Master's Thesis

Lifelong Learning of First-order Probabilistic Models for Everyday Robot Manipulation

(Lebenslanges Lernen von probabilistischen Modellen erster Stufe für alltägliche Roboter-Manipulationsaufgaben)

Author:	Marc Niehaus
Advisors:	Prof. Michael Beetz, PhD
	Dr. Jean Christoph Jung
Supervisor:	Daniel Nyga, M.Sc.

Submission date:	02.08.2016

Abstract

This thesis is about robots learning from their experience in a household domain. It describes how *Markov Logic Networks* can be learned from *log files* written during the execution of *CRAM* plans. *Markov Logic Networks* equip first-order logic formulas with a weight to state how often these formulas are true. *CRAM* is a system that allows high-level robot programming with Lisp. It is used in robots which can, for example, be used to set the breakfast table. *CRAM* plans have to be parametrized: If there is a plan to *pick up an object*, the *object* parameter might be *an apple on the cupboard*. The learned *Markov Logic Networks* enable the robots to reason about *CRAM* plans and their usual parameters. Thus, they can be used to extend incomplete plan parameters, such as *an apple* by information learned from experience, such as *on the cupboard*, to make them usable for a plan. Other queries, such as queries for the success probability for the parametrized plan *pick up an apple on the cupboard* or queries for the objects which are usually perceived *on the cupboard*, are also possible. Apart from that, a new inference algorithm for *Markov Logic Networks* is presented. An evaluation will show experimentally that the developed system works, even if it is not perfect. Furthermore, the strengths and weaknesses of the approach are discussed.

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. General Idea . . . 3
1.3. Contributions . . . 4

2. Foundations **7**
2.1. The PR2 Robot and ROS . . . 7
2.2. KnowRob . . . 9
2.3. CRAM (Cognitive Robot Abstract Machine) . . . 11
2.3.1. CRAM Plan Language . . . 11
2.3.2. Bullet Reasoning . . . 13
2.3.3. CRAM_M . . . 14
2.4. Markov Logic Networks . . . 17
2.4.1. Formula Structure . . . 17
2.4.2. Represented Probability Distribution . . . 19
2.4.3. MLNs and Markov Random Fields . . . 21
2.4.4. Inference Algorithms . . . 22
2.4.5. Learning Methods . . . 25
2.4.6. Common Pitfalls . . . 27
2.4.7. pracmln . . . 27
2.5. Usage in this Thesis . . . 29

3. Related Work **31**
3.1. Learning Models for Designators from Experience . . . 31
3.2. Improved Exact MLN Inference . . . 35

4. MLN Design **37**
4.1. A Naive Approach . . . 40
4.2. A Generic Approach . . . 43
4.3. Simple Weight Calculation . . . 48
4.4. A Refined Generic Approach . . . 49
4.5. A Designator Based Approach . . . 50
4.6. The Final Approach . . . 54

5. Improved Inference **59**
5.1. Algorithm Description . . . 60
5.1.1. Get Ground Formulas and Conjugate . . . 62

5.1.2. Combine Formulas . . . 65
5.1.3. Assign World Number . . . 66
5.1.4. Calculate Sum . . . 68
5.2. Applicability . . . 69

6. Implemented Software 71
6.1. Overview . . . 71
6.2. robot_memory . . . 75
6.3. cram_robot_memory . . . 77

7. Evaluation 81
7.1. Experiments . . . 81
7.1.1. Experiment Description . . . 81
7.1.2. Results . . . 86
7.1.3. Performance . . . 88
7.2. Opportunities and Limitations . . . 89

8. Conclusions 93

Appendices 95

A. Proofs 95

B. Code Listings 99

C. Affirmation 101

D. CD 103

List of Figures

1.1. Use Cases for this Thesis 3
1.2. Abstract Concept of the Software Developed in this Thesis 5

2.1. The PR2 Robot 7
2.2. Ontology Describing Bicycles 10
2.3. Description of the CRAM Language 11
2.4. The PR2 in the Bullet Visualization 14
2.5. Description of the Logging and Prediction Mechanism in CRAM 15
2.6. Task Tree Generated from the Data Logged by CRAM 15
2.7. Grammar for First-order Logic Formulas 18
2.8. MRF Generated from the MLN in Listing 2.5 22
2.9. The MLN from Listing 2.5 in pracmln 28
2.10. Assignment of Existing Software to the Thesis Concept 30

4.1. The Structure of the CRAM Log Files 38
4.2. An Example Log File as UML Object Diagram 39
4.3. State Machine for the Example in Figure 4.2 44

5.1. Top Level Activities in the Inference Algorithm 62
5.2. Calculating the Numerator or Denominator in the Inference Algorithm . . 63
5.3. Combining Formulas in the Inference Algorithm 66
5.4. Formula Graph for the Ground Formulas of Listing 5.3 67

6.1. Assignment of ROS Packages to the Learning Part of the Thesis Concept . 72
6.2. Assignment of ROS Packages to the Inference Part of the Thesis Concept 73
6.3. Communication Relationships Between the ROS Packages 74
6.4. Origin of the Different ROS Packages 75
6.5. Conversion of Task Trees to MLNs 76
6.6. The Designator Completion Algorithm 77
6.7. The completeKey Function of the Designator Completion 78
6.8. The completeValue Function of the Designator Completion 79

7.1. Training and Test Kitchens Used in the Evaluation 82
7.2. PR2 Robot in the first Training Kitchen 83

List of Tables

2.1. Worlds for Listing 2.5 20

5.1. Worlds for the MLN in Listing 5.3 61
5.2. Possible Ground Atoms of the Predicates from Listing 5.1 64
5.3. Combined Ground Formulas of the MLN in Listing 5.3 65
5.4. Containment of the Formula Combinations in Table 5.3 67

7.1. Objects Used in the Evaluation 84
7.2. Locations Used in the Evaluation 84
7.3. Evaluation Results 86
7.4. Performance for one Designator Completion 88

List of Listings

2.1. Definition of the Example ROS Message "distance" 8
2.2. Definition of the Example ROS Service "command" 8
2.3. OWL Queries from Prolog 10
2.4. Example CRAM Plan get-mug 13
2.5. MLN for the Computer Science Student / Pizza Example 18

4.1. Predicate Declarations for a Naive MLN 40
4.2. Extract of the Training Database for a Naive MLN 41
4.3. Extract of the Template Formulas for a Naive MLN 41
4.4. One Expanded Template Formula for a Naive MLN 42
4.5. Query for an Object Expected at a Location in the Naive MLN 42
4.6. Query for a Success Probability in the Naive MLN 42
4.7. Query for a Plan Parametrization in the Naive MLN 42
4.8. Predicate Declarations for the Generic MLNs 43
4.9. One Training Database for a Generic MLN 45
4.10. Example Formulas of a Generic Object MLN 45
4.11. Query for an Object Perceived at a Location in the Generic Object MLN . 45
4.12. Example Formula of a Generic State Machine MLN 46
4.13. Query for the Next Task in the State Machine MLN 46
4.14. Example Formula of a Generic Task MLN 47
4.15. Query for a Success Probability in the Task MLN 47
4.16. Query for a Successful Parametrization of a Task in the Task MLN 47
4.17. Example Formulas of a Refined Generic Object MLN 49
4.18. Grounded Formula of a Refined Generic Object MLN 49
4.19. Predicate Declarations of a Designator Based MLN 51
4.20. Extract of a Designator Based MLN 52
4.21. Training Databases for one Task in the Designator Based MLN 52
4.22. Query for Objects Perceived at a Location Using the Designator Based MLN 53
4.23. Query for a Success Probability Using the Designator Based MLN 53
4.24. Query for the Completion of a Designator Using the Designator Based MLN 53
4.25. Predicate Declarations of the Final MLN 54
4.26. Example Formulas of a MLN Using the Final Design 55
4.27. Query for Completing a Designator Property Key with the Final MLN . . 56
4.28. One Training Database for a MLN Using the Final Design 57
4.29. Query inferring what to Expect on a Cupboard Using the Final MLN . . . 57
4.30. Query for Failure and Success Probabilities Using the Final MLN 58
4.31. Query for Completing a Designator Property Value with the Final MLN . 58

5.1. Simplified Designator Based MLN 59
5.2. Query for the Success Probability of a Plan Parametrized with a Mug . . 59
5.3. Example MLN Used for Explaining the Inference Algorithm 61
5.4. Evidence Database for the MLN in Listing 5.3 61
5.5. Advanced Success Query in Listing 5.1 64

6.1. Evidence for the propertyKey Query in Listing 4.27 78

7.1. Example Object of a Training Kitchen 82
7.2. Example Object of a Test Kitchen 83
7.3. Designator Describing a Mug on the Kitchen Sink Block 89
7.4. Designator Describing a Mug with a Handle 90
7.5. Designator Describing a Mug 90
7.6. Designator Describing a Mug with a Category Property 90
7.7. Designator Describing a Glass with a Category Property 90
7.8. Designator Describing a Mug on the Pancake Table 91
7.9. Designator Describing a Plate on the Pancake Table 91
7.10. Designator Describing a Table Setting Action in Detail 92
7.11. Designator Describing a Part of the Table Setting Action 92

B.1. OWL File Describing Bicycles 99
B.2. Template Formulas for a Naive MLN 100

1. Introduction

1.1. Motivation

Imagine a robot which is helping in the household. It could set the table or prepare meals. When executing those tasks, it would be frustrating if the robot would fail on the same task over and over again. If one looks at the human species, they are able to learn from their experience their whole life long to avoid this. This is one of the reasons why humans are able to adapt to new environments and acquire new skills.

More precisely, the human is able to learn new activities. An example is baking a cake: If the human has never done this before, another human can teach him how to bake a cake. After the human has baked some cakes under the guidance of the teacher, he is able to bake one on his own. Moreover, the learning goes beyond the acquisition of new skills in this example. If the human has learned how to fold in beaten egg white, he learns to estimate the success of this action. In the case that he has few eggs available and he knows that he usually fails in folding in beaten egg white, he can ask another human to perform this task.

Another situation in which the human learns from its experience is when he bakes cakes in different kitchens. If the baking tin is located somewhere else in another kitchen and he finds out where, the human can remember this location and will also search at this location the next time. The human is also able to use these facts the other way around: If he wants to go shopping and if he is writing a shopping list, he can look at all food storage locations and he knows what is missing. Thus, he knows what he expects at a storage location.

In all these examples, the human updates what he learned the whole life long. He might try to simplify the baking process and use this simplification later on. The more often he tries to fold in beaten egg white, the more successful he gets. Hence, he updates his estimate on how often he fails. The more kitchens he uses, the more storage locations are in his mind when searching something. Thus, the learning process is never completed.

When robots execute a task in the household, they are usually following plans. Such a plan can be "*bake a cake*", but also "*pick up an object*". These plans are parametrized. A plan parameter for the plan to *pick up an object* might be an *apple in the fruit basket*. Therefore, the parametrized plan would be to *pick up an apple in the fruit basket*. Learning plans and thus new activities is a huge research field and will not be covered in this thesis. Instead, this thesis focusses on the combination of plans and parameters.

Why can it be necessary for a robot to learn about plans and their parameters its whole life long? The robot might receive vaguely specified commands from a human, such as *bring me an apple*. It might also have a plan for displacing an object that could be applied in this situation. Since plans usually consist of sub plans, it would first execute the sub plan to *pick up an object* with the *apple* as parameter. However, this sub plan also needs the location of the apple. Hence, the plan parameter is incomplete. Therefore, the robot must make the decision where to find the apple. This decision has to be made at runtime, which can be supported by the learned information.

If the robot learned how to parametrize a plan to be successful, it can use this information in such a case. Thus, it can automatically infer that it needs the information where the apple is placed to successfully execute the pick up plan. Moreover, it is able to infer that the apple is usually placed in the fruit basket. Without a learning mechanism, it would be required to embed this information in the robot plan which makes this plan less general. With the learning approach, the robot can be taught to pick up an apple in the fruit basket and it is able to use this information later on. Furthermore, the lifelong learning approach makes it possible to update this information. For instance, if apples are placed somewhere else in the kitchen afterwards.

