierarchy in Figure 3.10, there is another hierarchy describing the composition of complex actions from more basic ones. As an example, the action *PuttingSomethingSomewhere* for transporting an object from one position to another involves picking up an object, moving to the goal position, and putting the object down again. These sub-actions are described in the following OWL fragment:

```
Class: PuttingSomethingSomewhere
  SubClassOf:
      Movement-TranslationEvent
      TransportationEvent
      subAction some PickingUpAnObject
      subAction some CarryingWhileLocomoting
      subAction some PuttingDownAnObject
      orderingConstraints value SubEventOrdering1
      orderingConstraints value SubEventOrdering2
```

The ordering of *subActions* in a task can be specified by the partial ordering constraints given below which describe the relative pair-wise ordering among the sub-actions.

```
Individual: SubEventOrdering1
  Types:
      PartialOrdering-Strict
  Facts:
      occursBeforeInOrdering PickingUpAnObject
      occursAfterInOrdering  CarryingWhileLocomoting

Individual: SubEventOrdering2
  Types:
      PartialOrdering-Strict
  Facts:
      occursBeforeInOrdering CarryingWhileLocomoting
      occursAfterInOrdering  PuttingDownAnObject
```

Using these constructs, one can describe actions, encode knowledge about their hierarchical composition and about ordering constraints that need to hold among the parts of an action. In KNOWROB, this knowledge is mainly part of the main ontology. In the next section, we will look in more detail at how single actions can be composed to task descriptions in order to form a plan.

3.3.2 Composing actions to plans

Complex robot tasks are usually composed of many lower-level actions and movements. Therefore, plans can be hierarchically nested in order to keep the high-level plans short and concise, and to maximise code re-use. A typical plan definition derives specialized classes from the action classes in the KNOWROB ontology and extends them with task-specific action properties. These derived action classes are then arranged in a (partially) sequential order to form the task description. The code below is an excerpt of a plan for setting a table.

```
Class: SetATable
    Annotations: label "set a table"
  SubClassOf: Action
  EquivalentTo:
      subAction some PutPlaceMatInFrontOfChair
      subAction some PutPlateInCenterOfPlaceMat
      subAction some PutKnifeRightOfPlate
      subAction some [...]
      orderingConstraints value [...]


Class: PutPlaceMatInFrontOfChair
  EquivalentTo:
      PuttingSomethingSomewhere
      objectActedOn value PlaceMat1
      toLocation some Place1

Class: Place1
  EquivalentTo:
      inFrontOf-Generally some Chair-PieceOfFurniture

Individual: PlaceMat1
  Types: PlaceMat


Class: PutPlateInCenterOfPlaceMat
  EquivalentTo:
      PuttingSomethingSomewhere
      objectActedOn value DinnerPlate1
      toLocation some Place2

Class: Place2
  EquivalentTo:
      inCenterOf value PlaceMat1

Individual: DinnerPlate1
  Types: DinnerPlate

[...]
```

The upper part describes the task *SetATable* as a subclass of *Action* with a set of *subActions*. The lower part consists of definitions of task-specific subclasses of generic action classes. The class *PutPlaceMatInFrontOfChair*, for example, is defined as a subclass of the *PuttingSome-*

thingSomewhere action, with the additional restriction that the *objectActedOn* needs to be a *PlaceMat*, and the *toLocation* has to fulfill the requirements described for the class *Place1*, which by itself is described as some *Place* which is *inFrontOf-Generally* of some *Chair-PieceOfFurniture*.

Though most of the plan is described as part of the TBOX, i.e. using restrictions on the properties of action classes, the involved objects are described as instances. The reason is that, in order to ensure that subsequent actions are performed on the same object, one has to use an instance instead of a class restriction. A class restriction could otherwise be resolved to different instances so that the robot may choose to pick up one bottle and open another one. This kind of description is better suited to describe plans that are to be executed; before execution, the temporary object instances have to be grounded in the actually perceived objects in the robot's environment. Another option is to also describe the objects in terms of class restrictions, for example as

```
Class: PutPlateInCenterOfPlaceMat
  EquivalentTo:
      PuttingSomethingSomewhere
      objectActedOn some DinnerPlate
      toLocation some Place2

Class: Place2
  EquivalentTo:
      inCenterOf some PlaceMat
```

This representation does not guarantee that all actions are performed on the same object instance, but has different advantages: First, it can more easily be used to classify observed actions, i.e. to check whether they fit a task description. Second, this representation does not suffer from a problem that can arise when a plan described using the first method is executed several times on different objects: At the beginning of the first execution, the object instance in the plan, e.g. *PlaceMat1*, is unified with a perceived object, e.g. *PlaceMat2675*. The problem arises when the same plan is executed using a different object instance: Now, *PlaceMat1* is unified with e.g. *PlaceMat4231*, and the system could conclude that *PlaceMat2675* and *PlaceMat4231* were the same objects. Description Logics do not support variables, which would be needed here, but the problem can be circumvented by retracting the unification axiom after the plan has been executed.

3.3.3 Action instances

Whenever the robot is reasoning about actually performed actions, either by itself or by a human, it needs to describe action instances. These instances can, for example, be generated by an action recognition system that interacts with the knowledge base and populates the set of action instances based on observations of humans, or by the robot's executive that logs its actions into the

knowledge base. Based on these observations, the system can set properties like the *startTime*, the *objectActedOn*, or the *bodyPartUsed*.

Having the observations in the same language as the other pieces of knowledge in the knowledge base is important to relate the observations to background knowledge. Chapter 5 describes how action instances in the knowledge base can be used to interpret observations of humans performing everyday tasks.

3.3.4 Effects of actions on objects

With service robots extending their task spectrum, they need to reason about more and more complex effects of actions. Simple pick-and-place tasks already require the robot to represent changes in the positions of objects, but still keep the notion of "an object" as something that keeps on existing while the robot performs its task. When it comes to more complex activities, like cooking meals, this assumption is no longer valid since these activities affect objects in a much more fundamental way: Objects are created, destroyed, and can substantially change their types, appearance, and aggregate states. For example, vegetables are being cut into pieces (which appear as new objects while the original objects disappear), substances are mixed to cookie dough, which is transformed by the baking process from some liquid stuff to a rigid object.

To describe which effects an action has, a robot has to represent the world state both before and after the action has been performed, and it needs to track which objects got transformed into which other objects. Figure 3.11 visualizes the effects of a cracking and two mixing actions. After having cracked an egg, the egg ceases to exist and (at least) two pieces of eggshell, the egg yolk and the egg white appear. In parallel, the milk and flour are mixed to a dough, to which the eggyolk is added.

There are two use cases of the description of action effects that both have different requirements: When planning its actions such that they lead to the intended goal state, the robot needs declarative specifications of the inputs and outputs of an action. Using these specifications, it can search for actions that have the desired effects and verify that all required inputs are available. The properties for describing the relations between the actions and objects, namely their in- and outputs, pre- and postconditions, are described in the next section. The second main use case is projection, i.e. the prediction of the outcome of an action. To perform this prediction, the system needs methods to compute the changes the action induces to the world. In KNOWROB, this is realized using procedural descriptions of the effects of actions which are described in more detail in Section 3.3.4.2. That section further explains how structures like the one in Figure 3.11 can be

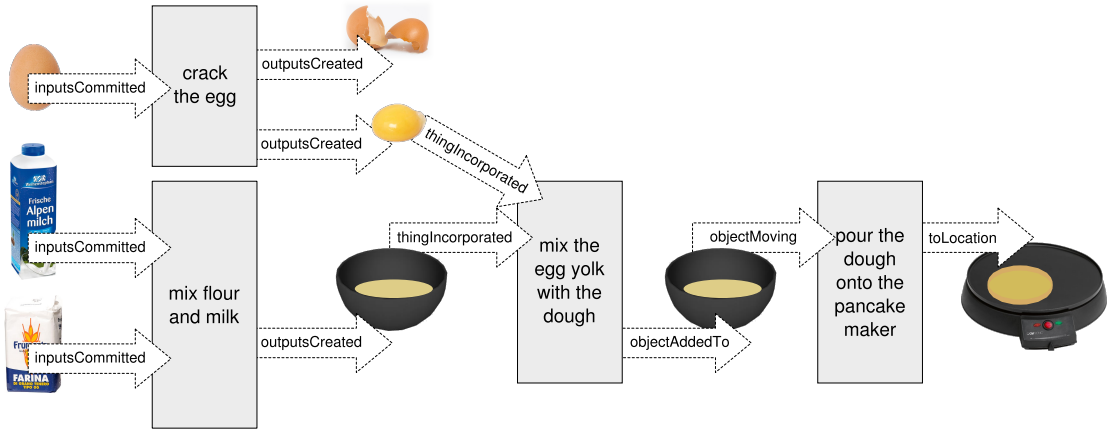


Figure 3.11 Object changes in a simple baking task: An egg is cracked, egg shells and egg yolk appear as single objects, and are mixed to a dough together with some milk and flour.

built up automatically as the result of the projection procedure. Section 3.3.4.3 will then discuss how these structures can be used for reasoning about object transformations.

3.3.4.1 Declarative descriptions of action effects

To perform reasoning about the effects of actions and the changes of the involved objects, it is necessary to describe the exact relation between actions and objects. KNOWROB therefore provides a detailed set of properties as shown in Figure 3.12. The sub-properties of *preActors*, displayed in the upper part of the figure, describe action properties that are supposed to hold before the action takes place. They include the agent (*doneBy*), the initial locations and states (*fromLocation*, *fromState*), and the different roles an object can play in an action. An object can be incorporated (*thingIncorporated*) into another one (*objectAddedTo*), can be removed from something, like the dirt in a cleaning action (*objectRemoved*), and can undergo state changes like freezing or melting (*objectOfStateChange*). Perceptual actions can detect an object (*detectedObject*), and actions can substantially change objects by transforming them into another one (*transformedObject*), destroying them (*inputsDestroyed*) or integrating them into another one (*inputsCommitted*). The *postActors* describe the outcome of an action: Outputs that are created by the action, for example a dough that emerges from flour and water (*outputsCreated*), or that were modified, but remained the same kind of object, like a bread from which a slice is cut off (*outputsRemaining*). Body movements lead to a *targetPosture*, transport actions move something to a *toLocation*, and state changes attain a target state (*toState*). In addition to the properties in Figure 3.12, actions can further be described using the event properties like *startTime*, *after*, or *temporallySubsumes* described in Section 3.1.

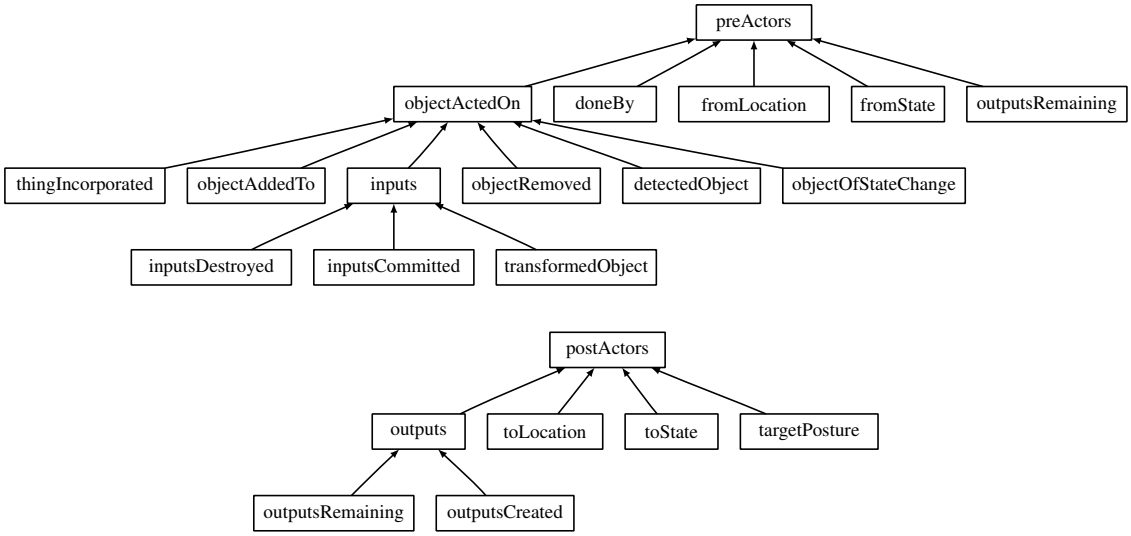


Figure 3.12 Hierarchy of action-related properties. The upper part lists specializations of *preActors*, which describe the inputs of an action and the situation at its beginning. The *postActors* describe the outputs and post-conditions.

By defining class restrictions using these properties, one can describe the inputs, outputs, pre- and postconditions of an action in terms of a declarative specification that can easily be queried to find an action that has the desired properties. For example, the robot can search for an action that turns a *PhysicalDevice* from *DeviceStateOff* to *DeviceStateOn* and receive the action *TurningOnPoweredDevice* which is defined as follows:

```
Class: TurningOnPoweredDevice
  SubClassOf:
      ControllingAPhysicalDevice
      objectOfStateChange some PhysicalDevice
      fromState value DeviceStateOff
      toState   value DeviceStateOn
      [...]
```

3.3.4.2 Temporal projection of action effects

Assuming a robot is about to perform an action on a given object instance, it can predict the outcome of the action and the resulting world state using its knowledge about the effects of actions. These effect axioms are described as predicates that create the links between the action instance and the involved objects in order to describe inputs, outputs, newly created or destroyed objects. In the current implementation, the projection rules are realized as prolog rules that describe the effects of an action on a rather coarse, symbolic level.

Below are two examples of projection rules for the actions “cracking an egg” and “mixing baking mix to a dough”. In the first lines, the predicate checks its applicability conditions: The

action needs to be of the expected type, the manipulated object has to be specified, and the outputs should not have been computed already. Then it creates the new object instance that are generated by the action using the *rdf_instance_from_class* predicate which generates a new instance with a unique name and the given type. Afterwards, the predicate asserts the relations between the input objects, the action, and the generated outputs. Before this projection, the generic *objectActedOn* was the only known relation, which can now be described in more detail using properties like *inputsDestroyed* or *thingIncorporated*.

```
% Cracking an egg
project_action_effects(?Action) :-

  owl_individual_of(?Action, 'Cracking'),
  \+ owl_has(?Action, outputsCreated, _),

  owl_has(?Action, objectActedOn, ?Obj),
  owl_individual_of(?Obj, 'Egg-Chickens'),!,

  % new objects
  rdf_instance_from_class('EggShell', ?Shell),
  rdf_instance_from_class('EggYolk-Food', ?Yolk),

  % new relations
  rdf_assert(?Action, inputsDestroyed, ?Obj),
  rdf_assert(?Action, outputsCreated,  ?Shell),
  rdf_assert(?Action, outputsCreated,  ?Yolk).


% Mixing baking mix to a dough
project_action_effects(?Action) :-

  owl_individual_of(?Action, 'Mixing'),
  \+ owl_has(?Action, outputsCreated, _),

  % at least one objectActedOn is a MixForBakedGoods
  owl_has(?Action, objectActedOn, ?Mix),
  owl_individual_of(?Mix, 'MixForBakedGoods'),

  findall(?Obj, owl_has(?Action, objectActedOn, ?Obj), ?Objs), !,

  % new objects
  rdf_instance_from_class('Dough',     ?Dough),
  rdf_assert(?Action, objectAddedTo,  ?Dough),
  rdf_assert(?Action, outputsCreated, ?Dough),

  % new relations
  findall(?O, (member(?O, ?Objs),
               rdf_assert(?Action, thingIncorporated, ?O) ), _).
```

While these simple descriptions help the robot to predict if objects appear, get destroyed, or change their types, they do not cover all effects and do not describe actions in sufficient detail. Moreover, they are still somewhat over-specialized for certain combinations of actions and objects. We thus intend to use them as the place where other, more advanced prediction mechanisms like physical simulation [Kunze et al., 2011a; Mösenlechner and Beetz, 2009] or more generic projection mechanisms can be plugged in.

The projection rules are implemented as computables for the *postActors* relation, which makes the projection transparent to the user, who just queries for the effects of an action. If they have not yet been computed, they are generated on demand during the reasoning process by the implementation of the computable, e.g. the rules shown before.

```
?- rdf_triple(postActors, 'put-mix-on-pan1', ?Post).
```

3.3.4.3 Reasoning about object transformations

The result of applying the projection methods is an object transformation graph like the one in Figure 3.11 that describes how the involved objects are transformed by the actions in task. The projection predicates assert the predicted outcome of the action and can further describe the input in more detail (e.g. asserting that the *objectActedOn* is in fact an *inputsDestroyed*). Due to the arrangement of the input and output properties in a hierarchy, one can query for the *preActors* or *postActors* and get the results for all sub-properties since the *rdf_has* query predicate also takes sub-properties into account.

```
?- rdf_has('put-mix-on-pan1', preActors, ?Post).

?- rdf_has('put-mix-on-pan1', postActors, ?Post).
```

To track the changes made to an object over a sequence of actions, we defined the *transformedInto* predicate as a transitive relation that covers all modifications of objects, including destruction, creation, and transformation. It is defined via a computable as a relation between all specializations of the *objectActedOn* of an action and all of the *outputs*:

```
transformed_into(?From, ?To) :-
  ( owl_has(?Event, thingIncorporated,   ?From);
    owl_has(?Event, objectAddedTo,       ?From);
    owl_has(?Event, inputsDestroyed,     ?From);
    owl_has(?Event, inputsCommitted,     ?From);
    owl_has(?Event, transformedObject,   ?From);
    owl_has(?Event, objectRemoved,       ?From);
    owl_has(?Event, objectOfStateChange, ?From);
    owl_has(?Event, outputsRemaining,    ?From) ),

  ( owl_has(?Event, outputsRemaining, ?To);
    owl_has(?Event, outputsCreated,   ?To)).
```

Figuratively speaking, this relation steps over the action and directly links the inputs and outputs, and due to its transitive definition, can create whole chains of transformation applied to an object. It allows, for instance, to retrieve all ingredients of a product, or to explain into which other objects an input object has been converted:

```
?- rdf_triple(transformedInto, ?From, ?To).
From = 'pancake-dough1',
To = 'Baked1'.
```

3.4 Processes and their effects

In the previous section, we have discussed how to model the direct effects of actions. Actions can also have indirect effects: When pouring liquid pancake dough into a hot pan, the direct effect is that the dough is in the pan, while the transformation from a liquid dough into a solid pancake is caused by an indirectly caused baking process. This process will only become active if its preconditions are fulfilled. There are external preconditions, namely that there needs to be a thermal connection between the dough and a heat source, as well as continuous preconditions, namely that the heat source needs to be hot enough for the dough to bake. In general, processes describe changes that happen in the world which are not directly and intentionally caused by an action. Examples are the process of melting ice, or the process of dough being transformed into cake by the surrounding heat. With our notion of processes we largely follow the Qualitative Process Theory (QPT) by Forbus [Forbus, 1984], the standard work for qualitative reasoning about processes.

The classical QPT only considers processes that happen more or less automatically because their preconditions become true for some reason. Its focus is on physical processes in industrial settings like the steam production in a boiler. In robotics, we are also strongly interested in the interaction between actions and processes: Robots can actively change the state of the world and either accidentally or intentionally start processes by their actions. Therefore, the representation should both include the effects of processes into the prediction of the outcome of actions, and support planning with processes, i.e. to perform an action in order to start a process.

Therefore, we extended the QPT representation in two ways: First, we added declarative descriptions of the requirements and outputs of processes that the robot can use in a planning context. These descriptions are very similar to the descriptions of the inputs and outputs of actions described in Section 3.3.4.1. Second, we included the process effect axioms into the action projection procedure: Each time the robot predicts the effects of an action, it also checks whether processes got started because their preconditions became true.

3.4.1 Process ontology

Similar to action classes, we also arrange the classes describing processes in an ontology whose upper part is visualized in Figure 3.13. In the upper half, the *IntrinsicStateChangeEvents* describe processes that mainly change the state of an object, e.g. if a device is switched on or off (*ChangingDeviceState*) or if a container is open or closed (*OpeningSomething/ClosingSomething*). This branch further comprises changes in temperature (*HeatingProcess/CoolingProcess*) and resulting changes in the aggregate state. The lower part, subclasses of *PhysicalEvent*, describes processes that result in the creation, destruction, or a different arrangement of objects.

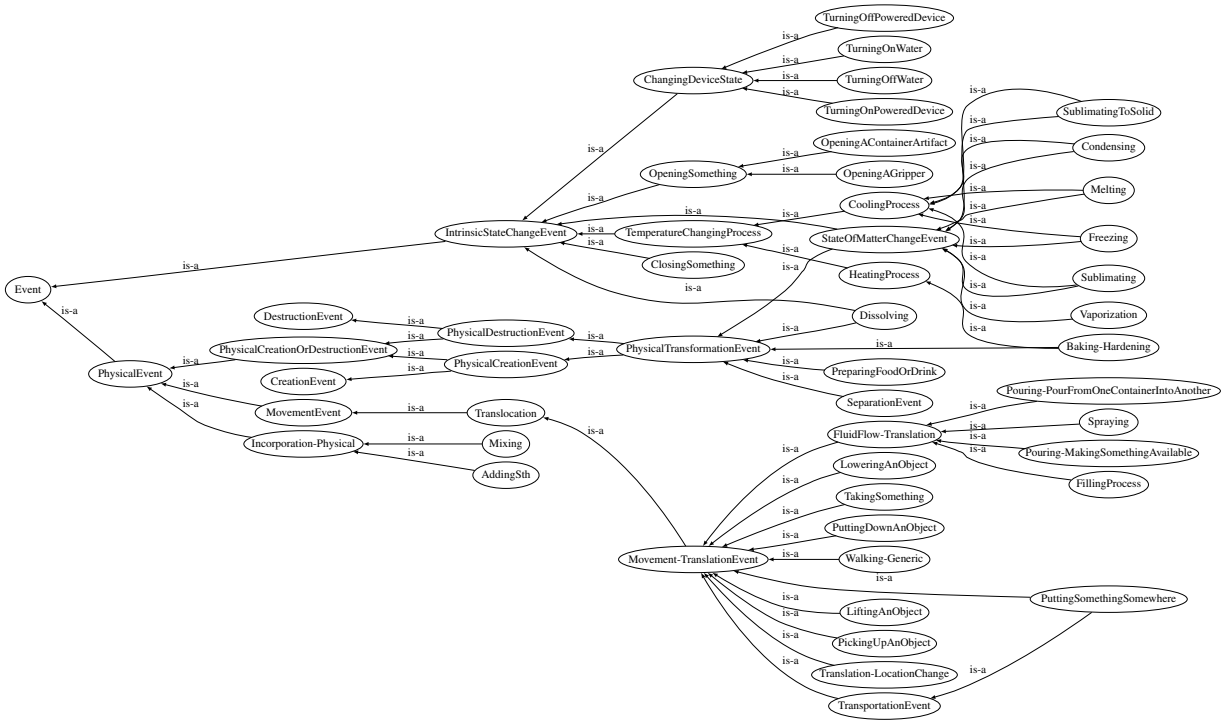


Figure 3.13 Part of the ontology of processes. For better readability, we omitted some branches of the ontology and did not fully expand the class hierarchy.

The process classes can be used like action classes in conjunction with the properties described in Section 3.3.4.1 to declaratively describe the required inputs for the process to become active and the outputs that are generated by the process. The limited expressiveness of description logics prohibits a complete and exact description of the changes induced by a process (for example, the lack of variables makes it hard to describe how a specific object is being changed using only pure description logics). For this reason, we combine the declarative descriptions with the projection rules described in the next sections.

3.4.2 Process definition

We adopt the definition of processes from Forbus' Qualitative Process Theory, which describes a process as follows (see [Forbus, 1984], p.105):

1. "the individuals it applies to;
2. a set of preconditions [...];
3. a set of quantity conditions [...];
4. a set of relations the process imposes beween the parameters of the individuals, along with any new entities that are created;
5. a set of influences imposed by the process on the parameters of the individuals."

The preconditions (2) are thereby external circumstances, not described inside the QPT, that need to be fulfilled for the process to become active, while the quantity conditions (3) are relations between the properties of the involved individuals that are part of the process theory. In a baking process, for example, the thermal connection between some dough and a heat source is an external precondition, while the relation of the temperatures (the temperature of the heat source needs to be above the baking temperature of the dough) is described as quantity condition.

Like in the QPT, we represent qualitative relations between values (like *larger than* or *smaller than*) instead of discretizing continuous values into concepts like *Hot*, *VeryHot* etc. First, those discrete concepts are not very well-defined: For humans, touching something of 70 degrees centigrade feels *VeryHot*, while this temperature may not even be *Hot* in the context of melting metal. Apart from that, qualitative relations go well with computables, which can easily derive them from numerical values.

3.4.3 Implementation

The representation of processes needs to provide a declarative description of the inputs and outputs as well as some procedural implementation that can be used for projection. The former is realized using a light-weight representation of the inputs and outputs of a process in OWL, which does not completely cover all effects, but is sufficient to enable some basic planning (e.g. to search for a process that creates the desired effects). On the other hand, there are detailed projection rules, written in Prolog, which are attached to the OWL class and which can be used to predict the outcome of a process that operates on a concrete set of entities. The combination of these two methods allows to search for suitable processes and to estimate their approximate effects.

Below is an example of a projection rule for a baking process. Like in QPT, it first describes the external preconditions (namely that some dough is thermically connected to a heat source, list the individuals that are changed (the dough), and describe the quantity conditions (the temperature of the heat source needs to be above the baking temperature of the dough). If these conditions are true, the process converts the *Dough* into some *Baked*, creates an instance of a *BakingFood* process linking them, and assigns the current time as the start time of the process. We simplified the QPT in several respects, for example by not yet looking at equivalence relations between values (the relations slot is empty here). However, this is no general limitation; these aspects are simply not yet implemented in the projection rules.

```
% Dough becomes Baked during a baking process
project_process_effects :-

  % Preconditions (outside of QP)
  rdf_triple(thermicallyConnectedTo, ?Dough, ?HeatSource),

  % Individuals changed in the process
  owl_individual_of(?Dough, 'Dough'),

  % QuantityConditions (prerequisites for the process to be active, inside of QP)

        % read temperature of the dough; default to 20 deg
        ( (rdf_triple(temperatureOfObject, ?Dough, ?TempDough),
           term_to_atom(?TDterm, ?TempDough)) ;
           ?TDterm=20 ),

        % read temperature of the heat source object; default to 20 deg
        ( (rdf_triple(temperatureOfObject, ?HeatSource, ?TempHeatSource),
           term_to_atom(?THSterm, ?TempHeatSource)) ;
           ?THSterm=20 ),!,

        ?TempBaked = 120,
        ?THSterm > ?TempBaked,
        ?THSterm > ?TDterm,

  % Relations (proportionality, newly generated instances like gas or a flow rate)
  % none

  % Influences of the process on the individuals
  rdf_instance_from_class('BakingFood', ?Ev),
  rdf_instance_from_class('Baked', ?Res),

  rdf_assert(?Ev, inputsDestroyed, ?Dough),
  rdf_assert(?Ev, outputsCreated,  ?Res),

  % remove references to the Dough (spatial relations)
  unlink_object(?Dough),

  get_timepoint(?NOW),
  rdf_assert(?Ev, startTime, ?NOW).
```

To facilitate reasoning about temperatures, we defined some default temperature values that cover most cases in kitchen activities. Specifying the exact temperature for each object is usually not feasible, but few approximate values are already sufficient for many tasks: On the one hand, we specify the *workingTemperatures* of the most important heating and cooling devices, namely

the refrigerator (+5°C), the freezer (-18°C), the oven (+180°C), a hot plate (+150°C) and the pancake maker (+150°C). On the other hand, the system knows the *minTempForProcess* and *maxTempForProcess* for some relevant processes: Water-like substances freeze at about 0°C, boil at about +100°C, and most dough starts to bake at around +120°C. The default temperature, which is used if no temperature is specified for an object, is set to 20°C.

3.4.4 Relation to actions

The original QPT does not deal with actions, but only considers processes that take place as a natural consequence of a given situation, e.g. a boiler heating up and producing steam. In a robotics context, however, it is crucial to take intentional actions into account, and to include the results of processes when reasoning about the consequences of actions.

We thus extended the QPT and combined it with our action representation. We regard processes as something that can be triggered as a side-effect of an action, in that the direct effects of an action make the processes' preconditions or quantity conditions true and thereby start it. The computation of process effects is realized as part of the action projection procedure: Having computed the direct effects of an action, the method calls the generic process projection predicate to check whether any processes became active.

In a planning context, the knowledge about processes can also be used to perform actions with the intention of starting a process that achieves a certain result. Actions and processes are both events that, given some preconditions are true, produce a certain output. Therefore, the planning system can handle them in a very similar way, with the distinction that actions need to be actively performed by the robot, while processes start automatically once their preconditions become true.

Forbus also described an extension of the QPT to include actions into the predictions [Forbus, 1988]: The system computes "action-augmented envisionments" by considering actions as changes in the background assumptions made by the QPT (the prerequisites and quantity conditions). This view on actions is somewhat different from ours: Actions are seen as something that influences processes, while we rather see processes as side-effects of actions. His representation also does not cover the planning aspect to determine which actions to perform in order to achieve the desired effect via processes.

3.4.5 Planning with actions and processes

In this section, we describe in more detail how the completion of action sequences, i.e. the planning using both actions and processes, works. Since the requirements of processes and the inputs of actions are described using the same properties, they can be handled equivalently. In the following, especially in the predicate names, we often refer to 'actions' when we mean 'actions and processes'. We will first introduce some predicates for reading the in- and outputs of an action or process and for checking whether a resource is available on the robot (i.e. if there is an instance of the respective kind in the knowledge base that has not yet been destroyed). Missing inputs are defined as inputs that are not available in the knowledge base at the current point in time.

```
action_inputs(?Action, ?Input) :-
  class_properties(?Action, preActors, ?Input).

action_outputs(?Action, ?Output) :-
  class_properties(?Action, postActors, ?Output).

action_missing_inputs(?Action, ?Missing) :-
  findall(?Pre, (action_inputs(?Action, ?Pre),
                 \+ resource_available(?Pre)), ?Missing).
```

On top of these basic query predicates, one can perform more advanced reasoning to infer if missing inputs of an action can be provided by adding (a sequence of) other actions. The *add_subactions_for_action* predicate succeeds if either there are no missing inputs, in which case the action is immediately feasible, or if it can find a sequence of actions that, starting from the currently available set of objects, can generate the missing inputs. The implementation iterates over all members of the list of missing inputs and calls the *resource_provided_by_actionseq* predicate to see whether they can be generated by a sequence of actions. The *resource_provided_by_actionseq* predicate first tries to find actions that produce the desired output and then verifies their feasibility by calling *add_subactions_for_action* again for this sub-action.

```
add_subactions_for_action(?Action, []) :-
  action_missing_inputs(?Action, []),!.

add_subactions_for_action(?Action, ?SubActions) :-
  action_missing_inputs(?Action, ?Ms),
  setof(?Sub, ((member(?M, ?Ms), resource_provided_by_actionseq(?M, ?Sub)) ; fail), ?Subs),
  flatten(?Subs, ?SubActions).


resource_provided_by_actionseq(?Resource, [?SubActions|?SubAction]) :-
  action_outputs(?SubAction, ?Resource),
  add_subactions_for_action(?SubAction, ?SubActions).
```

The predicate for planning which additional actions to perform in order to generate the inputs required for an action can be combined with the projection methods to complete whole action sequences. The *integrate_additional_actions* predicate iterates over the sequence of actions and first makes sure that all inputs of the current action are available by calling the *add_subactions_*

for_action predicate. If this first step is successful, the procedure computes the effects of the current action to predict the world state after its execution and continues the completion with the next action in the sequence. The projection step is required because to correctly evaluate the queries for available resources for the following actions, including those resources that have been generated by earlier actions.

```
integrate_additional_actions([],[]).
integrate_additional_actions([?A|?ActSeq], ?ResultActSeq) :-

  add_subactions_for_action(?A, ?AddActions),
  project_action_class(?A, _, _),!,
  integrate_additional_actions(?ActSeq, ?RestActSeq),
  append(?AddActions, [?A], ?ResultActSeq1),
  append(?ResultActSeq1, ?RestActSeq, ?ResultActSeq).
```

3.5 Robots and their capabilities

Self-models are a very important source of information for autonomous robots: Based on these models, they can determine which components they consist of and which capabilities they have in oder to decide if they will likely be able to perform a certain action. In the context of the pancake scenario, the robot needs information about its capabilities, especially in terms of object recognition. Robots also need this ability when they are to autonomously acquire new tasks, for example from task instructions found on web sites (as described in Section 4.1). Also when they exchange information, they need to check whether the information another robot sends can actually be used (see Section 6.3 for more information). Another example is a multi-robot scenario in which the distributions of actions in a task depends on the robots' capabilities. In all of these cases, the robot has to decide whether it thinks it can execute an action or if important components are missing.

Today's robot description formats like the commonly used URDF[3] already describe the kinematics and dynamics of a robot and allow to specify a collision model as well as a surface model for visualization purposes. What they are lacking is a description of the robot's semantics: Which of the components are sensors, which group of components forms a hand? [Kunze et al., 2011b] recently presented the Semantic Robot Description Language (SRDL) as a semantic extension of these languages. SRDL allows to semantically describe the components of a robot and to match them against the requirements of an action in order to check if something is missing. They introduce the notion of *Capabilities* as an intermediate layer between *Actions* and a robot's *Components*. Capabilities describe that a robot is able to perform an action, and can depend either on other capabilities (so-called *CompositeCapabilities*) or on components (as *PrimitiveCapabilities*). Since there are often different ways how a capability can be realized, the additional concept of a *CapabilityProvisionAlternative* is introduced.

We adopted some of these concepts and extended the system to better suit our requirements. Our main use case is the autonomous exchange of knowledge about tasks, objects, and environments between robots which is described in more detail in Section 6.3. Since the robots are to act autonomously and exchange descriptions of real-world tasks, we require the representation language to be on the one hand very expressive, and on the other hand easy to use, so that descriptions of new robots or new tasks can easily be created. To meet the challenges posed by this scenario, we had to extend the system in several respects:

[3] http://www.ros.org/wiki/urdf

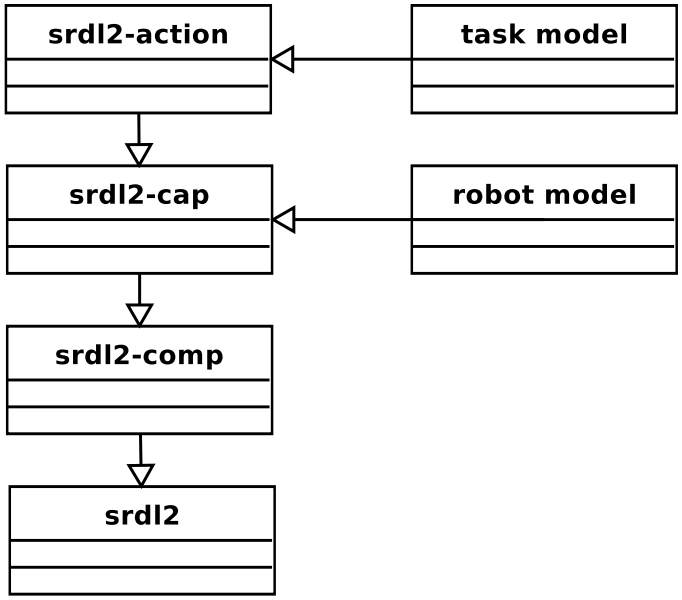


Figure 3.14 Overview of the different sub-ontologies of the SRDL2 language. The modular design allows to use only parts of the system, for example to describe only the hardware components of a robot.

Generality: We dropped the distinction between primitive capabilities, which can only have dependencies on components, and composite capabilities, which can only depend on other capabilities, in favor of a more flexible scheme in which actions, capabilities and components can all depend on components, and both actions and capabilities may have dependencies on capabilities.

Composability and modularity: In order to keep the descriptions of actions short, requirements can now be specified at a coarse level of granularity (e.g. that all actions involving arm motion depend on an *ArmComponent*), and are inherited by all derived action classes. The descriptions of components, capabilities and actions have been separated in order to facilitate the description of robots and actions, and to keep the system as modular as possible. Figure 3.14 shows the dependencies between the different parts of the ontology:

- *srdl2*: Generic relations like the *dependsOn* property
- *srdl2-comp*: Classes describing hardware and software components, and properties for the composition of components and their aggregation to kinematic chains,
- *srdl2-cap*: Classes of capabilities, including their dependencies on components or other capabilities
- *srdl2-action*: Dependency specifications for common action classes

- Robot model: Description of a concrete robot instance including its kinematic structure (auto-generated from URDF file using the tool by [Kunze et al., 2011b]), other hardware- and software components as well as capabilities the robot is asserted to have
- Task model: Description of the task at hand using the action classes defined in the *srdl-action* ontology

Expressiveness of requirement specifications: Often, it is not sufficient to define a dependency on a component of a certain class, but one needs to describe its properties in more detail, like the minimum resolution of a sensor, or the object class that an object recognition model can recognize. Such dependencies can now be described using OWL restrictions and are checked during the inference process.

Simplicity: To be practically usable, creating a description of a new robot should be as little effort as possible, and task instructions should ideally be free of anything else than the description of the involved actions. This allows, for example, to use the system for checking the capability requirements of task instructions that have been autonomously generated from web sites (see Section 4.1). We thus replaced the *CapabilityProvisionAlternatives* with simple sub-classes of capabilities (which they effectively are), and made full use of OWL features like transitivity and sub-properties to simplify the descriptions.

3.5.1 Robot components

In contrast to [Kunze et al., 2011b], we do not distinguish any more between robots and components. First, this makes the inference easier, and second, it also makes sense from a semantic point of view: Many of today's mobile manipulation platforms are assembled from components that can also be used as standalone robots, e.g. the arms and the robot's base.

3.5.1.1 Hardware components

Like in the original SRDL, the robot's hardware is, on the lowest level, described in terms of links and joints that can be imported from a URDF description (Figure 3.15 left). These links and joints can be composed to semantic components like arms and hands, assigning meaning to them (Figure 3.15 right). This association has to be done manually, see [Kunze et al., 2011b] for details.

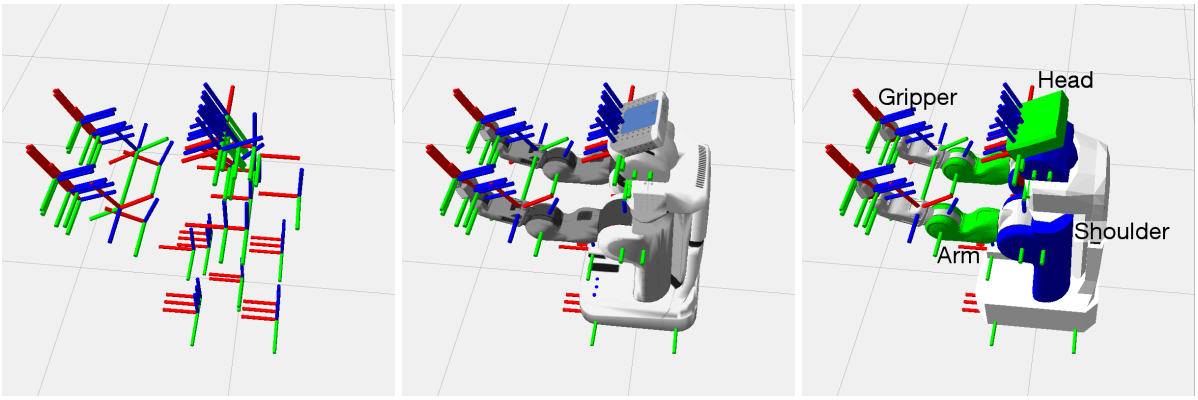


Figure 3.15 Common robot descriptions cover the kinematic and dynamic aspects (left) as well as surface models of a robot (center). A semantic robot description further adds information about the meaning of the different robot parts (right).

Some of the links may correspond to sensors: In these cases, we can assign an additional type to this link and assert that it is, in fact, a camera or a laser scanner. Especially for those sensors that perceive the outer world, it is important to know their type as well as their pose, for example to compute if a robot can see an object from a certain position, or to provide the robot with a custom map that fits the pose of its laser scanners. Sensors are often mounted on moving parts of the robot, like a pan-tilt unit. In these cases, their poses need to be determined during run-time based on the ID of the sensor's coordinate frame (which is part of the SRDL description).

3.5.1.2 Software components

Many of a robot's capabilities are determined by the software it runs. Including software components into the matching process is especially interesting since the robot can react on a negative result and try to retrieve the missing components somehow. While missing hardware components mean that an action can definitely not be performed, an unmet dependency on a software component can often be fixed by starting a program or downloading additional information. Important examples of software components are object recognition models (Section 3.2.1.2), which are required by all kinds of actions that somehow interact with objects, and environment models, which are crucial for navigation and for locating objects in the environment (Section 3.2.4).