It might also be necessary to estimate success probabilities for a given parametrization. If the robot learned a model for these success probabilities, it could decide to choose an alternative behavior based on the success probability. An example could be the command to set the table for a special occasion. This could involve placing expensive plates on the table. The robot might estimate that it frequently loses plates when setting the table. In this case, it might better ask a human to perform this task. When the success rate grows due to a software update, the estimate is also updated. Then, the robot will probably take the challenge to pick up the plate.

If the robot knows what to expect at a specific location, it is able to prepare for being at this location. It could, for example, prepare the perception mechanism to expect these objects. Moreover, it could abort the plan execution, for instance if an expensive plate is expected at a location where another object should be placed. Finally, it is able to infer which objects are probably missing if it wants to go shopping. All the use cases mentioned before are summarized in Figure 1.1.

At the same time, learning is not easy. There are some conditions that make the human learning very special. Humans are, for example, able to generalize: If a human is told to pick up an expensive plate and if he has no idea what "expensive" means, he will probably pick it up like any other plate. If the human learns how to pick up a white plate, he can also pick up a brown plate. However, more advanced example of generalization, such as transferring the ability to pick up a plate in order to pick up a wooden board, will not be covered by this thesis. Humans are also able to deal with contradictions, which is another condition. If another human tells that it is impossible to pick up a plate and nine other humans tell that it is possible, the human will probably believe it is possible. Thus, these conditions are important when applying learning in robotics.

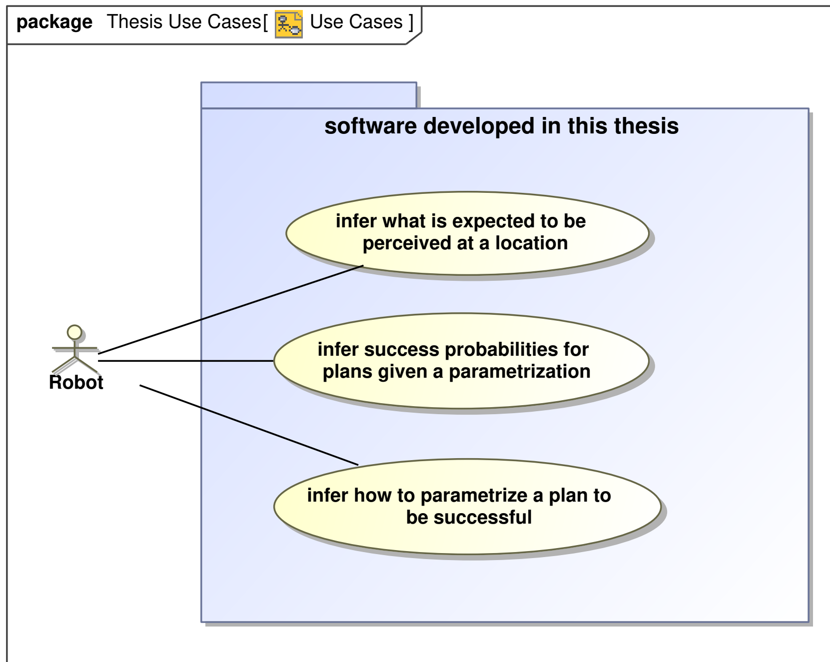


Figure 1.1.: Use Cases for this Thesis

In this thesis, an approach implementing learning for the use cases and conditions mentioned before is presented.

1.2. General Idea

Some of the household robots mentioned above can write down what they have done in form of logs. The idea of this thesis is to equip a robot with software learning and using a model from these logs. Since the robot shall be able to deal with uncertainty, probabilistic models are well suited as model. One of these model types is called *Markov Logic Network* (MLN). It is a formalism for combining first-order logic formulas such as "all apples are located in the fruit basket" with a probabilistic semantic stating how often this is true. Among other probabilistic models, MLNs have the advantage that they are based on first-order logic and thus they are very flexible. This is very useful here since different queries can be executed on one MLN. Hence, the robot can decide at runtime which information it needs and then create a appropriate query.

Figure 1.2 shows abstractly how the software developed in this thesis works: After the robot has executed a plan, it writes log files. Then, the log files are converted to a MLN. Afterwards, as shown in the activity at the bottom of Figure 1.2, the MLN can be used to infer probabilities for the queries mentioned in Figure 1.1. Results of these queries might be:

- The parameter *mug* for the *displace* plan must be extended with the location *on cupboard* to succeed.
- With this parametrization, the action will fail in 50% of the cases with a *Manipulation Failure*.
- Apart from the mug, an expensive plate is also expected to be on the cupboard.

The robot can then analyze the situation and it might decide to use a different plan, e.g. to ask a human to displace the mug. When the plan is executed, the plan parameters inferred by the MLN can be used. Though learning and inference are depicted separately in Figure 1.2, they can also be combined: The robot can use inferred plan parameters and success probabilities and learn its experiences with these later on.

1.3. Contributions

The previous sections already explained the scope of this thesis. In particular, this thesis contributes the following points:

- In Chapter 4, it shows how a Markov Logic Network capable of answering the queries shown in Figure 1.1 can be designed. Different MLN designs are presented and their advantages and disadvantages are worked out.
- In Chapter 5, it presents an algorithm able to perform efficient exact inference in one of the MLN designs developed before. The algorithm is also applicable to other MLNs as long as the formulas are conjunctions.
- In Chapter 6, it describes the software implemented in the scope of this thesis in order to use the Markov Logic Network designed in Chapter 4 with a (simulated) robot. Moreover, the algorithms manifested in this software are presented. The software, and thus also the MLN design, are experimentally evaluated and discussed in Chapter 7.
- In Appendix A, it provides a proof stating that the weights calculated for the MLNs in the implementation are correct.

Before these contributions are explained in detail, Chapter 2 introduces the frameworks and concepts used in this thesis and explains where they are used. Moreover, Chapter 3 presents works that are related to these contributions.

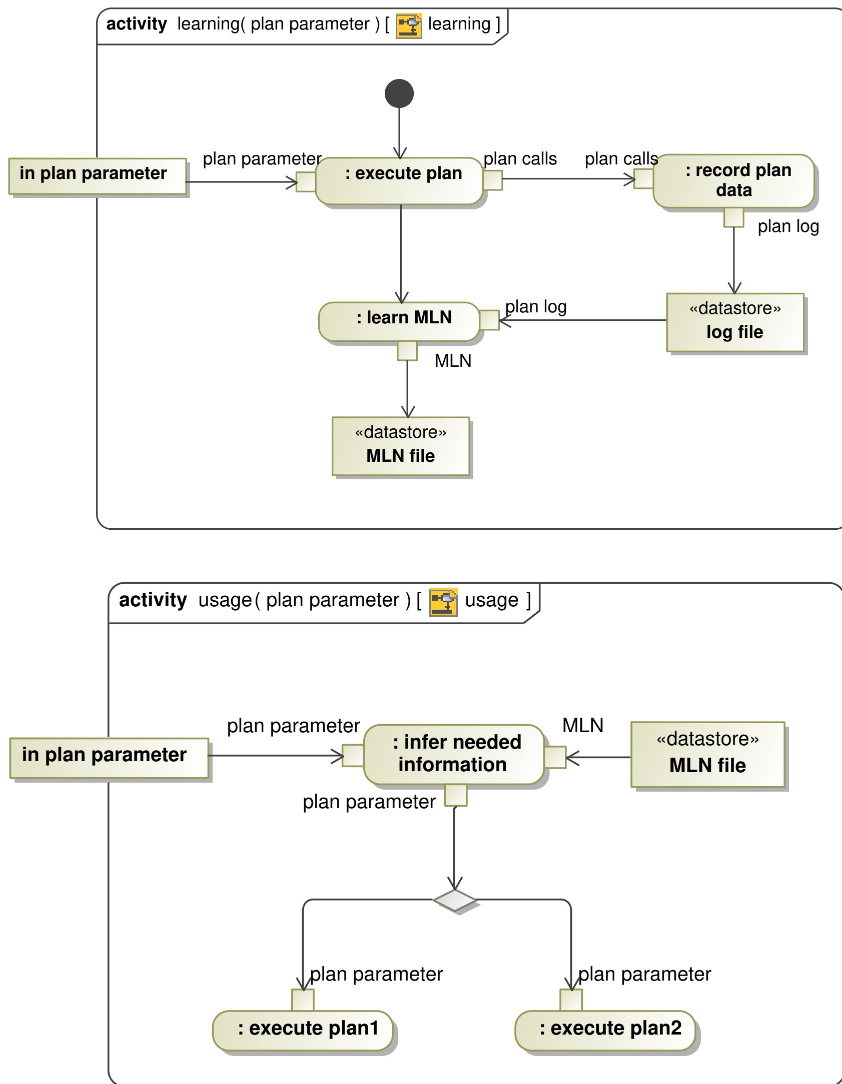


Figure 1.2.: Abstract Concept of the Software Developed in this Thesis

2. Foundations

This chapter introduces existing frameworks and algorithms used throughout this thesis. Furthermore, it describes where they are used in this thesis.

2.1. The PR2 Robot and ROS

Figure 2.1 shows the PR2. It is a mobile robot manufactured by the company Willow Garage. The PR2 is intended to operate in human environments such as the household. Thus, it has two arms with grippers at their end to be able to manipulate objects. Various sensors including cameras can be used to keep track of the environment. More information can be found in [OCC+09]. Though the algorithms presented in this work are not limited to the PR2, it is used as example. This is especially relevant in the case study in Chapter 7. Due to limited resources, a simulated version of the PR2 will be used.

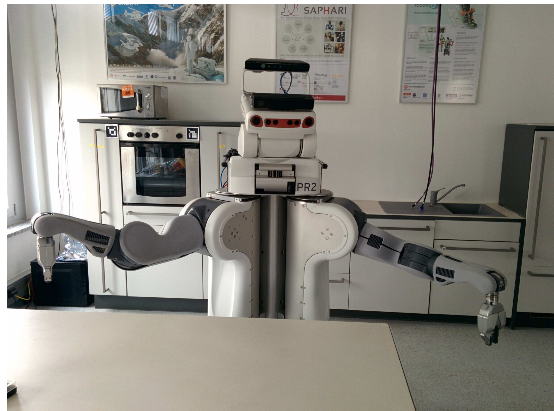


Figure 2.1.: The PR2 Robot

The software on the PR2 is based on *ROS* [Cou10]. ROS is the short form of *Robot Operating System* though it is actually an open source framework running on the actual operating system. It provides a mechanism for inter process communication. Processes in ROS are called *nodes*. They exchange *messages* through the ROS middleware. Nodes can run on different hosts in a peer-to-peer network. ROS uses a publish/subscribe based architecture: A node can *publish* a message to a *topic*. Other nodes *subscribed* to that

topic are notified when a new message arrives. Furthermore, a node can advertise a *service* for synchronous communication: A service consists of a request and a response message such that the service is called with the request message and responds with the response message. Message definitions are written in an own interface definition language converted by a code generator to classes or structures usable from different programming languages. The software instantiated as nodes is aggregated in packages. Several packages usable on different robots are available. Packages are built by a CMake based build system and, as mentioned before, they can be written in different languages. [QGC+09]

An example of a ROS node could be a simple safety node for a fictional robot: The node issues a stop command to the drive engine when the space in front of the robot falls below a certain threshold. In order to measure the space in front of the robot, a laser range finder is used as sensor. It periodically publishes a distance message on the topic "lrf/distance" whose IDL looks as in Listing 2.1.

```
float32 distance
```

Listing 2.1: Definition of the Example ROS Message "distance"

The ROS build system generates code for languages such as Python, C++ or Lisp. Now, a "safety" package can be created with a program called "distance" in one of these languages. The program calls a routine to start the node and afterwards it calls a routine to subscribe to the topic "lrf/distance". Every time the laser range finder publishes a new message, a callback is invoked in the safety program to process the distance message. It can now check whether the distance is below the threshold and send a stop command if necessary. The drive engine advertises a service "de/stop" for receiving commands. Thus, there must be a message definition for the request and response message which is shown in Listing 2.2.

```
uint8 STOP_COMMAND=0
# ... other commands ...
uint8 command
---
uint8 NO_ERROR=0
# ... other error codes ...
uint8 error
```

Listing 2.2: Definition of the Example ROS Service "command"

If the safety program decides to issue a stop command it calls the "de/command" service with an instance of the response message definition above. The drive engine processes the request message and returns an instance of the response message. Now the control flow returns to the safety node and it can have a look at the error code.

To summarize it: The PR2 is a mobile robot usable in the household with software in form of ROS packages communicating via the ROS middleware.

2.2. KnowRob

Some of the ROS packages that could be installed on robots like the PR2 are provided by *KnowRob*. It is a *knowledge base* that is implemented in SWI Prolog, a logic programming language. KnowRob can, for example, be used to infer missing information on vague specified tasks. An example is the information which objects are present in the kitchen. One use case for KnowRob is to represent encyclopedic knowledge such as "a glass is a container for fluids". For this purpose, KnowRob is able to load *OWL* (Web Ontology Language) files in Prolog and use them in queries. As the name states, OWL is a file format to represent ontologies. An ontology contains concepts in form of a taxonomy connected by relations. OWL files are based on XML. The formal background is provided by Description Logics. In Description Logics, there are the TBOX describing a taxonomy of concepts (e.g. a glass is a container for fluids) and the ABOX describing individuals (e.g. The glass that is in the sink). Moreover, there are roles representing properties and relations between individuals. The ontology that KnowRob primarily uses is an extended version of the OpenCyc ontology. Thus, KnowRob is able to reason about concepts and individuals. [TB13]

One use case for KnowRob mentioned in [TB13] is to gather the information where an object might be found. They propose different mechanisms: First, KnowRob can be queried for the location of similar objects. Another option is to use object properties and classes. For instance, perishable objects are usually stored in the fridge. However, they also mention that it is difficult to combine this information.

For information from other sources than the KnowRob ontology, KnowRob provides *virtual knowledge bases*. This means that the information is extracted form the source programmatically when it is needed in a query. It is achieved by using *computables*. Computables can be attached to description logic classes or properties and thus they either generate individuals or relations. One use case of computables is the integration of *ProbCog*, a framework for statistical relational learning (See Section 2.4 for details). This enables KnowRob to cope with uncertainty. [TB13]

Figure 2.2 shows an example of an ontology. The corresponding OWL file is shown in Listing B.1. It describes the classes *Human*, *Bicycle* and *RoadBike*. A RoadBike is a sub class of a Bicycle. Moreover, relations and properties are present. Bicycles have a *color* and they can have an *owner* which is a Human. Moreover, the file contains two instances: *Marc* is a Human. *MarcsRoadBike* is a RoadBike with the color black and the owner Marc.

Now, using the KnowRob Prolog predicates, queries to the OWL file using Prolog are possible. Listing 2.3 shows some queries.

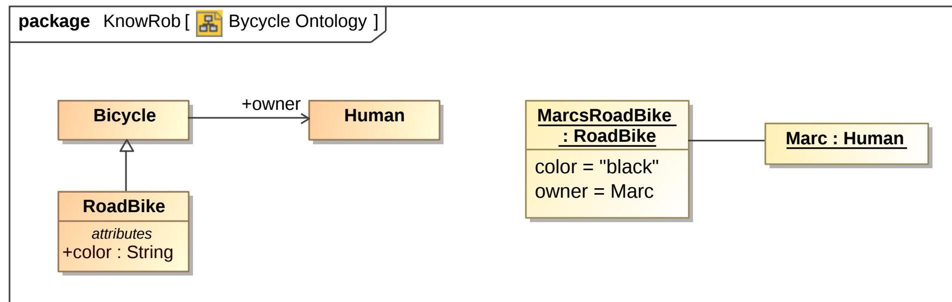


Figure 2.2.: Ontology Describing Bicycles

```
?- rdf_db:rdf_register_ns(bike, 'http://localhost/bicycle.owl#', [keep(true)]).
true.

?- owl_parse('/home/marc/bicycle.owl').
% Parsed "bicycle.owl" in 0.00 sec; 19 triples
true.

?- owl_individual_of(I, bike:'Bicycle').
I = bike:'MarcsRoadBike' ;
false.

?- owl_subclass_of(C, bike:'Bicycle').
C = bike:'Bicycle' ;
C = bike:'RoadBike' ;
false.

?- owl_has(bike:'MarcsRoadBike', K, V).
K = rdf:type,
V = owl:'NamedIndividual' ;
K = rdf:type,
V = bike:'RoadBike' ;
K = bike:color,
V = literal(type(xsd:string,black)) ;
K = bike:owner,
V = bike:'Marc' ;
false.
```

Listing 2.3: OWL Queries from Prolog

After registering the namespace *bike* for more comfortable queries and parsing the owl file, the system is asked for all individuals of the type Bicycle. Since MarcsRoadBike is of the type RoadBike and RoadBike is a sub class of Bicycle, MarcsRoadBike is returned. Then, the system is asked for all sub classes of the Bicycle which results in the Bicycle itself and the RoadBike. Finally, it is queried for the properties and relations of MarcsRoadBike which results in the type, the color and the owner. Similarly, the KnowRob ontology can be queried.

2.3. CRAM (Cognitive Robot Abstract Machine)

Robots like the PR2 are controlled by plans. These plans have very special requirements: In domains like the household, robots must execute plans for vaguely formulated tasks. An example of such a vague task is “Clean the windows”: This sentence omits information such as which windows have to be cleaned and where the windows are located. Moreover, the robot itself can only execute simple actions such as moving its arm or driving to a position specified by coordinates. However, the plan for cleaning the window should be as general possible in order to execute it on different robots. The *CRAM* language aims to solve these problems. Thus, CRAM is a framework for writing plans for robots like the PR2. [WBMB12] [BMT10]

2.3.1. CRAM Plan Language

Figure 2.3 visualizes the concepts of CRAM: In CRAM, A plan is basically a common Lisp function defined with special constructs from the *CRAM Plan Language*. This differs from other systems where plans are formed by partially ordered action sequences. The reason is that plan should be aware of failures or sensory information. [BMT10] [WBMB12]

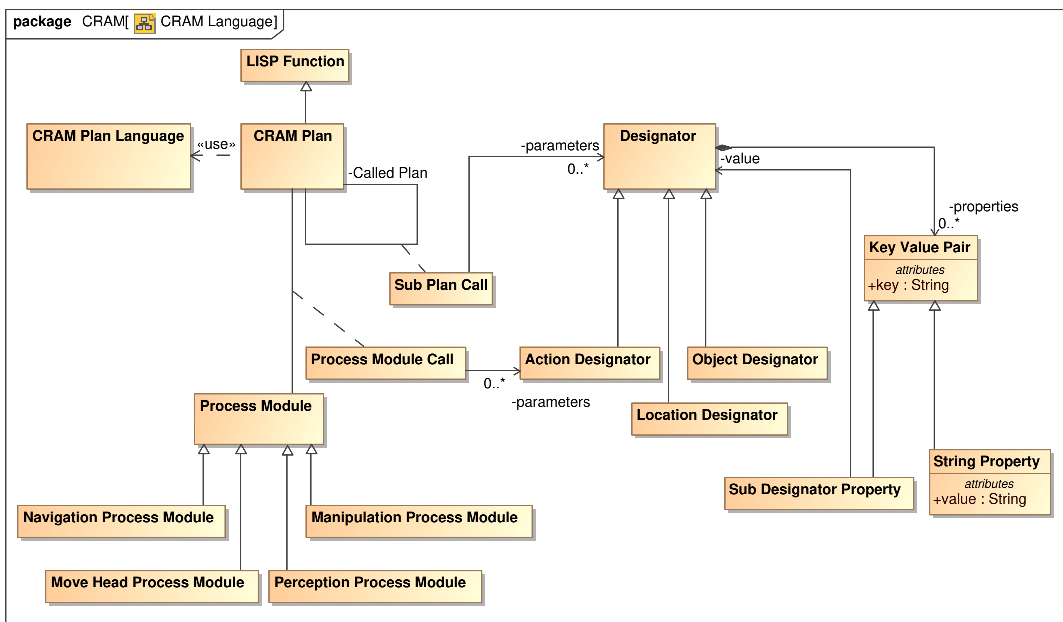


Figure 2.3.: Description of the CRAM Language

As a Lisp function, the plan can call other plans by executing the function. These sub plans might have parameters. As mentioned before, a parameter is usually a more or less

vague description of an action, object or a location. Therefore, CRAM provides a special construct for these parameters: *Designators*. A designator consists of key value pairs where the key is a string. The value can either be a string or another designator. There are different kinds of designators. The most important ones are action designators, object designators and location designators. Action designators describe an action, such as an action that tells the robot to perceive something. Object designators describe objects in the real worlds, such as glasses. Finally, there are location designators describing a location such as “on the table”. The Prolog reasoning mechanism in CRAM allows to *resolve* the designator into a data structure. The mentioned Prolog reasoning mechanism is written in Lisp and is not to be confused with KnowRob [cra16]. However, it can execute KnowRob queries to resolve a designator. This makes the designator a means to describe abstract concepts translatable to data structures. [WBMB12]

Figure 2.3 shows another call that can be made from a CRAM plan: The plan can call a *process module*. In contrast to the existing plans, a process module is robot dependent. It is capable of executing actions on the robot. Therefore, process modules are called with action designators as parameters which describe the action to be executed. Different process module types exist for different types of actions: First of all, there are the *navigation process modules*. They are used to move the robot from one place to another. Then, there are *perception process modules* used to perceive objects in the environment of the robot. *Manipulation process modules* can be used to manipulate (e.g. grasp) objects. Finally, *move head process modules* move the head of the robot. Thus, the process module allows to abstract away the properties of single robots. [WBMB12]

For the robot, grasping actions are especially difficult since the robot must know where to grasp an object. To solve this problem, CRAM has the concept of *virtual handles*. Each object designator specifying an object to be grasped has a sub designator as property describing where an object can be grasped. When the robot is supposed to grasp the object, it resolves the designator and grasps the object at that point. [WBMB12]

To be able to reason about plans, they can be annotated with Prolog expressions. Again, these Prolog annotations are only available in the Lisp Prolog interpreter and not in KnowRob. The annotation is done by using special constructs from the CRAM Plan Language mentioned before. A plan is therefore considered as a function that does something with a goal. An example is a function that *achieves* that an *object* is *in* the *hand*. The Prolog annotation for this plan would then be *achieve (object-in-hand ?object)*. Other plans might *perform* an action designator. With the help of these annotation, the robot knows what it does. [WBMB12]

Listing 2.4 provides an example of a CRAM plan which is adapted for the examples in [Win13] and [Kaz16]. It will be used throughout the thesis.

```
(cpl:def-cram-function get-mug ()
  (cram-language-designator-support:with-designators
    ((handle-location :location `((:pose , (cl-transforms-stamped:make-pose
        (cl-transforms:make-3d-vector -0.005 0.0 0.0)
        (cl-transforms:euler->quaternion :ax (/ pi 2))))))
     (mug-handle :object `((:type :handle) (:at ,handle-location)))
     (mug-location :location
       `((:on "Cupboard")
         (:name "kitchen_sink_block")))
     (mug :object `((:type :mug) (:at , mug-location) (:handle ,mug-handle))))
    (cram-plan-library:achieve `(cram-plan-library:object-in-hand ,mug))))
```

Listing 2.4: Example CRAM Plan get-mug

The plan tells the robot to get a mug standing on a cupboard named "kitchen_sink_block" in its hand. The *with-designators* function is used to create three designators: The first one (named handle-location) describes the location of the virtual handle of the mug. The second one describes the handle itself. It uses the designator created before to specify where the handle is located. The third designator describes the location of the mug vaguely. Finally, there is the designator that describes the mug itself using the location designator and the object designator for the handle created before. Then, these designators are used to call a sub plan that achieves that the mug is in the hand of the robot. The called sub plan is actually defined inside the CRAM framework using the goal semantics described above:

```
(def-goal (achieve (object-in-hand ?obj))
; ....
)
```

This is the definition for the mentioned prolog expression *achieve (object-in-hand ?object)*.