3.5.1.3 Realization

We re-arranged all properties for describing relations between components to be specializations of the *subComponent* property, for example the *successorInKinematicChain* and its subproperties *succeedingLink* and *succeedingJoint*. This facilitates querying for subcomponents without having to take their types or the exact kind of link into account. The *subComponent* property is also transitive, allowing to obtain all subsequent components with a single query. Robots are considered to be complex components, so all components of a robot can be retrieved by asking for sub-components of the robot instance:

```
?- sub_component('TUM_Rosie1', ?Sub).
Sub = 'TUM_Rosie_WheeledOmnidirPlatform1' ;
Sub = 'TUM_Rosie_Torso1' .
```

The availability of a component of a certain type on a given robot (or, in general, as part of another component) can be checked using the *comp_type_available* predicate:

```
?- comp_type_available('TUM_Rosie1', ?CompT).
CompT = 'VectorFieldArmController' ;
CompT = 'ArmMotionController' ;
CompT = 'MotionControllerComponent' ;
```

3.5.2 Robot capabilities

There are three possibilities to express that a capability is available on a robot: Either it is asserted to be available for the whole class of robots, for a specific robot instance, or it can be concluded that the capability should be available because all specified dependencies are fulfilled. Capability dependencies can be described using OWL restrictions that do not only specify the type of a required component or capability, but can further specify the required properties in more detail.

```
% capability asserted for robot instance
cap_available_on_robot(?Cap, ?Robot) :-
    class_properties(?Robot, hasCapability, ?Cap).

% capability asserted for robot class
cap_available_on_robot(?Cap, ?Robot) :-
    rdfs_individual_of(?Robot, ?RobotClass),
    class_properties(?RobotClass, hasCapability, ?Cap).

% capability depends only on available components or capabilities
cap_available_on_robot(?Cap, ?Robot) :-

    rdfs_subclass_of(?Cap, 'Capability'),

    forall( class_properties(?Cap, dependsOnComponent, ?CompT),
            comp_type_available(?Robot, ?CompT) ),

    forall( class_properties(?Cap, dependsOnCapability, ?SubCap),
            cap_available_on_robot(?SubCap, ?Robot) ).
```

3.5.2.1 Dynamic computation of capabilities

Which capabilities are actually available on a robot often depends on the current situation, e.g. which software modules are started. In the original SRDL implementation, the capabilities of a robot were either statically asserted, or the system could infer a capability to be available if it did not have any unmet dependencies. We extended the system by adding a third alternative: computing the currently available capabilities using computables.

The computables read information from the robot's communication middleware to determine which software packages are started and which services are available. We extended the description of capability classes with sets of indicators that can be used to check whether the respective capability can be assumed to be available. In the ROS middleware, for example, the *actionlib* interface[4] provides an abstraction layer that allows to easily check if certain kinds of capabilities are offered. If for instance the *move_base* action is available, one can assume that the robot has the capability to move in 2D while avoiding obstacles, and include this capability into the reasoning process.

Apart from a more flexible adaptation to changing software configurations, this system has the additional advantage that the setup of a new robot that uses standard software components becomes much easier: The robot's URDF description can be converted automatically, many higher-level capabilities can be inferred from the software infrastructure. Obviously, this requires the system to know the kind of software running on the robot, but since there is a trend towards using standard software for common skills like navigation, this will become less and less of a problem.

3.5.3 Action requirements

To check whether an action can be performed, its prerequisites need to be made explicit in a way that they can be checked against the robot's capabilities. In our system, actions can both depend on capabilities (like an object recognition capability) and on specific components (like an object recognition model for the specific kind of object that an action interacts with). Action requirements are usually specified on the most abstract level possible, for instance for the whole class of arm movements which all depend on some *ArmComponent*. Plans, on the other hand, will normally be described using more specific action classes that are derived from these generic ones and inherit their requirements. Therefore, the description of the plan can focus on the description of the action itself and does not have to deal with specifying the requirements. The

[4] http://www.ros.org/wiki/actionlib

set of capability dependencies of an action is composed of capabilities the action itself depends on and the capability dependencies defined for any sub-action:

```
required_cap_for_action(?Action, ?Cap) :-
    class_properties(?Action, dependsOnCapability, ?Cap).

required_cap_for_action(?Action, ?Cap) :-
    plan_subevents_recursive(?Action, ?SubAction),
    class_properties(?SubAction, dependsOnCapability, ?Cap).
```

Component dependencies of an action can either be described directly for the action, or indirectly via capabilities that are required for the action and which depend on some components. They can be queried using

```
?- required_comp_for_action('TableSetting', ?Comp).
Comp = 'MobileBase' ;
Comp = 'ArmMotionController' ;
Comp = 'ArmComponent' ;
```

Like capabilities, action requirements can also be computed automatically: Assuming that a robot needs a method to recognize each object it manipulates, a dependency on a suitable object recognition model can be added to each action.

3.5.4 Matching requirements to capabilities

Using the methods described in the previous sections, the system can determine which capabilities and which components are available, and by comparing them to the requirements of an action, determine which ones are missing for the action to be executable. A missing capability is thus defined as one that is required by an action, but not provided by the robot:

```
missing_cap_for_action(?Action, ?Robot, ?Cap) :-
    required_cap_for_action(?Action, ?Cap),
    \+ cap_available_on_robot(?Cap, ?Robot).
```

Missing dependencies can be queried using

```
?- missing_cap_for_action('TableSetting', 'TUM_Rosie1', ?Cap).
Cap = 'PuttingDownAnObjectCapability' ;
Cap = 'PickingUpAnObjectCapability' ;

?- missing_comp_for_action('TableSetting', 'TUM_Rosie1', ?Comp).
false.
```

Missing capabilities can often be provided by launching a software program, e.g. for object recognition, while missing software components can sometimes be downloaded from the Web (see Section 6.3).

3.6 Discussion and related work

In this chapter, we described our representations of actions, events, objects, environments and robot components and will now discuss how they relate to other methods in the literature. Already decades ago, projects like Cyc [Lenat, 1995] or SUMO [Niles and Pease, 2001] started to collect encyclopedic knowledge on a large scale in order to build a general upper ontology. To this end, researchers manually encoded very large amounts of knowledge in a machine-understandable format, usually variants of first-order logic. Recent efforts tried to automate the construction of knowledge bases by extracting encyclopedic knowledge from sources like Wikipedia [Wu and Weld, 2007; Suchanek et al., 2007], mainly focusing on already structured pieces of information such as categories and info-boxes. However, the knowledge bases mainly contain information about people and historic events and are thus not directly useful for robot manipulation tasks.

Cyc and SUMO have become huge, covering a wide range of phenomena. However, this increase in size came at a cost: Inference became rather slow, and ambiguities were created by adding knowledge – much of which a robot will never use. For example, "center" will mostly be meant as a spatial concept in a household robot context, not as a position in American Football. So neither Cyc nor SUMO are specialized for robotics, but were developed with the intention of understanding texts. For robot applications, it is thus often desirable to have less broad but deeper knowledge of the domain the robot is working in, like descriptions of different grasps or the concept of a "manipulation position" as the location where the robot should stand to manipulate objects.

In addition to the required knowledge, robots also need appropriate representations and reasoning methods for temporal reasoning, projection, and change modeling. Work coming from the area of knowledge representation, using robotics as application scenario, is unfortunately often over-formalized while lacking features needed in real-life scenarios: [Thielscher, 2000], for example, does not support any temporal reasoning, no detailed spatial representations, and neither information about object types nor about processes in the environment.

This was the motivation to develop specialized knowledge bases for autonomous robots, like KNOWROB or ORO [Lemaignan et al., 2010]. In KNOWROB, we integrate solutions from many areas into a coherent, implemented system. The upper ontology is partly taken from Cyc [Lenat, 1995], which emerged as quasi-standard, was extended with robotics-specific concepts and partly adapted to the limited expressiveness of description logics, our underlying representation formalism. The description logics way of representing knowledge in terms of classes and instances allows to separate between general principles and entity-specific information, with clear seman-

tics for the instantiation of a concept, for object instances as well as for actions and processes. These representations have been combined with powerful spatio-temporal representations and reasoning methods. Procedural attachments compute qualitative spatial and temporal relations. The temporal relations are realized according to Allen's interval calculus [Allen, 1983].

Object poses, the object properties that are changing most frequently in mobile manipulation, are represented by the event that created the robot's belief. Compared to similar approaches like the Fluent calculus [Thielscher, 1998], our representation carries additional information like the types of the "fluent" instances. This allows to further interpret the information, and also gracefully handles the description of multiple possible worlds in a single knowledge base. Describing different world states that are the results of different perception or prediction methods is possible without causing conflicts in the knowledge base.

The changes in object poses described before do not affect the objects by themselves: The objects remain the same, only their location changes. More substantial changes to objects can however happen, especially in the context of cooking activities, and are described using the actions or processes that induced these changes. This representation can describe the creation, destruction, aggregation and modification of objects in relation to the respective actions and processes. The methods for modeling processes are strongly inspired by the Qualitative Process Theory [Forbus, 1984], which we extended with actions as goal-directed intentional activities.

The action representation as hierarchical partially-ordered plans with prerequisites, effects, and temporal information is powerful enough to describe many real-world everyday activities, and is more expressive than for example STRIPS [Fikes and Nilsson, 1971] and rather related to hierarchical task network representations [Erol et al., 1994].

The description of inputs and outputs, however, is somewhat limited by the capabilities of OWL, and could be improved. Especially the lack of variables in description logics prevents the detailed description of the changes an action causes to the involved objects. This limitation has been overcome by the two-fold representation combining OWL descriptions with effect axioms in Prolog. These projection methods are still quite simple and mainly intended to be used in some very coarse-grained projection. They do however provide a well-defined interface to more sophisticated inference engines, for example using physical simulation [Kunze et al., 2011a], to predict the effects of actions with higher accuracy.

Since the representation of change is crucial for robots acting in real-world environments, we conclude with a discussion of the general properties of our approach. [Shoham, 1985] discusses which requirements a representation of change needs to fulfill and coined them into ten main properties – all of which are realized in KNOWROB or can be added with limited effort.

- **Interval semantics:** KNOWROB represents time in terms of intervals and supports reasoning about them.
- **Continuous change:** At the moment, continuous changes like a slowly filling container or the motion of an object are not explicitly represented. They could, however, be handled by the projection rules as well as by an a-posteriori analysis that interpolates between observations, for example of different poses of a moving object.
- **No inter-frame problem:** The question which parts of the world state change when performing an action is addressed by making the persistence assumption (the world remains static unless we conclude otherwise). Side-effects like changes in qualitative spatial relations induced by moving an object are avoided by computing these relations on demand based on the object's current position.
- **Concurrent actions:** There is no restriction preventing concurrent actions.
- **No intra-frame problem:** This problem refers to the influences of concurrent actions on each other. If two actions are performed on the same object, they can modify or annihilate the effect of the respective other action. Taking such side-effects into account is possible by checking for them in the action projection rules.
- **Suppressed causation:** Events that end without any apparent reason or that start at some later time, seemingly not directly triggered by an action, can be caused by incomplete knowledge about the processes taking place. They can be described and predicted by the projection rules.
- **Possible worlds:** The representation based on *MentalEvents* allows to describe multiple worlds like the results of different projection methods, the current and past world states, planned states etc. Since all of them are described in the same system, they can be compared inside KNOWROB, and can even be mixed to describe that an object is perceived at one location and intended to be somewhere else.
- **No cross-world identification problem:** This problem does not apply since KNOWROB represents different projected worlds inside the same system.
- **Modularity:** Adding new laws of change is easily realized by providing additional projection rules for actions and processes. Their results are combined with those the older rules generated.
- **Computational framework:** KNOWROB exists as implemented computational framework, including the methods for representing changes, for building up these representations based on the information the robot perceives, and for performing reasoning on these structures.

Chapter 4

Knowledge acquisition from the WWW

Skillfully performing everyday manipulation tasks like setting a table, doing the dishes, or cooking simple meals is highly challenging for autonomous robots and proved to be an extremely knowledge-intensive problem. Solving these problems means to scale the capabilities of today's robots in several dimensions: the number of tasks to perform, the parameters of actions (like different grasps to be used in different circumstances), the number of objects and required knowledge about their properties, and the necessary knowledge about the environment, to just name a few.

Figure 4.1 Performing complex tasks like making pancakes requires a lot of knowledge. In this section, we present some approaches to acquire such knowledge from public web resources.

The classical approach to solving such robot tasks is to generate a plan as a sequence of atomic actions which leads from the initial state to the goal state, taking the prerequisites and effects of the actions into account. The more complex a task is, the more difficult this gets: For common household activities, both the start and goal state are underspecified, there is no complete description of the prerequisites and effects of actions, and often, the mere generation of a sequence of actions is not sufficient, but more complex control structures including parallelism, timing etc. are needed.

Instead of generating a plan in this way, *web-enabled robots* can simply look up the appropriate courses of action on the web, making use of websites like *wikihow.com*. After having read the instructions and transformed them into a plan, the robot must find the ingredients and tools to

perform the task. To do so, it can use other websites such as online shops to find out how an object looks like. The robot also has to know the properties of these objects, e.g. if they are perishable, in order to decide where to search for them or how to handle them. Such information can also be found on the web, often on the same page as the product picture.

The web thus provides plenty of knowledge a robot can use to accomplish everyday tasks: *ehow.com* and *wikihow.com* contain thousands of step-by-step instructions for everyday activities like cracking and separating an egg, cooking different types of omelets, etc: about 92,000 on *wikihow.com* and even more than 1.5 million articles on *ehow.com*. Lexical databases like *wordnet.princeton.edu* group verbs, adverbs and nouns semantically into sets of synonyms (synsets), which are linked to concepts in encyclopedic knowledge bases like *opencyc.org*. These encyclopedic knowledge bases, which are represented in a variant of first-order logic, contain an abundance of knowledge about concepts such as eggs. They can inform the robot about the nutrition facts of eggs or tell it that eggs are products of birds and fish. However, they typically lack action-related information, e.g. the information that eggs can break easily and that they should be cooked or baked before consumption. This kind of knowledge is available at other websites such as the OpenMind Indoor Commonsense website (*openmind.hri-us.com*). But the web does not only contain knowledge about object usage, a robot could also retrieve appearances of objects (*images.google.com*, *germandeli.com*) and even geometric models (*sketchup.google.com/3dwarehouse*).

The following sections describe how robots can make use of these sources of knowledge, in particular how they can translate natural-language task instructions into executable robot plans, and how they can automatically generate large ontologies from information found on online shopping websites.

In the context of the pancake scenario, the work presented in this chapter serves for generating the initial, incomplete task descriptions that is used as a starting point to generate an effective robot plan. The object information derived from on-line shopping websites can then be used to generate missing object models, and to reason about storage locations of objects that need to be retrieved.

4.1 Task instructions from the WWW

In this section, we will present our approach to translating natural-language task instructions into formal representations that can be used by the robot. The resulting formal task descriptions need to be completed by adding missing actions, missing object information, and by inferring the right action parameters as described in the other sections. The different steps from the instruction in natural language to an executable plan are in the following explained using the example sentence "Place the cup on the table". Figure 4.2 explains the structure of our system, which is also described in [Tenorth et al., 2010b].

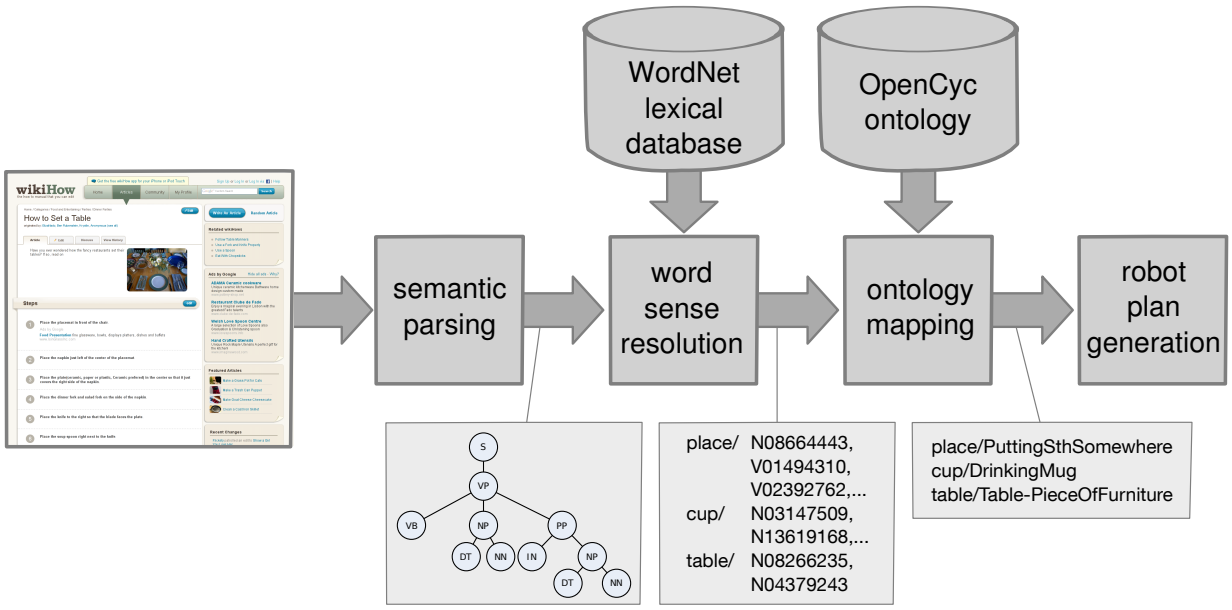


Figure 4.2 Overview of the import procedure for natural-language task instructions. After determining the syntactic structure, the system resolves the meaning of the words and builds up a formal task representation which can afterwards be transformed into an executable robot plan.

4.1.1 Semantic parsing

The first step after obtaining the natural-language instructions from the web site is to determine the parts of speech and the sentence structure using a natural-language parser. We employ the Stanford parser, an open-source Probabilistic Context Free Grammar (PCFG) parser [Klein and Manning, 2003], to perform the part-of-speech tagging and to generate the syntax tree (see Figure 4.3 left). Starting from this syntax tree, increasingly complex semantic concepts are generated in a bottom-up fashion using transformation rules similar to the approach in [Kate et al., 2005].

The leaves of the parse tree are words *Word(label, pos, synsets)*, consisting of a label, a part-of-speech (POS) tag and the synsets they belong to (see Section 4.1.2). Examples of POS tags are *NN* for a noun, *JJ* for an adjective or *CD* for a cardinal number. In the following, an underscore denotes a wildcard slot that can be filled with an arbitrary value.

Words can be accumulated to more complex structures, for example to quantifiers *Quant(Word(_,CD,_),Word(_,NN,_))* consisting of a cardinal number and a unit, or to objects *Obj(Word(_,NN,_),Word(_,JJ,_), Prep, Quant)* that are described by a noun, an adjective, prepositional statements and quantifiers. A prepositional phrase contains a preposition word and an object instance *Prep(Word(_,IN,_),Obj)*, and an instruction is described as *Instr(Word(_,VB,_), Obj, Prep, Word(_,CD,_))* with a verb, objects, prepositional postconditions and time constraints.

This approach allows to represent complex relations like "to the left of the top left corner of the place mat" by recursively combining these elements. Figure 4.3 exemplarily shows how the parse tree is translated into two *Obj* instances, one *Prep* and one *Instr*.

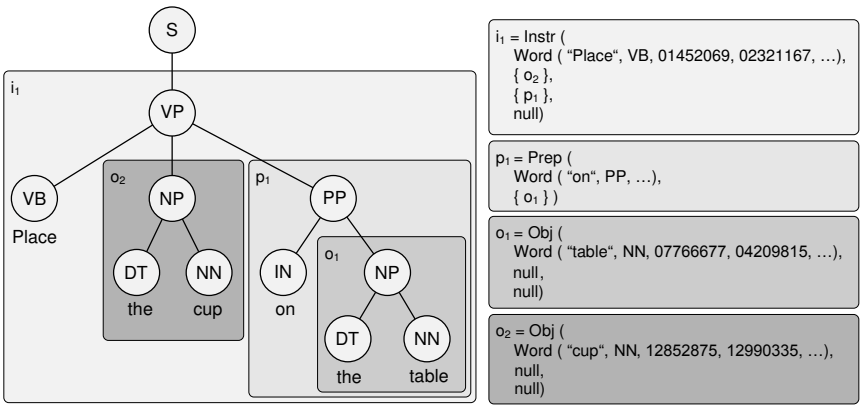


Figure 4.3 Parse tree for the sentence "Place the cup on the table" (left) and the resulting data structures representing the instruction that are created as an intermediate representation by our algorithm (right).

In an automatic post-processing of the generated data structures, object names consisting of multiple words (like "stove top"), phrasal verbs (like "turn on"), and pronominal anaphora (references using pronouns like "it") are resolved. Currently, we assume that "it" always refers to the last mentioned object, which proved to be a sensible heuristic in most cases. The system also handles conjunctions and alternative instructions ("and", "or"), negations, and sentences starting with modal verbs like "You should...", as long as the remainder of the sentence is an imperative statement. The slight difference in meaning presented by the modal verbs cannot be represented in the robot plan language and is therefore currently ignored.

4.1.2 Word sense retrieval and disambiguation

After this first processing step, the system has identified the structure of the instructions (i.e. the described actions, objects, prepositions, and their relations), but the instructions are still described using the original English words. The next step is to resolve their meaning by mapping these words to the corresponding concepts in the knowledge base. This mapping procedure combines the WordNet lexical database [Fellbaum, 1998] and the OpenCyc ontology [Matuszek et al., 2006]. The concepts in the KNOWROB ontology are largely derived from OpenCyc, i.e. they have the same identifiers and taxonomy, so the result of the mapping procedure can be used to represent the instructions both in KNOWROB and OpenCyc.

The WordNet database assigns words to one or multiple "synsets", corresponding to the different meanings a word can have. For thousands of synsets, there exist mappings from the synset in WordNet to the corresponding ontological concept in OpenCyc. "Cup" as a noun, for instance, is part of the synsets *N03033513* and *N12852875*, which are mapped to the concepts *DrinkingMug* and *Cup-UnitOfVolume* respectively.

Most words are part of multiple synsets, so a word sense disambiguation method has to select one of them. We use a very simple algorithm that is based on the observation that the word sense of the action verb is strongly related to the prepositions (e.g. "taking something from" as *TakingSomething* up vs. "taking something to" as *PuttingSomethingSomewhere*). Let $concepts(w)$ be the set of ontological concepts to which the word w could be mapped. For a single instruction (a_i, o_i, p_i) consisting of an action verb a_i, an object o_i and a set of prepositions $p_i \subseteq \{on, in, to, from, of, next\_to, with, without\}$, we are interested in the most probable pair of concepts $(A_i, O_i) \in concepts(a_i) \times concepts(o_i)$. Because the most appropriate concept for the action is, as mentioned above, largely dependent on the prepositions it co-occurs with, whereas it is reasonable to assume that the object sense is independent of the prepositions given the action sense, we compute the pair by maximizing

$$\begin{aligned} P(O_i, A_i \mid p_i) &= P(O_i|A_i) \cdot P(A_i|p_i) \\ &\propto \frac{P(O_i, A_i)}{P(A_i)} \cdot P(A_i, p_i) \end{aligned} \tag{4.1}$$

The required probability values appearing in the above formulas are determined based on a manually annotated training set. If there is no statistical evidence about any sense of a word, the algorithm chooses the meaning with the highest frequency rank in WordNet.

4.1.3 Formal instruction representation

With the ontological concepts being resolved, the instructions can be formally represented by creating the action description in KNOWROB that was introduced in Section 3.3. Action verbs are translated into specializations of the corresponding action concept in the ontology, e.g. subclasses of *PuttingSomethingSomewhere*. These action classes are further described by restrictions generated from the rest of the instructions: Locations where an object shall be taken from or where it shall be placed are described as restrictions on the *fromLocation* or *toLocation*. The object that is to be manipulated is linked by the *objectActedOn* relation. Temporal constraints, like waiting for a certain amount of time, are represented by the *WaitForPredefinedTimeInterval* action with a specified duration.

This formal instruction representation can be used for reasoning about the robot's capabilities with respect to this task (Section 3.5), for exchanging task instructions (Section 6.3), or for matching them against observed human behavior (Section 7.5). For executing the instructions on a robot, they have to be converted into the robot's plan language.

4.1.4 Plan generation

The imported instructions can be translated into plans in the CRAM Plan Language (CPL, [Beetz et al., 2010a]). CPL is a language for programming cognition-enabled control systems which includes data structures, primitive control structures, tools and libraries that are specifically designed to enable and support mobile manipulation as well as cognition-enabled control. Besides KNOWROB, CPL is the second main component of the CRAM (Cognitive Robot Abstract Machine) architecture.

CPL allows to describe abstract, high-level plans that are, during execution, decomposed into smaller, lower-level plans. For being able to successfully execute tasks based on the imported web instructions, the robot needs to have a library of plans for basic tasks like picking up objects or navigating to a position inside the environment, which we assume to exist.

In CPL, objects and locations are described by designators, qualitative specifications which are resolved during the plan execution. Designators do not directly reference specific objects in the environment, but rather describe necessary properties an object needs to have. This approach makes the plans more flexible because the robot can postpone the decision which objects to use until they are finally needed, and take the decision based on all information that is available at that moment. The designator concept also matches well with the action and object representation in KNOWROB, where object and location properties are described by restrictions that are concep-

tually similar to designators. The example sentence “Place the cup on the table” gets translated into the following plan, which first defines the designators *cup1*, *table1*, and *location1*, and then creates an *achieve* statement to specify that the *loc* of *cup1* shall be *location1*.

```
(def-top-level-plan put-cup-on-table1 ()
  (with-designators (
        (cup1       (an object   '((type cup))))
        (table1     (an object   '((type table))))
        (location1  (a location '((on ,table1)
                                  (of ,cup1)))))
  (achieve '(loc ,cup1 ,location1))))
```

A more complex example are the following instructions for making pancakes. These instructions have been used in the “making pancakes” demonstration during the CoTeSys workshop in October 2010 to show the system’s capability to make use of natural-language information from the Web.

1. Take the pancake mix from the refrigerator
2. Pour the mix into the frying pan.
3. Wait for 3 minutes.
4. Flip the pancake around.
5. Wait for 3 minutes.
6. Place the pancake onto a plate.

In the resulting plan, the six commands have been translated into four *achieve* and two *sleep* statements. Objects are described by their respective types, and locations relative to objects like in the original instructions.

```
(def-top-level-plan ehow-make-pancakes1 ()
    (with-designators (
          (refrigerator2 (an object '((type refrigerator))))
          (location1 (a location '((in ,refrigerator2)
                                   (of ,mixforbakedgoods2))))
          (mixforbakedgoods2 (some stuff '((type pancake-mix) (at ,location1))))

          (cookingvessel2 (an object '((type pan))))
          (location2 (a location '((in ,cookingvessel2)
                                   (of ,mixforbakedgoods2))))

          (pancake2 (an object '((type pancake))))

          (dinnerplate2 (an object '((type plate))))
          (location0 (a location '((on ,dinnerplate2)
                                   (for ,pancake2)))))

  (achieve '(object-in-hand ,mixforbakedgoods2))
  (achieve '(container-content-transfilled
                              ,mixforbakedgoods2
                              ,location2))
  (sleep 180)
  (achieve '(object-flipped ,pancake2))
  (sleep 180)
  (achieve '(loc ,pancake2 ,location0)))))
```

4.1.5 Plan debugging and optimization

As mentioned in the scenario description, the result of the instruction import procedure is not yet an effective task description that contains all information needed for actually executing the task. The natural-language instructions have been written by humans in a way that other humans can understand them, which often means that information that seems obvious has been omitted. Robots that are to execute these tasks need to detect and fill these gaps using different kinds of knowledge: Common-sense knowledge bases provide information about missing parameters or potential problems [Kunze et al., 2010], projecting the effects of actions using the methods described in Section 3.3.4 or executing the plan in a physics simulation can indicate plan flaws related to unexpected collisions [Mösenlechner and Beetz, 2009]. When such flaws are detected, the robot can try to acquire the information by downloading missing object models or by including missing actions into the plan. Section 7.3 describes in more detail how the different pieces of knowledge and inference methods are combined to complete the task instructions.

4.1.6 Evaluation

We tested the implemented system on 88 instructions from a training set and another 64 from a test set of howtos which are taken from `ehow.com` and `wikihow.com`[1]. Since many of the errors are caused by the syntax parser, we evaluate the system both with automatically parsed syntax trees and manually created ones in order to better show the performance of the other components. For the training set, we manually added 72 missing mappings from WordNet synsets to Cyc concepts; the test set was transformed without such manual intervention. We manually

[1]The complete training and test set can be downloaded from http://ias.cs.tum.edu/∼tenorth/icra10_ehow.txt

	aut. parsed		**man. parsed**	
Training Set:				
Actual Instructions	**88**	**100%**	**88**	**100%**
Correctly Recognized	59	67%	72	82%
False Negative	29	33%	16	18%
False Positive	4	5%	2	2%
Test Set:				
Actual Instructions	**64**	**100%**	**64**	**100%**
Correctly Recognized	44	69%	58	91%
False Negative	20	31%	6	9%
False Positive	3	5%	6	9%

Table 4.1 Summary of the evaluation on instruction level

determined the translation correctness by verifying that all relevant information from the natural language instruction was transformed into the formal representation.

First, we trained the disambiguator on the training set using manually created parse trees. Afterwards, we ran the system including the syntax parser on the same set of howtos, the results are shown in the upper part of Table 4.1. With correct parse trees, the system achieves a recognition rate of 82% on the training set and even 91% on the test set before the ontology mapping and the transformation of the instructions into the formal representation.

The remaining 18% resp. 9% have either been recognized incorrectly (missing object or preposition in the instruction) or not at all. The latter group also comprises instructions that are not expressed as imperative statements and, as such, are not supported by the current implementation. In both test runs, errors caused by the syntax parser result in a significant decrease in the recognition rate (15 percentage points in the training set, 22 in the test set).

Table 4.2 shows the results of the translation into the formal instruction representation. In the training set, 70 of the 72 instructions which have been recognized in the previous step could

	aut. parsed		**man. parsed**	
Training Set:				
Actual Instructions	**88**	**100%**	**88**	**100%**
Import Failures	**31**	**35%**	**18**	**20%**
Incorrectly/Not recognized	29	94%	16	89%
Missing WordNet entries	0		0	
caused Import Failures	0	0%	0	0%
Missing Cyc Mappings	0			
caused Import Failures	0	0%	0	0%
Misc. Import Errors	2	6%	2	11%
Disambiguation Errors	0		0	
Correctly imported into KB	**57**	**65%**	**70**	**80%**
.				
Test Set:				
Actual Instructions	**64**	**100%**	**64**	**100%**
Import Failures	**33**	**52%**	**28**	**44%**
Incorrectly/not recognized	20	61%	6	21%
Missing WordNet entries	3		3	
caused Import Failures	2	6%	2	7%
Missing Cyc Mappings	14		23	
caused Import Failures	11	33%	20	71%
Misc. Import Errors	0	0%	0	0%
Disambiguation Errors	2		3	
Correctly imported into KB	**31**	**48%**	**36**	**56%**

Table 4.2 Summary of the evaluation on knowledge base level

successfully be transformed, the two errors were caused by mappings of word senses to concepts that cannot be instantiated as objects in Cyc: the concept *PhysicalAmountSlot* in the commands "Use the amount that..." and the relation *half* in "Slice in half".

The results of the translation of the test set show that two external components are the main sources of error: 40% of the import failures are caused by the syntax parser, since a decrease from 61% to 21% of failures in the initial recognition step can be observed when switching to manually created syntax trees. In this case, missing Cyc mappings and WordNet entries are the main problem, causing about 78% of the remaining errors.

Test set of Howtos	**Instr. Level**	**KB Level**	**KB+maps**
How to Set a Table	100%	100%	100%
How to Wash Dishes	92%	46%	62%
How to Make a Pancake	93%	73%	81%
How to Make Ice Coffee	88%	63%	88%
How to Boil an Egg	78%	33%	57%

Table 4.3 Per-Howto evaluation of the import procedure.

An evaluation per howto (Table 4.3) shows that a reasonably large number of the instructions can be recognized correctly. The last column contains the results after having added in total eight mappings, including very common ones like *Saucepan* or *Carafe*, which will also be useful for many other instructions. The generation of a robot plan from the formally represented instruction is a simple translation from Cyc concepts to CPL statements which did not produce any further errors.

4.2 Object properties

For executing imported task instructions, robots need information about the referenced objects: On the one hand, they need to be able to recognize them, and on the other hand, they are supposed to handle them correctly, meaning they need to have at least basic knowledge about what these objects are. In the context of the scenario, the methods presented here can be used to generate object models that are needed for the task at hand; the knowledge about object properties is further useful to reason about where to search for objects. A large source of information about commercial products are online shopping websites: [Pangercic et al., 2011] show how images obtained from such websites can enable robots to recognize these objects in real life. They mined product pictures from the germandeli.com shopping website, extracted image features and learned recognition models that can be used to detect these objects in the image stream from the robot's camera.

We complement the system by additionally importing information about the object types and their properties, extracted from the website's category structure and from information on the product pages. To this end, we implemented a system that automatically translates the category structure of the website into a subclass structure in the knowledge base: For example, *Dallmayr Prodomo Coffee* is represented as a sub-class of *Dallmayr coffee*, *Coffee (German Brands)*, *Beverages*, and finally *Groceries*.

In addition to the category structure, online shops also provide detailed descriptions of the properties of products, such as pictures, the perishability status, price, ingredients, etc. Often, this information is already presented in a semi-structured way in form of tables or image icons. This information can easily and automatically be extracted and added to the knowledge base as properties of the respective object classes.

Using only the germandeli.com website, we generated an ontology of more than 7,000 object classes (Figure 4.4) which the robot can both recognize and reason about. For most of the objects, there is also information about the weight, the price, the brand, the country of origin, and product IDs like the EAN number (European Article Number) that link to external databases. Depending on the product, there can also be information if it is frozen, perishable (e.g. dairy products), or heat sensitive (e.g. chocolate products). In this case, the semantic information and the pictures that were used to construct the recognition model originate from the same source, so they can easily be combined and allow the robot not only to recognize objects, but also to know their properties and relations to other objects. Otherwise, establishing this correspondence can be very difficult since the products in an online shop are called slightly different from objects in WordNet, OpenCyc, ehow.com or other sources.

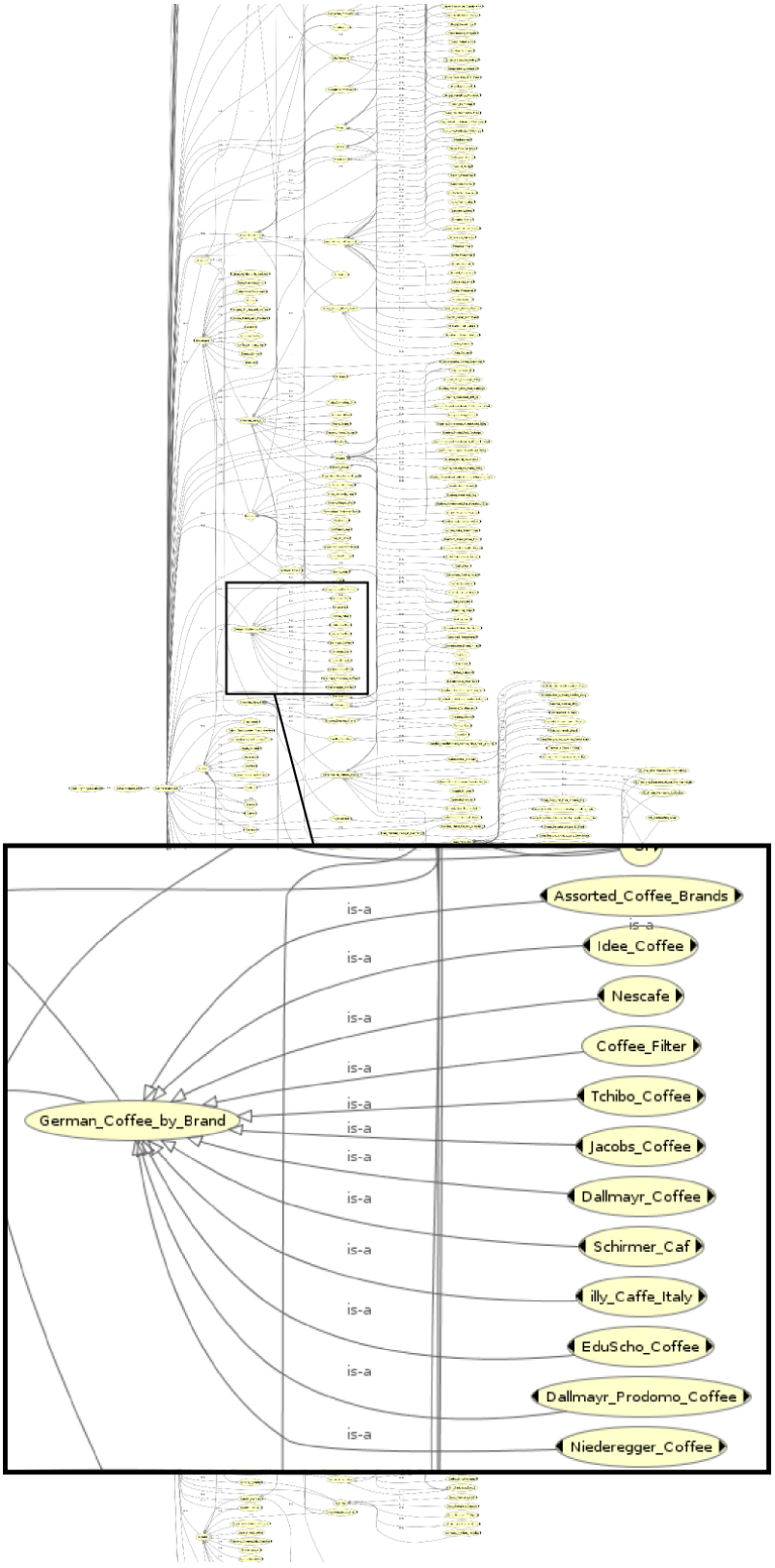


Figure 4.4 Ontology of about 7,000 object classes generated from information on the german-deli.com shopping website

This procedure allows to semi-automatically create large, domain-specific ontologies providing information about the objects a robot has to know about. Some adaptation to different web sites, some matching rules or some links to the existing ontology need to be added manually, but in general, the semi-automatic import greatly speeds up the generation of large knowledge bases.

4.3 Discussion and related work

On the one hand, the World Wide Web is the biggest resource of knowledge that has ever been available to a robot, consisting of billions of pages that cover a huge range of topics for many different audiences, and which are all, in principle, machine-readable: digital text, pictures, and videos. On the other hand, most web pages are intended to be used by humans – that is, they are written in different natural languages and in a way that humans find them convenient to read. This makes it difficult for machines to use the information because they first need to understand the meaning of the words and sentences in natural language.

The semantic web initiative [Lee et al., 2001] was founded to overcome this problem by creating a world wide web for machines. In the semantic web, information is encoded in machine-readable form instead of natural-language text, i.e. in a way that computers can retrieve, understand, relate and process the information in the documents. Briefly, the semantics of a document are not hidden in the text, but explicitly described in a logic-based format computers can understand. In theory, this allows computers to autonomously answer queries by searching the web for information. In practice, however, only a very small fraction of the information on the web is available in the Semantic Web or as web services [Paolucci et al., 2002], especially hardly any information required by autonomous robots in household environments. Therefore, we needed to develop techniques to translate the information from human-readable form – instructions in natural language, pictures, and 3D models of objects – into representations the robot can use – formally described knowledge, task descriptions and object models that can be used for recognition.

Unfortunately, most current work in web-mining and information retrieval does not provide these formal representations [Banko et al., 2008], mines information only from pre-structured sources like the Wikipedia infoboxes [Wu and Weld, 2007], or focuses on finding information for humans rather than understanding its content [Manning et al., 2008]. A closely related system by [Perkowitz et al., 2004] also used task descriptions from `ehow.com`, but only extracts sequences of object interactions for activity recognition purposes. For actually executing the instructions, robots need a much deeper understanding and need to convert the information from

the web into a formal representation to relate it to other pieces of knowledge. Kate et al. [Kate et al., 2005] use similar methods as we do for the semantic parsing, but do not apply them to web instructions and do not provide details of the knowledge processing and symbol grounding. Speech interfaces for robots also deal with the problem of converting natural language into commands, but can control the language that is being used and can thus afford to be quite limited in terms of vocabulary or allowed grammatical structures ([Zelek, 1997], [Tellex and Roy, 2006], [Mavridis and Roy, 2006]). [Smith and Arnold, 2009] recently presented very interesting results on the generation of hierarchical plans from short step-by-step instructions in the OpenMind Indoor Common Sense database. The system also translates the natural language instructions into a logical representation, though without ontology mapping. To the best of our knowledge ours is the first work that mines complex task descriptions from the web, maps natural-language words to ontological concepts, and translates the instructions into executable robot plans.