2.3.2. Bullet Reasoning

For some designators, the robot is actually required to imagine its environment. For example, if it has to navigate to a pose where it can see an object. In this case, the robot has to make sure that the line of sight between camera and object is not blocked by another object. There is a CRAM extension that allows to do this job using the physics engine *Bullet*[1]. It is usable through Prolog predicates that do a physics simulation of the current world until the world is stable so that nothing changes. When location designators should be resolved, the component uses a generative model: It generates pose candidates and then uses the physics engine to verify them. [MB11]

[1] http://www.bulletphysics.org/

Figure 2.4 shows a visualization of a bullet world representing the kitchen at the IAI Lab in the University of Bremen. It is generated during the execution of the plan shown in Listing 2.4 (The initialization is not shown there). In this situation CRAM tries to resolve a designator that describes a pose sufficient to see a mug standing on a cupboard with the name “kitchen_sink_block”. The red sector shows the area in which pose candidates are generated.

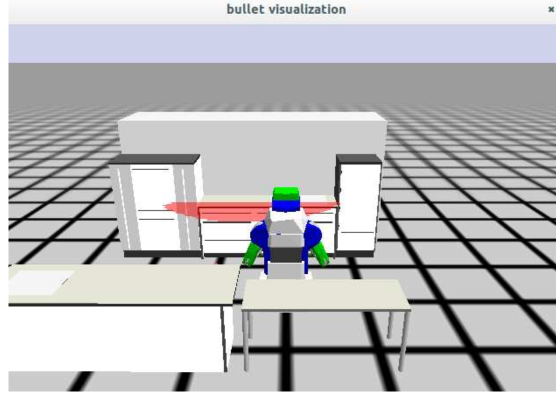


Figure 2.4.: The PR2 in the Bullet Visualization

Moreover, the simulation mechanism can do *temporal projection*: It can predict the effects of a plan over a longer period of time. This is implemented by providing own process modules. During the simulation, designator results are generated. Those designators for which a plan is successful are then stored and used in the actual plan. To improve the performance, only key points of a trajectory are simulated. This is a lack of precision but it is slightly faster than executing a complete simulation. Thus, CRAM has its own little simulation framework though it is not accurate. [MB13]

2.3.3. CRAM_M

To be able to reason about previous plan executions CRAM provides a logging mechanism called CRAM_M. During the plan execution, this mechanism records the executed plans in a task tree. This tree is similar to a stack trace in a debugger and contains executed CRAM functions as well as designators, time stamps, and failures. Figure 2.5 shows the components involved in the logging process: *semrec* is used to process symbolic information such as the task tree. It is able to export the task tree as OWL file. Continuous data such as poses as well as designators are processed by the component *mongodb_log*. It stores the information in the NoSQL database MongoDB, a database that stores its content in hierarchical documents. [WTBB14] [WB15]

Figure 2.6 shows a visualization of the log generated from the plan in Listing 2.4. Data from the OWL file and from the MongoDB are combined. On the left, tasks and designators are shown. There is the function *get-mug* itself and also the *with-designators*

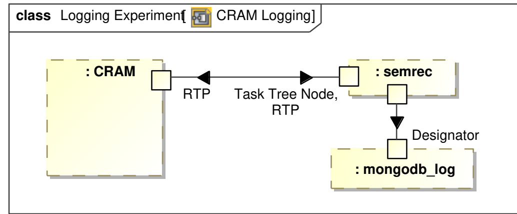


Figure 2.5.: Description of the Logging and Prediction Mechanism in CRAM

function as well as the *achieve* function. Furthermore, the transition to the data from the MongoDB is visible: All sub nodes of the *[MongoDB Designator] Document* node are stored in the MongoDB. This includes the whole designator structure. On the right, properties of the highlighted node (the call of the *achieve* function) are displayed. There, the mentioned Prolog representation of the plan appears. The parameter *?OBJ* is then displayed on the left (node *[?OBJ] Document*).

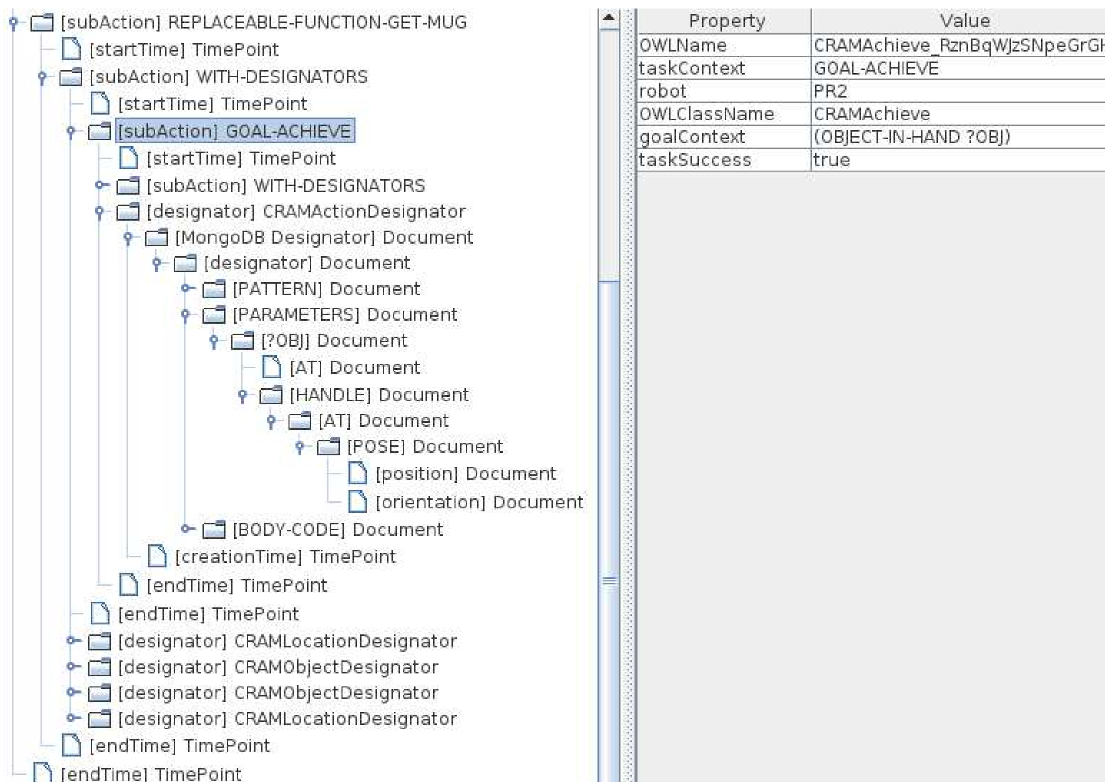


Figure 2.6.: Task Tree Generated from the Data Logged by CRAM

Since KnowRob can process OWL files, it can answer queries about the log files. To be able to also process the data in the MongoDB, computables are implemented. [WTBB14] mentions some example queries, such as the query for failed fetch tasks due to an object-not-found error or the query for the usual position of items in the refrigerator. Thus, it is possible to reason about previous experiments using Prolog queries.

[WB15] explains another function of the *semrec* component: It can be used to predict the result of a task in order to get the right task parameters using machine learning. The result can, for example, be an error or success. In this case, the parameters are not the designators but specially annotated parameters, called *RTP*s. An example of such a parameter might be a torque value or a location name. The parameters are annotated in the CRAM plans and after a learning phase, the best parameters are inferred automatically. Figure 2.5 depicts this: CRAM sends the RTPs to semrec and in another execution semrec sends parameters back to CRAM.

The reasoning is done using decision trees. On the one hand, the task trees of different plan executions are combined in a so called *condensed task tree* that share sub trees if they have an equal type in different execution. On the other hand, a decision tree is trained from the RTPs and from the task results. To get a success probability, semrec tracks the position of the current plan execution in the condensed task tree. Afterwards, the sub nodes in the condensed task tree are processed and the decision trees attached to the sub nodes are evaluated. Based on that, the probability for a result is calculated. Furthermore, the decision trees can be inverted to get parameter ranges. To summarize it: semrec allows to reason about parameters in logged executions specified previously during the plan design. [WB15]

2.4. Markov Logic Networks

A *MLN* (short form for *Markov Logic Network*) is an attempt to combine first-order logic with a probabilistic semantics in order to represent uncertainty [RD06]. Hence, they allow to represent knowledge by using formulas such as “All computer science students like pizza” as in first-order logic. Afterwards, queries can be answered. In first-order logic, one of these queries could be “Does Marc like pizza given that he is a computer science student” and the answer would be “yes” or “no”[2]. In contrast to first-order logic, MLNs assign a weight to the formulas that is relative to other formulas in the MLN. Moreover, the queries differ. The first-order query from above is translated to the MLN query “What is the probability that Marc likes pizza given that he is a computer science student”. Trough the weights it is possible that the resulting probability is between 0 and 100 percent.

A Markov Logic Network L is defined as a set of tuples (F_i, w_i). F_i is a first-order logic formula and $w_i \in \mathbb{R}$ is a weight for the formula [RD06]. The following sections explain MLNs in detail.

2.4.1. Formula Structure

Figure 2.7 shows the structure of a MLN formula. It is an adapted version of the grammar specified in [RN10] with the elements from [RD06]. The basic building block are atoms which are predicates applied to terms. An example is $ComputerScienceStudent(Marc)$ consisting of the predicate $ComputerScienceStudent$ and the constant $Marc$ which is a term. It is also possible to apply a function such as $FellowStudent(Marc)$ on a term. The return value of the function can then again be part of an atom. Thus, a construct like $ComputerScienceStudent(FellowStudent(Marc))$ is possible. However, due to the decidability, functions are only supported by MLNs if the function value is known in advance. [RD06]

More complex formulas such as $\forall s : ComputerScienceStudent(s) \wedge Likes(s, Pizza)$ can be constructed using the quantifiers $\forall$ and $\exists$ as well as the connectives $\Rightarrow$, $\wedge$, $\vee$ and $\Leftrightarrow$ and the *negation* $\neg$. The universal quantifier $\forall$ states that a formula is valid for all assignments of constants to a variable. The existential quantifier $\exists$ states that there is an assignment of a constant to a variable for which the formula is true. The logical connectives *and* ($\wedge$), *or* ($\vee$), *implication* ($\Rightarrow$) and *equivalence* ($\Leftrightarrow$) can be used to connect two formulas. To reduce the complexity of MLNs a typed logic is often used. This means that the arguments of predicates have a type. Thus, variables and constants used as arguments in atoms must have the type specified in the predicate declaration. The argument type of *ComputerScienceStudent* could for example be *human* with the domain *Marc*. In MLNs, unquantified formulas are implicitly assumed to be universal quantified. [RD06]

[2] Actually, the query is: Does “Marc likes pizza” hold given that “Marc is a computer science student and all computer science students like pizza” and the answer is true or false. For a better readability, it has been rewritten here.

$$
\begin{aligned}
Formula \rightarrow & Atom \\
& \mid (Formula\ Connective\ Formula) \\
& \mid Quantifier\ Variable, ... :\ Formula \\
& \mid \neg Formula \\
Literal \rightarrow & Atom \mid \neg Atom \\
Atom \rightarrow & Predicate(Term, ...) \\
Term \rightarrow & Function(Term, ...) \\
& \mid Constant \\
& \mid Variable \\
Connective \rightarrow & \Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow \\
Quantifier \rightarrow & \exists \mid \forall \\
Constant \rightarrow & String \\
Variable \rightarrow & string \\
Predicate \rightarrow & String \mid string \\
Function \rightarrow & String \mid string \\
\\
GroundFormula \rightarrow & GroundAtom \\
& \mid (GroundFormula\ Connective\ GroundFormula) \\
& \mid \neg GroundFormula \\
GroundLiteral \rightarrow & GroundAtom \mid \neg GroundAtom \\
GroundAtom \rightarrow & Predicate(Constant, ...)
\end{aligned}
$$

Figure 2.7.: Grammar for First-order Logic Formulas

Other important terms are *literals*, *ground formulas*, *ground literals* and *ground atoms*. *Literals* are atoms or their *negation*. *Ground atoms* are atoms containing only constants. As mentioned before, function values have to be known in advance and thus ground atoms do not contain functions. Similarly, *ground literals* are *ground atoms* and their negations. Since there are ground atoms and ground literals, it is obvious that there are also ground formulas which are formulas with atoms replaced by ground atoms. [RD06]

```
1  ComputerScienceStudent(human)
2  Likes(human, meal)
3  6.197  ComputerScienceStudent(s) ∧ Likes(s, Pizza)
4  4.000  ComputerScienceStudent(s) ∧ ¬Likes(s, Pizza)
5  9.794  ¬ComputerScienceStudent(s) ∧ Likes(s, Pizza)
6  9.794  ¬ComputerScienceStudent(s) ∧ ¬Likes(s, Pizza)
```

Listing 2.5: MLN for the Computer Science Student / Pizza Example

Listing 2.5 shows a MLN. The first two lines declare the predicates *ComputerScienceStudent* and *Likes* as well as their argument types *human* and *meal*. The formula in Line 3 has the weight 6.197 and it states that for all humans holds that they are computer science students and that they like pizza. In contrast to the first one, the formula in Line 4 states that for all humans holds that they are computer science students and that they do not like pizza. Similarly, the formulas in Line 5 states that all humans are no computer science students and that they like pizza. Finally, there is the formula in line six stating that all humans are no computer science students and that they do not like pizza. This depicts the usage of first-order logic formulas and implicitly universal quantified variables in MLNs.

2.4.2. Represented Probability Distribution

Listing 2.5 shows another interesting fact. On the one hand, all humans are computer science students. On the other hand, all humans are no computer science students. On the one hand, all humans like pizza. On the other hand, all humans do not like pizza. In first-order logic, this would be a contradiction in the knowledge base that is created by conjugating all formulas. Hence, the knowledge base would be useless. However, in MLNs these formulas make sense since a MLN defines a full joint probability distribution.

To calculate the exact full joint probability distribution it is necessary to create ground formulas and ground atoms from formulas, predicates and constants. Thus, the set of constants C has to be known when the joint probability distribution is calculated. Of course, C contains tuples of types and their domain values since a typed logic is usually used. As Figure 2.7 shows, ground atoms can be created from predicate declarations and constants by simply replacing the argument types by their domain elements. The creation of ground formulas is dependent on the quantifier type: Existentially quantified formulas are grounded by replacing them by disjunctions ($\vee$). The operands of the disjunctions are formed by replacing the existential quantified variable in the quantified formula by an element of the variable domain for each element in the variable domain. In contrast to this, universally quantified formulas are grounded by creating one ground formula for each element in the variable domain where the element replaces the variable. It might be necessary to apply these rules recursively to ground a MLN formula. [RD06]

In the example in Listing 2.5, the constants could be composed of $Marc, Svenja$ for the domain *human* and of $Pizza$ for the domain *meal*. Thus, the following ground atoms exist: $\{ComputerScienceStudent(Marc), Likes(Marc, Pizza), Likes(Svenja, Pizza), ComputerScienceStudent(Svenja)\}$. The formulas are all universally quantified using the variable s. Since the variable type of s is *human* and the domain of *human* consists of two constants, two ground formulas can be generated from each formula. For the first formula for example, the ground formulas are $ComputerScienceStudent(Marc) \wedge Likes(Marc, Pizza)$ and $ComputerScienceStudent(Svenja) \wedge Likes(Svenja, Pizza)$. If there was a formula $\exists s : ComputerScienceStudent(s)$, this formula would produce one ground formula: $ComputerScienceStudent(Marc) \vee ComputerScienceStudent(Svenja)$.

To make the grounding process and the MLNs in general work, the authors of [RD06] make three assumptions. First, it is assumed that different constants do not represent the same object. Then, there are no domain members that are not present in C, in a formula in L or in a function in L. Finally, as mentioned before, the value of each function has to be known in advance. However, the authors also name ways to get around of some of these restrictions.

Another important concept are worlds. A world is an assignment of the truth values *true* or *false* to each ground atom that can be created from L and C [RD06]. Table 2.1 shows the worlds that can be generated from Listing 2.5 using the constants mentioned before.

No.	ComputerScienceStudent(Marc)	Likes(Marc, Pizza)	ComputerScienceStudent(Svenja)	Likes(Svenja, Pizza)
1	false	false	false	false
2	false	false	false	true
3	false	false	true	false
4	false	false	true	true
5	false	true	false	false
6	false	true	false	true
7	false	true	true	false
8	false	true	true	true
9	true	false	false	false
10	true	false	false	true
11	true	false	true	false
12	true	false	true	true
13	true	true	false	false
14	true	true	false	true
15	true	true	true	false
16	true	true	true	true

Table 2.1.: Worlds for Listing 2.5

Now let L be an MLN with a set of m formula weight pairs (w_i, F_i), let $n_i(x)$ be the number of true groundings of the formula F_i in world x and let $\mathcal{X}_{L,C}$ be the set of possible worlds. Then the joint probability distribution in L grounded with the constants C is defined in [RD06] as follows:

$$P(X = x|L, C) = \frac{1}{Z} \exp\left(\sum_{i=1}^{m} w_i n_i(x)\right) = \frac{\exp\left(\sum_{i=1}^{m} w_i n_i(x)\right)}{\sum_{x \in \mathcal{X}_{L,C}} \exp\left(\sum_{i=1}^{m} w_i n_i(x)\right)} \tag{2.1}$$

If L is the MLN shown in Listing 2.5 the probability for the world no. 13 from Table 2.1 given the constants C from above would be:

$$\begin{aligned} P(X = \text{World no. } 13|L, C) &= \frac{\exp(1 \cdot 6.197 + 0 \cdot 4 + 0 \cdot 9.794 + 1 \cdot 9.794)}{Z} \\ &\approx \frac{8806494}{1324785576} \\ &\approx 66 \cdot 10^{-9} \end{aligned}$$

The joint probability distribution can then be used in queries as discussed in Section 2.4.4.

Apart from the formulas, the weights are the most important factor influencing the probability distribution. For some simple cases the weight of a formula F can be interpreted as $w = \ln\left(\frac{P(X=x_t)}{P(X=x_f)}\right)$ where x_t is a world where F is true and x_f is a world where F is not true while other things do not change. If the formulas are mutually exclusive and if possible exhaustive, the weights w_i can be set to $\ln(p_i)$ where p_i is the probability for the formula. A set of formulas is mutually exclusive if in one world maximally one formula is true. It is exhaustive if all combinations are modelled. Section 4.3 will analyze this case more detailed. If the weights of all formulas tend to infinity, the formulas in the MLN form a first-order knowledge base. This means that all worlds in which all formulas in the knowledge base are true are equally probable, whereas all other worlds have zero probability. However, in general there is no easy interpretation of the weights that is applicable to every case. [RD06] [Jai12].

2.4.3. MLNs and Markov Random Fields

The formula for the joint probability distribution has its origin in *Markov Random Fields* (*MRF*s). They are means to represent full joint probability distributions. Actually, a MRF is created when a MLN is grounded. [RD06]

Let $X = (X_2, X_2, ..., X_n)$ be random variables and $\mathcal{X}$ be the domain of X. A MRF consists of an undirected graph $G = (X, E)$ and a set of potential functions $\phi_k : X_{\{k\}} \rightarrow \mathbb{R}_{0+}$ where $X_{\{k\}}$ is the set of variable assignments in the kth clique of the graph G. A clique is a set of nodes that are all connected among each other. Let $x_{\{k\}}$ be the part of x representing the variable assignment of the kth clique of G. The joint probability distribution in a MRF calculates as follows:

$$P(X = x) = \frac{1}{Z} \prod_k \phi_k(x_{\{k\}}) = \frac{\prod_k \phi_k(x_{\{k\}})}{\sum_{x \in \mathcal{X}} \prod_k \phi_k(x_{\{k\}})} \tag{2.2}$$

In MLNs, the MRFs are represented as log linear models:

$$P(X = x) = \frac{1}{Z} \exp\left(\sum_j w_j f_j(x)\right) = \frac{\exp\left(\sum_j w_j f_j(x)\right)}{\sum_{x \in \mathcal{X}} \exp\left(\sum_j w_j f_j(x)\right)} \tag{2.3}$$

The $w_j \in \mathbb{R}$ are weights and the $f_j : \mathcal{X} \rightarrow \mathbb{R}$ are features. The clique potentials can be translated to the log linear form: For each variable assignment $x'_{\{k\}}$ of each clique k, a feature $f_i(x) = \begin{cases} 1 & \text{if } x = x' \\ 0 & \text{otherwise} \end{cases}$ and a weight $w_i = \ln(\phi_i(x_{\{k\}}))$ are created. This explains the structure of MRFs. [RD06]

Given some constants C, a MLN L and can be grounded to a MRF $M_{L,C}$ using the following procedure: First, a node is created for each ground atom. An edge between two nodes is created if they appear in the same ground formula. Figure 2.8 shows the MRF that is grounded from the MLN in Listing 2.5 and from the constants in Section 2.4.2. If the log linear representation is used, a feature is created for each ground formula which is 1 if the formula is *true* and 0 otherwise. The weight assigned to the feature is the formula weight. Of course, it would also be possible to create one feature per formula which returns the number of true groundings, as in Equation 2.1. Thus, a MLN is a template that can be used to generate MRFs. [RD06]

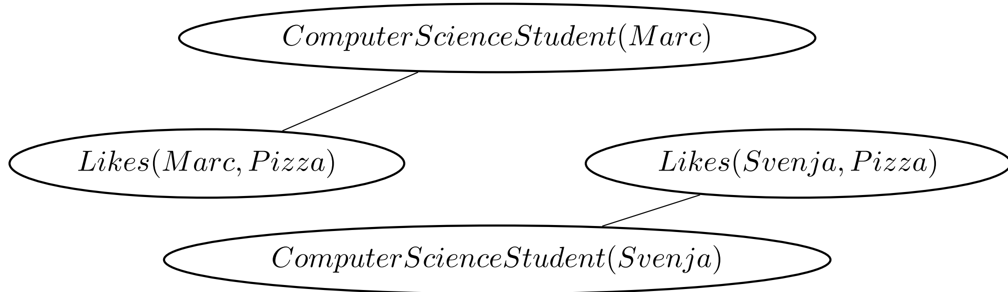


Figure 2.8.: MRF Generated from the MLN in Listing 2.5

2.4.4. Inference Algorithms

Though the full joint probability distribution is known, one might ask how to answer more complicated queries like "What is the probability that Marc likes pizza given that he is a computer science student". Since MLNs can be grounded to MRFs, there are different inference algorithms that can answer such questions. Moreover, *lifted* inferences might be possible such that the grounding process is not necessary. A small subset of algorithms is presented here.