While our methods are, in general, a step towards increasing scalability, the question is how well they generalize to new instructions and how much adaptation is needed to translate them. The translation system itself is very general: It uses a generic natural-language parser, covers many different language constructs, and reads information about actions and objects from two of the largest knowledge bases that are currently available. However, different prerequisites are needed for understanding a novel set of instructions: First, all words need to be in WordNet (which usually is the case), and there needs to be a mapping from the WordNet synset to the respective ontological concept. Though there are already mappings for tens of thousands of words, the coverage for household topics is still not complete. Preliminary investigations showed that somewhat less than 500 action verbs cover the whole range of food preparation, so creating mappings for this rather small set of actions seems reasonable. Other requirements for successful translation are that the corresponding class in the ontology exists and that the robot has a routine for executing each of the lower-level actions.

As an example, the adaptations that were needed to use the translation system, which had originally been developed for the set of instructions in Table 4.3, in the pancake experiment were limited to adding a few mappings from WordNet to Cyc (e.g. for the pancake mix), while the rest of the conversion process worked without modifications.

Inferring information that is missing in the instructions remains an open challenge: For instance, an instruction for setting a table states that items have to be put in front of the chair, but does not require them to be on top of the table. Other instructions fail to mention that an oven has to be turned off after use. Robots will have to detect these gaps and fill them appropriately. To fill in this missing knowledge, we looked into the acquisition of common-sense knowledge from web

resources [Kunze et al., 2010]. This *common-sense knowledge* is completely obvious to humans and therefore usually not explicitly described. Humans assume that their communication partner also has this kind of knowledge and therefore usually omit such "obvious" information when explaining something. The problem is that, since it is obvious to humans, most of this knowledge is typically not written down because humans acquire it already in their early childhood. Therefore, such knowledge has to be collected specifically for robots. Instead of letting a small group of experts create a knowledge, projects like the OpenMind Common Sense [Singh et al., 2002] initiative collect such data from Internet users by presenting them incomplete sentences and letting them fill in the gaps. While the OpenMind project collects general common-sense knowledge, the OpenMind Indoor Common Sense project (OMICS [Gupta and Kochenderfer, 2004]) focuses on the kind of knowledge required by robots acting in indoor environments. The users' responses are saved in semi-structured form in a relational database as sentence fragments in natural language. Several projects have started to convert the information into representations that support reasoning, for instance ConceptNet [Liu and Singh, 2004] or LifeNet [Singh and Williams, 2003]. [Kunze et al., 2010] translated the knowledge from the sentences in natural language into a formal logical representation in the KNOWROB knowledge base.

The problem of acquiring large amounts of common-sense knowledge is still an unsolved issue. "Crowdsourcing" the collection by distributing the task to voluntary Internet users helps to scale the system but creates other challenges: Ambiguities in natural language are hard to resolve and even harder when looking at the short sentence fragments provided by OMICS. Relations are also interpreted completely differently by different people: A sentence fragment like "if A, then B" is interpreted as either immediate and inevitable effect (switching on a dishwasher changes its state from "off" to "on"), long-term effect (switching on a dishwasher results in clean dishes) or as indirectly related consequence (loading a dishwasher causes dishes to be clean – if, what is omitted, the soap is filled in, the hatch is closed and the device is turned on), or even as "implies" (dishes are clean if the dishwasher has been turned on). Furthermore, there are gaps in the provided knowledge due to the way it was collected: being presented a sentence with placeholders to be filled in, people tend to enter the most obvious information. Presenting the same template to many people does not guarantee better coverage; instead, obvious statements occur several times, less obvious ones hardly ever. Nevertheless, such common-sense databases are a very useful source of knowledge that can hardly be found elsewhere, and the translation into semantic networks or description logics turns them into a useful resource for autonomous robots.

Even if the translation process works fine, there remains the problem of making sure the robot does not behave unsafe. Safety issues always arise when using information from other sources. Humans can usually estimate quite well how reliable a source of information is, and robots will need to develop such skills as well. For the moment, we only use selected pieces of knowledge, i.e. only a few websites and manually selected instructions, and the robot discards instructions it could not understand. This is surely an overly conservative strategy, and we risk that the robot loses information that would be required to successfully perform a task. More research is needed to enable the robot to estimate how well it has understood an instruction and to ask clarifying questions if needed. However, using our conservative strategy, we avoid the problem of the robot trying to execute either malicious instructions or task descriptions that were misunderstood or not even meant to be executed (e.g. in fictional texts). Furthermore, we do not see the import of instructions as a necessarily completely autonomous process, but rather as a semi-autonomous translation that facilitates the generation of plans using knowledge from the web. In case of problems, the robot can still ask the human if the result is what he intended to get.

Chapter 5

Observation and analysis of human activities

Observations of human activities are an important source of information for robots, especially for robots in a household environment. Many kinds of information, like the right motions or trajectories to use for a certain action, are very hard to obtain otherwise.

Acquiring this information by observing humans is a sensible option since humans are the natural domain experts who easily, naturally, and skillfully perform tasks that are still very complex for a robot. Since they also perform most of these tasks on a daily basis, their motions are highly optimized and often show very stereotypical behavior. Figure 5.1, for example, shows the hand trajectories of five table-setting episodes in the upper part, and the trajectories for specific motions like reaching towards an object, picking up an object, and opening a cupboard door in the lower three pictures. While the overall trajectory looks rather unstructured, the motions show a significant stereotypical structure when performing goal-directed activities. This structure suggests that robots can learn important action-related information from observing humans, from the low level of grasps or trajectories, to different alternative ways of reaching for an object, sequences of motions to perform in order to achieve a goal, up to actions and their parameters on the task level. These observations are an especially important source of knowledge since much of the information they provide – trajectories, motions, accelerations, different alternatives for achieving a goal, or the selection of the right kind of motion in a given context – can hardly be obtained otherwise and are hard to describe verbally.

To use this information for their tasks, robots have to relate it to the rest of their knowledge: If a plan involves a *Reaching* motion and the robot would like to check how a human performed this motion, it has to be able to find the right information in the large amount of data it has observed, which requires expressive semantic descriptions of the data. To maximize the success

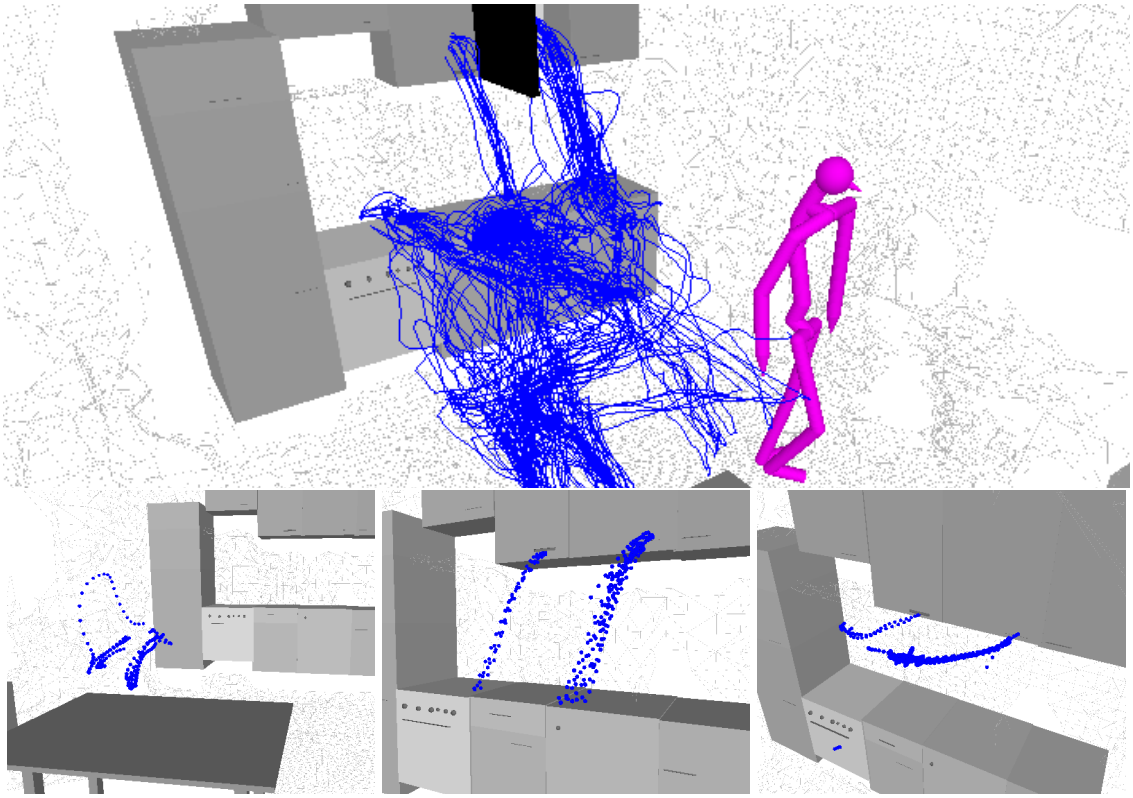


Figure 5.1 Upper picture: Hand trajectory data recorded in five table setting episodes. Lower pictures: Trajectories for picking an object from the table (left), for reaching towards a cupboard (center) and for opening a cupboard (right).

probability, a robot should select those that were performed on similar objects at a comparable position, performed with the same hand and maybe even by an agent of similar size. This raises the need for a formal description of the observation data, including semantic information like the agent that is performing a task or the object that is manipulated in an action, and for making it accessible to the robot's knowledge base.

Autonomously building up such a representation is a challenging task that involves the segmentation of continuous motions into segments with well-defined semantics, the fusion of the action data with information from other sensors in the environment to determine the objects the human interacts with, and the abstraction from the raw motions into higher-level action and task descriptions. The representation has to be powerful enough to deal with variations in human activities, from different ways of performing a motion up to different action sequences in an activity or different action parameters (e.g. different hands to perform the same action).

Having such a representation is not only beneficial for robots to learn from humans how to perform a task, it can also help to understand better how humans perform their everyday activities and to describe the impact of diseases that affect human motion and task execution capabilities like apraxia [Liepmann, 1920]. Apraxia is an impairment of motor planning skills that is caused by damage to certain brain areas. Patients lose the ability to perform motions and tasks they have already learned, though their muscular and sensory capabilities would still allow them to perform these tasks. This disorder is often observed after a stroke and results in a variety of complex effects including (list adapted from [Cooper et al., 2005]):

- Sequence errors: Addition, omission, or anticipation of a step in the action sequence
- Misuse: Actions are performed using an inappropriate object (e.g., hammering with a saw) or in the wrong way (e.g., cutting an orange with a knife as it if were butter)
- Mislocation: Actions are performed in completely the wrong place (e.g., pouring some liquid from the bottle onto the table rather than into the glass) or in a slightly wrong location (e.g., striking the match inside the matchbox)
- Tool omission: Using the hand instead of an obligatory tool (e.g., opening a bottle without using the bottle opener)
- Pantomime: Pantomiming the use of the tool/object instead of actually using it
- Perplexity: A delay or hesitation in performing an action
- Toying: Repeated touching or moving of an object without actually using it

In order to describe and detect such complex effects, a model needs to be able to describe both low-level motor defects, resulting in inappropriate motions, and high-level errors like a wrong sequence of actions or actions that are performed with the wrong tool or using the wrong object.

In this chapter, we present our methods for representing knowledge about human activities, for building up such a representation from observations, and for reasoning on these models. Our models allow to query for semantically specified observation data, to check if the observed activities comply with written instructions, and to compare the style how the activities are performed with models learned from observations. These observations may either be prior observations of the same subject, which can be used to detect changes in their performance, or observations of a whole group of subjects, which allows to assess how well a person performs in comparison to a reference group.

5.1 Automated models of everyday activities (AM-EvA)

Our approach to creating integrated models of human activities is the AM-EvA framework (Automated Models of Everyday Activities [Beetz et al., 2010c]). AM-EvA is designed as a system for automatically observing and interpreting human activities. It includes and integrates in a common reasoning framework different modules for the observation of human actions and motion tracking, for learning continuous models of the observed motions, for the segmentation and classification of the continuous movements, for the abstraction of the resulting action sequence, and for learning probabilistic relational models of complete activities.

Figure 5.2 illustrates the overall system architecture. Human motions are observed by a full-body motion tracking system and a sensor network which is embedded in the environment (see Section 5.2). The data has been manually annotated (Section 5.3) in order to train algorithms for segmenting and classifying the observed motions (Section 5.4). The result of the segmentation, a sequence of typed motion segments, is related to simultaneously observed events like object detections in order to determine action parameters, and then formally represented in terms of action instances in the KnowRob knowledge base. This combination of knowledge representation with classification methods allows to automatically generate formal descriptions of observed motions and perform logical reasoning about the observations. In order to further inspect the segments, for example to learn different alternatives how a kind of motion can be performed, the system applies methods for clustering trajectories and for characterizing them based on the context in which they are used (Section 5.5).

AM-EvA allows to interpret observed actions at very different levels of abstraction: The pose for performing an action, trajectories of different body parts, the sequence of motions to perform an action, different actions in an activity, and activity properties like the positions of objects or people involved are all accessible and semantically described. A central part of AM-EvA is the hierarchical representation of observed actions and the generation of this abstraction pyramid, which is described in Section 5.6. The pictures to the left and to the right of the pyramid in Figure 5.2 illustrate which information is available at different levels of description. Section 5.7 explains how activity-level models can be learned to describe the partial order of actions in a task as well as action parameters like the object or hand that are used.

In addition to the components described in this work, AM-EvA further comprises methods for tracking human motions [Bandouch et al., 2008] and for learning probabilistic activity models [Beetz et al., 2010c] from the observed and abstracted data.

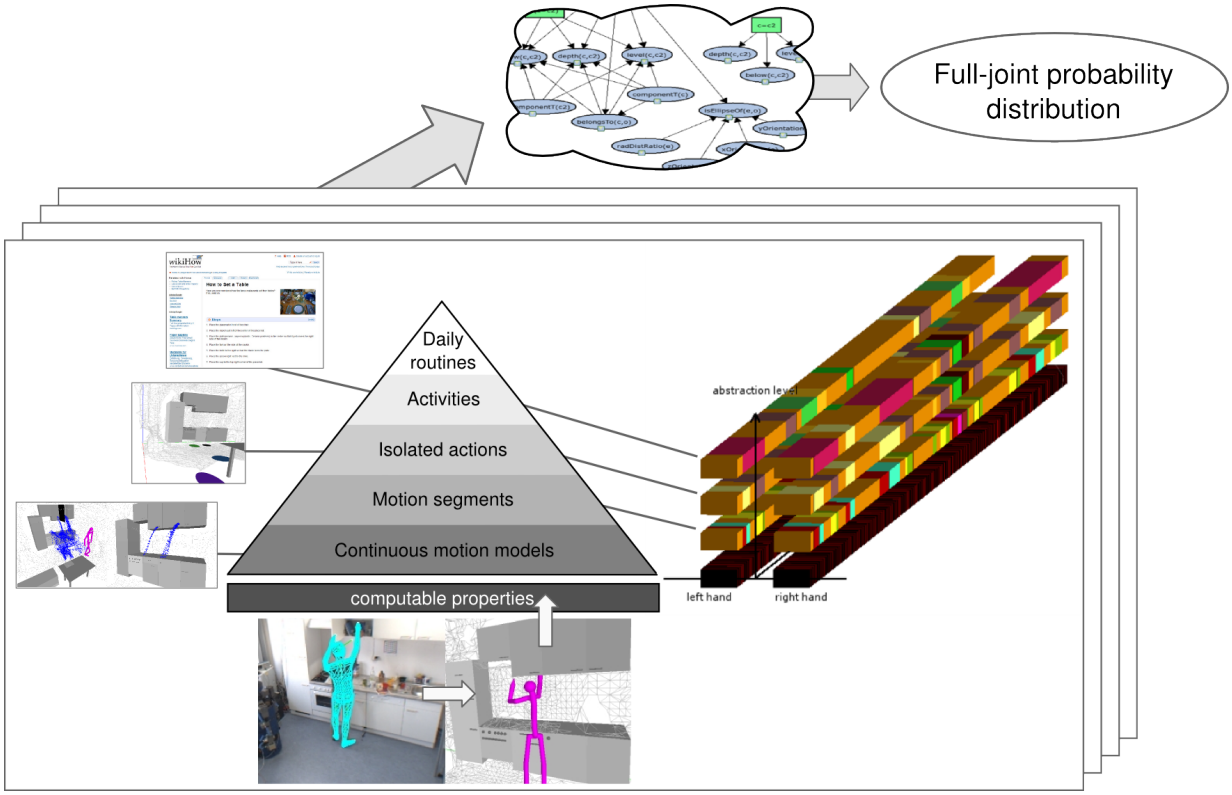


Figure 5.2 Activity observation, interpretation, and analysis using AM-EvAs. Motions are tracked from video data and descriptions at several levels of abstraction are generated that can be used for learning probabilistic relational models of human activities.

5.2 Data acquisition: The TUM Kitchen Data Set

The TUM Kitchen Data Set is the main source of data that is being used in this work and contains observations of several subjects setting a table in different ways (each recorded sequence is between 1-2 min). All subjects perform more or less the same activity, including the same objects and similar locations. Variations include differently ordered actions, a different distribution of tasks between the left and the right hand, and different poses that result from different body sizes. Since the subjects are performing the actions in a natural way – apart from the sometimes unnatural transport of only one object at a time – there are fluent transitions between sub-actions, and actions are performed in parallel using both the left and the right hand. Some subjects perform the activity like a robot would do, transporting the items one-by-one, other subjects behave more natural and grasp as many objects as they can at once. There are also a few sequences where the subject repetitively performed actions like picking up and putting down a cup (∼5 min each).

Due to considerable interest by other research groups, the data has been released in Octo-

ber 2009 and is publicly available for download at `http://kitchendata.cs.tum.edu`. Since then, it has generated more than 460 GB of downloads and attracts around 300 visitors a month, generating more than 10,000 hits. The data set is actively being used at several universities and already spawned several papers using the data. Users created additional annotations and data conversion tools and provided them back to the community.

5.2.0.1 Challenges

There are several requirements that make the acquisition of such a comprehensive data set a difficult task:

Non-intrusive data collection: Apart from privacy issues, monitoring systems in a household scenario should be as unintrusive as possible. Attaching accelerometers or markers, as required by many motion capture systems, to real people in their everyday lives, or asking them to wear special skin-tight clothes is certainly not feasible. Therefore, the system has to perform all analyses based on sensors embedded in the environment and has to be able to track people independently of their body size or their clothing.

Interaction with the environment: Opening doors, picking up objects or being occluded by pieces of furniture causes significant changes of the human silhouette that may well confuse most of today's state-of-the-art markerless motion capture systems.

Variation in performing activities: From the low motion level up to the sequence of actions, there are many degrees of freedom in how everyday activities can be performed (Figure 5.3). Both the motion capture system and the subsequent analysis methods need to be able to handle this high variability.

Continuous motions: People do not stop while performing everyday tasks, so there is no strict separation between single actions. To deal with this problem, sophisticated methods for motion segmentation are required, since actions can no longer be considered to be well-separated by default.

Parallelism: Humans often use both hands in parallel, while sometimes even walking at the same time (Figure 5.3). Actions performed with the left and the right hand start at different times, and may overlap both temporally and spatially. Sometimes actions are started with one and finished with the other hand, resulting in odd motion sequences for each hand.

Difference in body sizes: Differences in body size are not only challenging for the tracking system, but may also influence which motions are used to perform a task. For instance, we found that tall subjects put objects on the table mainly by flexing the spine instead of the elbow joint.

Required semantic information: For detailed analysis of the human activities, we need to observe the performed motions as well as semantic information like the identity of the objects and pieces of furniture the subject interacts with.
We therefore chose to use a markerless human motion tracking system (Section 5.2.0.2) that is able to estimate human motions performed in the context of everyday manipulation tasks, and to complement the motion capture data with data from sensors that are embedded in the environment (Section 5.2.0.3).

5.2.0.2 Human motion tracking

A system for the detailed and unintrusive observation of human motions is crucial for a thorough understanding of everyday manipulation activities. Commercial tracking systems typically rely on optical, magnetic or inertial markers attached to the human body for estimating its pose, which requires the instrumentation of the subjects. In addition, they often require quite a lot of post-processing of the generated data and are sensitive to occlusions as they occur when interacting with objects and the environment.

For recording the TUM Kitchen Data Set, we used the MeMoMan tracker that has been developed by Jan Bandouch, a markerless motion tracking system tailored towards application in everyday environments. Technical details of the tracking system can be found in [Bandouch and Beetz, 2009]. In the following, we only briefly describe the properties and advantages of the tracker with respect to our scenario.

Setting up the MeMoMan tracking system only requires to place cameras in the environment, no further modifications of the environment or instrumentation of the subjects are needed. For recording the TUM Kitchen Data Set, four fixed cameras were used, see Figure 5.3 for examples of the video images. Based on the silhouette of the observed human subjects, the tracker computes their full-body pose for each timestep based on an accurate 51 DOF articulated human model, providing both joint angles and joint positions in cartesian space (which can be used to derive trajectories of specific body parts, e.g. hand trajectories during a pick and place action). If the human silhouette is changed, e.g. when carrying objects or opening cupboards, this can be filtered out using an appearance model (Figure 5.4 right), which allows the system to accurately estimate the pose even in these challenging situations. Occlusions by static objects like the

Figure 5.3 The same action performed by different subjects. Notice the use of different hands for the same action, the fluent transitions or parallelism of actions (e.g. opening cupboard and grabbing cup), and the difference in motion based on the size of the human. Images taken from the TUM Kitchen Data Set [Tenorth et al., 2009].

kitchen table are handled by blocking masks that prevent the system from evaluating information that does not resemble the learned human appearance model.

Figure 5.4 shows the data used in the tracker: The leftmost picture is the inner skeleton model, which provides the input data for the subsequent processing steps (joint angles and cartesian joint positions). The center view is the outer model that is adapted to the shape of the human subject before the experiments, and the right view is the appearance model the tracker learns during operation.

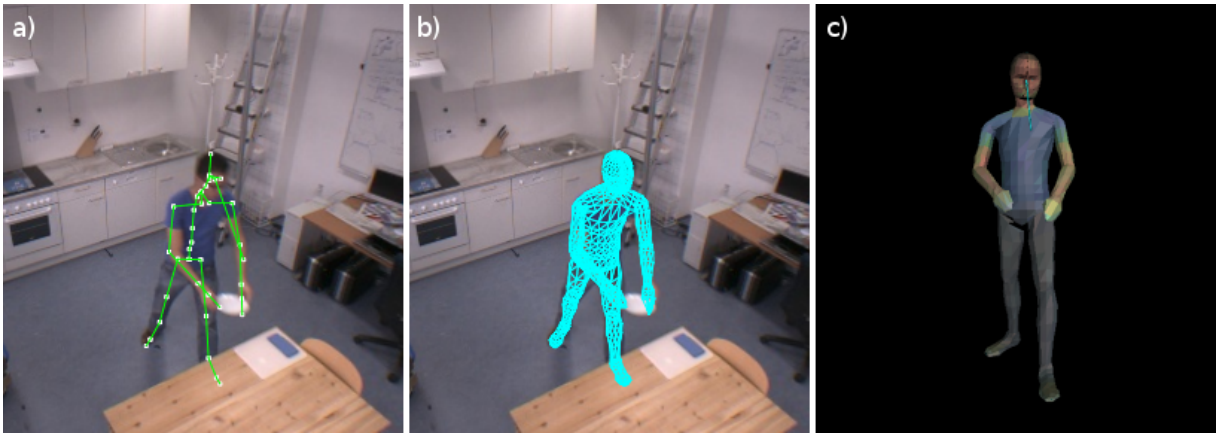


Figure 5.4 Human motion tracking (one of four cameras): a) inner model b) outer model c) virtual 3D view with appearance model. Images courtesy of Jan Bandouch [Bandouch and Beetz, 2009].

The recording of the video images and the actual tracking of the motion capture data has been done by Jan Bandouch, the creator of the MeMoMan tracker. Whenever tracking problems occurred (mostly while raising the right hand to reach towards the right outer cupboard, a motion insufficiently covered by the camera setup), the data was manually post-processed to ensure high quality.

5.2.0.3 Sensor-equipped kitchen environment

Data from RFID (Radio Frequency IDentification) readers for detecting objects and magnetic reed sensors that detect if doors or drawers are being opened was recorded using the sensor network in the assistive kitchen described in [Beetz et al., 2008]. Objects equipped with RFID tags are the place mat, the napkin, the plate and the cup. Metallic items like pieces of silverware cannot be recognized by the RFID system, and are therefore not recorded in the data set. The approximate positions of the RFID sensors and the cameras are given in Figure 5.5.

The subjects were asked to set the table according to the layout shown in red in Figure 5.5, plus a place mat and a napkin (which are not shown here). Initially, the place mat and the napkin were placed on the stove top (large yellow ellipse on the left), the cup and plate in an overhead cupboard (yellow ellipse on the right). Silverware was in a drawer, marked by the yellow ellipse in the center. There were two recording sessions in which the exact placement on the table shifted along the long side of the table, while the relative arrangement of the objects remained the same.

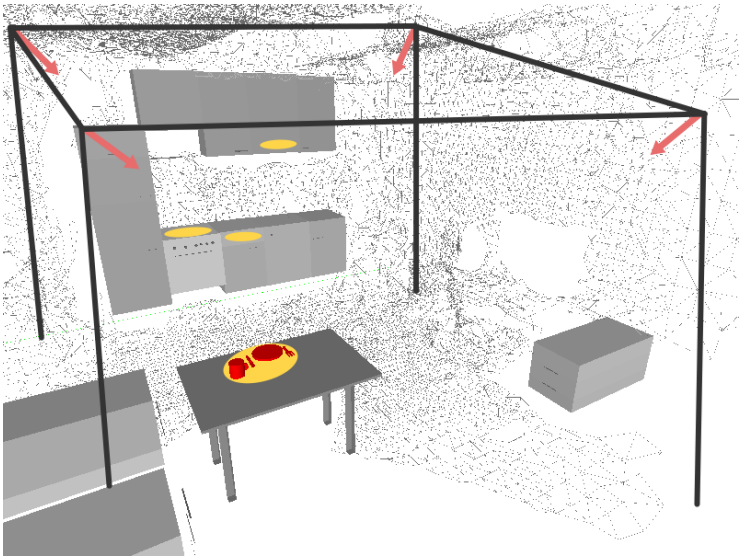


Figure 5.5 Spatial layout during the recording sessions: approximate camera locations (red arrows), RFID sensors (yellow ellipses), and approximate layout of the items on the table (red objects).

5.2.0.4 Modalities provided by the data set

The data has been recorded in the TUM kitchen [Beetz et al., 2008], a sensor-equipped kitchen environment used to perform research on assistive technologies for helping elderly or disabled people to improve their quality of life. The recorded (synchronized) data consists of calibrated videos, motion capture data and recordings from the sensor network, and is well-suited for evaluating approaches for motion tracking as well as for activity recognition. In particular, the following modalities are provided (see also Figure 5.6):

- Video data (25 Hz) from four static overhead cameras (Fig. 5.5, 5.3), provided as 384x288 pixel RGB color image sequences (JPEG) or compressed video files (AVI). Additionally, full resolution (780x582 pixel) raw Bayer pattern video files are available.
- Motion capture data extracted from the videos using the markerless full-body motion tracking system [Bandouch and Beetz, 2009]. Data is provided in the BVH file format which contains the 6 DOF pose and the joint angle values. In addition to the joint angles, global joint positions are available as a comma-separated text file (CSV, one row per frame, the first row describes the column datatype). The data has been post-processed when necessary.

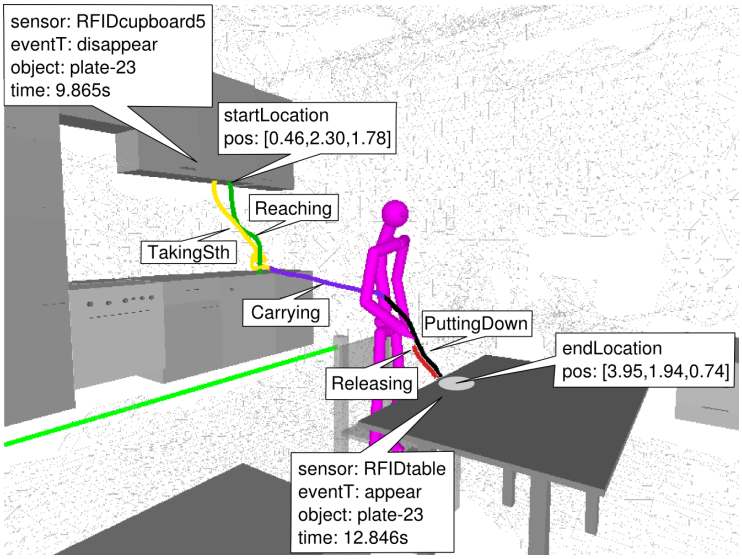


Figure 5.6 Illustration of the information provided by the TUM Kitchen Data Set, including annotated pose and trajectory data and RFID readings.

- RFID tag readings from three fixed readers embedded in the environment (sample rate 2Hz).
- Magnetic (reed) sensors detecting when a door or drawer is opened (sample rate 10Hz).
- Each frame has been labeled manually, as explained in Section 5.3. These labels are also provided as CSV files with one row per frame.

5.3 Labeling

The recorded data is to be used in the context of our knowledge representation. To integrate the observations with other pieces of knowledge in the system, they need to be described in a compatible format that relates the continuous motion data to a semantic description of the action that is being performed. Therefore, we manually annotated the data by assigning semantic labels to motion segments. The labels correspond to the action classes in the KNOWROB ontology (Figure 5.7), i.e. the observed motions are represented as instances of these classes. This segmentation into different kinds of actions provides ground truth for segmentation algorithms and also allows to train the algorithms in a supervised way. The primary goal of the segmentation is to obtain a sequence of short motion primitives that forms the basis for further, more semantic analysis. For this reason, unsupervised segmentation approaches are not usable as they cannot provide this link between the segments and their semantics required in the subsequent processing steps.

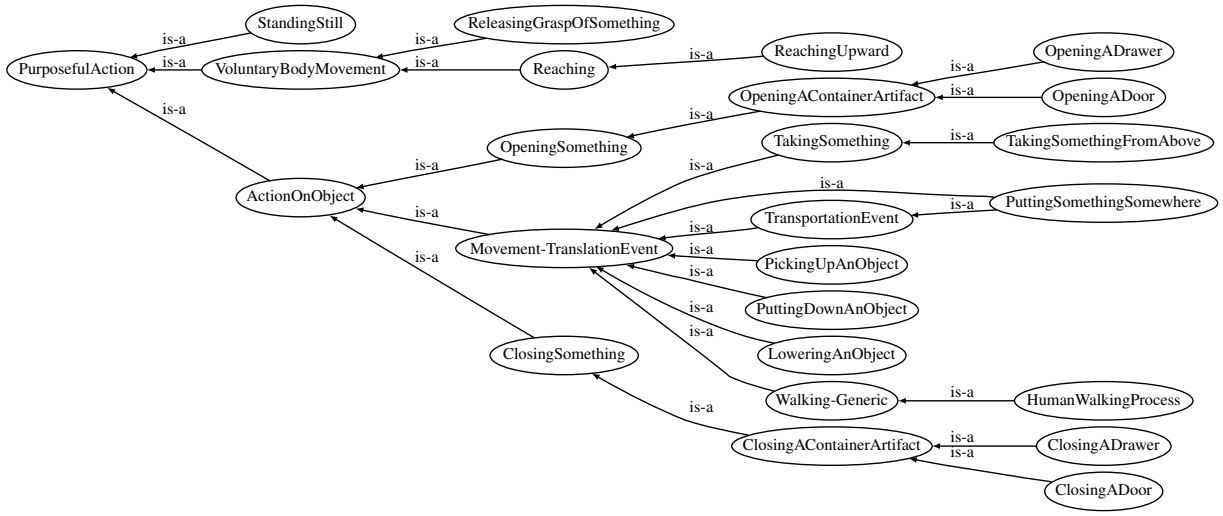


Figure 5.7 Excerpt from the action ontology used for labeling the data set. Special motions like *ReachingUpward* are defined as subclasses of the more general *Reaching* motion.

To account for the high degree of parallelism in the activities, we labeled the left hand, the right hand and the trunk of the person separately: In a common action, the subject is opening a cupboard door with one hand, while the other one is reaching towards a cup, and while the body is still moving towards the cupboard. The set of labels describing the hand motion consists of *Reaching* towards an object or the handle of a cupboard or drawer, *TakingSomething*, *LoweringAnObject*, *ReleasingGraspOfSomething* after having put down an object or having closed a door, *OpeningADoor*, *ClosingADoor*, *OpeningADrawer*, *ClosingADrawer*, and *CarryingWhileLocomoting*. To take the very different postures for object interactions with the overhead cupboards into account, we added the classes *ReachingUpward* and *TakingSomethingFromAbove* which are specializations of the respective motion classes. The motion of the person in the environment is described by the *trunk* label sequence which only has two possible values: *StandingStill* and *HumanWalkingProcess*, which indicates that the subject is moving.

The class *CarryingWhileLocomoting* describes all frames when the subject is carrying an item or, since the difference in the pose is often negligible, when the subject is moving from one place to another without carrying anything. It further serves as a catch-all class and contains the background motion between the actions. When acting naturally, humans perform surprisingly many irrelevant movements, such as swinging the arms or reaching halfway towards an object, thinking about it, retracting the hands and doing something else first. If these motions are not long and articulated enough to be reliably recognized, they are put into the catch-all class, which therefore comprises rather diverse movements. The alternative, labeling every short, partial motion, would result in a large set of classes, ambiguities in the labeling, and hardly detectable short motion segments.

An important choice when annotating data is the selection of which granularity to use. We chose to label the data at a rather fine-grained level, and use the methods described in Section 5.6 to generate coarser, more abstract descriptions when needed.

5.3.0.5 Labeling tool

A visual labeling tool has been developed in order to efficiently annotate the tracking data (Figure 5.8). Labeling three streams of action using very fine-grained annotations can be a tedious task, and the tool helps to significantly speed up the process. Compared to the paper-based method used before, the data annotation became about six to nine times faster.

The upper part of the program displays the images of the four cameras for the current time frame to provide the user with the best possible view on the scene. Below, there is a timeline to quickly sift through the sequence, and three label sequences for the left hand, the right hand, and the trunk. The lowest part are radio buttons to select which label to set. During the labeling process, the user sets a mark at the last frame of a segment by clicking on one of these buttons; this label is then assigned to all frames since the previous mark.

In order to verify if the labeling is correct, the tool also supports loading and visualizing label segments, so that a user can jump between the labels and inspect if they are correct.

5.3.0.6 Linking actions to manipulated objects

For semantically interpreting the data, knowing only the type of a motion segment is usually not sufficient but more detailed information about objects and locations is needed. The motion segments can thus be related to simultaneously observed sensory events from the RFID tag readers or the door sensors, in order to determine action properties, e.g. which object is being picked up or which cupboard is being opened.

This assignment is being done automatically using the *computable* classes and properties described in Section 2.6 that allow to read external data sources, like the RFID events, and related them to properties in the knowledge base, like the *objectActedOn*. We define a computable class to read instances of RFID events (Section 3.1), use a computable property to compute which events occur during an action segment, and compute the *objectActedOn* of an action by the objects detected in simultaneous RFID events. The same method is used to assign the door that is being opened to the correct action segment based on information from the magnetic sensors.

Locations, like the *fromLocation* or the *toLocation* of a transport action, can be approximately determined by the locations of the RFID tag readers. When a reader does not detect an object

Figure 5.8 Screenshot of the labeling tool developed for annotating the human motions in the TUM Kitchen Data Set.

any more while performing a pick-up action, we assume that object has been picked up from the location of the RFID tag reader.

5.4 Segmentation

This section is about our approach to automatically segment and classify the motion capture data, i.e. to infer the start time, the end time and the label of a segment from the data. These methods allow to transform newly observed data into the semantic representation used for further analysis.

In theory, many segments can be defined by the observable events that happen at either their beginning or their end: For instance, when an object has been picked up, it is not detected any more by the respective RFID tag reader, resulting in an RFID event. In reality, this simple segmentation approach is not possible because the sampling rates of the sensors are too low compared to the average duration of the motions (many motions only take about half a second,

which is the sampling rate of the RFID readers). Moreover, the rather large detection range of the RFID sensors lets them detect objects that have already been picked up, further delaying the end-of-detection signal. For this reason, the sensor data can only be used as one indicator amongst others and needs to be combined with further information in order to reliably and exactly estimate the beginning and end of each segment.

The features used for classification combine information from the RFID and door sensors with additional pose-related and environment-related information. Despite the timing problems, the RFID and door sensor data still provide valuable information to distinguish between actions that comprise very similar motions, but have different semantics. Putting an object down on the counter top and closing a drawer, for example, both include a rather straight forward motion of the hand, while they have completely different meaning. An RFID event registered at about the same time suggests a put-down motion, while the detection of the respective drawer being opened indicates an *OpeningADrawer* motion. If the events are used in this way, their exact timing becomes less of an issue.

The pose-related features denote e.g. if the human is extending or retracting the hands, if the hands are expanded beyond a certain threshold, or if they are lifted above a certain limit. We also use discretized angles of the joints that are significant for the task at hand, like shoulder and elbow joints. In particular, we use the following pose-related features:

- Joint angles, discretized into ten bins, of the spine (Joints ULW, OLW, BRK, KO in [Tenorth et al., 2009]) and the respective shoulder, arm and hand (SBL, OAL, UAL, HAL, FIL for the left hand)
- Finger joint position in a shoulder-centric coordinate system and gradient of the hand motion in the same coordinate system above or below zero (binary feature, separate for x,y,z components)
- Velocity of the hand in local coordinates above or below 0.35 m/s (empirically determined)
- Arms being extended or retracted (relative to the shoulder position)
- Person moving with more than 0.35 m/s (in global coordinates)

These features are to capture aspects like 'hand moving', 'hand in front of person', 'hand above shoulder', and 'arm being extended'. Some of them are a rather rough approximation, but computing them in a more principled way would be very difficult due to the many degrees of freedom and the deformable human body. For example, if the shoulders are not parallel to the hips, the notion of "in front of" becomes quite hard to define. The pose-related features are combined with information from the environment model and the sensor network, namely:

- Doors that are currently open (detected by the magnetic door sensors)
- Handle of a cupboard or drawer a hand is close to (read from the environment model)
- Object that is being carried (in the time span between two RFID readings)

We tested different classification methods on this feature set. In each case, we performed an leave-one-episode-out cross validation, i.e. the classifier is trained on all sequences but one and tested on the remaining one. Randomly sampling the frames to use for the test set yields wrong results for sequential data since for each element of the test set, the neighboring frame in the original motion sequence, that is almost identical, will be in the training set. The classification then effectively becomes a kind of nearest-neighbor matching with very close neighbors being available, leading to overly positive results.

The classifiers we chose are a C4.5 decision tree [Quinlan, 1993], a support vector machine (SVM, [Platt, 1999]), and a conditional random fields-based classifier (CRF, [Lafferty et al., 2001]). The former two classify each frame independently, while the CRF takes the sequence of labels into account. For the decision tree and SVM, we used the implementation from the Weka library [Witten and Frank, 2005], for the CRF the implementation in Mallet [McCallum, 2002]. Both libraries are integrated into KNOWROB and can be called from within the knowledge processing system.