Exact Inference

As mentioned before, the joint probability distribution can be used to answer a *query* Q given an *evidence* E as well as constants C and a MLN L. [RD06] points that out: Let Q be a first-order formula describing the query and let E be a first-order formula describing the evidence. Moreover, let $\mathcal{X}_{Q,L,C}$ be the worlds where Q is true and let $\mathcal{X}_{E,L,C}$ be the worlds where E is true. Then

$$P(Q|E,L,C) = \frac{P(Q \wedge E|L,C)}{P(E|L,C)} = \frac{\sum\limits_{x \in \mathcal{X}_{Q,L,C} \cap \mathcal{X}_{E,L,C}} P(X = x|L,C)}{\sum\limits_{x \in \mathcal{X}_{E,L,C}} P(X = x|L,C)} \tag{2.4}$$

With the joint probability distribution from Equation 2.1 this formula becomes

$$P(Q|E,L,C) = \frac{\sum\limits_{x \in \mathcal{X}_{Q,L,C} \cap \mathcal{X}_{E,L,C}} \exp\left(\sum\limits_{i=1}^{m} w_i n_i(x)\right)}{\sum\limits_{x \in \mathcal{X}_{E,L,C}} \exp\left(\sum\limits_{i=1}^{m} w_i n_i(x)\right)} \tag{2.5}$$

In the MLN in Listing 2.5, the question "What is the probability that Marc likes pizza given that he is a computer science student and given the constants from Section 2.4.2" can be answered using this formula. The worlds where both the formula $Likes(Marc, Pizza)$ and the formula $ComputerScienceStudent(Marc)$ are true are the worlds no. 13 to 16 from Table 2.1. The worlds where the formula $ComputerScienceStudent(Marc)$ is true are the worlds 9 to 16 from Table 2.1.

$$\begin{aligned}
&P(Likes(Marc, Pizza)|ComputerScienceStudent(Marc), L, C) \\
&= \frac{\sum\limits_{x \in \{\text{World no. } 13,14,15,16\}} \exp\left(\sum\limits_{i=1}^{m} w_i n_i(x)\right)}{\sum\limits_{x \in \{\text{World no. } 9,10,11,12,13,14,15,16\}} \exp\left(\sum\limits_{i=1}^{m} w_i n_i(x)\right)} \\
&\approx \frac{17881160}{19868402} \\
&\approx 0.9
\end{aligned}$$

Thus, there is a 90% probability that Marc likes pizza.

Equation 2.4 can easily be implemented. Unfortunately, this kind of inference is $P\#$ complete and thus unusable in larger domains [RD06]. Therefore, a brief introduction of other inference methods follows.

Gibbs Sampling

A solution for those cases where exact inference is infeasible due to its complexity are approximate inference methods. One class of approximate methods are *Markov Chain Monte Carlo (MCMC)* methods. These methods work by drawing state samples from a *Markov chain*. [Jai12]

The Markov chain defines a distribution $\pi_t : \mathcal{X} \to [0, 1]$ assigning a probability to each state (in this case: world) for each time step t. This is done by providing an initial distribution π_0 and a transition model $\mathcal{T}(x \to x') = P(x'|x)$ providing the next state x' given the current state x. For cases where $t > 0$ holds, π_t is defined as $\pi_t(x) = \sum\limits_{x' \in \mathcal{X}} \pi_{t-1}(x') \cdot \mathcal{T}(x' \to x)$. If no further transitions are possible, i.e. if $\pi_t = \pi_{t-1}$, the stationary distribution is reached. Important properties of Markov chains for MCMC

algorithms are *ergodicity*, which causes the chain to finally reach the stationary distribution, and being in *detailed balance with a distribution* π, causing π being the Markov chains stationary distribution. [Jai12]

In the MCMC algorithms a Markov chain with the stationary distribution $P(x|E,L,C)$ is used to calculate $P(Q|E,L,C)$. In the beginning, a state is sampled from the initial distribution. That means that the start state x_0 is randomly drawn from the set of state whereas with probability $\pi_0(x')$ holds $x_0 = x'$ for all $x_0 \in \mathcal{X}$. Then, $n-1$ other states x_i are sampled from the distribution $\pi_i(x_{i-1})$. If a state x_i satisfies Q a counter c starting at 0 is increased. The probability is then $P(Q|E,L,C) = \frac{c}{n}$. [Jai12]

[RD06] show how to apply *Gibbs sampling* to MLNs. First, the grounded MRF is reduced: Starting at the ground atoms in the query all nodes are expanded except if they are evidence nodes. Nodes that are not expanded do not need to be considered. The parts that vary in the different MCMC algorithm are the initial distribution and the calculation of the transition model. In the *Gibbs sampling* algorithms, the initial state is sampled using the *MaxWalkSat* SAT solver. The successor states are calculated by sampling a new ground atom X_l given its Markov blanket B_l formed by all nodes directly connected in the MRF by using the following probability:

$$P(X_l = x_l | B_l = b_l) = \frac{\exp(\sum_{f_i \in F_l} w_i f_i(X_l = x_l, B_l = b_l))}{\exp(\sum_{f_i \in F_l} w_i f_i(X_l = 0, B_l = b_l)) + \exp(\sum_{f_i \in F_l} w_i f_i(X_l = 1, B_l = b_l))} \tag{2.6}$$

In this case, F_l is a set of ground formulas containing X_l. $f_i(X_l = x_l, B_l = b_l)$ is 1 if the ground formula f_l is true in a state where X_l has the value x_l and B_l has the value b_l and 0 otherwise. This makes Gibbs sampling a random experiment to determine a probability.

MC-SAT

Gibbs sampling has one disadvantage: If the weights get very large, the Markov chain is not ergodic. This is not the case in another MCMC algorithm called *MC-SAT*. It uses a SAT to sample successor states. A SAT solver is a tool able to determine a world given logic formulas. [PD06]

Before the actual algorithm starts, all formulas having a negative weight are converted to formulas with positive weight by negating the formula and multiplying the weight with -1. Those formulas which have an infinite weight are called *hard*. The initial state is sampled by sampling a world where all hard formulas are true. The next samples are generated by creating a set M where formulas f_k which were true in the last assignment are added to with probability $1-\exp(-w_k)$. Then, the next world is created by sampling from the uniform distribution of the worlds which are true in M. This can be done using the SampleSAT algorithm. [PD06]

Since the Markov chain generated by MC-SAT is in theory ergodic, it is an improvement compared to Gibbs sampling. However, the samples that SampleSAT generates are only nearly uniformly distributed. Moreover, the search for a solution for the SAT problem takes time. Therefore, the algorithm is also imperfect. [PD06]

WCSP Inference

Sometimes, a probability distribution is not necessary. Thus, it is enough to get the most probable world. This kind of inference is called *maximum a posteriori* (*MAP*) inference. A MAP inference calculates

$$\arg\max_{x} P(X = x|L, C) \tag{2.7}$$

If evidence should be considered, it can be added to the MLN as a hard formula. A method to perform a MAP inference is to convert the ground MRF to a *Weighted Constraint Satisfaction Problem* (*WCSP*). This problem can then be solved using a WCSP solver. [JMW09].

A WCSP is defined as $\mathcal{R} = (Y, D, C)$ where $Y = \{Y_1, ..., Y_n\}$ contains variables, $D = \{\text{dom}(Y_1), .., \text{dom}(Y_n)\}$ contains the domains of the variables and $C = \{c_1, ..., c_r\}$ is a set of constraints. Each constraint c_i is a function from a set of variables to a cost value $v \in \mathbb{N}$. The WCSP can be solved by finding a consistent variable assignment for all variables which minimizes $\sum_{i=1}^{r} c_i(y)$. [JMW09].

To convert a ground MRF to a WCSP, the formula weights are first converted to positive numbers as explained above. Then, the weights have to be scaled to natural numbers since costs are usually natural numbers. Afterwards, mutually exclusive and exhaustive ground atoms are mapped to one variable. The other ground atoms are mapped to an own variable. Finally, one constraint per ground formula is defined: It returns w_i if the formula is false and 0 otherwise. After solving the WCSP problem, the variable assignment that is the solution can be transferred back to the MRF. [JMW09].

In summary, different inference algorithms for different use cases are available.

2.4.5. Learning Methods

In some cases, one wants to automatically create MLNs from given training data. This can be done with machine learning techniques. Since formulas and weights can be learned independently, there is a difference between *structure learning* and *parameter learning*. For this work, only parameter learning is needed. Parameter learning requires training data in form of a database x which is essentially a world. Then, the learner tries to determine weights w maximizing the probability $P(w|x, L, C)$. [Jai12]

Finding appropriate weights can be done using different techniques. The first is the *Maximum A Posteriori* method. It uses the Bayes' theorem to rearrange the desired formula for the desired weight $w^* = \arg\max_w P(w|x)$.

$$w^* = \arg\max_w P(w|x) = \arg\max \frac{P(x|w) \cdot P(w)}{P(x)} = \arg\max_w P(x|w) \cdot P(w) \tag{2.8}$$

$P(x|w)$ is an abbreviation for $P(X = x|L_w, C)$ and can thus be calculated using Equation 2.1. Since the exact weight distribution $P(w)$ is not known, a Gaussian distribution is assumed. Another option is to leave out the $P(w)$ resulting in

$$w^* = \arg\max_w P(x|w) \tag{2.9}$$

This is called the *Maximum Likelihood* method. If multiple databases $x^{(i)}$ should be used for learning, they can be assumed to be independent and thus the weight can be learned using

$$w^* = \arg\max_w \prod_{i=1}^{n} P(x^{(i)}|w) \tag{2.10}$$

Compared to the maximum a posteriori method, the maximum likelihood method is more vulnerable for overfitting which results in weights that only model the training data and that do not generalize. However, it is necessary to determine the parameters for the Gaussian distribution. [Jai12]

Instead of computing Equation 2.9, it is also possible to use the logarithmized version since it has the same maximum:

$$\arg\max_w \ln P(x|w) \tag{2.11}$$

Unfortunately, computing Equation 2.9 or Equation 2.11 is P# complete. A workaround is to use more approximation and maximize the pseudo likelihood:

$$P_w^*(X = x) = \prod_{l=1}^{n} P_w(X_l = x_l|MB_x(X_l)) \tag{2.12}$$

The definition of $P_w(X_l = x_l|MB_x(X_l))$ is given in Equation 2.6. A drawback of this solution is that it is only an approximation and thus it might not work in some cases. The actual optimization is then done using an optimization algorithm such as L-BFGS which employs the gradient of the pseudo log likelihood. [Jai12] [RD06]

In summary, learning MLN weights from example worlds is possible but might be computationally expensive.

2.4.6. Common Pitfalls

[Jai12] names some important things that have to be considered during the design of MLN formulas. The perhaps most important one is that implications in formulas do not model conditional probabilities. If an implication $A \Rightarrow B$ with A and B being formulas is used as MLN formula, the probabilities $P(A)$ and $P(B)$ are modified by this formula. Instead of using implications, the author proposes to replace the implication by conjunctions. To model a conditional probability, the truth table of A and B has to be represented in the formulas. This is the reason why the MLN in Listing 2.5 contains four conjunctions instead of one implication $ComputerScienceStudent(s) \Rightarrow Likes(s, Pizza)$.

Another possible problem that [Jai12] mentions is that the so called *shallow transfer* might not work in any case. Shallow transfer means that the MLN and thus the probability distribution is usable with domains of different size. Some formulas such as implications having a hard weight might unintentionally change the probabilities of single ground literals. A proposed solution is to extend MLNs with special constraints fixing these probabilities. There is another issue when the domain size changes: Cardinality restrictions (e.g. a tandem bicycle has maximally two drivers) might not be preserved. To solve these restrictions, [Jai12] proposes AMLNs which are able to construct MLNs dependent on the attributes of the domain. Finally, for a given domain there might be more than one weight vector maximizing the likelihood. However, if the domain is then enlarged, the wrong probability is the result. Thus, learning in MLNs can be difficult.

In summary, compared to first-order knowledge bases, additional thoughts have to be considered during the MLN design.

2.4.7. pracmln

There are frameworks providing learning and inference methods for MLNs. One of them is *pracmln*. It is a fork of the ProbCog framework. Among others, it provides the inference and learning algorithms in Section 2.4.4 and Section 2.4.5. There is a python API for accessing pracmln programmatically. [Nyg16]

pracmln provides different grammars for creating MLNs. The *StandardGrammar* is the one presented in Figure 2.7. Variables are identified by having lower case letters in this grammar. Apart from the StandardGrammar, there is the PRACGrammar. The most important difference to the StandardGrammar is that variables are represented by a question mark in front of the variable.

Apart from that, pracmln offers different extensions. One example are *soft functional constraints*: Predicate declarations can contain predicate types ending with a question mark. In this case, worlds in which two true ground atoms of that predicate with the same arguments except from the specified one exist are not considered during the inference. An example is a predicate *bikeRider(bikeName, human?)*: A world in which *bikeRider(MarcsRoadBike, Marc)* and *bikeRider(MarcsRoadBike, Svenja)* are true is not

considered during inference and learning. Additionally, there are *functional constraints* which are applied by using an exclamation mark instead of a question mark: They additionally exclude worlds where no ground atom is true given some arguments. To realize functional and soft functional constraints, pracmln uses variables which are not to be confused with the variables in first-order logic. Without the constraints, every ground atom gets its own variable which might be true or false. For functional or soft functional constraints, there are special variables for each combination of non (soft) function parameters whose values are specified by the domain of the constrained argument.

Another extension are Fuzzy-MLNs. They allow predicates whose ground atoms are exclusively appearing in the evidence to be declared as fuzzy predicates. Afterwards, the ground atoms of these predicates can have a truth value between 0 and 1 instead of just being true or false. During the inference, the formulas are evaluated using fuzzy logic replacing the conjunction by the minimum and the disjunction by the maximum. In Equation 2.1, the weights have to be multiplied by the number of true ground formulas. FuzzyMLNs multiply the weights with the fuzzy truth value of each ground formula instead. The fuzzy ground atoms are intended to represent similarities between concepts in description logics. This allows to use description logics which can, for example, be used in word sense disambiguation. [NB15]

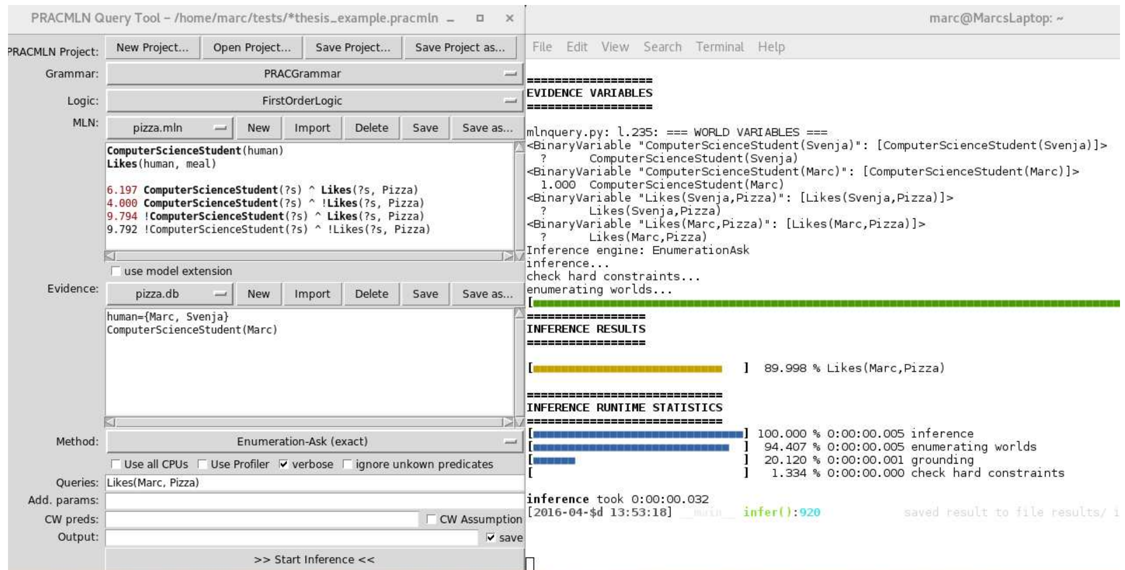


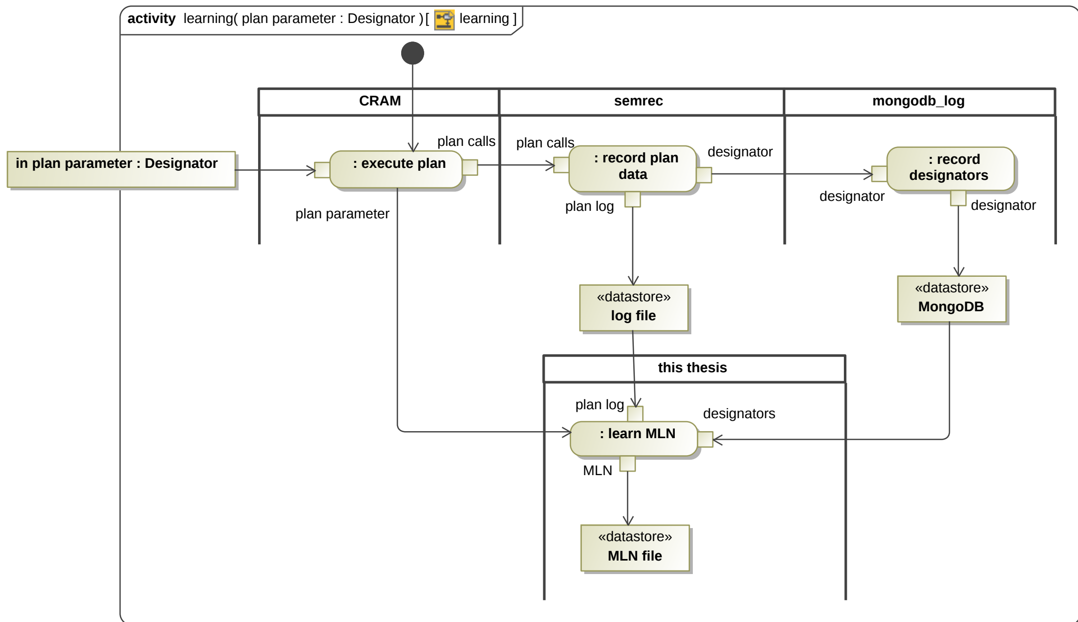
Figure 2.9.: The MLN from Listing 2.5 in pracmln

Figure 2.9 shows a query asking whether Marc likes Pizza given that he is a computer science student and given the domain of human in pracmln. On the right side, the variables are visible. The inference algorithm used is the algorithm shown in Equation 2.4 and the resulting probability is 89.998 which is the probability calculated in Section 2.4.4. All in all, *Markov Logic Network* combine first-order logic and probability theory and the framework pracmln implements inference and learning algorithms.

2.5. Usage in this Thesis

Figure 2.10 shows how the software frameworks presented before are used in this thesis. The executed plans are actually *CRAM* plans. Hence, the activities displayed in Figure 2.10 are CRAM plans, too. Moreover, plan parameters are the *Designators*. Though these plans are not necessarily executed on a *PR2* robot, the PR2 is the robot used as example. *Semrec* and *mongodb_log* from Figure 2.5 are used to record the plan logs. In combination with the inference algorithm developed in this thesis, *pracmln* is used to perform the MLN inference. The software implemented in Chapter 6 is used for the MLN creation as well as for converting designators and plan calls to queries and vice versa.

For the use cases in Figure 1.1, this means that it shall be possible to perform inferences on designators and tasks. More precisely, it must be possible to infer success probabilities of CRAM plans as well as information about objects which is located in the designators. A parameter of an incompletely parametrized CRAM plan can be a second designator, but it is more likely a property of an existing designator. If, for example, the robot should execute a *object-in-hand* plan for a mug, the software must infer that the *at* property of the *mug* designator is needed and it must infer a value for this property. Thus, it is also necessary to complete designators.



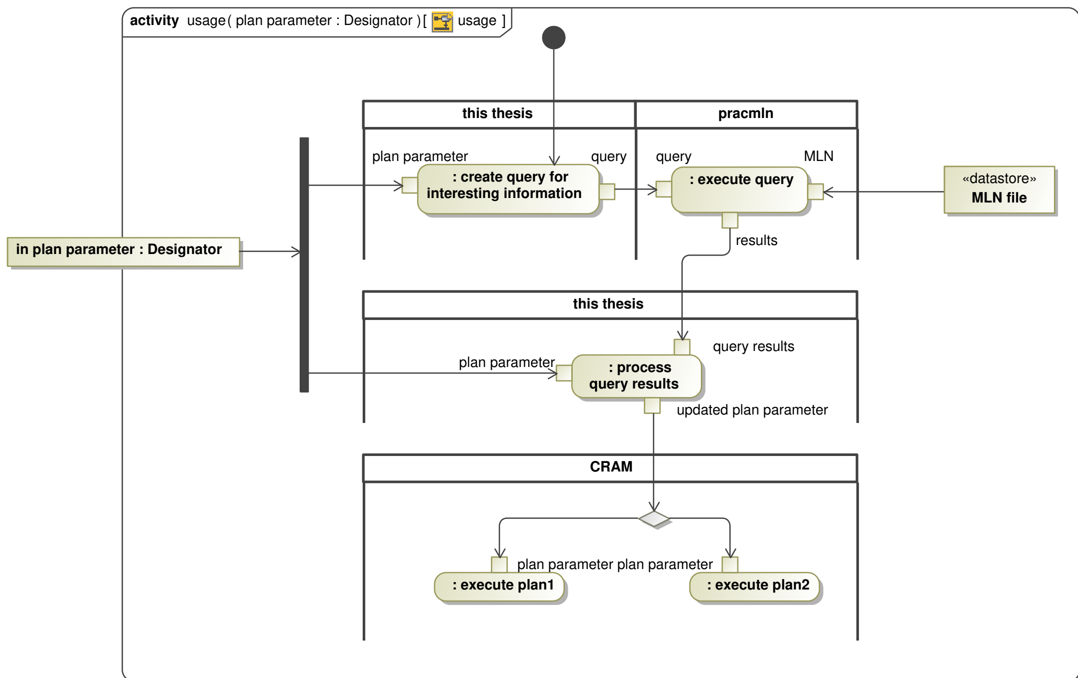


Figure 2.10.: Assignment of Existing Software to the Thesis Concept

3. Related Work

For the different contributions of this thesis there are different related publications.

3.1. Learning Models for Designators from Experience

Learning from logged data has already been investigated by the AG AI of the University of Bremen in different approaches. First of all, as mentioned in Section 2.3.3, [WTBB14] describes how the logged data can be used for reasoning in a Prolog knowledge base. In theory, this knowledge base could also be queried for the probabilities calculated in this work. However, there are some downsides: First of all, the Prolog knowledge base needs the whole logs for reasoning while the approach presented here compresses these data. Then, the probability calculation would have to be part of the query. This is difficult since it would require a similarity measure for designators to achieve generalization. Finally, the approach in this work is able to combine different experiments in one MLN. On the other side, one does not have to worry that much about the effects of computational complexity when querying the Prolog knowledge base.

Another approach, which is also mentioned in Section 2.3.3, is presented in [WB15]: As in this work, tasks that occur more than once are aggregated in the memory representation. Furthermore, reasoning about success probabilities and calculating the best plan parameters using decision trees is possible. But there are some differences: The parameters used for the decision trees are conceptually on a different level than the parameters in this work. They are marked in the plan using a special plan construct. On the one hand, this allows to concentrate on parameters which are really important. On the other hand, the parameters used in this work are also available if they are not explicitly tagged[1] and thus parameters from very different plans can be used for reasoning. Moreover, this work allows to reason about the parameters themselves. For instance if a parameter specifies a telephone, it is possible to infer that telephones usually have a location and to infer the most probable place. Another difference is that the decision tree approach in [WB15] provides a better generalization for numeric values and that it is probably faster.