The decision tree is a rather simple classifier that has the advantages of producing human-readable models and of performing inherent feature selection. We use the *weka.classifiers.trees.J48* class with the parameters *-C 0.25 -M 2*. The classifier achieves an accuracy (correctly classified

class \ classified as	Carrying	ClosingADoor	ClosingADrawer	LoweringAnObject	OpeningADoor	OpeningADrawer	Reaching	ReleasingGrasp	TakingSomething
Carrying	15893	69	77	353	41	214	753	881	167
ClosingADoor	46	939	0	27	181	0	127	84	3
ClosingADrawer	216	0	367	31	0	165	301	72	55
LoweringAnObject	472	9	44	435	0	51	110	198	169
OpeningADoor	6	143	0	5	1470	0	362	67	7
OpeningADrawer	245	7	140	7	0	797	130	162	83
Reaching	777	107	90	109	170	185	1260	106	197
ReleasingGrasp	862	130	108	146	51	79	177	1057	300
TakingSomething	291	21	34	25	14	80	81	250	371

Table 5.1 Confusion matrix for the J48 decision tree and the data of the left hand. The classifier reaches a per-frame accuracy of 0.68.

frames divided by the total number of frames) of 0.68 – the detailed results are listed in Table 5.1. There is a noticable bias towards the *CarryingWhileLocomoting* class that is due to two reasons: First, this class is much larger than all the others, in fact containing more frames than all other classes together. Second, this class is also the most diverse one that contains all in-between motion the subject exhibits while not performing any goal-directed activity. Therefore, it contains a large variety of motions, increasing the likelihood that an observed frame is classified as belonging to this class. The same bias can be observed in the results of the other classifiers, though much less for the CRF. The reason is that the CRF does not only consider the single frames (which may be similar to frames in the *CarryingWhileLocomoting* class), but also takes the sequence context into account that can make an idle frame much less likely at many positions in the sequence.

The SVM achieves the best overall accuracy correctly classifying 77% of the frames (trained using the *weka.classifiers.functions.SMO* class with the parameters *-C 1.0 -L 0.0010 -P 1.0E-12 -N 0 -V -1 -W 1 -K "weka.classifiers.functions.supportVector.PolyKernel -C 250007 -E 1.0"*). The confusion matrix in Table 5.2 shows a significant improvement compared to the decision tree. However, the results do not show a problem both classifiers share: The strict per-frame classification tends to create many short segments, and the mis-classified frames are scattered throughout the sequence. Though this leads to a high overall per-frame accuracy, the resulting sequence differs quite a lot from the actually performed sequence of actions. This is problematic for the intended higher-level processing, in which a correct action sequence is more important

class \ classified as	Carrying	ClosingADoor	ClosingADrawer	LoweringAnObject	OpeningADoor	OpeningADrawer	Reaching	ReleasingGrasp	TakingSomething
Carrying	17565	18	65	108	56	115	257	194	70
ClosingADoor	30	949	3	35	184	0	92	112	2
ClosingADrawer	50	0	742	33	0	86	172	97	27
LoweringAnObject	436	0	28	653	0	4	122	233	12
OpeningADoor	13	143	0	15	1657	0	159	59	14
OpeningADrawer	110	0	127	1	0	1112	107	91	23
Reaching	831	71	226	151	141	76	1416	47	42
ReleasingGrasp	955	94	117	165	57	106	41	1243	132
TakingSomething	273	13	58	92	3	124	105	242	257

Table 5.2 Confusion matrix for the support vector machine and the data of the left hand. The classifier reaches a per-frame accuracy of 0.77.

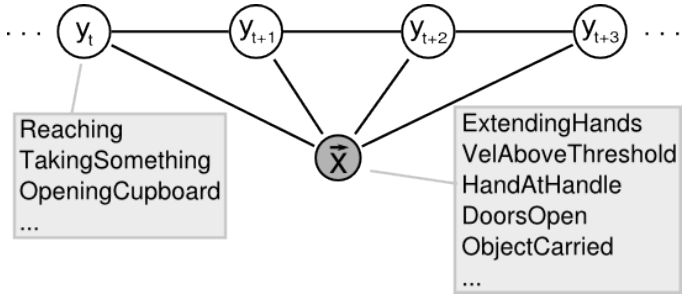


Figure 5.9 Structure of the CRF for segmenting human motions.

than the exact segment borders.

For this reason, we finally chose the segmentation based on a linear-chain conditional random field (Figure 5.9, Table 5.3). Its per-frame accuracy of 0.73 is slightly lower than for the SVM, but the errors in the resulting sequence are rather shifted boundaries between the segments or segments that have been missed completely. In total, this better reflects the actual sequence of actions what is what we are interested in here.

The classifiers are called from within KNOWROB and return a sequence of labeled pose instances. Consecutive poses with the same label are combined into one instance of the respective motion class (e.g. an instance *Reaching1* of type *Reaching* with all pose observations as subevents, the time stamp of the first pose as *startTime*, and the one of the last pose as *endTime*. This is depicted in the lower part of Figure 5.12, where the transition from pose vectors to typed motion segments via the CRF-based segmentation procedure is shown.

class \ classified as	Carrying	ClosingADoor	ClosingADrawer	LoweringAnObject	OpeningADoor	OpeningADrawer	Reaching	ReleasingGrasp	TakingSomething
Carrying	13598	2	24	617	10	14	636	618	208
ClosingADoor	2	808	22	0	85	11	59	148	24
ClosingADrawer	1	0	467	38	0	90	187	125	37
LoweringAnObject	406	0	0	424	0	0	110	260	60
OpeningADoor	54	159	0	0	1186	0	168	48	31
OpeningADrawer	12	0	92	8	0	874	39	106	137
Reaching	539	65	71	42	59	25	1498	149	81
ReleasingGrasp	572	66	67	255	17	40	154	1173	115
TakingSomething	230	4	64	0	27	59	131	121	326

Table 5.3 Confusion matrix for the CRF and the data of the left hand. The classifier reaches a per-frame accuracy of 0.73.

5.5 Trajectory models

The segmentation procedure splits the continuous motions into short segments and assigns them a semantic class label. These classes, however, are rather coarse descriptions: For instance, there can be very different ways of *Reaching* towards an object, like reaching into an overhead cupboard or reaching straight down towards an object on top of the table, and each of these trajectory styles may be more or less appropriate in a given context. The context may thereby depend on different things: the type of the object to be manipulated, its position (e.g. on the counter or in an overhead cupboard), the size of the human subject performing the task, or also the action that is to be performed next with that object.

In this section, we describe methods for (1) discovering different ways how an action can be performed, leading to different shapes of the resulting trajectories, and (2) for learning semantic characterizations that describe which of these trajectory clusters are used in which context. The proposed system consists of two main components: A trajectory clustering module, and models that describe the semantics of those clusters. For learning the semantic models, we employ the GrAM (Grounded Action Models) framework described in [v. Hoyningen-Huene et al., 2007]. GrAMs have been applied to the analysis of soccer games and for learning places used in manipulation actions [Tenorth and Beetz, 2009]; here, we learn trajectories instead. The input data for the following methods are pre-segmented trajectories for motions like *Reaching* or *LoweringAnObject* that have been created by manual annotation or by the segmentation methods described earlier.

5.5.1 Clustering trajectories

The algorithm used for clustering was developed in collaboration with Daniel Nyga and is described in more detail in [Nyga et al., 2011]. It is motivated by Classification and Regression Trees (CART [Breiman, 1984]), a classification method to create binary decision trees by iteratively partitioning a data set into two cuboid sub-regions, each time taking only few dimensions into account that allow to distinguish the two sub-sets with high accuracy. This splitting rule is typically given by a simple threshold (also called decision stump, a decision tree with only one node). The process is repeated until a stopping criterion is reached, in our case when the cohesion within a cluster, the mean value of the average distances of each member to each other member, does not significantly improve any more.

The Hopkins index [Hopkins and Skellam, 1954] indicates if clusters exist in a data set and is used as criterion to select which components to use for clustering. It is computed by uniformly sampling two sets of $m << n$ points: a set $S = \{s_1, \ldots, s_m\} \subset X$ from the dataset $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^p$, and another set $R = \{r_1, \ldots, r_m\}$ from the convex hull around the data. The algorithm then computes, for each point in these data sets R and S, the distance to the nearest neighbor in X:

$$d_R = \{d_{r_i} \mid \min_j \|r_i - x_j\|, 1 \leq j \leq n\},\ 1 \leq i \leq m \tag{5.1}$$

$$d_S = \{d_{s_i} \mid \min_j \|s_i - x_j\|, 1 \leq j \leq n\},\ 1 \leq i \leq m \tag{5.2}$$

The ratio of these distances yields the Hopkins index $h \in [0, 1]$ that describes how the points inside the data set are distributed with respect to arbitrary points in the convex hull.

$$h(X) = \frac{\sum_{i=1}^{m} d_{r_i}^p}{\sum_{i=1}^{m} d_{r_i}^p + \sum_{i=1}^{m} d_{s_i}^p} \tag{5.3}$$

If there is no significant cluster structure, the distances inside the data set do not differ much from those between random other points in the convex hull. The resulting Hopkins index is thus near to $h \approx 0.5$. A very small Hopkins index $h \approx 0$ results from data that exhibits a regular structure, whereas a Hopkins index $h \approx 1$ is a sign that significant clusters are present in the data. Intuitively, a large Hopkins index means that the distances from arbitrary points in the convex hull to the nearest data point are large, while most points in the data set have a neighbor nearby. Since the Hopkins index highly depends on the sets R and S, it is recommended to sample several different sets R and S and average over the results.

The trajectories are first re-sampled to a length of 100 points using spline interpolation. Since the provided labels often do not exactly match the beginning and end of a trajectory segment, we cut off 25% of the frames, namely the first 16% and the last 9%. The trajectories are then transformed into a person-intrinsic coordinate frame defined by the left and right shoulder *SBL* and *SBR*. This transformation only affects the x and y coordinates, the z coordinate stays unaffected. In the resulting representation, person-related spatial relations like “in front of”, “left of” or “above” have a direct correspondence to the trajectory values. Since the human body is a deformable system, there is no optimal person-intrinsic coordinate frame, but our experience shows that this shoulder-centric coordinate system yields the best results. Finally, the ending points of the trajectories, corresponding to the object positions, are aligned. After these pre-processing steps, the clustering algorithm iteratively selects those components of the trajectories that exhibit

the highest Hopkins indices and applies the CART clustering. Once the CART tree has been built up from the training data, it can be used to classify novel test data and assign it to one of the clusters.

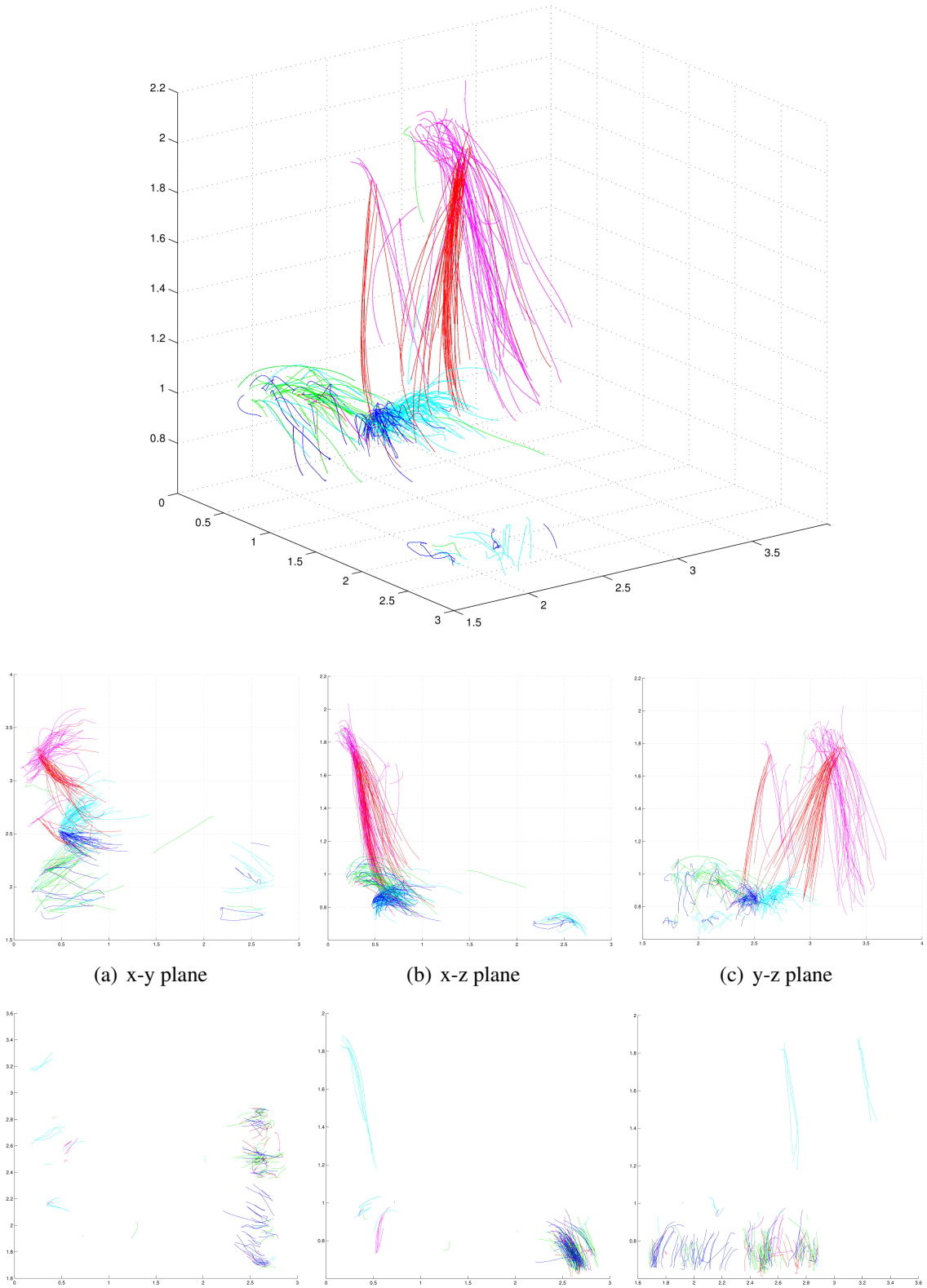
We evaluate the system on the trajectories of the left and right hand from the TUM Kitchen Data Set. Figure 5.10 shows the trajectory clusters identified for the motions *Reaching* and *LoweringAnObject*. These results indicate that the algorithm is able to distinguish different forms of trajectories even if they are in similar regions of the environment, like the upwards motions in clusters 1 and 4. The trajectories for putting down objects are not as well separated as those for reaching, but their shapes are also well distinguished. The blue cluster, for instance, is mainly directed to the left, while the green one points to the right.

5.5.2 Context-dependent trajectory selection

Having identified the different kinds of trajectories in a motion class, the system automatically learns models characterizing their semantics and describing in which context they shall be used. This semantic model is learned based on an intensional specification of the learning problem using the Grounded Action Models (GrAMs) method [v. Hoyningen-Huene et al., 2007]. Such an intensional specification lists a set of features (called *observables*) which can serve for predicting the value of a property (called *predictable*). Given a training set of observed action instances of a class like *Reaching*, the system learns the relation between their observable properties and the one to be predicted. In our case, the observable properties describe things like the manipulated object or the hand that is used, while the shape of the trajectory (i.e. the cluster ID) is to be predicted. For example, the intensional model *reachTrajModel* is defined as an instance of an *ActionModel* that is to be learned from the training set *reachTraj* (all trajectories of *type Reaching*) with the observables *objectActedOn* and *bodyPartsUsed*:

```
Individual: reachTraj
    Types: Restriction
    Facts: onProperty type
           hasValue   Reaching

Individual: reachTrajModel
    Types: ActionModel
    Facts: forAction    reachTraj
           observable   objectActedOn
           observable   bodyPartsUsed
           predictable  trajectory-Mean
```



(a) x-y plane (b) x-z plane (c) y-z plane

(d) x-y plane (e) x-z plane (f) y-z plane

Figure 5.10 Cluster assignments, indicated by the trajectory color, for *Reaching* trajectories (upper diagram and center row) and trajectories for *LoweringAnObject* (lower row).

The action model is learned as a classifier on top of the cluster assignment that uses the values of the *observable* features in order to learn rules that explain the values of the *predictable* features. These rules form the *extensional* action model and semantically describe in which context a trajectory cluster is being used. They are equivalent to a sub-class definition in Description Logic, like a class "*Reaching* actions performed with the right hand towards a cup inside the cupboard". These sub-class definitions are learned autonomously from data as those rules that best explain the relations in the training set. Action models can either be used for classification (inferring the context given an observed trajectory), or for selecting a trajectory given a certain context.

Table 5.5 shows the associations between observable properties and the cluster IDs that are learned based on the specification of the intensional model. The system is able to distinguish trajectories with different meaning, though they have similar shapes and are in similar regions of the environment like reaching for a cupboard handle (Cluster 4) and taking an object out of that cupboard (Cluster 1). For some objects, our algorithm also found differences in the reaching behavior of the left and right hand, e.g. for *SilverwarePiece* (Cluster 2/5 resp.). Other objects at approximately the same position, such as *Cup* and *DinnerPlate*, or *Napkin* and *PlaceMat*, exhibit reaching trajectories that are too similar to be discernible for the algorithm. Since most of the objects are always picked up with the same hand, there is little variation in clusters for the *bodyPartsUsed* property.

bodyPartsUsed	objectActedOn	Cluster Assignment
LeftHand	PlaceMat	3
	Cup	4
	DinnerPlate	4
	Napkin	3
	SilverwarePiece	2
	Drawer	3
	Cupboard	1
RightHand	PlaceMat	3
	Cup	4
	DinnerPlate	4
	Napkin	3
	SilverwarePiece	5
	Drawer	3
	Cupboard	1

Table 5.5 Extensional action model for *Reaching* motions

5.6 Hierarchical models

A fine-grained action segmentation does provide local information, e.g. about the trajectory used for opening a cupboard, but it cannot be used to determine more abstract, higher-level information like which objects are moved from where to where in a table-setting task. To obtain such higher-level action descriptions, we need to combine motions to simple actions, these simple actions to more complex ones, and subsequently build more and more abstract action models.

The hierarchical action models (HAM) presented here exploit knowledge about sub-actions that an action can have to recursively create higher levels of abstraction. This knowledge is available in the knowledge base and was described in more detail in Section 3.3. Figure 5.11 exemplarily shows the information that is available for the class *PuttingSomethingSomewhere*, namely that the action *PuttingSomethingSomewhere*, a transport action, involves picking up an object, carrying it to its destination, and putting it down. In addition, there are partial-ordering constraints among the sub-actions requiring that the pick-up action has be done before moving the object, followed by putting it down again. This knowledge is described on the class level; observations are instantiated as instances of these classes and thus inherit their properties. In the following, we describe how this abstract knowledge is used to build up hierarchical action models like the one in Figure 5.12.

A first step that is performed after the segmentation and before the abstraction into hierarchical

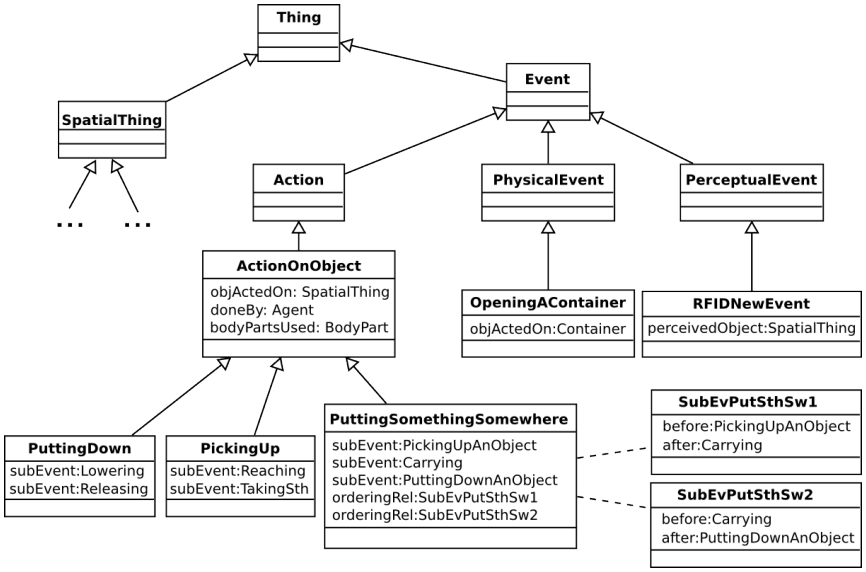


Figure 5.11 Knowledge about classes of actions, events and objects that is represented in KNOWROB and used in the abstraction process.

models is to condense the label sequence, i.e. to aggregate all subsequent frames with the same label to one action instance, to assert its type (based on the label), and its start- and end-times (based on the times of the first and last frame in the segment). In this step, a sequence of motion segments is created in the knowledge base (*Reaching-14*, *Moving-27* etc. in Figure 5.12).

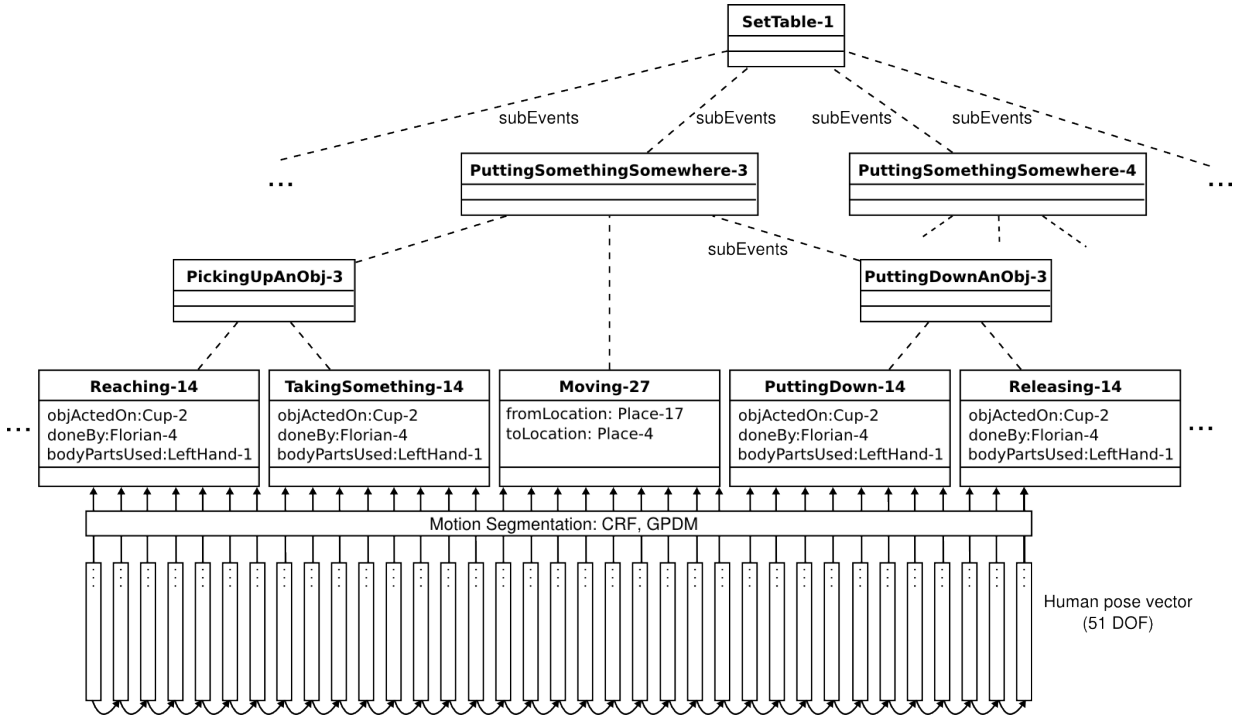


Figure 5.12 Hierarchical action model describing the abstraction from motion tracking data to more and more abstract action instances.

5.6.1 Definition

We start with a formal description of the hierarchical models and the transformation rules that are used for creating them.

Def. 1: A *Hierarchical Action Model* is defined as a 4-tuple

$$\mathcal{H} = \{\{\mathcal{A}_i\}, T_a, T_p, T_r\},$$

consisting of a set of action sequences $\mathcal{A}_i$, a set of action classes T_a, e.g. *Reaching* or *Taking-Something*, a set of properties T_p which relate the actions to other instances, e.g. to objects with the *objectActedOn* property, and a set of relations between actions T_r like *subAction* or *nextAc-*

tion. There can be multiple action sequences on the same level of abstraction (e.g. if the left and the right hand perform independent actions), and multiple levels of abstraction that also potentially combine independent action sequences into a single one (e.g. when a pick-up action of the left and a put-down action of the right hand are combined into a joint transport action).

Def. 2: An *action* $a = \{t, P\}$ is described by its type $t \in T_a$ and a set of properties $P = \{p_j^a \in T_p\}$ that describe for example its start time or the manipulated object.

Def. 3: Let $A_s^k = \{a_0^k, a_1^k, \dots a_n^k\}$ be a *sequence of actions* a_i of length n at abstraction level k. The lower index s refers to the label sequence, the upper index k describes the level of abstraction. Then we can describe an *activity* at abstraction level k as a set of action sequences

$$\mathcal{A}^k = \{A_s^k\}$$

Sequences with $k = 0$ describe the most fine-grained description, potentially even single image frames (as in Figure 5.12). Activities can consist of more than one action sequence per abstraction level if there are independent strings of actions, e.g. $s \in \{l, r\}$ for the left and right hand, as in our implementation.

5.6.2 Construction from action sequences

Hierarchical action models are constructed by iteratively applying two kinds of rules to the segmented action sequence until no further change can be achieved:

- *Abstraction rules* generate a new action sequence on a higher level of abstraction.
- *Transformation rules* propagate information inside and among action sequences on a fixed level of abstraction.

Abstraction rules are, for instance, used to create a pick-up action when a *Reaching* motion followed by a *TakingSomething* motion is observed. Transformation rules are for example used to propagate information from instantaneous events, like the detection of an object, to neighboring actions. This is often needed to set the *objectActedOn* for a *Reaching* motion, since the corresponding RFID tag event usually occurs during the following *TakingSomething* action when the object is actually removed from the detection range of the reader.

Def. 5: *Abstraction rules* combine several action segments of one action sequence at level k to form more abstract representations on a higher level $k + 1$

$$R_A : A_s^k \rightarrow A_{s'}^{k+1}$$

The formulation of the rules is kept very general in order to make them applicable to a large variety of object manipulation actions. They operate based on the knowledge about sub-actions that is declaratively specified in the knowledge base like the example given in Figure 5.11. This allows to easily apply the system to new tasks as longs as a declarative description of actions and their sub-actions is available. Algorithm 1 describes a slightly simplified version of the abstraction procedure:

Algorithm 1 abstract(ActSequence, PriorSuperActs, Segment)

$Cur \leftarrow$ first($ActSequence$)
$Rest \leftarrow$ rest($ActSequence$)
$SuperActs \leftarrow$ compatible_super_actions(Cur, $PriorSuperActs$)

if ordering_constraints_met($Segment$, $SuperActs$) **then**
  if segment_complete($Segment$, $SuperActs$) **then**
    $Super \leftarrow$ create_superaction($Segment$, $SuperActs$)
    $HighLRest \leftarrow$ abstract($Rest$, [], [])
    $HighL \leftarrow$ append($Super$, $HighLRest$)

  else
    $Segment \leftarrow$ append(Cur, $Segment$)
    $HighL \leftarrow$ abstract($Rest$, $SuperActs$, $Segment$)
  end if
else
  $HighL \leftarrow$ append($Segment$, $HighLRest$)
end if
return HighL

The abstraction procedure iterates over the sequence of low-level actions *LowL* and creates a more abstract sequence *HighL*. It maintains a set of hypotheses of potential higher-level actions *ActHypos* that so far fitted to the other actions in the segment. In each step, the system checks which of these super-action hypotheses can also be a super-action of the current action using the *compatible_super_actions* and discards the other alternatives. In the next step, the set of hypotheses is further filtered by removing those whose ordering constraints would be violated by the current segment. The algorithm then checks whether the segment is complete, i.e. if all required sub-actions are there and if all ordering constraints are met for any of the hypotheses. In this case, it creates the super-action and continues the abstraction with a new segment and an empty set of hypotheses. Otherwise, it continues with the old segment until it is either complete or until all super-action hypotheses turned out to be wrong. In this case, the system just adds all actions from the current segment to the higher-level sequence. Often, actions can

be incorporated later at a higher level of abstraction, for example *Moving-27* as sub-action of *PuttingSomethingSomewhere-3* on the second level.

Def. 6: *Transformation rules* operate on a constant level of abstraction and either change the sequence of actions, their properties, or combine sequences of actions, thus transform

$$R_T : \{A_s^k\} \rightarrow \overline{A}_{s'}^k$$

Transformation rules can also add relations $r \in T_r$ between actions, for example to link each action to the next one in the sequence, or combine two sequences into a single one. These relations are described by asserting the respective OWL properties that link the action instances in the knowledge base.

The example in Algorithm 2 is used to propagate RFID events between neighboring actions. If the event has been detected during the *Reaching* phase and no object is detected for the following *TakingSomething* phase, the detection is propagated. There is an analogous rule for the opposite case, propagating events from a *TakingSomething* to the preceding *Reaching* segment.

Algorithm 2 transform(ActSequence)

```
First ← first(ActSequence)
Rest ← rest(ActSequence)
Second ← first(Rest)

if (type(First) = Reaching) and (type(Next)=TakingSomething) then
  if objectActedOn(First) and not objectActedOn(Next) then
    Obj ← objectActedOn(First)
    assert(objectActedOn(Next), Obj)
    transform(Rest)
  end if
end if
```

5.6.3 Evaluation

To evaluate our approach, we first show on two real-world data sets of different activities how hierarchical models can be built from a flat sequence of actions. In addition, we explain how the hierarchical action representation can be used to answer queries about the observed activities.

5.6.3.1 Abstraction into hierarchical models

We first present the results of the abstraction from a flat segmentation into a hierarchical action model. The experiments were performed the experiments on two different data sets, the TUM Kitchen Data Set (see Section 5.2), which mainly contains observations of table setting activities, and the CMU MMAC data set [De la Torre et al., 2009], which consists of data from different cooking tasks. The CMU MMAC data set does not provide the data required by our motion segmentation algorithm (Section 5.4), so this hierarchical model is based on the manual labels provided with the data.

Figure 5.13 visualizes the models constructed from sequence 1-0 of the TUM data set (sub-figures (1) and (2)) and the 'Eggs' sequence of subject 12 in the CMU data set (subfigure (3)). In sub-figures (1) and (3), the colors correspond to action classes; the pink color on the topmost level in subfigure (1), for instance, is a *PuttingSomethingSomewhere* action.

In sub-figure (2), the color describes the manipulated object. This kind of coloring also shows how actions are combined, e.g. how *Reaching*, *TakingSth*, *LoweringAnOject* and *Releasing-GraspOfSomething* form a *PuttingSomethingSomewhere* action. This is visually apparent as the 'inverted U' shape. For the object colored in blue, the *PuttingSomethingSomewhere* action was started with the right and finished with the left hand; combining these sections is in principle possible, though not yet implemented. The missing objects in the middle of the sequence are pieces of silverware that cannot be detected by the RFID sensors, and since the *objectActedOn* is missing, the low-level motions are not aggregated to higher-level actions.

In the CMU data, the actions of the left and right hand are not labeled separately, so there is only one stack of sequences. Since the data is also labeled on an already more abstract level, the benefit of the abstraction procedure is not as large as for the TUM data. However, it can still be seen how e.g. the actions *OpeningACupboard*, *TakingSomething* and *ClosingSomething* are combined to the *GettingSomethingFromTheCupboard* instances depicted in yellow.

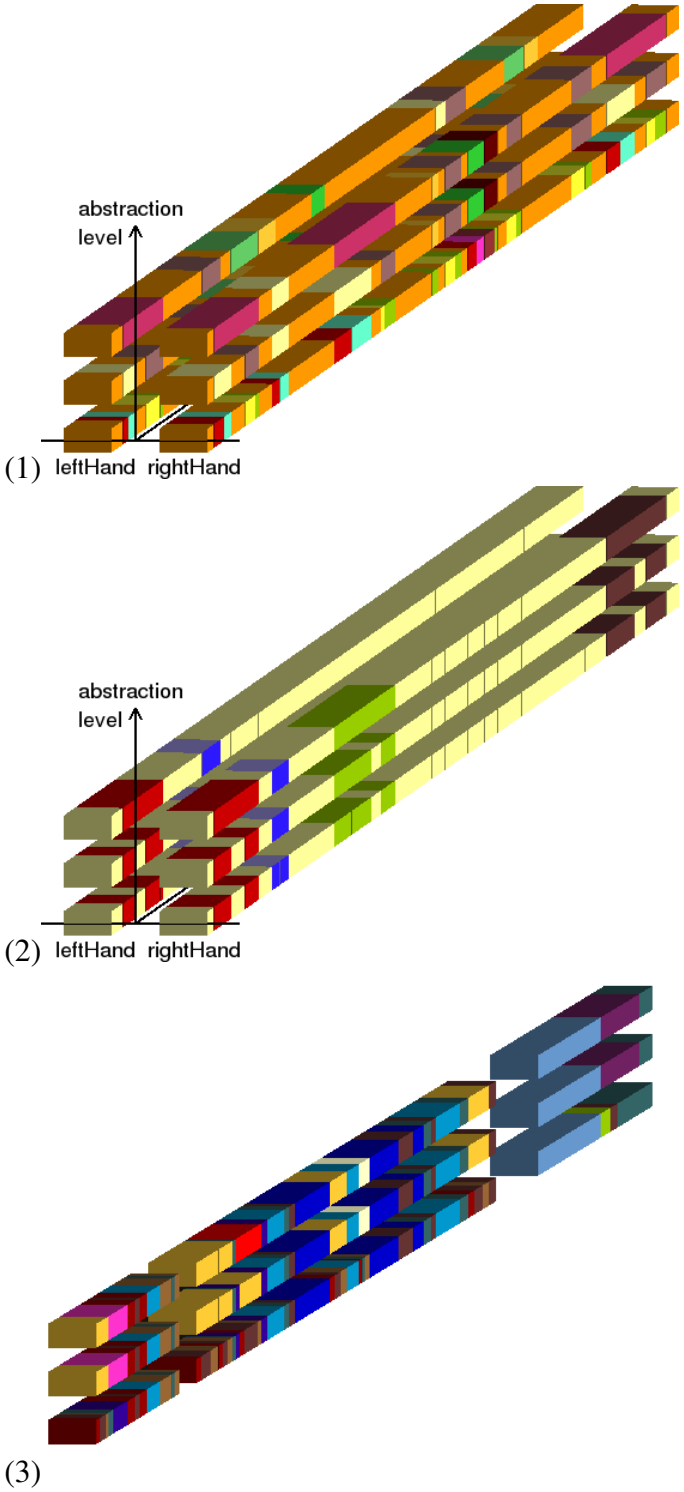


Figure 5.13 Visualization of the action abstraction on the TUM Kitchen Data set and the CMU MMAC data set. (Left: TUM data, color based on the action type. Center: TUM data, color based on the manipulated object. Right: CMU data, color based on the action type. In the TUM data, there are separate stacks of sequences for the left and right hand, in the CMU data only one for both. Higher levels correspond to more abstract descriptions, the time dimension points towards the back.

Figure 5.14 Locations where a place mat is taken from (dark red) and put to (light green) during a table-setting task.

5.6.3.2 Queries on HAMs

To demonstrate the usefulness of the models, we now present different queries they enable a robot to do. We start with asking for everything that is known about the action segment *PickingUpAnObject100*:

```
?- owl_has('PuttingSomethingSomewhere100', ?Pred, ?O).
   Pred=type,             O=PuttingSomethingSomewhere ;
   Pred=subAction,        O=PickingUpAnObject150 ;
   Pred=subAction,        O=CarryingWhileLocomoting53 ;
   Pred=subAction,        O=PuttingDownAnObject151 ;
   Pred=objectActedOn,    O=placemat-1 ;
   Pred=doneBy,           O=florian ;
   Pred=bodyPartsUsed,    O=rightHand ;
   Pred=fromLocation,     O=loc(0.32,1.98,1.08) ;
   Pred=toLocation,       O=loc(3.2,2,0.74) ;
   Pred=startTime,        O=0.722562 ;
   Pred=endTime,          O=5.45968
```

In a robotics context, it is useful to be able to select trajectories based on their purpose, e.g. for imitation. The trajectory for a *TakingSth* motion with a *fromLocation* on the table is obtained with the following query and visualized in Figure 5.1, bottom left:

```
?- type(?A,'TakingSth'), fromLocation(?A,?From),
   on_Physical(?From,?T), type(?T,'Table'),
   trajForAction(?A,'rightHand',?Traj).
```

The models also allow to query for action-related information, for example from which location to which location an object is transported (visualized in Figure 5.14):

```
?- type(?A, 'PuttingSomethingSomewhere'),
   objectActedOn(?A, ?O), instance_of(?O, 'PlaceMat'),
   fromLocation(?A, ?FL), highlight_location(?FL),
   toLocation(?A, ?TL),   highlight_location(?TL).
```

We can also query for habits of a person, e.g. if she always opens cupboards with the left hand

```
?- forall(type(?A, 'OpeningACupboard'),
   bodyPartsUsed(?A, 'leftHand')).
Yes
```

The following query asks for objects that are carried with both hands, i.e. where two simultaneous *PuttingSomethingSomewhere* actions, performed by different hands on the same object instance, exist:

```
?- type(?A1,'PuttingSomethingSomewhere'),
   type(?A2,'PuttingSomethingSomewhere'),
   not(?A1=?A2),
   bodyPartsUsed(?A1,'leftHand'),
   bodyPartsUsed(?A2,'rightHand'),
   objectActedOn(?A1,?O),objectActedOn(?A2,?O),
   timeInterval(?A1,?I1),timeInterval(?A2,?I2),
   timeOverlap(?I1,?I2).

O = 'placemat-1'
```

5.7 Partial-order models

It is surprising how different people cook the same meal: Some people first prepare all the tools and ingredients, others start to cook and get the things they need just in time, others do something in between. In addition, people get distracted, perform irrelevant actions like cleaning the countertop, or forget to take something out of the refrigerator. This is possible since many human activities allow a lot of freedom in how they are performed: The same goal can be reached by significantly different action sequences, corresponding to different 'styles' that are often typical for a person. Technically speaking, there is usually no total ordering among the actions in a task, but only a partial order imposed by causal dependencies among the actions, and this partial order often depends on the person performing the task.

For analyzing human activities, it is desirable to recognize these different styles, to spot differences and anomalies, to compare activities, and to find out which actions are relevant for the task at hand. Models of these activities should not get confused by the variety of different action sequences, but rather learn the characteristic ordering constraints that need to be met, i.e. the partial order or dependency structure of the actions in a task.

Figure 5.15 illustrates this on the example of making brownies. The left part shows three different action sequences for the task. In the leftmost sequence, the subject only retrieves objects that are immediately needed for the next action. The subject in the center, in contrast, first prepares all ingredients and tools and then starts with the cooking. The colors indicate the dependencies among the actions, which are also shown in the partial-order graph in the right part of the picture. The arrows indicate the precedence relation between actions; an arrow from A to B means that A happens before B. Our goal is to learn the partial-order graph from a multitude

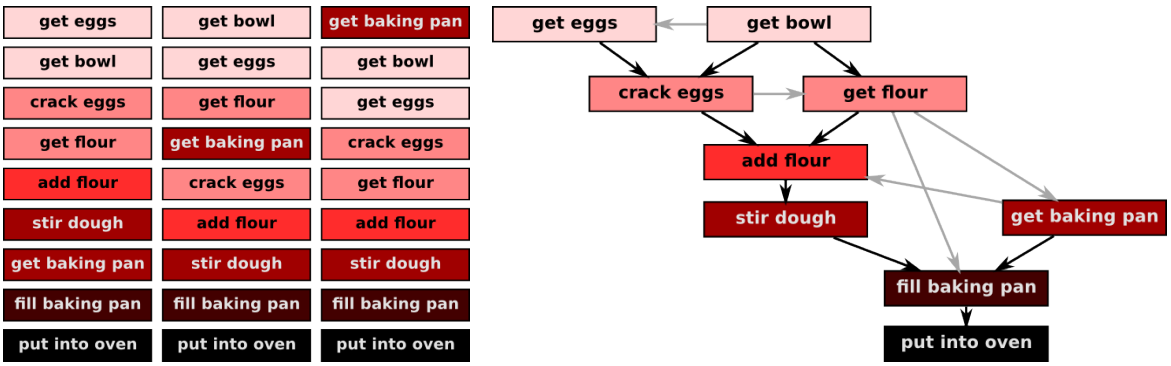


Figure 5.15 From several observations of the same task (left), the system learns the partial order of actions in that task (right) using statistical relational learning methods.

of diverse action sequences like those in the left part of Figure 5.15. In many cases, the training set does not equally cover all alternatives how an action can be performed, but shows some bias, introducing soft precedence constraints in addition to the causal dependencies between the actions. These soft constraints can be represented using a statistical model that can describe the probability of a precedence relation based on how consistently it was observed in the training data (visualized by the gray arrows in Figure 5.15).

The implementation is based on the Bayesian Logic Networks (BLNs) introduced in Section 2.5.1. By learning a BLN, we extract a joint probability distribution over the types of actions in an activity, their properties and their pairwise ordering constraints. Combined, these pairwise ordering constraints result in a partial order imposed on all actions in a task. The resulting models do not only describe the partial order, but general relations between consecutive actions and their properties. From a diverse training set, the models can learn which actions are relevant and which ordering relations are important. Actions that occur in all observations of a task are considered more relevant than those that are only rarely observed, and ordering relations that consistently hold are also more likely to be important. With this approach, the system can for example learn that wiping the countertop is less important for making an omelette than cracking eggs, and that one first needs to take the eggs out of the refrigerator before cracking them. The learned full-joint probability distribution can thus be used for various inference tasks:

- Classifying and verifying activities
- Identifying relevant actions in the activity
- Inferring missing data, e.g. the type of an action or object given the overall activity model
- Manually analyzing the learned models can show how structured a person performs a task

5.7.1 Modeling partially-ordered tasks

In contrast to the previously described task representations, tasks are now considered to be abstract partially-ordered action descriptions, and are distinct from the (linear) observed action sequences. Furthermore, the partial-order constraints are no longer assumed to be given and deterministic, but are now soft constraints that need to be learned from the training data. We thus have to extend the prior notation to take this additional information into account and will explain the notation used for learning partial-oder models in this section. Note, however, that the learned soft ordering constraints can be translated into the notation used earlier by choosing only the most likely constraints or those that have a likelihood beyond a certain threshold. We start with a formal description of the representation of tasks and actions. $\mathcal{T}$ denotes a set of tasks, each of

which is described by a set of actions $\mathcal{A}_t$, a possibly empty set of action properties $\mathcal{P}_t$ and an ordering relation $\mathcal{O}_t$ among the actions.

$$\mathcal{T} = \{T_t | T_t = \langle \mathcal{A}_t, \mathcal{P}_t, \mathcal{O}_t \rangle\} \tag{5.4}$$

Tasks are not to be confused with the observed action sequences S, which are instances created by performing a task. A task model describes the partial order inherent in an activity, action sequences are sequential samples following this partial order. The number of actions they comprise may differ due to non-observed actions or additional irrelevant actions. Action sequences are described as

$$\mathcal{S} = \{S_s^T | S_s^T = \langle a_0, a_1, \dots \rangle\} \tag{5.5}$$

Observed actions in an action sequence are marked with a subscript index a_i, the prototypical actions in a task model have a superscript index a^i. Action sequences are related to tasks via the $activityT$ predicate.

$$activityT(S^T) = T \tag{5.6}$$

Each task model comprises a set of n actions, which have one of m different types $A^0, \dots A^m$

$$\mathcal{A}_t = \{a^0, a^1, \dots, a^n\} \tag{5.7}$$

$$\forall i \in [0, n] : actionT(a^i) \in \{A^0, A^1, \dots, A^m\} \tag{5.8}$$

Actions may have different properties like the object manipulated or the hand used. $\mathcal{P}_t$ assigns a probability value to each property $\pi_j \in \Pi$ of each action a^i:

$$\mathcal{P}_t : \mathcal{A}_t \times \Pi \to \mathbb{R} \tag{5.9}$$

$$\Pi = \{\pi_0, \pi_1, \dots, \pi_p\} \tag{5.10}$$

$$P_{ij} = P(\pi_j(a^i) = True) \tag{5.11}$$

For *action sequences*, this reduces to a simple indicator matrix that, for each action-property-pair, contains a probability value that this combination is present. In the case of reliable observations, this probability will be 1, in other cases it reflects the observation uncertainty. For *tasks*, $\mathcal{P}_t$ is more complicated and depends on the properties of the problem at hand, as explained in the following sections.

The ordering relation $\mathcal{O}_t$ for a task T describes the probability that an action a^i is executed before an action a^j in the respective task context. In our system, this relation is learned from the

training set of sequences S^T_{train} and assigns a probability to each pair of actions $a^i, a^j \in \mathcal{A}_t$

$$\mathcal{O}_t : \mathcal{A}_t \times \mathcal{A}_t \rightarrow \mathbb{R} \tag{5.12}$$

The relative ordering of two actions is expressed using the $precedes$ predicate defined as

$$\forall a_i, a_j \in S_s : (i < j) \Leftrightarrow precedes(a_i, a_j, S_s) \tag{5.13}$$

Figure 5.16 illustrates how a sequence $1-2-3-4-5$ is translated into a set of pairwise ordering constraints, depicted by the black arrows. Sequences of observed actions are described by giving the types of actions ($actionT$), their ordering ($precedes$) and optionally their parameters (e.g. $objectActedOn$), for instance as

$$\begin{aligned} &activityT(Act_0) = MakeToast \\ &\quad \wedge\, actionT(N_1) = N1 \wedge objectActedOn(N_1, O_1) \\ &\quad \wedge\, objectT(O_1) = O3 \\ &\quad \wedge\, actionT(N_2) = N3 \wedge actionT(N_3) = N4 \ldots \\ &\quad \wedge\, precedes(N_1, N_2, Act_0) = True \\ &\quad \wedge\, precedes(N_1, N_3, Act_0) = True \wedge \ldots \\ &\quad \wedge\, precedes(N_1, N_2, Act_0) = True \wedge \ldots \end{aligned}$$

These models are implemented using Bayesian Logic Networks [Jain et al., 2009] from the ProbCog statistical relational learning library. Section 2.5.1 describes how the ProbCog algorithms are integrated into the KNOWROB knowledge base. Examples of the fragments of conditional probability distributions are shown in Figure 5.18, where the oval nodes denote random variables and the rectangular nodes contain preconditions for the respective fragments to be ap-

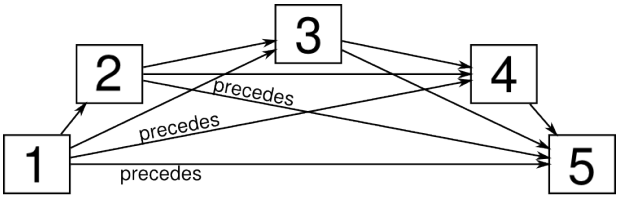


Figure 5.16 Describing the partial order in the sequence $1-2-3-4-5$ by pairwise precedence relations.

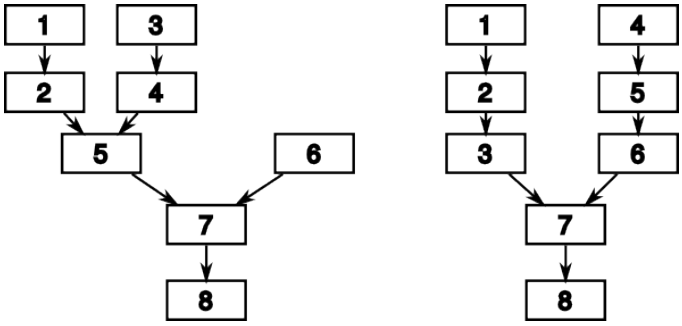


Figure 5.17 Precedence graphs for the fictional activities *MakeCoffee* (left) and *MakeToast* (right) which were used for sampling the synthetic action data .

plicable. Note that the manually specified structure only describes that "some action can precede some other action", whereas the actual relations between specific action classes are learned by the system.

5.7.2 Evaluation

We evaluate the system first on synthetic data, and then on two real-world data sets of human activities. The synthetic data set allows to test how the methods are affected by different noise levels in the data and to check if the actual partial-order graph can be reconstructed. The more complex real data sets, the TUM Kitchen Data Set and the CMU MMAC Data Set, provide realistic test data to verify that the methods perform well on real observations of human activities.

5.7.2.1 Synthetic Data

First, we tested the methods on synthetic action sequences sampled from the two precedence graphs in Figure 5.17. Note that both graphs consist of the same actions, i.e. no single action can be used as a hint which activity is performed, only the order contains information. This is certainly more difficult than most real-world applications, but required, for instance, when distinguishing between different styles of performing the same activity.

The sampling is performed using the following procedure: Let $\mathcal{N}$ represent the set of nodes whose ordering constraints are met and who can thus be selected in the next step, and let $prereq(n)$ be the set of nodes that are prerequisites for node n. The sampling starts with the set of nodes

$$\mathcal{N}_0 = \{n_n : \forall n_k \neq n_n \Rightarrow n_n \notin prereq(n_k)\}, \tag{5.14}$$

the set of all actions that are not prerequisites for any other action. At each sampling step i, a

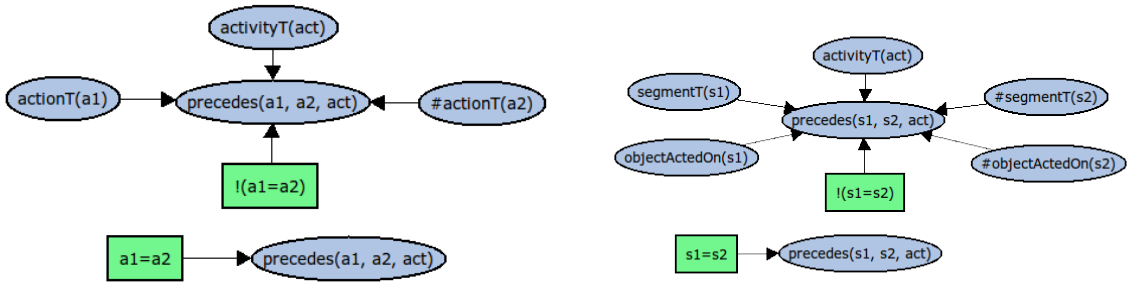


Figure 5.18 The model structure for the synthetic data (left) and the TUM kitchen data (right) with dependencies as conditional probability distribution fragments.

random element n_i is chosen, and the sampling continues with

$$\mathcal{N}_{i+1} = (\mathcal{N}_i \cup prereq(n_i)) \setminus n_i \tag{5.15}$$

All actions occur exactly once in this data set, i.e. for both graphs is $m = n = 8$, and there are no action properties, i.e. $\mathcal{P} = \emptyset$. The data can be modeled with the very simple BLN in Figure 5.18 (left). The blue, ellipse-shaped nodes represent the random variables, i.e. the logical terms and predicates. Square nodes are *decision nodes* that activate or deactivate a fragment. Here, they indicate if the two actions are identical, in which case an ordering relation would make no sense.

Learning the partial order The learning algorithm should be able to recover the partial order from the data. Figure 5.19 visualizes the conditional probabilities inside the $precedes$-node of the BLN. In this visualization, redundant relations have been pruned, i.e. when $P(precedes(A, B)) = 1$, $P(precedes(A, C)) = 1$ and $P(precedes(B, C)) = 1$, we did not draw the edge $A - C$ to improve clarity. As can be seen in the picture, the algorithm successfully recovered the partial-order structure the data was sampled from.

Interconnections that are not present in the original graph, for instance between the nodes $N1$, $N2$, $N3$, and $N4$, reflect the properties of the sampling algorithm. It is equally likely to switch to a different branch of the activity (i.e. between $N1 - N2$ and $N3 - N4$) and to continue the same branch. In observations of humans, such interconnections reflect an alternating behavior, as opposed to a stringent execution of each string of actions.

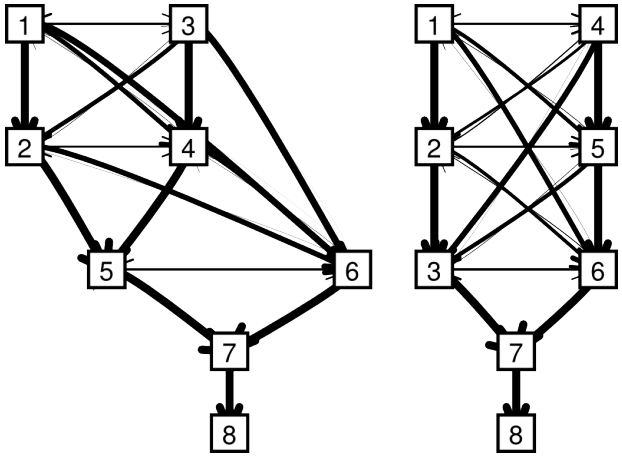


Figure 5.19 Learned dependencies in the synthetic data set. The thickness of the lines depicts the probability that one action is performed before another. The partial-order structure could successfully be recovered from the observed data.

Classification in the presence of noise Observations of activities are often obscured by irrelevant actions that are performed in between the essential actions for a task, like cleaning the countertop or drinking some water while cooking a meal. Errors during the segmentation into single actions also creates such action noise.

To test the influence of irrelevant actions in between the important ones, we modified the sampling algorithm described earlier so that, in each step, a noise action may be chosen instead of one of the relevant actions with a certain probability. Formally, equation (5.15) changes to

$$\mathcal{N}_{i+1} = (\mathcal{N}_i \cup prereq(n_i) \cup \mathcal{X}) \setminus n_i \tag{5.16}$$

where $\mathcal{X}$ is a set of noise actions, i.e. actions that are irrelevant to the activity. In the experiments, we sampled from $\|\mathcal{X}\| = 10$ noise actions, denoted $x_0 \ldots x_9$, with a probability of 10%, 20% and 50% respectively. The sequences in both the training and the testing set comprised these noise actions, so the system does not know a priori which actions are actually relevant.

Figure 5.20 (right) shows the classification performance (F1 value) of our system (inference on the ground BLN performed using the Backward Sampling algorithm [Fung and Favero, 1994] using 5000 samples). Even with the very noisy sequences, in which about half of the actions are not relevant to the activity, the system is still able to learn a model that allows for good classification. If there is few noise (lines without markers), as few as five examples suffice for reasonable performance, while the more noisy data requires about 15 samples for similar results.

We compare the classification results to those obtained from Hidden Conditional Random

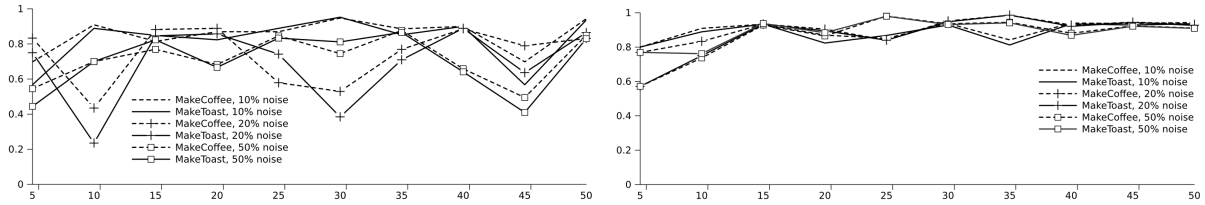


Figure 5.20 Recognition rates (F1 value) on synthetic data with different noise levels (10%, 20% and 50% probability of choosing a noise action) and sizes of the training and testing set (5 to 50 samples, see x-axis). Left: HCRF. Right: BLN (our approach).

Fields (HCRF, [Quattoni et al., 2004]), which have shown to outperform Hidden Markov Models and Conditional Random Fields, which are among the most common methods in action recognition. HCRFs directly model the sequence of actions, but cannot take longer-range dependencies like global ordering constraints into account. The results in Figure 5.20 suggest that the model gets confused by the large variation in the data and the significant amount of noise. While the results are still rather stable for low noise data (lines without markers), they get much worse when the proportion of irrelevant actions increases.

Inferring the types of single actions Besides classification, the models can also be used to infer the type of a single action in a sequence, e.g. in case of an ambiguous classification, as

$$argmax(P(actionT(a_i)|S^T)) \tag{5.17}$$

We randomly sampled sequences from the nosiest version of both activities (50% noise actions), removed the type of an arbitrary action in the test sequence, and inferred this type given the rest of the sequence. The exemplary results in Table 5.6 show that it is possible to infer the type of an action given the type of the activity and the surrounding actions.

ID	activityT	actionT	most likely types
12	MakeCoffee	N8	N8(0.5760), N7(0.4135), X5(0.0042)
25	MakeCoffee	N7	N7(0.4837), N5(0.2022), N6(0.0846)
33	MakeCoffee	N8	N8(0.7667), N7(0.2211), X5(0.0117)
43	MakeCoffee	N1	N1(0.5303), N3(0.4243), N2(0.0447)
24	MakeToast	N6	N6(0.2867), N3(0.2498), N7(0.1395)
37	MakeToast	N4	N4(0.5940), N1(0.3800), N5(0.0220)
48	MakeToast	N4	N4(0.3950), N2(0.2860), N5(0.1860)

Table 5.6 Inferring the type of unknown actions in an activity.

In addition, the results show which actions are easy to identify. Action N8, for example, is always the last non-noise action in every sequence and can thus easily be identified (seq. 12, 33).

When there is confusion, it is mostly between actions on a similar level of the precedence graph (e.g. N4 and N1 in seq. 37) or between direct predecessors and successors (as in seq. 25, where N5 and N6 are direct predecessors of N7).

Identifying (ir)relevant actions A priori, the system does not know which actions are relevant and which are just noise. Using the proposed models, the probability of an action given the activity can be calculated to decide which actions are most relevant to the task at hand.

$$P(actionT(a_i) = A^j | activityT(S^T)). \tag{5.18}$$

Table 5.7 shows that, even in the extreme case of 50% noise actions, the relevant actions are more consistent across the observed episodes and therefore have a higher probability. Since both activities consist of the same number of actions, the results are identical for both the $MakeCoffee$ and $MakeToast$ activity.

action	probability	action	probability
N1	0.83	X0	0.29
N2	0.83	X1	0.31
N3	0.83	X2	0.39
N4	0.83	X3	0.40
N5	0.83	X4	0.27
N6	0.83	X5	0.26
N7	0.83	X6	0.36
N8	0.83	X7	0.34
		X8	0.37
		X9	0.31

Table 5.7 Relevance of an action as its probability of occurring in an activity.

5.7.2.2 TUM Kitchen Data Set

We further evaluate the system on the TUM Kitchen Data Set described in Section 5.2. Since we are interested in modeling the actions on a rather abstract level of detail, we do not deal with the problem of segmenting the continuous motion (Section 5.4), but rather use the manually created labels provided with the data set.

All subjects in the data set perform the same activity (setting the table for one person), using the same objects, but in different order: Some behave like an inefficient robot that transports the objects one-by-one, others are more human-like in carrying several objects at once. On the one hand, this makes this data set quite structured, but on the other hand, it creates a difficult classification challenge since all objects and actions are identical for all classes.

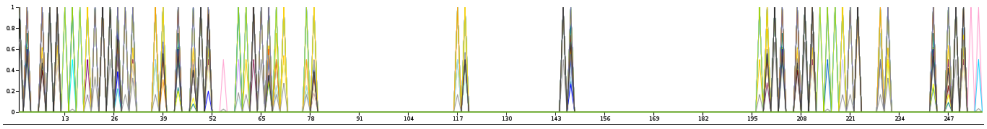


Figure 5.21 Conditional probability distribution of the precedes-node in the TUM data set. Each curve corresponds to the first action in a pair (a_1), the values on the x-axis denote the set $o_1 \times a_2 \times o_2$, and the value of the curve is the conditional probability that a_1 performed on o_1 precedes a_2 performed on o_2. The very peaked distribution indicates distinct ordering constraints.

In total, there are $m = 8$ classes like $Reaching$ or $OpeningACupboard$, and the observation sequences have a length k of about 70 action segments. $\mathcal{P} = \{objectActedOn\}$, the object an action is performed on, is the only property. The BLN structure for this data set is shown in Figure 5.18 (right).

Visualizing the learned model becomes difficult due to the influence of the object on the order. However, when plotting the conditional probability for each action a_1 over the object o_1 $\times$ the subsequent action a_2 with object o_2, a peaked, sparse distribution can be observed (Figure 5.21). Many values are zero because several object-action pairs never occur (like *opening a knife*). Some actions always occur before others (conditional probability of one), others have softer ordering constraints as can be seen by the lower peaks in the diagram. We noticed in our experiments that such sparse, peaked distributions are typical for problems that show a partial order.

Classification performance We tested the model by discriminating between two different styles of setting the table, in the following referred to as *robot-like* (transporting one object at a time) and *human-like* (a more natural behavior, including e.g. grasping all pieces of silverware at once). Due to a lack of sufficient data for the different cases, the test sequences were manually created to cover the different cases: A typical example of each activity style similar to the training data, though with a different order of the transported objects, additional noise actions, and shorter sequences where some object interactions were omitted. One sequence (*HumanRobot*) was constructed by concatenating the first half of a human-like and the second half of a robot-like sequence.

Table 5.8 presents the inference results obtained using Backward Sampling with 5000 samples and, as a comparison, the classification obtained from the HCRF (identical results for $m = 3, 5, 10, 20$ hidden states). Features for the classification were the action class and the manipulated object.The HCRF fails to classify the sequences and labels all of them as $Human$, supposedly because it did not learn the subtle differences in the ordering. Our system correctly

classified almost all the sequences, only the $HumanRobot$ sequence was classified as $Human$, whereas an indecisive result would have been expected. Apparently, the parts of the $Human$ sub-sequence are more salient than those in the $Robot$ part of the sequence. As mentioned before, all actions and objects are identical for both classes and only the order differs. In other cases, the distinction between different activities would obviously become much easier.

	BLN		HCRF	
activityT	SttHuman	SttRobot	SttHuman	SttRobot
Human1	**1.0000**	0.0000	**1**	0
Human2	**1.0000**	0.0000	**1**	0
Human1short	**1.0000**	0.0000	**1**	0
HumanRobot1	**1.0000**	0.0000	**1**	0
Robot	0.0009	**0.9991**	1	0
Robot1	0.0001	**0.9999**	1	0
Robot1short	0.2678	**0.7322**	1	0
Robot1noisy	0.2680	**0.7320**	1	0

Table 5.8 Classification results of different table-setting test sequences. Correct results are printed in bold font.

5.7.2.3 CMU MMAC Data Set

The CMU MMAC Data Set [De la Torre et al., 2009] provides observations of 43 subjects cooking 5 different recipes. So far, only part of the data has been labeled, namely a subset of the '*making brownies*' and the '*cooking an omelette*' recipes, which we use for learning the models. On this data, we will present some queries that show that the models do not only represent the ordering, but a complete joint probability distribution over different aspects of the observed actions.

Identifying (ir)relevant actions and objects The system does not know which actions or objects are relevant for a task, but can compute it using the learned models which contain information about which actions and objects appear consistently across all sequences.

P(actionT(A1) | inActivity(A1, Act)=True
∧ activityT(Act)=MakingBrownies
=⟨ TakingSomething:0.25,
PuttingSomethingSomewhere:0.15, Pouring:0.13,
OpeningSomething:0.13, ClosingSomething:0.08,
Stirring:0.08, Walking:0.03, TurningOnDevice:0.04,
Reading:0.03, [...] ⟩

P(objectActedOn(A1) | inActivity(A1, Act)=True
∧ activityT(Act)=CookingOmelette
= ⟨ Egg-Chickens:0.19, Cupboard:0.15, FryingPan:0.12,
VegetableOil:0.08, TableSalt:0.06, Bowl-Mixing:0.05,
Fork-SilverwarePiece: 0.05, [...] ⟩

Person-specific preferences As mentioned in the introduction, different people show substantially different behavior when performing the same task. One example are different ways of cleaning up after finishing a task: Some subjects put the frying pan back onto the stove, others put it into the sink after cooking. Such preferences are implicitly learned by the models.

P(doneBy(A1) | inActivity(A1, Act)=True
∧ activityT(Act)=CookingOmelette
∧ actionT(A1)=PuttingSomethingSomewhere
∧ objectActedOn(A1)=FryingPan
toLocation(A1)=Sink)
= ⟨ P3: 0.32, P5: 0.31, P4: 0.18, P0: 0.17 ⟩

The subjects P3 and P5 put the frying pan into the sink several times while cooking in order to drain some spare oil. The remaining subjects put the pan back onto the stove and not into the sink.

5.8 Discussion and related work

In this chapter, we presented AM-EvA, an integrated system for the observation and analysis of human everyday tasks. AM-EvA allows to automatically combine and integrate different kinds of information and information processing routines and to build up models that describe actions from the low motion level over the levels of motion segments and actions up to the task level. The knowledge-based representation allows robots to use the derived information directly for planning and for improving their own actions since the action classes used for interpreting human activities are the same as in the planning context.

Data sets The TUM Kitchen Data Set which has been recorded as part of this work was among the first to provide detailed observations of human everyday activities. The motion tracking data recorded in real-world scenes is still a unique feature of the TUM dataset. It turned out to be very well-accepted by the research community, which can be seen by the continously high number of downloads, but also by the papers that use the data. Though the data set is rather new, there are already papers at high-quality conferences (ECCV 2010 [Gall et al., 2010], ICPR [Krausz and Bauckhage, 2010]) and journals (PAMI [Gall et al., 2011]) that use the data for evaluation.

Other datasets of everyday activities, partially larger than ours, do not provide motion capture data that can be used by robots to learn motions from observations. The CMU Kitchen Data Set [De la Torre et al., 2009] contains multi-modal observations of several cooking tasks, including video data from calibrated cameras but only very few episodes have motion capture data. Due to the large number of actions and the high variation between the actors, this data set is extremely challenging for action recognition. Furthermore, the actors are heavily equipped with technical devices, making it difficult to evaluate e.g. markerless motion trackers on the video data. The Opportunity dataset [Roggen et al., 2010], recorded later than the TUM dataset, provides 25 hours of observations of twelve subjects performing activities of daily living like making sandwiches. The human subjects are highly instrumented and the dataset contains 10 modalities of 72 sensors, including accelerometers and microphones attached to the human body and to objects of interest, cameras, position tracking system, and reed sensors in furniture. The data set is much larger than the TUM data set, but less detailed in terms of observed motions and therefore less suited to learning motion models. Uncalibrated video data and RFID readings (without motion capture data) are provided by the LACE Indoor Activity Benchmark Data Set[1], which is mainly targeted towards a coarser, general description of activities. Even coarser, more higher-level data

[1] http://www.cs.rochester.edu/∼spark/muri/

is available in the MIT House_n Data Set[2], which is aimed at recognizing activities as a whole, while we are also interested in modeling the single actions and even different motions an activity consists of.

Several motion-capture-only data sets such as the CMU Motion Capture Database[3] or the MPI HDM05 Motion Capture Database[4] are available that provide large collections of data. These motions are extremely articulated, well separated, and do not resemble natural everyday activities, nor do they involve manipulation or interaction tasks. The Karlsruhe Human Motion Library [Azad et al., 2007] consists of motion capture data specifically recorded for human motion imitation on humanoid robots.

Segmentation of human motions The segmentation and classification of motions is often referred to as "activity recognition". An excellent overview of different approaches and representations can be found in Krüger's survey paper [Krüger et al., 2007]. In general, there are two main kinds of segmentation methods, either working in an unsupervised way, or supervised based on manually annotated training data.

Unsupervised segmentation methods usually perform either some kind of clustering of similar recurrent motions [Zhou and De la Torre, 2009; Kulic et al., 2009] or split motions based on intrinsic properties like a minimum of motion energy [Weinland et al., 2006]. The former class of systems performs especially well for structured, repetitive motions like walking or running, since correspondences can easily be found and repetitions are numerous. The latter are well-suited if actions are well-separated and if foreground actions involve significantly more motion than background movements. Both is not the case for mobile manipulation actions, which makes unsupervised methods less suited for this kind of problem. Another reason why we opted for a supervised method, though this requires manual annotation of training data, is that we would like the system to identify segments that do not only resemble each other from a kinematic point of view, but are also meaningful for humans. Such segments with well-defined semantics can only be obtained by a supervised segmentation technique. In our system, a standard classifier based on Conditional Random Fields proved to perform well enough for our tests.

Trajectory models The work on learning trajectory models can be seen in the context of imitation learning [Schaal, 1999]: Robots observe human actions, analyze and abstract the observed data, and use them to imitate the actions. An overview of the research in this area can be found

[2] http://architecture.mit.edu/house_n/data/
[3] http://mocap.cs.cmu.edu
[4] http://www.mpi-inf.mpg.de/resources/HDM05/

in [Billard et al., 2008]. Recent approaches learn motions for instance as differential equations [Pastor et al., 2009] or Gaussian Mixture Models [Calinon et al., 2010]. These methods assume that the motions are explicitly demonstrated, i.e. that everything that has been observed is to be imitated. In contrast, we approach the problems of deciding *what* to imitate and of learning in the task context: The robot observes complete activities like setting a table and extracts models of single motions from this data. This requires methods to identify distinct trajectory clusters and to select a suitable one based on its semantics.

An approach for clustering human reaching trajectories using point distribution models (PDM [Roduit et al., 2007]) has been presented by Stulp [Stulp et al., 2009]. PDMs are learned by first performing a Principal Component Analysis (PCA) and then clustering the data using the Mahalanobis distance. However, their experiments use very clean, pre-segmented trajectories and the approach did not perform well on our more noisy data. Further, both [Roduit et al., 2007] and [Stulp et al., 2009] assume that high variance implies the existence of clusters, but, as pointed out by [Ding et al., 2002], the subspace obtained using PCA does not necessarily coincide with the subspace spanned by the cluster centers, especially in real-world clustering problems without well-separated clusters.

Hierarchical action models The logic-based action representation helps to overcome the problem of different granularities at which actions are described, since descriptions on several levels of abstraction are integrated into a single model. The model further describes the semantics of the actions on the different levels by linking them to objects, locations, and other action properties. To build up the hierarchy, the abstraction procedure exploits the same action information that is also used for planning, i.e. the information about sub-actions has to be provided only once and can be used both for task decomposition in the planning context and for action recognition. There are few other systems that use logical representations for interpreting observed actions. Landwehr [Landwehr et al., 2009] uses logical transformation rules for modeling activities, but without representing a hierarchical structure. Plan recognition systems provide in-depth action analyses, but require detailed specifications of the complete plans which are then matched against the observations [Kautz and Allen, 1986; Goldman et al., 1999]. In contrast, our approach does only require local sub-action relations and no knowledge of the global plan.

Since the abstraction is based on deterministic rules, it can potentially fail if the sequences do not match completely and that are sensitive to noise in the action sequences. A probabilistic abstraction method could probably improve robustness: There are approaches to building hierarchical models using statistical models, for instance multi-layer Hidden Markov Models [Padoy

et al., 2009; Luhr et al., 2003] or Dynamic Bayesian Networks [Park and Kautz, 2008]. However, these systems are limited to two fixed levels of abstraction and are not relational, i.e. the types of actions they can be applied to needs to be known when creating the model. Statistical relational models would help, but are likely to suffer from scalability problems, especially for large numbers of action classes and longer action sequences. Stochastic parsing techniques have been applied to action recognition and also allow to create hierarchical representations if the grammar supports it [Ryoo and Aggarwal, 2006; Ivanov and Bobick, 2000]. However, the resulting models do not describe the semantics of the involved actions and their relations to objects.

In the current implementation, the estimated types of the higher-level actions are not used to improve the analysis on the lower levels of the hierarchy. The combination of the bottom-up abstraction with some top-down information exchange will be an interesting aspect to investigate in the future.

Partially-ordered action models Modeling the partial order inherent in human activities is a rather new topic. Many of today's approaches for activity recognition are using sequence-based methods like Hidden Markov Models (HMMs) [Patterson et al., 2005], Conditional Random Fields (CRFs) [Vail et al., 2007] or Suffix Trees [Hamid et al., 2007]. These models have in common that they directly describe the observed sequences by local action transitions, and they are based on the Markov assumption that the transition to the next action only depends on the current action.

This assumption is valid for actions described on a lower level: After reaching towards an object, a person is usually either picking it up or, if the object is the handle of a door or drawer, opens this container. The state transition is rather independent of prior actions in the task, which is why models that make the Markov assumption (HMM, CRF) work fine for this kind of data.

Interestingly, the Markov assumption does not hold any more on a more abstract level, and the partial order becomes dominant: The likelihood for picking up a specific kind of object changes significantly depending on which task is being performed and which objects have been manipulated beforehand. This is the case for household activities, assembly tasks in a factory, or many games.

Many of today's action recognition data sets do not show a partial order since they have been recorded in a controlled setting in which the sequence of actions is completely determined. In this case, there is a total ordering among all actions, and no real dependency constraints between the actions can be learned. This is, however, not due to the structure of the activities, but only due to the unnaturally constrained recording setting.

There are a few other systems in the area of action recognition that can deal with partially ordered tasks, like the one by [Shi et al., 2004] using manually specified Dynamic Bayesian Networks to represent the partial order, or the system presented by [Gupta et al., 2009]. In that paper, actions in baseball games are arranged in an AND/OR graph that describes possible action sequences and alternatives. [Bui et al., 2008] present the "Hidden Permutation Model" that learns the partial order of actions in a similar way as our system with a focus on increasing scalability. However, these models do not describe action properties and thus do not allow reasoning beyond the partial order of action types.

In the field of planning (e.g. [Penberthy and Weld, 1992]) and plan recognition, there is much work about partially ordered plans. Kautz and Allen's seminal paper [Kautz and Allen, 1986] formalizes plan recognition as a logical inference problem. [Goldman et al., 1999] extend this work to a probabilistic model that can handle partially ordered and interleaved plans. These approaches rely on a manually created model and have mainly been applied to synthetic problems so far. The system presented in this paper differs from those approaches in that it *learns* a model that is able to describe complex tasks including their partial order from observed data.

The area of preference learning also deals with learning and representing orderings, though 'partial order' is usually meant as a total order among the top-k elements in a set, as opposed to a partial ordering of all actions. [Kirshner et al., 2003] learn partial-order relations in astronomic data.

In terms of scalability, models representing a partial order are much more complex compared to those describing only a sequence, having huge space requirements depending on the length of the sequence, the number of actions and the parameters. In practice, the conditional probability table representing the precedence relation is often sparse: Some combinations of actions and objects do not make sense and thus have zero probability, see Figure 5.21. Therefore, the table can efficiently be represented using decision trees [Friedman and Goldszmidt, 1996]. Even without such optimizations, our implementation smoothly handles inference in models of about 40 segments with about 10 action and object classes. Learning BLNs is generally much easier than inference because parameter learning of Bayesian networks comes down to counting. Training on 20,000 sequences runs very fast without problems.

Chapter 6

Knowledge-enabled decision making

This chapter deals with the problem of acquiring information during operation, namely from the perception system, and combining this information with the robot's background knowledge for taking informed decisions. In the example scenario, the robot is at this step supposed to have generated an effective task description that contains all required actions and objects. To actually execute this task, it now needs to ground the abstract object descriptions into its perception, make sure all required object models are available and retrieve them otherwise, and it needs to reason about how to translate abstract action descriptions like "open the container where you think cups are stored in" into real-world actions.

While the previous chapters introduced the knowledge processing framework and the representations for the different kinds of knowledge, we now focus on those components that are needed for on-line operation on the robot and the integration with the robot's perception and control system that are required to perform grounded symbolic inference about the robot's percepts and actions. This integration is thereby more than just a technical aspect: The symbolic descriptions in the knowledge base are only meaningful if they are *grounded*, that is, if they are closely linked to the data the robot perceives and the actions it does.

The first section in this chapter discusses how the knowledge base can exchange information with the perception system, both information about object models and information about the actually perceived objects. The following one describes how the environment model represented using the methods described in Section 3.2.4 can help in the decision process. We continue with a description of RoboEarth, a project that targets at building a large web-based knowledge base that robots can use to autonomously exchange knowledge between each other. In the pancake example, RoboEarth could be used to download object models as well as environment models, and can be used to share the effective task description generated as the result of the process described here.

6.1 Integration of perceptual information

Interfacing a knowledge base with a perception system requires integration on two levels: the level of object recognition models, which enables the robot to reason about which objects can potentially be recognized, and the level of object detections using these recognition models, which allows it to describe and reason about object instances in the environment. KNOWROB includes interfaces to several vision components like the CoP perception system [Klank et al., 2009], the K-COPMAN system [Pangercic et al., 2010], and the RoboEarth object recognition component [Waibel et al., 2011].

In the following, we will focus on the integration with the CoP perception system, which is the most mature and complete one. We created an interface to synchronize CoP's model database with KNOWROB to inform the knowledge base about newly created object models. Whenever a new object is added to the CoP database, the signature of the model is announced to KNOWROB and added to the set of available models in the knowledge base. Based on this set of models, the robot can decide if it can recognize a kind of object or not.

Representation of object recognition models and algorithms When introducing our approach to representing object poses (Section 3.2), we described the taxonomy of *MentalEvents* depicted in Figure 3.4. Those classes form the upper level, categorizing the main kinds of events that can create beliefs about object poses. We thus extended the taxonomy of *MentalEvents*, namely the *VisualPerception* branch, with the variety of vision algorithms provided by the CoP system, in order to describe in much more detail which algorithm was used in conjunction with which recognition model to detect the respective object. Figure 6.1 gives an overview of the various algorithms for detecting objects and for estimating their positions. Most of these are general-purpose algorithms that need to be parametrized with a suitable object recognition model to detect objects of a certain kind. The corresponding taxonomy of object recognition models describes the different kinds of models (Figure 6.2). Detections of objects are linked to instances of these models using the *perceivedUsingModel* property, which allows to describe exactly using which algorithm and which model some belief about an object entered into the knowledge base.

Different algorithms have different properties in terms of input data they require and output data they produce: Some can only provide information about the presence or absence of objects, others compute the position in space or even the six-dimensional pose of an object. They further differ in the sensors they can use and in the computation complexity. If the robot does not intend to manipulate an object, information if something is present or not may be sufficient and can often be provided by rather simple methods, while manipulating an object requires to have exact

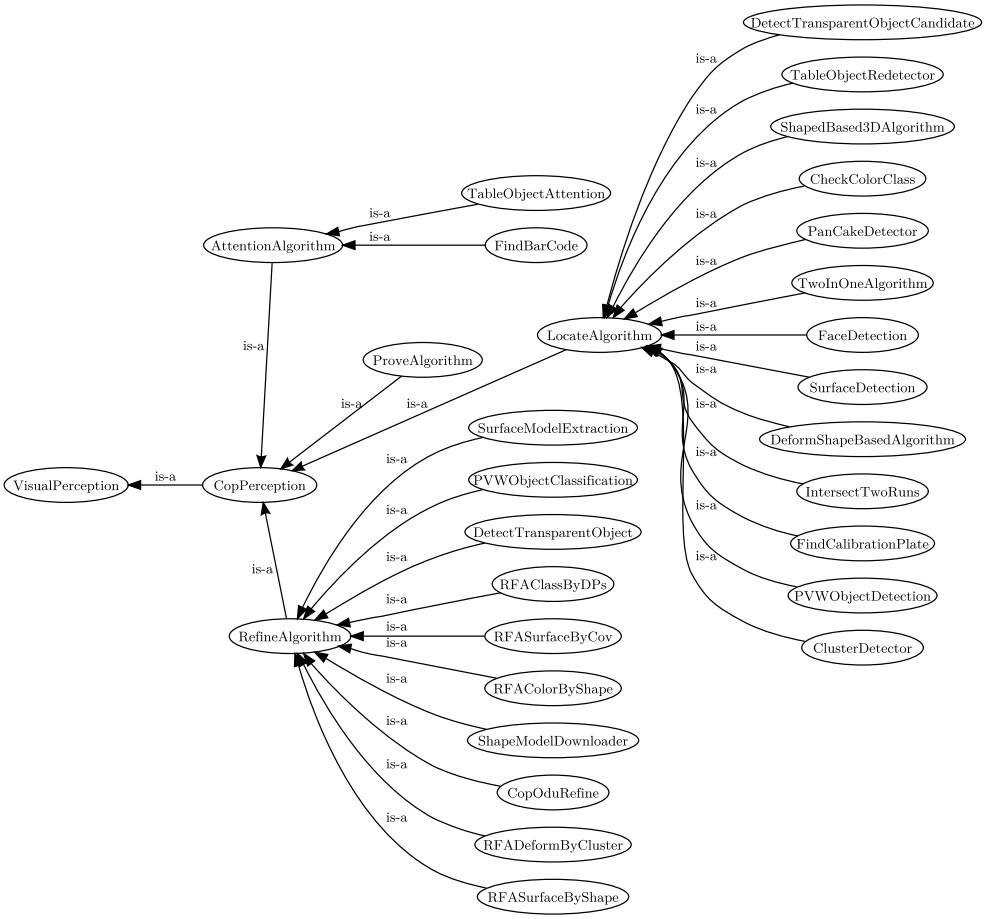


Figure 6.1 Classes of perception algorithms provided by the Cop perception library, represented as specializations of *VisualPerception*.

information about its pose. Detailed descriptions of models and algorithms are also important for exchanging information (see Section 6.3). To determine which algorithms are required to make use of a model, and which models can be used to recognize which object, the system has to be able to describe them in sufficient detail.

Object perceptions There are different perception systems that need to be interfaced in different ways: Some perception methods perform recognition on demand, others continuously match models against the current sensor data. In the former case, the communication is performed synchronously using a request-response based scheme, in the latter one asynchronously by passively listening to the published object detections. In the context of the ROS middle-ware, the former is interfaced using service calls, the latter one by listening to detection results that are continuously published on a so-called “topic”.

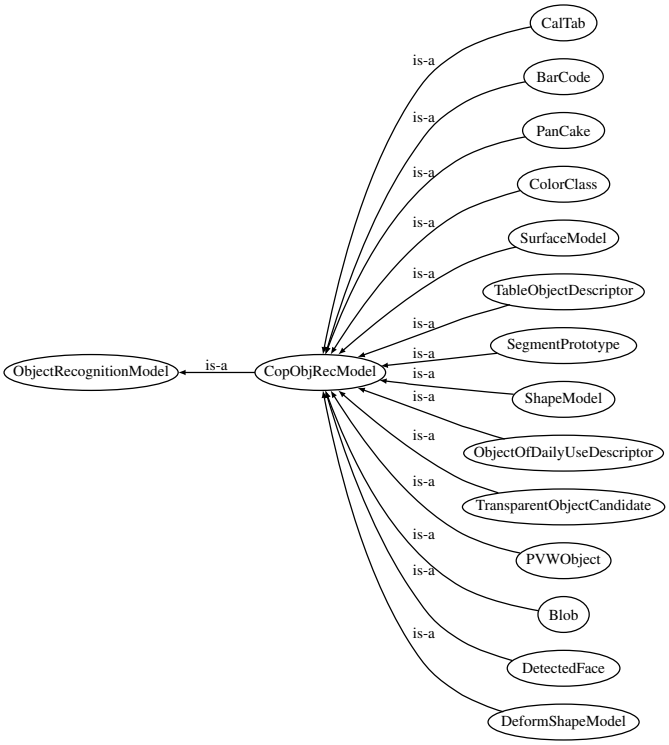


Figure 6.2 Types of object recognition models in the Cop perception system.

Figure 6.3 visualizes the two kinds of interfaces. In the case of synchronous communication (upper left), the integration with KNOWROB is solved using computables: A computable class with an appropriate *target* is defined which is called automatically whenever someone queries for objects of that respective type. The computable then sends a request to recognize this kind of object to the perception system, and creates the object representation described in Section 3.2 for all detected objects returned in the result set. The types of these object instances are determined based on the response of the perception system. This allows to create a computable for a generic class like *HumanScaleObject*, which then generates instances of more specific classes like cups, plates, or forks. This kind of interface is for example used to read information from the Willow Garage *tabletop_object_detector* [1].

The second kind of interface, based on asynchronous communication, listens to all perception results and adds them to the knowledge base. This listener is running in a separate thread in parallel to the KNOWROB engine, receives all object detections that are published on the topic and creates the respective perception instances for them (lower left block in Figure 6.3). This second kind of interface is for example used to read information from the CoP vision system.

[1] http://www.ros.org/wiki/tabletop_object_detector

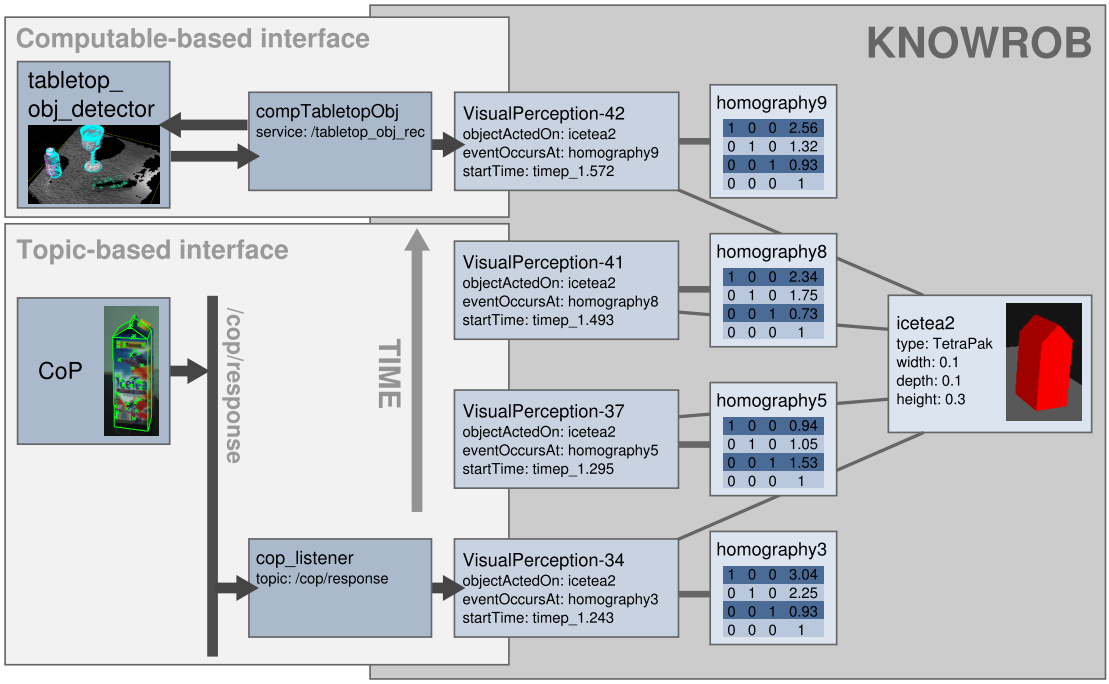


Figure 6.3 Query-base and topic-based interface between the knowledge base and the perception system to include detections of objects into the knowledge representation.

6.2 Environment information

Semantic models of the environment are of major importance to a robot performing object manipulation tasks. In Section 3.2.4, we explained how environment maps are represented in KNOWROB and will now report on how this information is used to accomplish tasks. Several environments have been represented in this format. Figure 6.4 shows some examples how this information can be used to locate appliances based on their types or the purpose for which they are used, like a *HeatingDevice* (left), a device for *WashingDishes* (center), or a *CoolingDevice* (right).

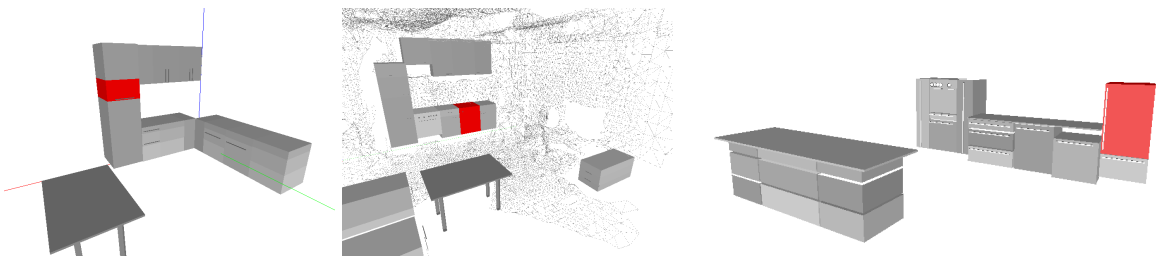


Figure 6.4 Examples of semantic environment maps represented as object instances in KNOWROB.

In order to ground the abstract object descriptions that are used in plans, a robot needs to locate the respective objects in the environment and decide where to search for them. A similar problem arises when the robot needs to bring something back where it belongs. In both cases, the robot needs knowledge about the organizational structure of the environment. A simple solution is to use general knowledge about the types of objects and their properties. Perishable objects, for example, must be cooled and are stored in the refrigerator, frozen food belongs into the freezer, silverware into a drawer, and tableware into cupboards. Figure 6.5 explains how such information can be encoded and used. The upper left part of the picture shows the taxonomy of household appliances and other super-classes of *Refrigerator*, while the upper right section is a part of the food ontology. The description of the concept *Refrigerator* contains the statement that refrigerators are storage places for perishable goods. Due to the hierarchical structure and concept inheritance, this relation applies to all instances of sub-classes of *Perishable*, namely *DairyProduct* and *CowsMilk-Product*. This kind of inference is especially useful if the robot does not have much knowledge about the environment it is operating in, apart from generic class-level knowledge.

If the robot knows the locations of some objects in the environment, it can estimate the correct locations for novel objects based on their similarity to known objects. If we assume that

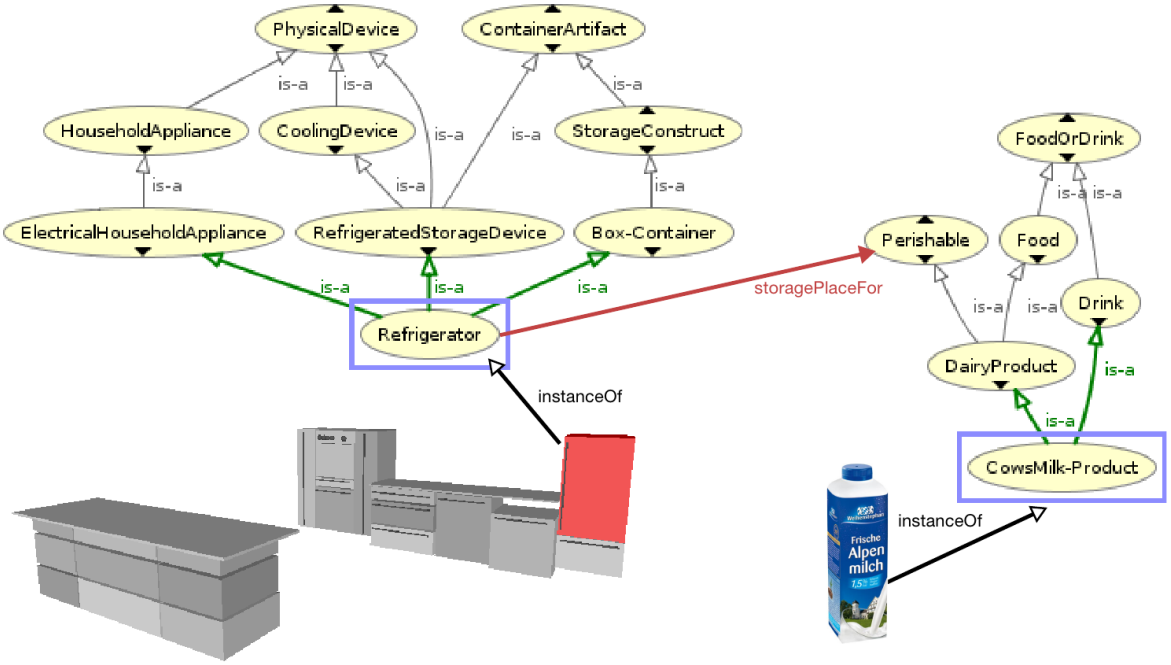


Figure 6.5 Locating objects based on knowledge about their properties and about the environment.

similar objects are usually placed together, the "semantic similarity" of the object classes in the ontology proved to be a good indicator where to search for objects. We chose the *wup* similarity measure [Wu and Palmer, 1994] that is defined as

$$sim(C_1, C_2) = \frac{2 \cdot d(S)}{d_S(C_1) + d_S(C_2)} \tag{6.1}$$

where S is the least common superconcept of C_1 and C_2, $d(C)$ is the (lowest) depth of concept C in the ontology, and $d_S(C)$ is the (lowest) depth of concept C in the ontology when taking a path through superconcept S of C. Table 6.1 shows some examples of objects and their similarity to cups, cooking pots, and cutlery.

	glass	plate	salad bowl	platter	knife	spatula
Cup	**0.78**	0.67	0.67	0.67	0.52	0.52
Pot	0.67	0.67	0.67	0.67	0.6	0.7
Cutlery	0.58	0.58	0.58	0.58	**0.78**	0.76

	cakepan	colander	pasta	cereals	mop	detergent
Cup	0.67	0.53	0.5	0.53	0.53	0.53
Pot	**0.78**	0.7	0.5	0.53	0.6	0.6
Cutlery	0.6	0.6	0.48	0.5	0.6	0.6

Table 6.1 Concept similarity based on the KNOWROB ontology.

After the initial publication in [Tenorth et al., 2010a], the concept of using semantic similarity measures to discover organizational principles in kitchens has been further investigated afterwards by Martin Schuster, leading to a paper that compares different techniques for learning the organizational structure [Schuster et al., 2011].

Having chosen a location for an object using either of the methods presented beforehand, the robot needs to generate actions to retrieve them. This includes the translation of abstract instructions like "open the container in which you expect to find a cup" into a trajectory for the robot to pull open the correct drawer or container door. These trajectories can be obtained from articulation models that have been generated by the robot that created the semantic environment map [Blodow et al., 2011], and that are stored in the semantic map as attachment to the respective containers. Figure 6.6 visualizes the different kinds of opening trajectories (linear, circular, s-shaped) for the different containers (left picture) and the result for a query for the container in which the system expects to find a cup (right picture).

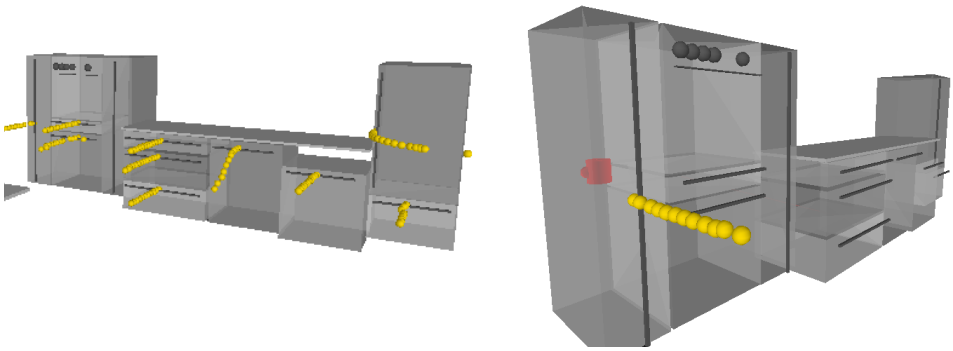


Figure 6.6 Semantic environment maps with opening trajectories for different kinds of containers in the kitchen.

6.3 Knowledge exchange between robots

The Web 2.0 has changed the way how web content is generated. Instead of professional content providers, it is now often the users who fill web sites with text and images, forming a community of people helping each other by providing information they consider useful to others. The free encyclopedia Wikipedia grew up to millions of articles, sites like *cooking.com* or *epicurious.com* collect tens of thousands of cooking recipes, and *ehow.com* and *wikihow.com* offer instructions for all kinds of everyday tasks. "Crowdsourcing" knowledge acquisition has proven to greatly speed up the generation of web content, and we are trying to make use of it in order to improve the performance of our robots. We are working towards a similar "World Wide Web for Robots", a web-based community in which robots can exchange knowledge *among each other*. With this approach, we intend to speed up the time-consuming knowledge acquisition process and let robots profit from tasks other robots have already learned, object models they created, and environments they explored.

If such information is to be generated and used by robots, that is, without human intervention, it has to be represented in a machine-understandable format. In this respect, we have much in common with the Semantic Web [Lee et al., 2001], in which computers are to exchange information between each other: Content should be separated from rendering, it has to be represented in terms of logical axioms that a computer can understand, and these logical axioms need to be well-defined, for example in a central ontology. Such an explicit representation of the semantics is important to enable a robot to *understand* the content, i.e. to set different pieces of information into relation. Only when it knows the semantics of the exchanged information, a robot can decide if some information can be useful for a task it has to perform, find out which piece of information

to choose among different alternatives, and decide if it has all requirements for using it.

Conveying these semantics needs to be supported by the representation language, but though there are several languages for describing different kinds of robot information, they usually do not provide semantic descriptions. Examples are AutomationML [Drath et al., 2008], used in industrial applications, the FIPA [O'Brien and Nicol, 1998] standard, which primary deals with the definition of communication standards for software agents, or XABSL [Loetzsch et al., 2006], mainly used in the RoboCup soccer context. Object description formats like the proprietary DXF [Rudolph et al., 1993] or the open Collada [Arnaud and Barnes, 2006] standard describe objects using meshes and textures, but without further specifying semantic properties.

Here, we describe our approach to defining a semantic representation language which we realize as a combination of the following components:

- Representation of the information to be exchanged, like sequences of actions and their parameters, or positions of objects in the environment;
- Meta-information, describing which kind of data is being exchanged (e.g. a CAD model of an object class, saved as DXF file, together with the units used, etc);
- Specifications of requirements for a piece of information to be usable (e.g. certain sensors or higher-level capabilities);
- Robot self-models that describe the robot's configuration and capabilities; and
- Methods for matching the requirements of a piece of information to the capabilities of the robot to determine what is missing and needs to be retrieved.

The representation language is realized by extending the representations for actions, objects, and environment maps in KNOWROB with representations for those kinds of information that are specifically needed in the context of exchanging knowledge. The work on knowledge exchange is done as part of the RoboEarth project [Zweigle et al., 2009] which targets at building a "World Wide Web for Robots" and covers different aspects like the generation and execution of task descriptions, sensing, learning, a central web knowledge base, apart from the methods for representing and reasoning about the exchanged knowledge described in this work. We here assume the other components of the system to exist.

Figure 6.7 shows the part of RoboEarth that is relevant here: On the left is the central RoboEarth knowledge base, containing descriptions of actions (called "action recipes"), objects, and environments. These pieces of information have been created by different robots with different sensing, acting and processing capabilities. Therefore, all of them have different requirements on capabilities a robot must have in order to use them. The proposed language thus provides

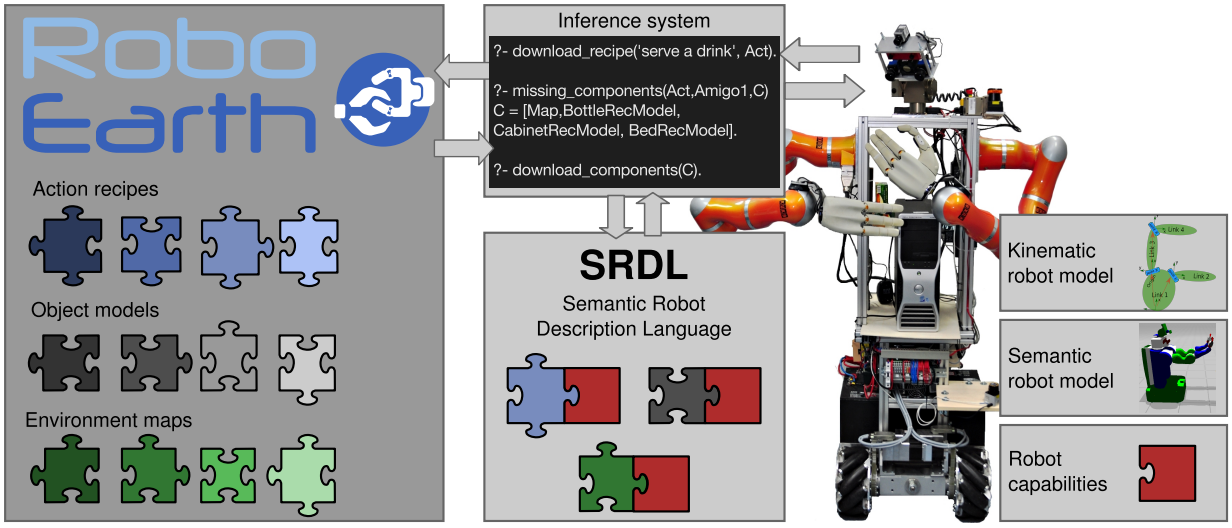


Figure 6.7 Overview of the RoboEarth system: A central database provides information about actions, objects, and environments. The robot can up- and download information and determine if it can use it based on a semantic model of its own capabilities.

methods for matching these required capabilities against those available on the robot. Each robot has a self-model consisting of a description of its kinematic structure, including the positions of sensors and actuators, a semantic model that describes the meaning of the robot's parts (e.g. that certain joints form a gripper), and a set of software components like object recognition systems. We use the Semantic Robot Description Language (Section 3.5) to describe these components and the capabilities they provide, and to match them to the requirements specified for action recipes. Section 6.3.3 explains the matching process in more detail.

6.3.1 Knowledge representation for exchange

The exchange of knowledge creates additional challenges for the robot's knowledge representation in addition to the description of the information itself. First, one needs to ensure that the uploading and the receiving robots share a common vocabulary to describe information, and that this vocabulary can be extended in a distributed way. Another requirement is that the descriptions provide meta-data about the information that is being exchanged to allow robots to interpret the data and decide if it is useful to them at all. Further, the description formats should remain compatible to existing data formats and rather link to files in established formats than to create new ones. The following paragraphs discuss these problems and our solution approaches.

Central upper ontology, distributed extensions RoboEarth is intended to be a large, distributed system that can be used by various different robots, performing many different actions in very diverse environments. In such a setting, it is impossible to predict all kinds of actions or objects that need to be described. Instead, we need a flexible way to extend the language to new application domains. At the same time, it is important to have a common basic language that describes the most important and generic concepts (like the relations between actions, objects, grasps, and trajectories) that is the same for all communicating partners. Otherwise, a robot would not be able to understand a description it downloads from RoboEarth. We therefore chose the following setup: A common central ontology describes the main concepts that are required. This ontology provides the language elements described in this report and supports the reasoning capabilities. It can be extended by every user by deriving special classes or properties from the generic ones. This way, robots still have an approximate idea of the meaning of a piece of information and can set it in relation to other information. This central ontology is an extension of the KNOWROB ontology with the exchange-specific concepts described in this section.

Meta-data for describing knowledge Information about the creator of a recipe, the creation time or the location may be important for selecting a good recipe: If a very similar robot created it, the likelihood that it will work will be higher. The older a map is, the more likely it is outdated. This information can be described using the *createdBy* and *creationDateTime* properties. The number of times a recipe has been downloaded, the amount of successful executions, or the kinds of robots it has been tried on are valuable pieces of information when trying to select one recipe among several alternatives. Another kind of meta-data are dependencies specified in SRDL which can be used for matching requirements of pieces of information to the capabilities of a robot.

Links to external data files While the language provides means to describe various kinds of information, there are already established and optimized file formats for many applications: Collada [Arnaud and Barnes, 2006], for instance, is a widely used format for describing kinematics. Other modules, like object recognition systems, have their own file formats that allow to efficiently store the required information. On the one hand, we try to keep these data formats to ensure compatibility and to profit from existing tools. On the other hand, we need to describe the semantics of the exchanged data. The language thus allows to add links to external data files to the OWL representation. In this case, the information content (an object model, an occupancy grid map, etc) is stored in an external file, and a description in the RoboEarth language adds meta-information like the prerequisites that are required to use the model.

6.3.2 Action descriptions

Figure 6.8 visualizes an action recipe for serving a drink to a patient in bed. In this picture, action classes are visualized as blocks, properties of these classes are listed inside the block, and

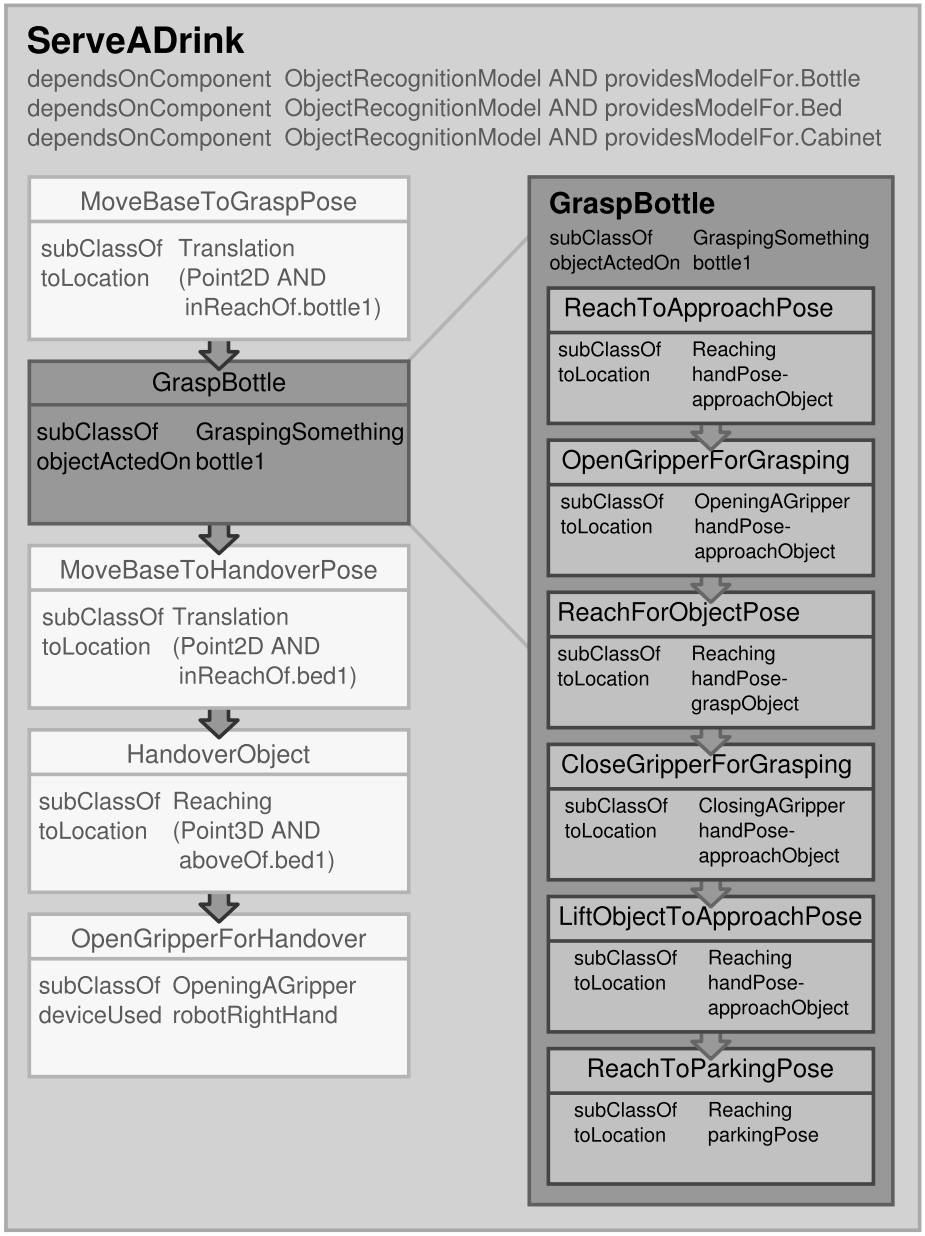


Figure 6.8 Representation of a "serving a drink" task, called "action recipe" in the RoboEarth terminology, which is composed of five sub-actions that themselves can be described by another action recipe.

ordering constraints among the actions are shown as arrows between the blocks. There are three levels of hierarchy: The recipe for the *ServeADrink* action includes the *GraspBottle* action that, by itself, is defined by an action recipe (shown on the right side) consisting of single actions. Both recipes consist of a sequence of actions that are described as task-specific subclasses of generic actions, like *Reaching* or *Translation*, with additional parameters, like the *toLocation* or the *objectActedOn* as described in Section 3.3. The hierarchical structure allows the robot to recursively retrieve additional action recipes for sub-tasks if it does not know how to execute them. If it already has a plan for these tasks, it does not need to download these descriptions any more.

The action recipe further lists dependencies on components that have to be available on the robot in order to successfully perform the task. In this example, there is a list of object recognition models that are necessary for the robot to be able to recognize all objects involved in the task. In addition to these requirements are those defined at higher levels of the abstraction hierarchy. All specializations of *Translation*, for instance, require some kind of moving base to be executable.

6.3.3 Matching available and required capabilities

In order to find out if the robot is able to execute a recipe and, if not, whether the robot can *be enabled* to do so by downloading additional information, the system matches the requirements of the action recipe to the robot's capability model. If needed, the robot can for instance load more detailed recipes, object models, or an environment map. While we cannot guarantee that the robot will be able to successfully execute the action described in a recipe – which can still fail for various reasons –, we can at least compute if the robot has all information it needs based on the information we have about the task. The goal is thus not to guarantee that an action can be executed, but rather to check whether something is definitely missing and if that missing part can be provided by RoboEarth.

The matching process is realized using the Semantic Robot Description Language (Section 3.5) and visualized in Figure 6.9. The robot first queries for an action recipe and, together with the query, sends a description of its own capabilities to the inference engine, which then checks whether all requirements of the recipe are available on the robot. After the first check, the *EnvironmentMap* is found to be missing on the robot.

Without semantics, the system would reject the request since it does not know anything called *EnvironmentMap*. But knowing the semantics of an *EnvironmentMap*, it can infer that both maps are specializations of an *EnvironmentMap* and can thus be used to fulfill the dependency. The matching process is applied recursively until the system finds a combination of action recipes,

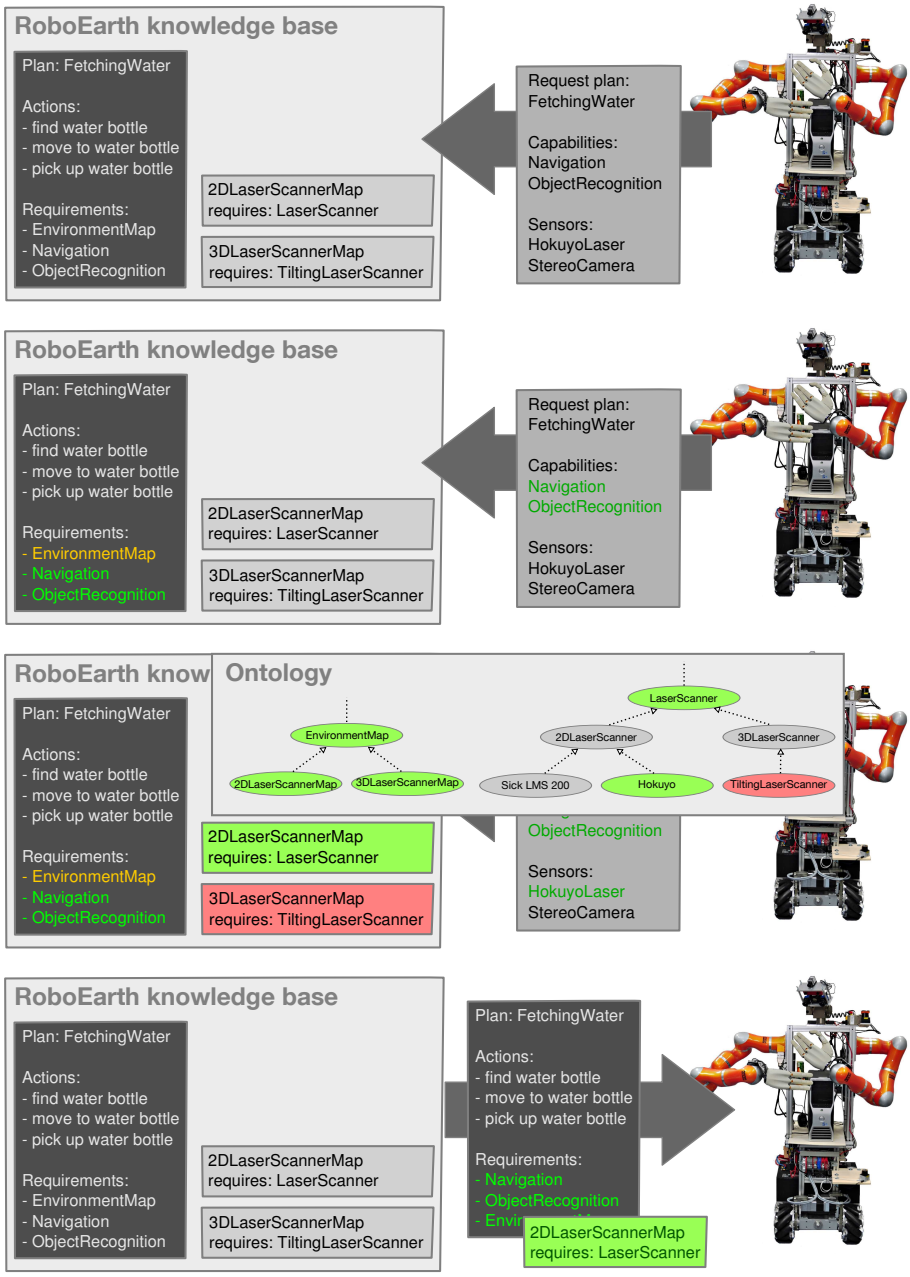


Figure 6.9 Matching requirements of action recipes against robot capabilities to determine which further information is still missing and has to be downloaded. The matching becomes more flexible by taking the robot's knowledge into account and selecting the *2DLaserScannerMap* to fulfill the general requirement on an *EnvironmentMap*.

object- and environment models that fits the robot and does not have any unmet dependencies. The result is a set of software components that need to be retrieved from RoboEarth in order to be able to perform the task.

6.3.4 Grounding abstract descriptions of actions and objects

For executing the abstractly specified action recipes, the robot has to ground the instructions into its perception and action execution system.

Grounding action descriptions Action recipes can be decomposed hierarchically into more and more basic actions, but at some point, these action descriptions have to be translated into executable code on the robot. For executing the recipe, the robot then needs implementations of all primitive actions the task could be decomposed into, and an execution engine that coordinates these action primitives. The mapping from action descriptions to executable primitives is done using the *providedByMotionPrimitive* property – usually not in the action recipe itself, but on the next higher level of abstraction. The subclass in the action recipe then parametrizes the primitive with respect to the task at hand. In the above example, the mapping is defined between the class *Reaching* and the *move_gripper* primitive. This reduces redundancy because the mapping can be performed for a whole class of actions at once, which helps to keep the exchanged recipe simple. For the rather simple scenario in the experiment, only three primitives are used: *move_gripper*, *open_gripper* and *navigate*.

There is intentionally no fixed level of granularity at which the transition between action recipes and the executable action primitives takes place. Both large, monolithic implementations of functionality, having the threshold at a rather high level, and combinations of small functions are supported. The hierarchical structure of recipes allows to flexibly decide if the robot is able to execute a high-level recipe using implemented functionality or if it has to download specialized recipes that are further decomposing an action.

Grounding object instances Inside action recipes, objects are described using temporary object instances. This description on the instance level is required to ensure that all actions in a plan are performed on the same object, not on arbitrary instances of an object class. During execution, these temporary instances need to be unified with perceived objects in the robot's actual environment. New perceptions can easily be incorporated by adding a new perception instance (see Section 3.2.3), but the system has to decide *which* object instance to update. In the described experiment, the execution engine makes the simplifying assumption that only one object of a kind

exists, making the object grounding problem becomes much simpler. Each further detection of an object, e.g. a bottle, updates the location of the only existing instance of that kind. We are investigating methods to overcome this limitation: The executive should consciously choose the objects to use, and a symbol anchoring component should ensure that the same object is being used throughout the whole recipe [Blodow et al., 2010].

6.4 Discussion

This chapter deals with the integration of the knowledge processing system into the robot control program. One important aspect is the integration of perception components to generate object instances in the knowledge base from the object detections the robot makes. There are two kinds of interfaces for two types of perception systems: One of them uses computables to forward queries to the vision system, which is used for active, task-directed perception systems. The other interface is just a passive listener that records all results the perception system generates and saves them to the knowledge base. Both interfaces build up a detailed internal representation of the object detection that does not only allow to perform reasoning about the changing poses of objects, but can also describe the source of information including the vision algorithm and the object recognition model that was used.

This representation also forms the basis for the semantic environment maps that describe the poses and types of pieces of furniture in the environment. They are very important for a mobile robot to plan actions to fetch objects from their storage location. We present two ways to infer the most likely storage location for an object: If no information about the distribution of other objects over the different places in the environment is available, the system can use general rules like “perishable items belong into the refrigerator” to make an informed guess where to search first. If the robot has information about the storage places of similar objects, it can use a second kind of inference based on the semantic similarity, assuming that similar objects are often stored together. This assumption proved to be a very good approximation of the actual organizational principles in human kitchen environments.

We further presented methods for exchanging knowledge between robots. They are based on the representations developed in this work, and extend them with meta-data describing the requirements for the information to be used by a robot. Based on this meta-data, a robot can autonomously decide if it can use a piece of information given its components and capabilities. These methods are important components of the RoboEarth system through which robots can autonomously exchange information about plans, objects and environments.

Chapter 7

Evaluation and experiments

Evaluating a knowledge processing system for robots is a complex task for which there are neither established benchmarks nor evaluation methods so far. Using knowledge processing techniques on a robot has its specific challenges that are often different from those encountered in classical, purely symbolic reasoning as commonly investigated in artificial intelligence research. Part of these challenges are related to using the methods on a physical system, other challenges are posed by the complex scenarios the robots are acting in.

Since the robot is acting in the physical world, its knowledge base has to operate in real-time and compute results fast enough not to slow down the other components of the robot. The symbolic knowledge base also needs to be grounded in the robot's sensing and actuation methods to enable the robot to reason about objects it has perceived and about actions it has performed. Abstractly defined actions in the knowledge base need to be linked to parametrizations of executable action components on the robot, and the robot must be able to deal with the uncertain and incomplete information provided by its sensors.

The application in complex realistic scenarios is challenging in terms of scalability, expressiveness, extensiveness and usefulness. The knowledge representation needs to be scalable enough to handle a significant amount of knowledge both on the class level, for instance for describing a large number of object types, and on the instance level, for example to store all observations of objects recorded over an extended period of time. The knowledge should be extensive enough to cover the most important phenomena the robot interacts with including spatial and temporal information, actions, and robot self-models. At the same time, the representation has to be expressive enough to represent the complex relations between these pieces of information: spatial and temporal relations, situations that change over time, the effects of actions, continuous and discrete, probabilistic and deterministic information, etc.

Capturing all these aspects in a few aggregated numbers is hardly feasible. We therefore eval-

uate the quality of the system by a combination of different methods: A quantitative evaluation is performed for those aspects where this is possible, like the scalability of the system and its response time. We then compare the developed system with other robot knowledge bases in the literature with respect to the implemented features. Four complex usage scenarios are described to show how the different components of our system contribute to solving complex realistic tasks. In the end, we discuss how the release of the software and data sets as open source contributes to ensuring their quality.

7.1 Scalability and responsiveness

To assess the system's scalability we performed tests with the internal object representation. The representation of objects is one of the more complex descriptions since it consists of the object instance, the homography matrix and the perception instance for each detection of the object. At the same time, it is one of the most important ones since novel detections of objects are the kind of information that continuously comes in during operation of the robot. We thus created a test program that creates a large number of object detections in a loop, corresponding to thousands of virtual perceptions of this object:

```
n_objs(0).
n_objs(?Num) :-
  new_obj(?Num),
  ?Num1 is ?Num - 1,
  n_objs(?Num1).

new_obj(ID) :-
  create_perception_instance(?Perception),
  set_perception_pose(?Perception, [?ID,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3]),
  create_object_instance(['bed'], ?ID, ?Obj),
  set_object_perception(?Obj, ?Perception).
```

The *n_objs* predicate just calls the *new_obj* predicate *Num* times, which then creates a dummy detection of a bed at a dummy pose. We tested the performance with Prolog's *time()* predicate that lists the CPU time consumed by evaluating a predicate. We used this value to avoid problems caused by other processes or multi-core processor effects.

```
?- time(n_objs(65000)).
% 3,120,050 inferences, 2.880 CPU in 3.058 seconds (94% CPU, 1083351 Lips)
```

We tested the system with up to 65,000 perception instances which correspond to about 43 minutes during which the robot continuously detects the object with 25Hz and asserts every detection to the knowledge base. A more realistic case is that the robot detects an object every 10 seconds, for example with its tilting laser scanner, in which 65,000 detections correspond to 180 hours of continuous operation – which is more than a full week. Figure 7.1 visualizes the time needed for

this (blue square markers). The time for the creation of new instances scales linearly with their number, and 65,000 object detections can be created in about 2.88 seconds. The maximum frame rate for creating object instances is thus about 22,000 detections per second, corresponding to 170,000 triples per second. As a comparison, the ORO paper [Lemaignan et al., 2010] gives a rate of 7,245 statements per second for creating triples.

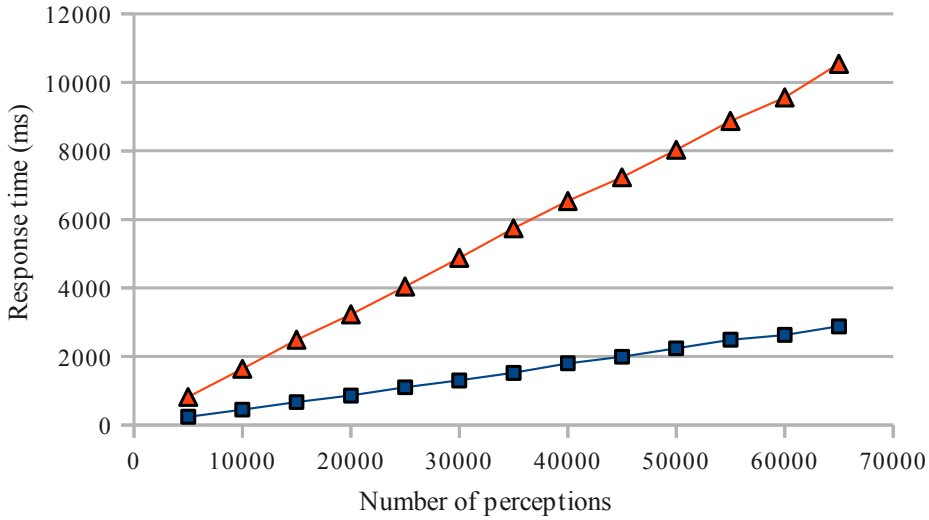


Figure 7.1 Both the creation of a large number of object perceptions (blue square markers) and the query for the latest one (orange triangles) scale linearly with the number of objects and are in the area of few seconds even for 65,000 observations.

As a query benchmark we chose the *latest_detection_of_instance('bed', LatestDetection)* predicate that reads the latest perception of the respective object instance. At the moment, there is no temporal indexing, so the predicate needs to read all perceptions of the object and sort them by their time stamps. This is clearly a sub-optimal implementation, but as Figure 7.1 shows, is still reasonably fast for still large numbers of perceptions, and also scales linearly with the number of objects. Simple queries like asking if an object instance is of a certain type (which requires the system to exploit several steps in the sub-class hierarchy) or querying for all 65,000 objects of a kind are possible in much shorter time.

```
?- time(owl_individual_of('Bed-PieceOfFurniture5000', 'HumanScaleObject')).
% 15 inferences, 0.000 CPU in 0.000 seconds (0% CPU, Infinite Lips)
true.

?- time(findall(?A, owl_individual_of(?A, 'VisualPerception'), ?As)).
% 65,213 inferences, 0.060 CPU in 0.061 seconds (99% CPU, 1086883 Lips)
true.
```

The 65,000 object perceptions correspond to about 490,000 triples in the knowledge base. Up to about 7 Million triples have been used on a common laptop computer (Intel Core 2 Duo

P8700, 2.53GHz, 4GB PC3-8500 RAM, Ubuntu 10.10 32 bit) without noticeably slowing down neither other programs nor the creation of new triples. The 490,000 triples increase the memory consumption of the whole Prolog process from 16.5 to 47.8 MB.

In total, this shows that even large-scale applications like observing an object every ten seconds over a whole week's time can be handled with comparatively moderate memory and at reasonable speed. Spatial and temporal indexing would help to speed up the computation of qualitative relations and other queries that heavily use the object pose representation.

7.2 Comparison to related knowledge bases

In this section, we compare KNOWROB against three other state-of-the-art knowledge processing system that have been developed for being used on robots. The comparison is based on the referenced publications; there may be additional features that have been implemented in the meantime.

ORO The focus of the ORO ontology [Lemaignan et al., 2010] is on human-robot interaction and on resolving ambiguities in dialog situations. This capability was for example described in [Ros et al., 2010], where the robot inferred based on its knowledge about the objects in the environment and their properties which queries it should ask to disambiguate a command.

ORO uses OWL as representation format and a standard DL reasoner for inference. An underlying 3D geometrical environment representation serves for computing spatial information and for updating the internal belief state about the positions of objects [Siméon et al., 2001]. Whenever new knowledge is added to the system, the reasoner classifies the whole knowledge base. This continuous classification of everything is useful to detect inconsistencies, at least those that can be detected on the logical level. However, detecting inconsistencies without being able to remove them is not very practical: Each incorrect sensor reading can lead to wrong beliefs about the environment, and if the knowledge base just refuses any information that contradicts the available knowledge, this wrong belief will never be changed. The continuous classification can also cause scalability problems as all knowledge has to be re-classified whenever some part of it changes, though many areas of knowledge may not be affected at all.

ORO's base ontology has been aligned to the KNOWROB ontology in Spring 2010 and though it has since been extended in both projects it still remains very similar. ORO has several features that are especially suited to a human interaction scenario, like the possibility to represent multiple knowledge bases for different cognitive agents. Such separate knowledge bases are for instance

required to describe the estimated human belief that may differ from the robot's own view on the world. In addition, there are methods that allow components to be notified once a logical statement becomes true, as well as techniques for selecting which properties to use in order to categorize a set of objects. Separate working and long-term memory components allow to store knowledge with different life times.

Compared to KNOWROB, the representations of objects, spatial relations, actions and processes are rather shallow, and there is no support for temporal or spatio-temporal reasoning. Knowledge in ORO is encoded manually or by human dialog, no methods for large-scale knowledge acquisition are presented. Object poses are only described using qualitative spatial relations that are asserted to the knowledge base, which has some disadvantages compared to the on-demand computation in KNOWROB when it comes to updating knowledge.

PEIS The knowledge base presented in [Daoutis et al., 2009] is an important part of the PEIS ecology project (Physically Embedded Intelligent Systems). PEIS investigates distributed intelligent systems consisting of mobile robots, but also of sensors embedded into the environment which are all integrated into a common framework. The PEIS knowledge base is realized as an extension of the Cyc inference engine. On the one hand, this gives the system full access to the large Cyc ontology, but it comes at the cost of slower inference, of irrelevant knowledge in several branches of the ontology, and of a lack of knowledge in areas like robotics or mobile manipulation.

The main focus of this work is the grounding of symbols in percepts, which is realized using a dedicated perceptual anchoring layer in between the perception system and the knowledge base. This explicit grounding layer is more advanced than the solutions in KNOWROB itself, although similar capabilities are developed in a research project [Blodow et al., 2010] in parallel to KNOWROB. The result of the grounding procedure is asserted to the knowledge base including pre-computed spatial relations. This approach, which was also chosen in ORO, requires a component in addition to the knowledge base itself which tracks the state of the internal belief state and asserts, updates and retracts knowledge whenever something has changed. Daoutis et al. call this component "Knowledge Base Synchronizer". To work properly, this component needs a lot of knowledge about the implications of an observed change in order to correctly decide which new knowledge can be derived from these observations and which parts of the knowledge base have become outdated. Errors in this step can lead to outdated knowledge and therefore to inconsistent beliefs. In KNOWROB, we avoid these problems and also the need for such a synchronization component by using an on-demand computation scheme. Information is stored

only once, and more abstract views on this information (e.g. spatial relations) are computed once they are needed and do not have to be retracted afterwards.

The PEIS ontology distinguishes two kinds of memory: the working memory describes objects that are currently in view, while the archive stores the poses of objects that have been detected earlier and got out of view. This approach assumes that the robot continuously detects objects so that the concept "in view" is well-defined. KNOWROB does not make this distinction since this assumption cannot be made if the robot just occasionally searches for an object.

The system includes some simple integration of natural-language information based on the natural-language processing capabilities in Cyc. This is on the one hand used to give explanations to a user, and on the other hand to resolve ambiguities by asking a user. Such ambiguities about the right concept to describe an observed object are avoided in KNOWROB by explicitly modeling object recognition models and the object classes they provide models for.

In contrast to KNOWROB, PEIS does not include temporal and spatio-temporal information and explicit models of change. Also the memory components only store the fact that something has been observed, but not when and how. Therefore, the system can only model the current belief state, and the archive memory just describes those objects of the current belief that are out of view. Detailed descriptions of actions and processes are also missing as well as models to compute their effects.

OUR-K The OUR-K system presented in [Lim et al., 2011] is the successor of the OMRKF framework [Suh et al., 2007]. OUR-K is a very advanced and extensive system that describes a variety of aspects centered around five main kinds of knowledge: contexts, objects, spaces, actions and features. Each of these kinds of knowledge is described at three levels: The context on the level of situations, temporal and spatial context; spaces in terms of semantic, topological and metric maps; actions as tasks, sub-tasks and primitive behaviors; objects as composed objects, normal objects, and object parts; and features as perceptual concepts and perceptual features. Regarding the range of information that can be described, OUR-K is close to the KNOWROB ontology, though lacking the notion of processes, robot self-models, and having simpler action descriptions. The actions the robot is to perform in the examples are mainly navigation based on a semantic map and object recognition. They do not require the deep knowledge about actions and objects that is needed for manipulation activities.

The representation format is based on DL for the concept hierarchies and Horn clauses for the rules, which is also very similar to the approach in KNOWROB in which we use DL and Horn clauses in Prolog for more advanced reasoning like computables or projection rules. OUR-K

chose a somewhat special way of representing knowledge by reifying almost all relations: The "before" relation, for example, is described as an instance of the class *Before* that is linked to the two respective time intervals using the *hasSubjective* and *hasObjective* relations. While this allows to qualify all knowledge with its spatial or temporal context, one loses the elegance of describing properties of objects or actions using roles in DL.

7.3 Completing underspecified instructions

Over the course of the previous chapters, we considered the scenario of generating an effective description for the task of making pancakes. While we referred to this example in the individual sections, we will now briefly summarize which kinds of knowledge are integrated and where the robot can obtain them from. Figure 7.2 visualizes the process of completing the incomplete instructions. The different colors correspond to the different kinds of information and the mechanisms how they can be acquired and are explained in the legend in the upper right.

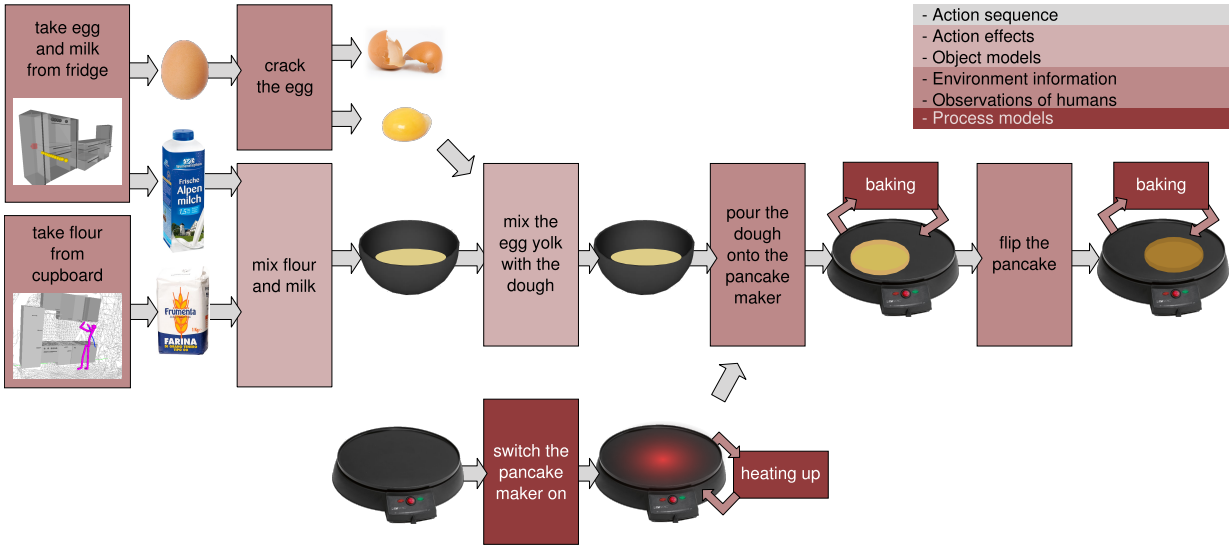


Figure 7.2 Visualization of the different kinds of knowledge used to complete the instructions for making pancakes.

The initial processing step is to search for and download natural-language task instructions from the Internet and to interpret them in order to derive a formal task specification. This conversion process has been described in Section 4.1. It is integrated into the knowledge base using computables for the *forCommand* relation that links a plan to a natural-language command (as described in Section 2.6). The conversion process can therefore easily be called using the query

below, which starts the download and conversion of the natural-language instructions and returns the name of the class describing the plan (as in Section 3.3). As result of this procedure, the robot has an approximate description of the actions and their ordering.

```
?- rdf_triple(forCommand, ?P, 'make␣pancakes').
P = 'MakePancakes'
```

Using the projection rules for actions and processes, it can then compute which objects are supposed to appear and disappear at which step and how they are related to actions in terms of inputs and outputs. The rules for projecting action effects have been described in Section 3.3.4. In Figure 7.2, the predicted objects are drawn in between the actions. Using its knowledge about its own capabilities (Section 3.5) and the knowledge about the models in the perception system (Section 6.1, the robot can then check whether it has recognition models for all of these objects. If models are missing, they can for instance be downloaded from the RoboEarth database (Section 6.3).

To ground the abstractly described objects in actual objects in the environment, the robot needs to add actions to search for these objects and to retrieve them from their storage locations. This combines knowledge from the environment model (Section 6.2) with knowledge about object properties (Section 3.2.1, Section 4.2) These additional actions are shown in the left part of Figure 7.2. If the objects are inferred to be inside a container, the system can use the articulation model encoded in the semantic environment map (Section 6.2) in order to create actions to open the container. The following query is an example how to obtain the opening trajectory of the container that is inferred to be the most likely storage location for milk. It uses the methods described in Section 6.2 to infer the right location and to read the opening trajectory from the semantic environment model. Its result is show in Figure 7.3 (left).

```
?- storagePlaceFor(?StPlace, 'CowsMilk-Product'),
   rdf_has(?StPlace, openingTrajectory, ?Traj),
   findall(?P, (rdf_has(?Traj, pointOnTrajectory, ?P) ), ?Traj).
```

For other kinds of motions, which are not part of the environment model, the robot can query for semantically similar movements it has observed from a human. An example query is given below, which is for the motion of taking a dinner plate out of a cupboard (Figure 7.3 (right)). The methods for the semantic analysis of human motions described in Section 5.6 enable the robot to send such semantic queries to retrieve matching poses and trajectories from its observations.

```
?- rdf_triple(type, ?A, 'TakingSomething'),
   rdf_triple(objectActedOn, ?A, ?Obj),
   owl_individual_of(?Obj, 'DinnerPlate'),
   rdf_triple(trajectory-Arm, ?A, ?Tr),
   rdf_triple(pointOnArmTrajectory, ?Tr, ?P)
```

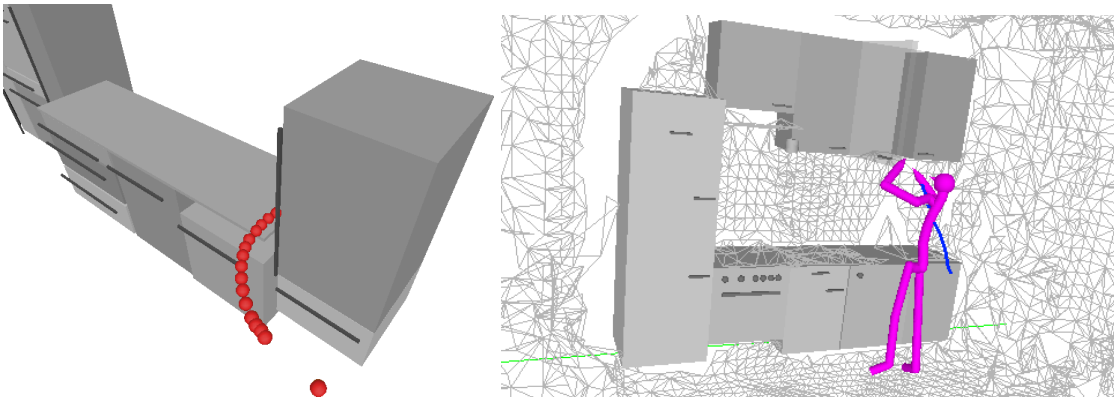


Figure 7.3 Left: Opening trajectory of the refrigerator as result of a query for how to open the most likely storage place for milk. Right: Trajectory for taking a plate out of the cupboard selected from observations of a human performing a table-setting task.

The robot has now grounded the object descriptions, created actions for fetching the objects from their most likely storage locations, and retrieved examples how to perform the different motions from human demonstrations. However, the action sequence is not complete yet: The original instructions did not contain the command to switch on the pancake maker, so the dough does not bake to a pancake. This plan flaw is detected by checking if the final result of the plan is as expected and by verifying that all inputs of all actions are provided either by objects that can be retrieved from the environment or by objects that are generated by previous actions. If this is not the case, the declarative descriptions of the effects of actions and processes can be used for planning in order to generate all required objects, either as direct effects of actions or indirectly by adding actions that trigger processes which achieve the desired effects. This planning with actions and processes was described in Section 3.4.5. The following query first reads all sub-actions of the original action plan that was generated from the web instructions, and then calls the *integrate_additional_actions* predicate which recursively adds actions to ensure that all inputs of the actions in the original plan are available at the time they are executed. In this case, an action of type *TurningOnHeatingDevice* is added to the sequence that triggers the process *BakingFood*. At the moment, the system does not support reasoning over the temporal extent of actions and processes, which would be necessary to turn on the pancake maker beforehand to give it some time to heat up.

```
?- plan_subevents('MakingPancakes', ?OrigActionSeq),
   integrate_additional_actions(?OrigActionSeq, ?DebuggedActionSeq).

egg1 -> EggShell1
egg1 -> EggYolk-Food2
milk1 added to -> Dough4
flour1 added to -> Dough4
Dough4 added to -> Dough6
EggYolk-Food2 added to -> Dough6
Dough4 on top of pancakemaker1

OrigActionSeq = ['CrackingAnEgg',
                 'MixFlourAndMilk',
                 'MixEggAndDough',
                 'PourDoughOntoPancakeMaker',
                 'FlippingAPancake'],

DebuggedActionSeq = ['CrackingAnEgg',
                     'MixFlourAndMilk',
                     'MixEggAndDough',
                     'PourDoughOntoPancakeMaker',
                     'TurningOnHeatingDevice',
                     'BakingFood',
                     'FlippingAPancake'].
```

The result is an effective task specification that can be used for generating a robot plan, but also for projection. The projection methods are realized as computables for the *postActors* relation, which makes the projection procedure transparent to the user: The relations can be queried in the same way independent if they have already been computed or are just generated during the inference process. The object transformation graph in Figure 7.2 is thus effectively built up during the inference. The following queries show how the projection can be performed; the outputs are described in a more informal way for better readability:

```
?- rdf_triple(postActors, cracking1, ?Post).
egg1 -> EggShell0
egg1 -> EggYolk-Food1

?- rdf_triple(postActors, mixing1, ?Post).
EggYolk-Food1 added to -> Dough2
milk1 added to -> Dough2
pancakemix1 added to -> Dough2

?- rdf_triple(postActors, pour1, ?Post).
Dough2 on top of pancakemaker1

?- rdf_triple(postActors, turnon1, ?Post).
pancakemaker1 switched on
Dough2 -> Baked4

?- rdf_triple(postActors, put1, ?Post).
Baked4 on top of plate1
```

The first actions create, destroy and join objects to create the dough. Then the pancake maker is switched on, changing its device state from off to on. The action projection predicate checks whether any processes became active due to the results of the actions. In this case, a heating process is being started, causing the temperature of the pancake maker to rise to its working temperature since the pancake maker is a heating device with device state 'on'. As a result of the heating process, the dough is in thermal contact with a heat source, which starts the baking process which transforms the piece of pancake dough into a baked pancake.

Once this object transformation graph has been built up, the robot can perform reasoning about it. For example, it can use its new knowledge about when objects are created or destroyed to perform situation-specific computation of spatial relations. In order to determine if a relation holds at a specific point in time, either in the past or in the predicted future, the computation needs to take the creation and destruction of objects into account. In the example, the egg is initially assumed to be on the table, but gets destroyed over the course of the actions so it cannot be assumed to be on the table afterwards.

```
# Initially, the egg is computed to be on the table
?- rdf_triple('on-Physical', ?A, table1).
A = 'egg1'.

# Perform projection as described above
?- [...]

# The egg got destroyed and therefore not on the table any more
?- rdf_triple('on-Physical', ?A, table1).
false.
```

Using the transitive property *transformedInto* that is computed based on the object transformation graph, one can perform reasoning about which objects are transformed into which other ones and determine for example where the ingredients of a product have been taken from.

```
# What has been transformed into Baked4?
?- rdf_triple(transformedInto, ?A, 'Baked4').
A = 'Dough2' ;
A = 'EggYolk-Food1' ;
A = 'egg1' ;
A = 'milk1' ;
A = 'pancakemix1' ;
false.

# What did the egg get transformed into?
?- rdf_triple(transformedInto, egg1, ?Res).
Res = 'Baked4' ;
Res = 'Dough2' ;
Res = 'EggShell0' ;
Res = 'EggYolk-Food1' ;
false.
```

The effective task specification is as complete as possible given the robot's knowledge. However, some decisions are intentionally postponed to execution time, for example where to stand while performing an action or where exactly to put down an object. These action parameters strongly depend on the situation at hand, for example the configuration of obstacles around the object of interest and therefore cannot be determined a priori. A parallel research project investigates how these kinds of decisions can be taken using physical reasoning [Lorenz Mösenlechner and Michael Beetz, 2011].

7.4 Exchanging information between robots

In this section we report on an experiment that was implemented and successfully demonstrated to the public during the RoboEarth workshop in Eindhoven in January 2011. It was the first demonstration of the complete RoboEarth system that is being developed as a web-based knowledge base through which robots can exchange information about actions, object models and environment maps. An important aspect of this project is the language to describe the exchanged information. It needs to be expressive enough to encode the information to be transferred, but also needs to provide constructs for storing meta-data that describe the pieces of information and which can be used for selecting those that fit the robot's capabilities.

The task the robot was to perform in this experiment was to serve a drink to a patient in bed in a hospital room, i.e. to first find a bottle in the environment, to pick it up, to move to the patient and to hand over the bottle. The information to be used was to be downloaded from the RoboEarth knowledge base and needed to be grounded in the robot's perception and action system. Figure 7.4 shows the course of actions performed during the demonstration. A video of the experiment can be found at http://www.youtube.com/watch?v=RUJrZJyqftU. KNOWROB, including the extensions developed as part of this thesis, contributed to the demonstration in the following ways:

- Formal representation of the action recipe, the environment model, and the object models
- Download of the recipe and the other pieces of information from the central RoboEarth knowledge base

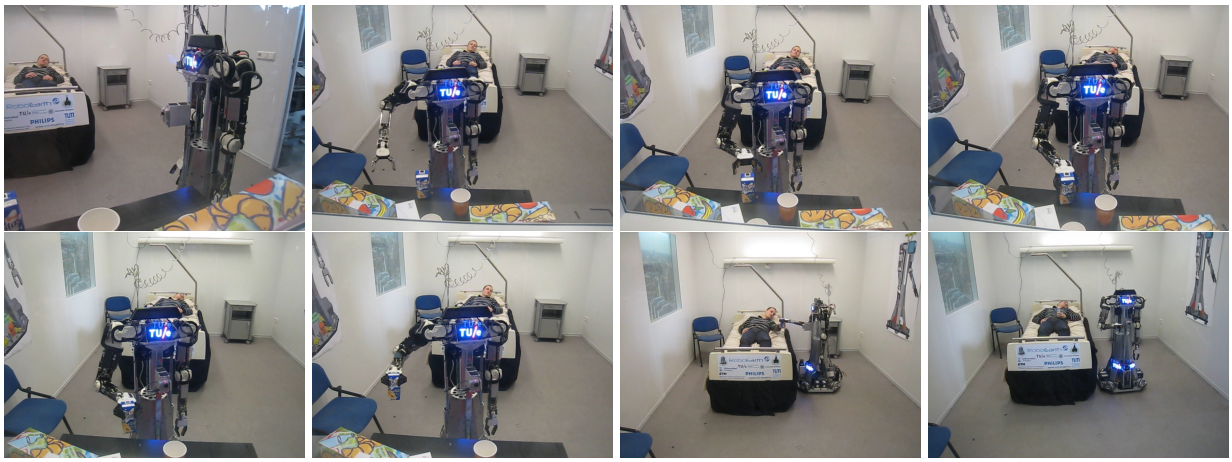


Figure 7.4 Pictures taken during the "serve a drink" experiment, a task that was performed based on information downloaded from the RoboEarth Internet database.

- Matching of the requirements of the "serve a drink" recipe to the capabilities of the Amigo robot
- Run-time knowledge base interfacing the executive to the vision and world modeling components, providing the executive with information about the actions to perform and their parameters, the locations of objects

The language and the accompanying reasoning methods have successfully been used to exchange tasks, object, and environment information on a physical mobile manipulation robot and execute the abstractly described task. This experiment showed that the presented methods enable a robot to download the information needed to perform a mobile manipulation task, including descriptions of the actions to perform, models of the objects to manipulate, and a description of the environment, from the RoboEarth database on the Internet.

7.5 Matching observations against task specifications

This experiment investigates how abstract task descriptions can be compared with observations of human manipulation actions, which can be useful to check if a task has been performed correctly and to determine where the observations match the specification and where they differ from each other. The comparison is non-trivial, there is a large gap to bridge between the two kinds of action descriptions. On the one hand, there are very abstract task-level descriptions like "put the place mat in front of the chair", on the other hand, there are the unstructured continuous motions that have been observed from the human subject. To compare these very different descriptions, the system needs to create a structured representation of the observed motions and abstract them from the level of continuous motions up to the task level at which action properties like the *fromLocation* and *toLocation* can sensibly be described.

To this end, we combine the generation of hierarchical action models described in Section 5.6 with the import of natural-language task instructions using the methods described in Chapter 4. The hierarchical action models describe observed actions as a hierarchical structure of action instances which are linked to the objects and locations that are set as action parameters. These structures are to be matched against a formal task specification to determine missing actions, a wrong order of actions, or wrong tools that have been used for the right action. An example is illustrated in Figure 7.5: The lower part with the blue background is the hierarchical action model generated from observations of humans, the upper part is a description of the same task that has been imported from the WWW.

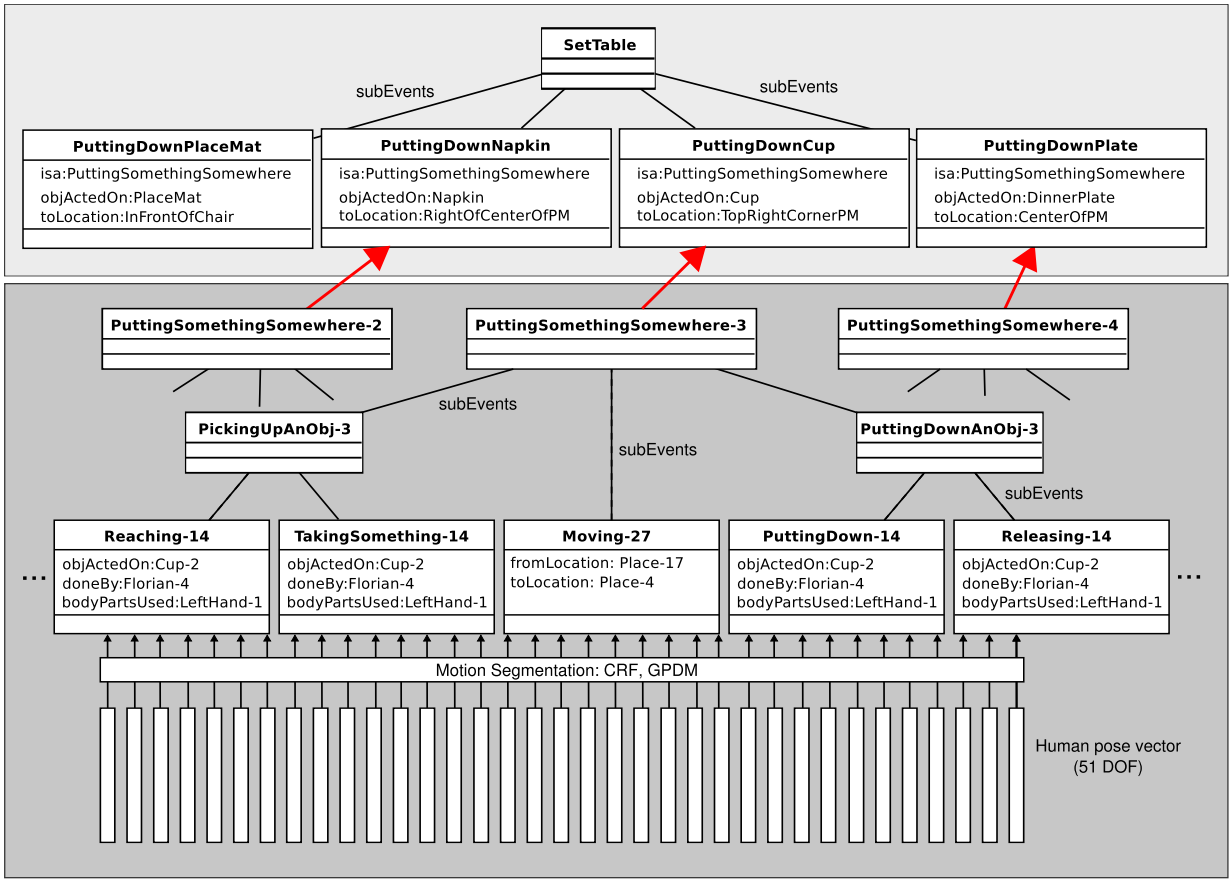


Figure 7.5 Matching abstracted observations of human activities against formal task specifications imported from the Internet.

The following query exemplarily reads all information the system has about two *PuttingSomethingSomewhere* actions which has been created automatically from the observed data. The *objectActedOn* was determined based on the RFID tags attached to the objects, the *startTime* and *endTime* from the first and the last frame in the segment. The original data in the TUM Kitchen Data Set does not contain the exact object positions, because the objects have been detected using RFID tag readers which only provide information about the presence of an object inside the rather large detection range. To demonstrate the ability of the system to perform reasoning about object poses, we adapted the object locations in the database manually for this experiment to reflect the spatial setup in a more detailed fashion.

```
?- owl_has('PuttingSomethingSomewhere191', ?P, ?O).

P = 'doneBy',        O = florian ;
P = 'objectActedOn', O = 'plate-1' ;
```

```
P = 'startTime',      O = 'timepoint_13.5289101601' ;
P = 'endTime',        O = 'timepoint_19.9923501015' ;

P = 'fromLocation',   O = 'loc_0.32_2.2_1.71' ;
P = 'toLocation',     O = 'loc_3.2_2_0.74' ;

P = 'subEvents',      O = 'PickingUpAnObject153' ;
P = 'subEvents',      O = 'CarryingWhileLocomoting62' ;
P = 'subEvents',      O = 'PuttingDownAnObject154' ;

?- owl_has('PuttingSomethingSomewhere192', ?P, ?O).
P = 'objectActedOn', O = 'cup-1' ;
P = 'fromLocation',  O = 'loc_0.32_2.2_1.71' ;
P = 'toLocation',    O = 'loc_3.2_2_0.74' ;
[...]
```

Matching these observations against the abstract task descriptions goes much beyond just comparing the action sequences: Both the action descriptions and their parameters can be quite complex, hierarchical structures. Locations, for example, are often described as a sequence of qualitative spatial relations like “to the left of the center of the plate” or “to the top right corner of the place mat”. During the conversion procedure, these natural-language descriptions are translated into nested class restrictions in OWL. The tasks themselves are also represented in terms of action classes, as described in Section 3.3, while the observations of the human subject are described as instances. The matching problem can therefore be regarded as a classification problem, namely to check whether the observed instances can be assigned to the classes in the task specifications. The *matching_actions* predicate checks for each sub-action of the current plan if there is corresponding action instance that fits the specification:

```
matching_actions(?Plan, ?Act) :-
  plan_subevents(?Plan, ?SubEvents),
  member(?ActCl, ?SubEvents),
  owl_individual_of(?Act, ?ActCl).
```

In the following, we will consider a plan for setting a table as example. The hierarchical action model generated from observation is shown in Figure 5.13 (top), the natural-language sentences describing this tasks are as follows:

1. Place the place mat in front of the chair.
2. Place the napkin just to the left of the center of the place mat.
3. Place the plate (ceramic, paper or plastic, ceramic preferred) in the center so that it just covers the right side of the napkin.
4. Place the fork on the side of the napkin.
5. Place the knife to the right so that the blade faces the plate.
6. Place the spoon right next to the knife.
7. Place the cup to the top right corner of the place mat.

These natural-language instructions can be converted into a class-level action description in OWL as explained in Section 3.3.1 by calling the import procedure using the computable defined for the *forCommand* property.

```
?- rdf_triple(knowrob:forCommand, ?P, 'set␣a␣table').
P = 'SetATable'
```

Once the formal task representation has been generated it can be inspected, for example using the *plan_subevents* predicate that allows to read all sub-actions of the plan. The instructions for the table-setting plan have been translated into seven transport actions for bringing the different objects to the right positions on the table, described as sub-classes of the *PuttingSomething-Somewhere* class. When calling the *matching_actions* predicate, two of the specifications can successfully be matched against the observations, namely those transporting the dinner plate and the cup. The other objects were either not visible to the system (since there were no RFID tags attached to them) or the reference object could not be found (the chair is not part of the semantic map, so the 'in front of' relation could not be evaluated).

```
?- rdf_triple(forCommand, ?P, 'set␣a␣table').
P = 'SetATable'

?- plan_subevents('SetATable', ?SubEvents).
SubEvents = ['PuttingSomethingSomewhere1',
             'PuttingSomethingSomewhere2',
             'PuttingSomethingSomewhere3',
             'PuttingSomethingSomewhere4',
             'PuttingSomethingSomewhere5',
             'PuttingSomethingSomewhere6',
             'PuttingSomethingSomewhere7']

?- matching_actions('SetATable', ?MatchingAction).
MatchingAction = 'PuttingSomethingSomewhere191' ;
MatchingAction = 'PuttingSomethingSomewhere192' ;
```

To establish these matches, the system had to ground the abstract descriptions of spatial configurations like "the top right corner of the place mat" in the observed data. The positions of objects are usually observed as points in space which need to be translated into qualitative descriptions to check whether they comply with the abstract specifications. We use the computables described in Section 3.2.5 for this task which allow to calculate, based on the objects' positions and dimensions, if a spatial relation like "in center of" or "right of" holds.

7.6 Inferring missing objects

Another integrated experiment showing how the robot can use its knowledge to accomplish complex tasks is the inference which objects are missing in an incomplete table setup. To complete it, the robot must first find out which meal will take place and which objects are required for it. Therefore, it needs detailed knowledge about the types of food and utensils used in different meals and the preferences of the participating persons. This scenario therefore is also a demonstration how the different components in KNOWROB can jointly be used to solve a complex task. The system is described in more detail in [Pangercic et al., 2010] and was realized in joint work with Dejan Pangercic, who realized the perception components, and Dominik Jain, who contributed the statistical relational models of meal preparation that perform the actual inference.

KNOWROB serves as the integration platform for perceptual information about the objects that can currently be seen on the table and the learned statistical relational models about which objects are commonly used by different people in different meals. The objects detected by the vision system are published on a topic and added to the knowledge base using the techniques described in Section 6.1. By combining the detected object poses with the environment information, the system can compute which objects are on a given table based on the spatio-temporal computables for calculating qualitative spatial relations which have been described in Section 3.2.5.

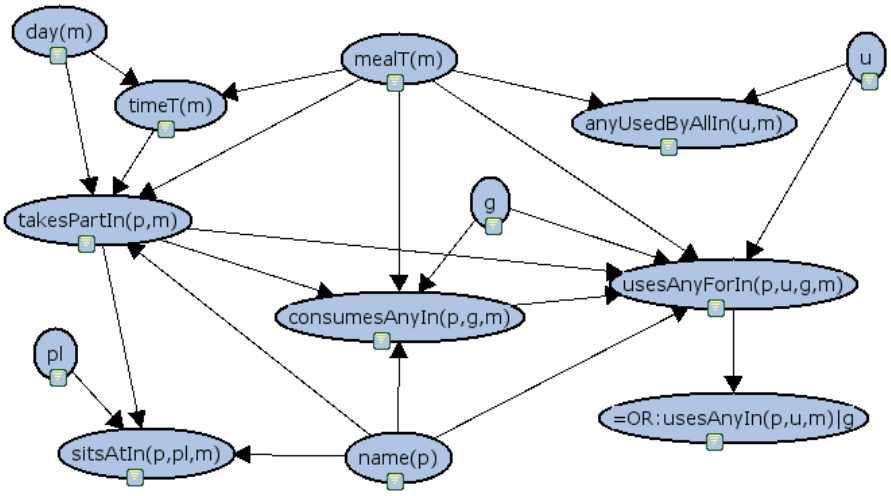


Figure 7.6 Bayesian logic network for the meal preparation context, describing the relations between people participating in a meal and the foodstuff and utensils they use (courtesy of Dominik Jain).

This set of objects serves as evidence for the statistical relational inference methods that are integrated as described in Section 2.5. The inference is performed based on models that have been learned on (simulated) observations of human meals and that describe the type of the meal, who

took part, who consumed which goods using which utensils. Figure 7.6 shows the dependency structure of the BLN that, after having been trained on this data, effectively represents a joint probability distribution over all the different aspects of human meals.

During the execution, the robot control program sends a query to KNOWROB asking for the set of missing objects, KNOWROB forwards this query to the ProbCog inference engine, which then loads the respective model, reads evidence from KNOWROB (the set of objects that have been perceived on the table) and performs the inference. An example query with its result can look like follows:

$$
\begin{aligned}
&P(usesAnyIn(P, ?u, M), consumesAnyIn(P, ?f, M) \mid mealT(M) = Lunch \wedge \\
&\quad usesAnyIn(P, Plate, M) \wedge usesAnyIn(P, Knife, M) \wedge \\
&\quad usesAnyIn(P, Fork, M) \wedge usesAnyIn(P, Spoon, M) \wedge \\
&\quad usesAnyIn(P, Napkin, M) \wedge consumesAnyIn(P, Salad, M) \wedge \\
&\quad consumesAnyIn(P, Pizza, M) \wedge consumesAnyIn(P, Juice, M) \wedge \\
&\quad consumesAnyIn(P, Water, M) \wedge takesPartIn(P, M)) \\
&\approx \langle\langle\ Glass: 1.00, Bowl: 0.85, Cup: 0.51, \ldots \rangle, \\
&\quad \langle\ Soup: 0.82, Coffee: 0.41, Tea: 0.14, \ldots \rangle\rangle
\end{aligned}
$$

The result of the inference process is a set of object types with assigned probabilities that denote which objects are supposed to be on the table. After subtracting those objects that are already there, KNOWROB determines which ones still need to be transported there and determines based on its perceptual memory (Section 3.2.3) and its knowledge about storage locations in the environment (Section 6.2) where to search for them.

Figure 7.7 visualizes the results of some exemplary queries. The upper row shows the input images in which the different objects have been recognized. These objects are shown in the lower images in red on top of the table. The results of the inference process, indicating which objects are concluded to be missing, are visualized behind the table, with the hue value corresponding to the probability from low (blue) to high (red). In the left image, the system inferred a knife and a glass to be certainly missing – which makes sense, given that there is juice, but no drinking vessel, and no knife for cutting the cake and the sausage. The center image shows a setup where silverware is already placed on the table, but a cup and a plate are obviously missing. In the right pictures, it is again the glass that is needed to drink the water and juice that has been detected on the table.

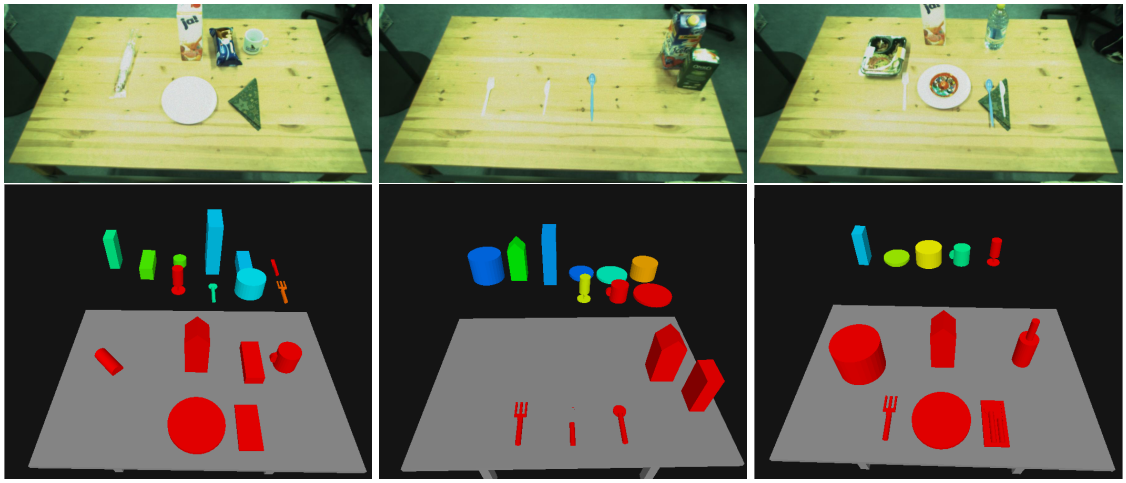


Figure 7.7 Results of queries for missing objects. Upper row: Input camera images (courtesy of Dejan Pangercic). Lower row: Visualized inference results in which the hue indicates the probability with which the object is inferred to be missing (blue: low, red: high)

7.7 Open source software contributions

All the software that has been developed as part of the presented knowledge processing system, the ontologies and models that have been created as well as the data set of human motion tracking data which was used to evaluate the action interpretation techniques have been released to the public as open-source software. By releasing the code and data to the public, we enable others to replicate the experiments made and to profit from the experiences and implementations in the system.

The KNOWROB system has become the main knowledge base in the ROS environment, is part of the standard ROS package distribution, and is used by several other research institutions (e.g. ETH Zurich, University of Stuttgart, Technical University Eindhoven, University of Pisa, University of Graz).

The TUM Kitchen Data Set has generated more than 460 GB of downloads and attracts around 300 visitors a month. The data seems to be actively used; users have even contributed tools they developed and additional annotations they made. The data has been used for the evaluation of several external publications at ECCV [Gall et al., 2010], at ICPR [Krausz and Bauckhage, 2010], and in the PAMI journal [Gall et al., 2011]).

Chapter 8

Conclusions

In this thesis, we reported on our work on knowledge representation and reasoning methods that help autonomous robots improve their problem-solving skills. The KNOWROB knowledge base developed as part of this thesis is a fast and scalable knowledge processing system specialized for being used on and by autonomous robots. It is based on well-established standards like the Web Ontology Language (OWL) and uses Prolog as underlying inference mechanism. KNOWROB is integrated into the robot's control system and grounded in the robot's internal data structures and the perception system.

We introduce the concept of on-demand computation of semantic information based on data structures in the robot control program and perception modules. Procedural descriptions, called "computables", can be attached to the classes and properties in the knowledge base to describe how these classes and properties can be computed, either based on information that already exists inside the knowledge base, thereby generating different abstract views on the same original data, or by loading data from external sources, for instance object detections from the vision system.

The KNOWROB ontology is one of the largest ontologies of the robotics and household domain that has been developed so far, containing hundreds of classes that describe objects, actions and events. These classes, together with a number of relations between them, form the basic language elements which the robot can use to describe its actions and observations. Such a common semantic language is extremely important to integrate information from different sources. To combine these pieces of information without asking for human assistance, a robot needs integrated representations that represent the different kinds of knowledge and the relations between them in a common formal language.

We have developed novel representations and reasoning schemes for several kinds of information: Descriptions of events and temporal information are required to account for the dynamic nature of the environments. The techniques in KNOWROB allow to describe and reason about

the start and end time of events, their duration, time points and time spans. One of the most important kinds of information for robots are descriptions of objects and the environment. The KNOWROB ontology provides a large taxonomy of object types, methods for describing the poses and dimensions of objects.

To reason about changing object configurations, we developed novel spatio-temporal representations that describe not only the poses of objects, but also the times at which the object has been detected and the methods that have been used. Based on these spatio-temporal representations and the computables mentioned earlier, the robot can compute qualitative spatial relations like 'in', 'on', or 'next to' for different points in time based on the object poses stored as metric positions. These techniques have been used to describe semantic environment maps in the knowledge base in terms of object instances. Such environment models are for example very useful for finding objects in the environment.

We further equipped the robot with tools to decide if it can recognize an object, namely with descriptions of the object recognition models it has and the kinds of objects they allow to recognize. Such descriptions can for instance be used to make sure that all objects which appear in a task description can be recognized, and to trigger methods for obtaining the required models otherwise.

Besides objects, another very important kind of information a robot has to reason about are actions. KNOWROB contains a detailed action ontology that describes many different kinds of actions the robot can perform. The ontology covers most actions needed for mobile manipulation and simple meal preparation tasks like transporting objects, mixing and pouring, and opening and closing different kinds of containers. These single actions can be hierarchically composed to complex tasks, and ordering constraints can be specified among them, forming expressive task specifications. The required inputs and the effects of actions are described in a way that both projection and planning are supported. They consist of two parts: A declarative description of the preconditions and effects can be used to search for suitable actions in a planning context, whereas procedural projection rules allow to qualitatively predict the effects of these actions. The projection rules describe the effects that actions have on objects using a number of very detailed relations that specify for instance if an input object is being destroyed over the course of an action.

Robots that are to plan their actions, to learn from past experiences, and to diagnose errors in task executions need expressive and detailed representations of changing world states and the causes of these changes. Object poses that are changed by transportation actions are rather simple to describe, while much more substantial changes like the destruction, creation or transformation

of objects can be caused for instance by cooking actions. Our representations can describe how actions move, split, destroy, create, join, switch on, open and close objects based and are the basis for a detailed representation of change.

To account for changes that are not directly caused by actions, we introduce the notion of processes that are often triggered as side-effects of actions. For example, when switching on a heating device, the direct effect is that the device state changes from off to on, but in addition, a heating process is started that causes the device to heat up to its working temperature. The representation of processes is similar to the action representation and also supports projection as well as planning. Planning with processes thereby means to add actions that trigger the processes that achieve the desired effects.

The last major class of information in KNOWROB are models of the robot itself. They combine a description of its kinematics with semantic models of the robot's components which attach meaning to sub-components, like a hand or an arm, and further add descriptions of software components like environment maps or object models. On top of the component-level descriptions, one can define capabilities, and the system can automatically infer which of these capabilities are available on a given robot platform by checking which dependencies are met. By matching the capabilities and components against formally described action requirements, the robot can decide whether or not it can perform a task and react accordingly. If software components are missing, they can potentially be downloaded; if capabilities are missing, the robot may start a program that provides them.

All of these pieces of knowledge can also be exchanged between robots, and using some meta-information and the matching between required and available capabilities, the robot can determine autonomously if some information can be useful and if all requirements are fulfilled. KNOWROB is interfaced with the web-based RoboEarth knowledge base that provides task descriptions, object models and environment maps described in the same format that is also used inside of KNOWROB.

In addition to the representational methods, we also developed several ways to automate and scale up the knowledge acquisition process. One source of information that we regarded is the Internet, more specifically web sites which contain thousands of step-by-step instructions that describe in detail how to perform everyday tasks. In this work, we presented a novel system for translating these instructions, written in natural language, into a formal representation that can be used in conjunction with the rest of the knowledge base to reason about the task and to generate an executable robot plan. Translating the instructions involves semantic parsing to determine the structure of the instructions, the interpretation of the natural-language descriptions

in order to resolve ambiguities, and the conversion into a logical representation. In addition to task instructions, we also made use of information about products and their properties that can be obtained from online shopping websites. The product descriptions have been used to automatically generate an ontology of that arranges about 7,000 classes of household products in a taxonomy and annotates each of them with information found in the product description, like its weight, price, or perishability status. These methods help to automate the knowledge acquisition process, and though some manual adjustments are often helpful to improve the quality of the automatically imported data, the system allows to scale the knowledge base to a much larger number of classes with minimal effort.

We further described the acquisition of knowledge-based models of human everyday activities as a another important source of information. By representing observed activities in the same language that is also used to describe the robot's actions and the objects in the environment, the robot can directly derive useful information from the observations. The presented models are a unique combination of formal logic-based action descriptions and stochastic action recognition methods. They integrate several state-of-the-art methods for interpreting human actions at different granularities in a formal, coherent framework. The interpretation starts with a segmentation of the continuous motions and a classification of the segments that relates them to the action classes in the ontology. In this context, we exploit a novel combination of pose-related and environment-related features to reliably perform the segmentation of the highly challenging input data and evaluated several classification methods. The initial segmentation is used as starting point to further explore different ways how an action can be performed – for example different kinds of reaching motions. For this purpose, we presented clustering methods that can distinguish different shapes of the trajectories inside one semantic class of motions, and combined the clustering methods with techniques to automatically learn the context in which these motions shall be used based on abstract definitions of the learning problem. While the initial segmentation is rather fine-grained, describing single motions like reaching towards an object or opening a cupboard door, the system should also understand higher-level descriptions of the same actions, e.g. on the level of bringing a cup from the cupboard to the table. Such descriptions can be generated automatically using the methods presented in this work by abstracting action sequences from the level of continuous motions up to the task level. These hierarchical models are constructed by exploiting information about sub-actions that more complex actions are composed of that is already available in the robot's action ontology and otherwise often used for planning robot actions. If there are several examples that show how a task can be performed, the robot can combine them and learn the partial ordering of actions inside this task. Our novel

approach is based on statistical relational learning and extracts not only information about the ordering, but also information about objects and other action parameters from the observations of human activities.

While we evaluated the different components independently in the respective chapters, we also presented several integrated scenarios that combine multiple modules and representations to accomplish a complex reasoning task. The overall quality of a robot knowledge processing system depends on a variety of factors and can hardly be described by a few numbers, but the range of tasks it can be used for can serve as a measure for how effective and useful it is. We considered four usage scenarios: Completing action sequences that lack important information, exchanging knowledge between robots, comparing observations of human actions with formal task specifications generated from descriptions on the Internet, and inferring which objects are missing in an incomplete table setup.

In the first scenario, the robot received the command to perform a task and needed to generate an effective action specification that contains all required information. After having downloaded a step-by-step description of the task and having translated it into a formal representation, the robot had to determine which pieces of information were missing and where it could obtain them from. In a first step, it applied its knowledge about the involved actions, predicted their effects and thereby completed the set of objects in the task. It then made sure that all of them could be recognized and downloaded missing object models otherwise. To ground the abstract object specifications, the robot inferred where to search for these kinds of objects based on its knowledge about their properties and the environment, and added actions to retrieve them from their storage locations. By checking if the predicted and desired results match, the robot could also verify whether the task achieved the desired effects and otherwise add additional actions to the task. Action parameters like the trajectories to be used could be generated from the abstracted and formally described observations of humans.

The second integrated scenario investigated how knowledge about actions, objects and the environment can be exchanged between robots. In the experiment, a hierarchical task description of a mobile manipulation task was encoded using the representations developed in this thesis and downloaded by a robot that then determined which further information it needed to perform this task. Using the model of itself and its capabilities in conjunction with the descriptions of action requirements, the robot autonomously inferred that it was lacking an environment map and some object models. After having downloaded them from RoboEarth, a web-based knowledge repository for robots, it was able to recognize the required objects, to infer their locations based on the environment map, and to ground the abstract action descriptions by parametrizing executable

action primitives.

Our third scenario combined the analysis of human activities with formal task instructions imported from the Internet. This combination allows to verify if a task has been performed correctly, to check whether mistakes were made or to find alternative ways of performing an action. To enable the robot to do these analyses, it needs to translate instructions from natural language into a logical format, segment the observed human motions and represent them in the knowledge base, abstract them up to the level at which the task instructions are described, and compare the observations with the task descriptions by classifying them into the classes defined in the task description.

The last scenario dealt with inferring which objects were missing in a tabletop setup and mainly required the integration of the probabilistic inference and the perception methods with the knowledge base. The system combined learned models of the foodstuff and utensils used in different meals with objects that had already been perceived on the table to jointly infer which kind of meal was to be prepared and which items were missing. While the objects on the table served as evidence for the probabilistic inference procedure, its environment model helped the robot to locate the missing items.

These usage scenarios show that the developed knowledge processing system can be used to solve complex reasoning tasks in realistic scenarios which integrate several kinds of knowledge from very different sources. In order to further scale the system towards novel applications, it can be extended in different ways:

Memory management: Although KNOWROB scales well up to large amounts of knowledge, a more active memory management could further improve its performance. The temporally and spatially very detailed observations of objects, for example, can be compressed by pruning subsequent detections of an object at the same place. In general, computing expectations and storing only surprising data is an effective way of reducing the amount of data that needs to be stored. Older pieces of information could then be swapped into an external long-term memory and only be retrieved if the robot needs to reason about this range in time. This raises the problems how to decide which data is to be swapped out, how it is processed, when it is to be loaded again, and how this memory structure can be made as transparent as possible to the rest of the system.

Spatial knowledge: A more thorough representation of spatial information, including explicit and semantic representations of units of measure, coordinate frames and transformations between them would make the system more suitable for distributed real-world applications in which these aspects cannot be assumed to be standardized any more. Especially when integrating information

from multiple sources, as required to exchange information over the Internet, a robot needs a deeper understanding of these spatial concepts. When representing relative poses, the robot must know their dependencies to decide whether they remain valid once the robot has moved. An object pose that is stored relative to the robot's base, for example, becomes invalid if the robot moves and would have to be updated.

Symbol anchoring: While object instances in KNOWROB are grounded in the robot's perception by the computables that generate them from the object recognition results, there is currently no explicit anchoring procedure that ensures that an object instance remains bound to the same physical object over several perceptions. This issue is investigated in a parallel research project [Blodow et al., 2010] whose solutions should be integrated into the KNOWROB architecture.

On-line activity monitoring: Recent developments in real-time human motion tracking in range sensor data [Shotton et al., 2011] suggest to apply the methods presented in this work to on-line activity recognition and interpretation. While all our computations are much faster than real time, they are currently realized in a batch processing mode since the original tracking system could only be applied off-line. Applying them to on-line tracking opens up several new application domains: A cognitive reminding system could estimate what the human subject is currently doing, compare this to a description of what should be done, and suggest how to improve. Another option would be to let the robot help the human in a way that best fits the task context and the action the human is currently performing.

8.1 Publications

The work presented in this thesis has led to several publications in international journals and conferences. When the work described in a chapter had previously been published, we referred to the respective publication; below is a list of all prior publications for completeness:

Journal publications

Moritz Tenorth, Ulrich Klank, Dejan Pangercic, and Michael Beetz. Web-enabled Robots – Robots that use the Web as an Information Resource. *Robotics & Automation Magazine*, 18(2), 2011.

Moritz Tenorth, Dominik Jain, and Michael Beetz. Knowledge Representation for Cognitive Robots. *Künstliche Intelligenz*, 24(3):233–240, 2010.

Michael Beetz, Moritz Tenorth, Dominik Jain, and Jan Bandouch. Towards Automated Models of Activities of Daily Life. *Technology and Disability*, 22(2):27-40, 2010.

Michael Beetz, Dominik Jain, Lorenz Mösenlechner, and Moritz Tenorth. Towards Performing Everyday Manipulation Activities. *Robotics and Autonomous Systems*, 58(9):1085-1095, 2010.

Markus Waibel, Michael Beetz, Raffaello D'Andrea, Rob Janssen, Moritz Tenorth, Javier Civera, Jos Elfring, Dorian Gálvez-López, Kai Häussermann, J.M.M. Montiel, Alexander Perzylo, Björn Schießle, Oliver Zweigle, and René van de Molengraft. RoboEarth - A World Wide Web for Robots. *Robotics & Automation Magazine*, 18(2), 2011.

Conference papers

Daniel Nyga, Moritz Tenorth, and Michael Beetz. How-models of human reaching movements in the context of everyday manipulation activities. In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9–13 2011.

Moritz Tenorth, Lars Kunze, Dominik Jain, and Michael Beetz. Knowrob-map – knowledge-linked semantic object maps. In *Proceedings of 2010 IEEE-RAS International Conference on Humanoid Robots*, Nashville, TN, USA, December 6-8 2010.

Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In *IEEE/RSJ International Conference on Intelligent RObots and Systems*, Taipei, Taiwan, October 18-22 2010.

Dejan Pangercic, Moritz Tenorth, Dominik Jain, and Michael Beetz. Combining perception and knowledge processing for everyday manipulation. In *IEEE/RSJ International Conference on Intelligent RObots and Systems*, Taipei, Taiwan, October 18-22 2010.

Moritz Tenorth and Michael Beetz. Priming Transformational Planning with Observations of Human Activities. In *IEEE International Conference on Robotics and Automation (ICRA)*, Anchorage, Alaska, May 3-8 2010.

Moritz Tenorth, Daniel Nyga, and Michael Beetz. Understanding and Executing Instructions for Everyday Manipulation Tasks from the World Wide Web. In *IEEE International Conference on Robotics and Automation (ICRA)*, Anchorage, Alaska, May 3-8 2010.

Lars Kunze, Moritz Tenorth, and Michael Beetz. Putting People's Common Sense into Knowledge Bases of Household Robots. In *33rd Annual German Conference on Artificial Intelligence (KI 2010)*, Karlsruhe, Germany, September 21-24 2010.

Moritz Tenorth and Michael Beetz. KnowRob — Knowledge Processing for Autonomous Personal Robots. In *IEEE/RSJ International Conference on Intelligent RObots and Systems*, 2009.

Freek Stulp, Andreas Fedrizzi, Franziska Zacharias, Moritz Tenorth, Jan Bandouch, and Michael Beetz. Combining analysis, imitation, and experience-based learning to acquire a concept of reachability. In *9th IEEE-RAS International Conference on Humanoid Robots*, 2009.

Dejan Pangercic, Rok Tavcar, Moritz Tenorth, and Michael Beetz. Visual scene detection and interpretation using encyclopedic knowledge and formal description logic. In *Proceedings of the International Conference on Advanced Robotics (ICAR)*, Munich, Germany, June 22 - 26 2009.

Michael Beetz, Jan Bandouch, Dominik Jain, and Moritz Tenorth. Towards Automated Models of Activities of Daily Life. In *First International Symposium on Quality of Life Technology -*

Intelligent Systems for Better Living, Pittsburgh, Pennsylvania USA, 2009.

Michael Beetz, Freek Stulp, Bernd Radig, Jan Bandouch, Nico Blodow, Mihai Dolha, Andreas Fedrizzi, Dominik Jain, Uli Klank, Ingo Kresse, Alexis Maldonado, Zoltan Marton, Lorenz Mösenlechner, Federico Ruiz, Radu Bogdan Rusu, and Moritz Tenorth. The assistive kitchen — a demonstration scenario for cognitive technical systems. In *IEEE 17th International Symposium on Robot and Human Interactive Communication (RO-MAN), Muenchen, Germany*, 2008. Invited paper.

Workshop papers

Nico Blodow, Zoltan Csaba Marton, Dejan Pangercic, Thomas Rühr, Moritz Tenorth and Michael Beetz. Inferring generalized pick-and-place tasks from pointing gestures. In *IEEE International Conference on Robotics and Automation (ICRA), Workshop on Semantic Perception, Mapping and Exploration*, 2011.

Moritz Tenorth, Jan Bandouch, and Michael Beetz. The TUM Kitchen Data Set of Everyday Manipulation Activities for Motion Tracking and Action Recognition. In *IEEE Int. Workshop on Tracking Humans for the Evaluation of their Motion in Image Sequences (THEMIS). In conjunction with ICCV 2009*, 2009.

Moritz Tenorth and Michael Beetz. Towards practical and grounded knowledge representation systems for autonomous household robots. In *Proceedings of the 1st International Workshop on Cognition for Technical Systems*, 2008.

Zoltan Csaba Marton, Nico Blodow, Mihai Dolha, Moritz Tenorth, Radu Bogdan Rusu, and Michael Beetz. Autonomous Mapping of Kitchen Environments and Applications. In *Proceedings of the 1st International Workshop on Cognition for Technical Systems*, 2008.

Bibliography

Allen, J. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843.

Arnaud, R. and Barnes, M. (2006). *COLLADA: sailing the gulf of 3D digital content creation*. AK Peters, Ltd.

Azad, P., Asfour, T., and Dillmann, R. (2007). Toward an Unified Representation for Imitation of Human Motion on Humanoids. In *IEEE International Conference on Robotics and Automation (ICRA)*.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. (2007). *The description logic handbook*. Cambridge University Press New York, NY, USA.

Baader, F., Horrocks, I., and Sattler, U. (2008). Chapter 3 description logics. In Frank van Harmelen, V. L. and Porter, B., editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 135–179. Elsevier.

Bandouch, J. and Beetz, M. (2009). Tracking humans interacting with the environment using efficient hierarchical sampling and layered observation models. In *IEEE Int. Workshop on Human-Computer Interaction (HCI). In conjunction with ICCV2009*.

Bandouch, J., Engstler, F., and Beetz, M. (2008). Accurate human motion capture using an ergonomics-based anthropometric human model. In *Proceedings of the Fifth International Conference on Articulated Motion and Deformable Objects (AMDO)*.

Banko, M., Etzioni, O., and Center, T. (2008). The tradeoffs between open and traditional relation extraction. *Proceedings of ACL-08: HLT*, pages 28–36.

Beckett, D. (2004). RDF/XML syntax specification. Technical report, W3C.

Beetz, M., Mösenlechner, L., and Tenorth, M. (2010a). CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*

Beetz, M., Stulp, F., Esden-Tempski, P., Fedrizzi, A., Klank, U., Kresse, I., Maldonado, A., and Ruiz, F. (2010b). Generality and legibility in mobile manipulation. *Autonomous Robots Journal (Special Issue on Mobile Manipulation)*, 28(1):21–44.

Beetz, M., Stulp, F., Radig, B., Bandouch, J., Blodow, N., Dolha, M., Fedrizzi, A., Jain, D., Klank, U., Kresse, I., Maldonado, A., Marton, Z., Mösenlechner, L., Ruiz, F., Rusu, R. B., and Tenorth, M. (2008). The assistive kitchen — a demonstration scenario for cognitive technical systems. In *IEEE 17th International Symposium on Robot and Human Interactive Communication (RO-MAN), Muenchen, Germany*. Invited paper.

Beetz, M., Tenorth, M., Jain, D., and Bandouch, J. (2010c). Towards Automated Models of Activities of Daily Life. *Technology and Disability*, 22(1-2):27–40.

Billard, A., Calinon, S., Dillmann, R., and Schaal, S. (2008). *Springer Handbook of Robotics*, chapter 59. Robot programming by demonstration. Springer.

Blodow, N., Jain, D., Marton, Z.-C., and Beetz, M. (2010). Perception and probabilistic anchoring for dynamic world state logging. In *Proceedings of 2010 IEEE-RAS International Conference on Humanoid Robots*, Nashville, TN, USA.

Blodow, N., Marton, Z.-C., Pangercic, D., Rühr, T., Tenorth, M., and Beetz, M. (2011). Inferring generalized pick-and-place tasks from pointing gestures. In *IEEE International Conference on Robotics and Automation (ICRA), Workshop on Semantic Perception, Mapping and Exploration*.

Bohren, J., Rusu, R. B., Jones, E. G., Marder-Eppstein, E., Pantofaru, C., Wise, M., Mosenlechner, L., Meeussen, W., and Holzer, S. (2011). Towards autonomous robotic butlers: Lessons learned with the pr2. In *ICRA*, Shanghai, China.

Breiman, L. (1984). *Classification and Regression Trees*. Chapman & Hall/CRC, Boca Raton, Florida.

Bui, H. H., Phung, D., Venkatesh, S., and Phan, H. (2008). The Hidden Permutation Model and Location-Based Activity Recognition. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, pages 1345–1350. AAAI Press.

Calinon, S., D'halluin, F., Sauser, E. L., Caldwell, D. G., and Billard, A. G. (2010). Learning and reproduction of gestures by imitation: An approach based on hidden Markov model and Gaussian mixture regression. *IEEE Robotics and Automation Magazine*, 17(2):44–54.

Cooper, R., Schwartz, M., Yule, P., and Shallice, T. (2005). The simulation of action disorganisation in complex activities of daily living. *Cognitive Neuropsychology*, 22(8):959–1004.

Daoutis, M., Coradeshi, S., and Loutfi, A. (2009). Grounding commonsense knowledge in intelligent systems. *J. Ambient Intell. Smart Environ.*, 1(4):311–321.

De la Torre, F., Hodgins, J., Montano, J., Valcarcel, S., and Macey, J. (2009). Guide to the Carnegie Mellon University Multimodal Activity (CMU-MMAC) Database. Technical report, CMU-RI-TR-08-22, Robotics Institute, Carnegie Mellon University.

Ding, C., He, X., Zha, H., and Simon, H. D. (2002). Adaptive dimension reduction for clustering high dimensional data. *Data Mining, IEEE International Conference on*, 0:147.

Drath, R., Luder, A., Peschke, J., and Hundt, L. (2008). AutomationML-the glue for seamless automation engineering. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 616–623. IEEE.

Erol, K., Hendler, J., and Nau, D. (1994). Htn planning: Complexity and expressivity. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1123–1123. John Wiley & Sons LTD.

Fahlman, S. (2006). Marker-passing inference in the scone knowledge-base system. *Knowledge Science, Engineering and Management*, pages 114–126.

Fellbaum, C. (1998). *WordNet: an electronic lexical database*. MIT Press USA.

Fikes, R. O. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. Technical Report 43r, AI Center, SRI International.

Forbus, K. (1984). Qualitative process theory. *Artificial Intelligence*, 24:85–168.

Forbus, K. (1988). Introducing actions into qualitative simulation. In *2nd Qualitative Physics Workshop*, Paris, France.

Frege, G. (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Louis Nebert.

Friedman, N. and Goldszmidt, M. (1996). Learning Bayesian Networks with Local Structure. In *Proceedings CUAI*.

Fung, R. M. and Favero, B. D. (1994). Backward Simulation in Bayesian Networks. In *UAI*, pages 227–234.

Gall, J., Yao, A., Razavi, N., Van Gool, L., and Lempitsky, V. (2011). Hough forests for object detection, tracking, and action recognition. To appear.

Gall, J., Yao, A., and Van Gool, L. (2010). 2d action recognition serves 3d human pose estimation. *Computer Vision–ECCV 2010*, pages 425–438.

Getoor, L. and Taskar, B. (2007). *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Goldman, R., Geib, C., and Miller, C. (1999). A new model of plan recognition. In *Proceedings of the 1999 conference on uncertainty in artificial intelligence*, volume 13, page 14.

Gupta, A., Srinivasan, P., Shi, J., and Davis, L. (2009). Understanding videos, constructing plots learning a visually grounded storyline model from annotated videos. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 2012–2019. Citeseer.

Gupta, R. and Kochenderfer, M. J. (2004). Common Sense Data Acquisition for Indoor Mobile Robots. In *AAAI*, pages 605–610.

Haarslev, V. and Müller, R. (2001). RACER system description. *Automated Reasoning*, pages 701–705.

Hamid, R., Maddi, S., Bobick, A., and Essa, I. (2007). Structure from statistics-unsupervised activity analysis using suffix trees. *IEEE*, 206:7.

Harnad, S. (1990). The symbol grounding problem. *Physica D*, 42:335–346.

Hopkins, B. and Skellam, J. (1954). A new method for determining the type of distribution of plant individuals. *Annals of Botany*, 18(2):213.

Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., and Wang, H. (2006). The manchester owl syntax. *OWL: Experiences and Directions*, pages 10–11.

ISO/IEC 13250:2000 (1999). *Information Technology – Document Description and Processing Languages – Topic Maps.* International Organization for Standardization, Geneva, Switzerland.

Ivanov, Y. and Bobick, A. (2000). Recognition of visual activities and interactions by stochastic parsing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):852–872.

Jain, D., Waldherr, S., and Beetz, M. (2009). Bayesian Logic Networks. Technical report, IAS Group, Fakultät für Informatik, Technische Universität München.

Kate, R., Wong, Y. W., and Mooney, R. (2005). Learning to Transform Natural to Formal Languages. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 1062–1068.

Kautz, H. and Allen, J. (1986). Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32–38.

Kirshner, S., Parise, S., and Smyth, P. (2003). Unsupervised learning with permuted data. In *International Conference on Machine Learning*.

Klank, U., Pangercic, D., Rusu, R. B., and Beetz, M. (2009). Real-time cad model matching for mobile manipulation and grasping. In *9th IEEE-RAS International Conference on Humanoid Robots*, pages 290–296, Paris, France.

Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 423–430, Morristown, NJ, USA. Association for Computational Linguistics.

Krausz, B. and Bauckhage, C. (2010). Action recognition in videos using nonnegative tensor factorization. In *20th International Conference on Pattern Recognition (ICPR)*, pages 1763 –1766.

Krüger, V., Kragic, D., Ude, A., and Geib, C. (2007). The meaning of action: a review on action recognition and mapping. *Advanced Robotics*, 21(13):1473–1501.

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

Kunze, L., Dolha, M. E., Guzman, E., and Beetz, M. (2011a). Simulation-based temporal projection of everyday robot object manipulation. In Yolum, Tumer, Stone, and Sonenberg, editors, *Proc. of the 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Taipei, Taiwan. IFAAMAS.

Kunze, L., Roehm, T., and Beetz, M. (2011b). Towards semantic robot description languages. In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China.

Kunze, L., Tenorth, M., and Beetz, M. (2010). Putting People's Common Sense into Knowledge Bases of Household Robots. In *33rd Annual German Conference on Artificial Intelligence (KI 2010)*, pages 151–159, Karlsruhe, Germany. Springer.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th International Conf. on Machine Learning*, pages 282–289.

Landwehr, N., Gutmann, B., Thon, I., De Raedt, L., and Philipose, M. (2009). Relational transformation-based tagging for activity recognition. *Fundam. Inf.*, 89(1):111–129.

Laskey, K. B. (2008). MEBN: A language for first-order Bayesian knowledge bases. *Artif. Intell.*, 172(2-3):140–178.

Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5):34–43.

Lemaignan, S., R., R., L., M., R., A., and Beetz, M. (2010). Oro, a knowledge management module for cognitive architectures in robotics. In *Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3548–3553.

Lenat, D. (1995). CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38.

Liepmann, H. (1920). Apraxie. *Ergebnisse der gesamten Medizin*, 1:516–543.

Lim, G. H., Suh, I. H., and Suh, H. (2011). Ontology-based unified robot knowledge for service robots in indoor environments. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 41(3):492–509.

Liu, H. and Singh, P. (2004). ConceptNet: A Practical Commonsense Reasoning Toolkit. *BT Technology Journal*, 22:211–226.

Loetzsch, M., Risler, M., and Jüngel, M. (2006). XABSL-a pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129. Citeseer.

Lorenz Mösenlechner and Michael Beetz (2011). Parameterizing Actions to have the Appropriate Effects. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Under review.

Luhr, S., Bui, H., Venkatesh, S., and West, G. (2003). Recognition of human activity through hierarchical stochastic learning. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 416–422.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, 1 edition.

Matuszek, C., Cabral, J., Witbrock, M., and DeOliveira, J. (2006). An introduction to the syntax and content of Cyc. *Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, pages 44–49.

Mavridis, N. and Roy, D. (2006). Grounded situation models for robots: Where words and percepts meet. In *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4690–4697.

McCallum, A. K. (2002). Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu.

McDermott, D. (1992). Robot planning. *AI Magazine*, 13(2):55–79.

Mösenlechner, L. and Beetz, M. (2009). Using physics- and sensor-based simulation for high-fidelity temporal projection of realistic robot behavior. In *19th International Conference on Automated Planning and Scheduling (ICAPS'09)*.

Motik, B., Patel-Schneider, P. F., Parsia, B., Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., and Smith, M. (2009). OWL 2 web ontology language: Structural specification and functional-style syntax. Technical report, W3C.

Niles, I. and Pease, A. (2001). Towards a standard upper ontology. In *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001*, pages 2–9. ACM.

Nyga, D., Tenorth, M., and Beetz, M. (2011). How-models of human reaching movements in the context of everyday manipulation activities. In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China.

O'Brien, P. and Nicol, R. (1998). FIPA – towards a standard for software agents. *BT Technology Journal*, 16(3):51–59.

Padoy, N., Mateus, D., Weinland, D., Berger, M.-O., and Navab, N. (2009). Workflow monitoring based on 3d motion features. In *Proceedings of the International Conference on Computer Vision Workshops, IEEE Workshop on Video-oriented Object and Event Classification*.

Pangercic, D., Haltakov, V., Holzer, S., and Beetz, M. (2011). Fast and robust object detection in household environments using dot descriptors and vocabulary trees with sift descriptors. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Under review.

Pangercic, D., Tenorth, M., Jain, D., and Beetz, M. (2010). Combining perception and knowledge processing for everyday manipulation. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, Taipei, Taiwan.

Paolucci, M., Kawamura, T., Payne, T., and Sycara, K. (2002). Semantic matching of web services capabilities. *The Semantic Web?ISWC 2002*, pages 333–347.

Park, S. and Kautz, H. (2008). Hierarchical Recognition of Activities of Daily Living using Multi-Scale, Multi-Perspective Vision and RFID. In *Intelligent Environments, 2008 IET 4th International Conference on*, pages 1–4.

Pastor, P., Hoffmann, H., Asfour, T., and Schaal, S. (2009). Learning and generalization of motor skills by learning from demonstration. In *Proceedings of the International Conference on Robotics and Automation (icra2009)*.

Patterson, D., Fox, D., Kautz, H., and Philipose, M. (2005). Fine-grained activity recognition by aggregating abstract object usage. In *Ninth IEEE International Symposium on Wearable Computers, 2005. Proceedings*, pages 44–51.

Penberthy, J. and Weld, D. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *proceedings of the third international conference on knowledge representation and reasoning*, pages 103–114. Citeseer.

Perkowitz, M., Philipose, M., Fishkin, K., and Patterson, D. J. (2004). Mining models of human activities from the web. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 573–582. ACM.

Platt, J. (1999). Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods*, pages 185–208. MIT Press.

Quattoni, A., Collins, M., and Darrell, T. (2004). Conditional random fields for object recognition. In *NIPS*, pages 1097–1104. MIT Press.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California.

Reiser, U., Connette, C., Fischer, J., Kubacki, J., Bubeck, A., Weisshardt, F., Jacobs, T., Parlitz, C., Hagele, M., and Verl, A. (2009). Care-o-bot 3 - creating a product vision for service robot applications by integrating design and technology. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1992 –1998.

Richardson, M. and Domingos, P. (2006). Markov Logic Networks. *Mach. Learn.*, 62(1-2):107–136.

Roduit, P., Martinoli, A., and Jacot, J. (2007). A quantitative method for comparing trajectories of mobile robots using point distribution models. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2441–2448.

Roggen, D., Calatroni, A., Rossi, M., Holleczek, T., Forster, K., Troster, G., Lukowicz, P., Bannach, D., Pirkl, G., Ferscha, A., Doppler, J., Holzmann, C., Kurz, M., Holl, G., Chavarriaga, R., Sagha, H., Bayati, H., Creatura, M., and del R Millan, J. (2010). Collecting complex activity datasets in highly rich networked sensor environments. In *Networked Sensing Systems (INSS), 2010 Seventh International Conference on*, pages 233 –240.

Ros, R., Lemaignan, S., Sisbot, E., Alami, R., Steinwender, J., Hamann, K., and Warneken, F. (2010). Which one? grounding the referent based on efficient human-robot interaction. In *19th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 570 –575.

Rudolph, D., Stürznickel, T., and Weissenberger, L. (1993). *Der DXF-Standard*. Rossipaul.

Ryoo, M. and Aggarwal, J. (2006). Recognition of composite human activities through context-free grammar based representation. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 1709–1718.

Schaal, S. (1999). Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6):233–242.

Schuster, M., Jain, D., Tenorth, M., and Beetz, M. (2011). Learning Organizational Principles in Human Environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Under review.

Shearer, R., Motik, B., and Horrocks, I. (2008). Hermit: A highly-efficient owl reasoner. In *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, pages 26–27. Citeseer.

Shi, Y., Huang, Y., Minnen, D., Bobick, A., and Essa, I. (2004). Propagation networks for recognition of partially ordered sequential action. *cvpr*, 02:862–869.

Shoham, Y. (1985). Ten requirements for a theory of change. *New Generation Computing*, 3(4):467–477.

Shotton, J., Fitzgibbon, A., Cook, M., Sharp, T., Finocchio, M., Moore, R., Kipman, A., and Blake, A. (2011). Real-time human pose recognition in parts from single depth images. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

Siméon, T., Laumond, J.-P., and Lamiraux, F. (2001). Move3D: a generic platform for path planning. In *4th International Symposium on Assembly and Task Planning*.

Singh, P., Lin, T., Mueller, E. T., Lim, G., Perkins, T., and Zhu, W. L. (2002). Open mind common sense: Knowledge acquisition from the general public. In *Proceedings of the First International Conference on Ontologies, Databases, and Applications of Semantics for Large Scale Information Systems*, pages 1223–1237.

Singh, P. and Williams, W. (2003). LifeNet: A Propositional Model of Ordinary Human Activity. In *Workshop on Distributed and Collaborative Knowledge Capture (DC-KCAP)*.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Web Semant.*, pages 51–53.

Smith, D. and Arnold, K. (2009). Learning hierarchical plans by reading simple english narratives. In *Proceedings of the Commonsense Workshop at IUI-09*.

Srinivasa, S., Ferguson, D., Helfrich, C., Berenson, D., Romea, A. C., Diankov, R., Gallagher, G., Hollinger, G., Kuffner, J., and Vandeweghe, J. M. (2010). Herb: a home exploring robotic butler. *Autonomous Robots*, 28(1):5–20.

Sterling, L. and Shapiro, E. (1994). *The Art of Prolog: Advanced Programming Techniques*. MIT Press.

Stulp, F., Oztop, E., Pastor, P., Beetz, M., and Schaal, S. (2009). Compact models of motor primitive variations for predictable reaching and obstacle avoidance. In *9th IEEE-RAS International Conference on Humanoid Robots*.

Suchanek, F., Kasneci, G., and Weikum, G. (2007). Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM.

Suh, I. H., Lim, G. H., Hwang, W., Suh, H., Choi, J.-H., and Park, Y.-T. (2007). Ontology-based Multi-Layered Robot Knowledge Framework (OMRKF) for Robot Intelligence. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 429–436.

Tellex, S. and Roy, D. (2006). Spatial routines for a simulated speech-controlled vehicle. In *HRI '06: Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, New York, NY, USA. ACM.

Tenorth, M., Bandouch, J., and Beetz, M. (2009). The TUM Kitchen Data Set of Everyday Manipulation Activities for Motion Tracking and Action Recognition. In *IEEE Int. Workshop on Tracking Humans for the Evaluation of their Motion in Image Sequences (THEMIS). In conjunction with ICCV2009*.

Tenorth, M. and Beetz, M. (2009). KnowRob — Knowledge Processing for Autonomous Personal Robots. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, pages 4261–4266.

Tenorth, M., Kunze, L., Jain, D., and Beetz, M. (2010a). KNOWROB-MAP – Knowledge-Linked Semantic Object Maps. In *Proceedings of 2010 IEEE-RAS International Conference on Humanoid Robots*, Nashville, TN, USA.

Tenorth, M., Nyga, D., and Beetz, M. (2010b). Understanding and Executing Instructions for Everyday Manipulation Tasks from the World Wide Web. In *IEEE International Conference on Robotics and Automation (ICRA).*, pages 1486–1491.

Thielscher, M. (1998). Introduction to the fluent calculus. *Electron. Trans. Artif. Intell.*, 2:179–192.

Thielscher, M. (2000). Representing the knowledge of a robot. In *PRINCIPLES OF KNOWLEDGE REPRESENTATION AND REASONING-INTERNATIONAL CONFERENCE-*, pages 109–120. Citeseer.

Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic Robotics*. MIT Press, Cambridge.

v. Hoyningen-Huene, N., Kirchlechner, B., and Beetz, M. (2007). GrAM: Reasoning with grounded action models by combining knowledge representation and data mining. In *Towards Affordance-based Robot Control*.

Vail, D. L., Veloso, M. M., and Lafferty, J. D. (2007). Conditional random fields for activity recognition. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA. ACM.

Vassiliadis, V., Wielemaker, J., and Mungall, C. (2009). Processing OWL2 ontologies using Thea: An application of logic programming. In *OWLED, CEUR Workshop Proceedings*, volume 529.

Waibel, M., Beetz, M., D'Andrea, R., Janssen, R., Tenorth, M., Civera, J., Elfring, J., Gálvez-López, D., Häussermann, K., Montiel, J., Perzylo, A., Schießle, B., Zweigle, O., and van de Molengraft, R. (2011). RoboEarth - A World Wide Web for Robots. *Robotics & Automation Magazine*, 18(2). Accepted for publication.

Weinland, D., Ronfard, R., and Boyer, E. (2006). Free viewpoint action recognition using motion history volumes. *Computer Vision and Image Understanding*, 104(2-3):249–257.

Wielemaker, J., Schreiber, G., and Wielinga, B. (2003). Prolog-based infrastructure for RDF: performance and scalability. In Fensel, D., Sycara, K., and Mylopoulos, J., editors, *The Semantic Web - Proceedings ISWC'03, Sanibel Island, Florida*, pages 644–658, Berlin, Germany. Springer Verlag. LNCS 2870.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition.

Wu, F. and Weld, D. S. (2007). Autonomously semantifying wikipedia. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 41–50, New York, NY, USA. ACM.

Wu, Z. and Palmer, M. (1994). Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics Morristown, NJ, USA.

Zelek, J. (1997). Human-robot interaction with minimal spanning natural language template for autonomous and tele-operated control. In *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems, 1997. IROS '97.*, volume 1.

Zhou, F. and De la Torre, F. (2009). Canonical time warping for alignment of human behavior. In *Advances in Neural Information Processing Systems Conference (NIPS)*.

Zweigle, O., van de Molengraft, R., d'Andrea, R., and Häussermann, K. (2009). RoboEarth: connecting robots worldwide. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 184–191. ACM.