[Jai12] describes a Bayesian Logic Network (a concept similar to MLNs) for inferences in a pick an place scenario where objects are picked from six different locations. It allows to infer different probabilities such as the best hand to use, the field name or the

[1] Of course, it is necessary to use the correct functions to make the parameters appear in the logs, but they are not explicitly tagged as parameters for reasoning.

outcome. However, it models designator properties such as field name or the hand to use as predicates. Thus, the network design has to be changed for different plans. Moreover, it is not possible to complete the structure of a designator. The approach in this work does not model designator properties as predicates and is thus able to do this kind of reasoning.

[WB15] mentions different approaches that have been developed in the past to give robots the opportunity to learn from experiences. Three of them are especially relevant here. The first one is [Moo90]. It describes how to use nearest neighbor learning of models to infer an action given a robot state and a desired behavior or to infer a behavior given an action and a robot state. While the model learned in this work is close to a nearest neighbor method, the scope is another: This work learns about actions and their parameters and the parameters are mostly symbolic rather than n dimensional vectors of real numbers.

[TM95] provides a definition of the term "lifelong learning" in the context of reinforcement learning: It is stated that "for each different environment and each reward function the robot agent must find a different control policy". Moreover, the principle of knowledge transfer is shown in different scenarios where neural networks are combined. The networks perform different tasks such as predicting the next robot state given a state and an action, modeling the evaluation function used in reinforcement learning or modeling collision reward or sensor reliability given sensor data. As the title states, the lifelong learning connects [TM95] with this work. However, there are large differences: In [TM95] reinforcement learning is used while in this work a statistical relational model is learned. Moreover, this work learns models of designators representing abstract concepts while in [TM95] concrete inputs and outputs such as sensor values or states are used.

For RPL, a predecessor of the CRAM planning language, [Kir09] shows how learning from experiences can be integrated into the planning language with minimal changes to the plan itself. This is done by mapping the plan to a hybrid automation to define experiences. They can then be abstracted and the abstracted experiences are available for learning algorithms. Special language extensions allow the use of leaned models. In this work, the experiences are already present in form of logs. Therefore, they only need to be integrated into the plan and thus no such framework is used for the integration of the learning itself.

Another framework that incorporates learning is [RNZ+13]: Knowledge is represented using OWL ontologies. Learning in this framework includes an approach called *Constraint Generalization*. It is used to learn models for object categorization. Moreover, the paper mentions a method for learning how to perform a task such as serving a dish from examples which described in detail in [NHRL14]. This is similar to the plan parameter completion learning in this work. Anyhow, they describe the single parts of a task while the focus of this work lies in inferring parameters for an existing plan.

In general, there are different examples for learning from experience. In [HPK+12] experiences are used in a distance metric matching perceived object descriptors to object

descriptors from a database in order to find appropriate grasp parameters. In [HLRB13] support vector regression is used to learn a function from experience that returns a pushing score given an object shape. This score can then be used to predict how good a robot can push an object from a location.

One relation that is represented in the MLNs created in this work is the relation between objects having certain properties and actions. This relation is similar to *affordances* as introduced by [Gib14]: An affordance describes what something or someone having certain properties is good for. An example is a mug: As an object having a handle it is *graspable* for a human (and for some robots). Since the concept of affordances is also used in robotics, there exist different approaches for learning affordances. [PG15] for example presents a solution where robots learn which combination of objects affords which action in a dialogue with a human. The connection between objects and actions is modeled with an m-estimate. Apart from the different representation, the data are collected by interacting with a human and the object properties are not considered in contrast to this work. In [WHB14] a neural network is used to predict the effect of an action given an object state and the action itself. There is an exploration phase guided by reinforcement learning in which the neural network is trained. However, the approach is different to this work since it focusses on object poses and numeric values for the actions rather than on symbolic descriptions. Additionally, only the outcome can be predicted.

[MLBSV07] shows how to learn Bayesian networks representing affordances from data collected during an experimentation phase: A variable in the Bayesian network is created for each object feature (e.g. color, shape or size). Moreover, there is one variable for the action (e.g. grasp) and one for each effect caused by the action (e.g. a velocity). To be able to use continuous variables, the values are clustered. A similar approach is presented in [SHKK10] where affordance models for grasping are learned from data collected by a GraspIt![2] simulation and labelled manually. In this approach, the effect is not modeled but there are additional variables for action parameters (for the GraspIt! framework) as well as for environment information (e.g. the free space). Moreover, the structure of the Bayesian network is modeled manually in this approach. Both [MLBSV07] and [SHKK10] are similar to the approach presented in this work since probabilities about objects and actions can be calculated.

In spite of that, there are differences: The affordances are not exactly comparable to the designators since designators can describe objects as well as actions or locations while there are only objects along with their affordances in the Bayesian networks. Furthermore, the MLNs do not contain effects. Then, the object features are different: The Bayesian network approach assumes them to be equal for all objects whereas this work allows different objects to have different properties. Apart from that, this work allows reasoning about the designator structure itself such as inferring additional properties which is difficult to realize with the Bayesian network approach. The reasoning about multiple objects is also impossible using the Bayesian approach. Thus, there are some different use cases for this work and the Bayesian network approaches.

[2] `http://graspit-simulator.github.io/`

However, there are different extensions to the approach presented in [MLBSV07]. [MMO+12] uses the *ProbLog* language to learn a probabilistic knowledge base for multi object affordances from experience. ProbLog is, similar to MLNs, a means for statistical relational learning. It is an extension of the Prolog programing language. First, a Bayesian network is learned as in [MLBSV07] which is then converted to a ProbLog knowledge base. Due to additional, manually modeled formulas it is possible to reason about multiple objects. This approach is extended by [MDR14a]. There, *Distributional Clauses* are used which are in turn an extension of ProbLog allowing continuous variables. That allows learning a model for one-armed object manipulation as Bayesian network, converting it to Distributional Clauses, adding some more formulas and using the knowledge for two-armed object manipulation. Distributional clauses are also used by [MDR14b] to find objects hidden behind other objects.

These approaches are quite similar to this work, but there are still differences. First of all, there is the difference between object affordances and designators mentioned above. The approaches presented above focus on object properties relevant for object classification and thus all objects have the same property types such as an object weight. In contrast to this, the designators consist of arbitrary key-value pairs where values can be strings or designators. This is considered in this work. Then, there is the different modeling language. The Distributional Clauses allow continuous variables such as an object size, while this work can only handle symbolic variables. Thus, continuous values are treated as symbols. Another difference is that this work focusses on lifelong learning and thus the model can be updated incrementally which is not considered by the other approaches. So there are some differences to this approach primarily due to the use cases.

More similar concerning the model created are approaches using MLNs. [ZFFF14] explains how a MLN representing object attributes, visual attributes of objects and object affordances is created. The information used to train the MLN is extracted from the web, from databases as WordNet and from images. The first difference to this work is the information source: This work uses the experience of the robot as source. Additionally, the MLN is different: In this work, object properties are not represented by predicates. Thus, it is possible to infer which properties are present without knowing the possible properties.

[AMPSQ14] presents another MLN based approach. They convert a *semantic knowledge base* in form of an OWL ontology to a MLN. The MLN is used in a robot task planner, for example to infer an object location. Though the CRAM log files are OWL files, this approach is not applicable here since the objects are not part of the log file. Moreover, the approach represents relations between objects, such as the object location, as predicates. The predicates are also generated from the semantic knowledge base. In contrast to that, this work uses a fixed set of predicates which allows to query for the relations themselves.

To summarize, various publications treat learning from experience. There are some approaches having similar use case as this work and some using the same probabilistic representation. However, to the best of my knowledge no work is combining exactly the use cases, the modeling language and the designators that this work combines.

3.2. Improved Exact MLN Inference

There is also some related work for the inference algorithm developed in this thesis. The probably first improvement of exact inference algorithms is presented in the paper introducing MLNs: [RD06]. Instead of reasoning over the full grounded MRF, only those variables between query and evidence are considered. The approach in this work also reduces the search space. However, it removes different and, in some cases, more variables. On the other hand, the approach presented in this work is not applicable to all kinds of formulas compared to the approach presented in [RD06].

Especially in the field of *lifted inference*, there are algorithms speeding up the exact inference by omitting the grounding process. An important algorithm is the *FOVE* algorithm [DSBAR05]. It tries to eliminate ground atoms in some cases by replacing them. There are several extensions such as [MZK+08]. The process described there is especially useful if the individual ground atom is unimportant for calculating a probability. In contrast to that, the approach in this work does not try to avoid the grounding process. Instead, it tries to generate the all the worlds having a larger probability from the formulas. It is thus suited if the number worlds in which a formula is true is much smaller than the total number of worlds.

Similar to the inference algorithm developed in this work [JGMS10] uses the logical formulas in the inference. This is accomplished by converting the MLNs to another formalism called *counting programs*. In the counting programs, simplification rules are applied until the program is so far decomposed into other counting programs that the inference task is easy to solve. Although both approaches perform inference on the logical formulas, decomposing the formulas is the opposite of the technique used in this work. Here, the formulas are composed to generate worlds. Again, a conceptual difference is that the algorithm in this work is not lifted.

The formula structure is used in the algorithm shown in the lifted inference algorithm in [GD10]. They create a so called *AND/OR* tree describing the variable assignment in clauses which are disjunctions of literals. By introducing special nodes, repeated structures are combined. To be able to exploit clauses, the formulas are assumed to be in conjunctive normal form while the algorithm developed here assumes formulas to be conjunctions. Furthermore, the tree in the algorithm developed in this work represents the combination of formulas rather than the assignment of variables.

Another algorithm using the formula structure is the *FROG* algorithm presented in [SN09]. FROG reduces the number of groundings by not considering those which are definitely true given the evidence. Instead, only the number of those groundings is stored to use it later. Again, in contrast to this algorithm assuming formulas to be conjunctions, FROG assumes the formulas to be a set of clauses. The FROG algorithm yields especially high performance if the closed world assumption is used and there are negated literals in the clauses. In contrast to that, the closed world assumption is rather unimportant in this work while it is more important that there are few true ground formulas.

Similar to the algorithm in this work, the *probabilistic theorem proving* algorithm presented in [GD16] uses the number of worlds in which a formula is true in the inference. It also exploits the formula structure. More specifically, the authors provide a lifted version of the DPLL algorithm able to cope with formula weights. However, the algorithm works differently than the algorithm in this work. It can cope with formulas in CNF while the algorithm in this work works with formulas which are conjunctions. Furthermore, it decomposes the formulas while the algorithm developed in this work combines them.

In summary, there are different inference algorithms performing well on different kinds of MLNs. This work adds another one for another special case.

4. MLN Design

The design of the MLN predicates and formulas is the key factor to a successfully learning robot. Unfortunately, finding such a design is nontrivial as pointed out by [Jai12] and summarized in Section 2.4.6. Therefore, this chapter describes the development of a MLN design extracting as much useful information from the CRAM log files as possible. At the same time, the design has to omit as much useless information as possible. Of course, a good MLN design must also keep the use cases in Figure 1.1 in mind.

The fact that the MLNs shall be learned from CRAM log files raises the question which information is available in these log files. Section 2.3.3 and especially Figure 2.6 already mentioned that the CRAM log files are OWL files and they stated that designators are stored in a MongoDB. Figure 4.1 presents the log files in detail. It shows the OWL classes as well as data and object properties as UML class diagram. The most important classes in Figure 4.1 are the sub classes of the *Task*[1] class: They represent classes of CRAM plans. *CRAMAchieve* for example represents all the *achieve* plans, for example *object-in-hand ?obj* or *loc ?obj ?loc*.

However, the name of a CRAM plan can better be reconstructed from the *taskContext* property. It contains a more precise description of the plan name, especially for plans defined with *def-cram-function*. The Prolog annotation introduced in Section 2.3.1 is represented in the property *goalContext*[2]. An example of such a Prolog annotation has already been mentioned: *OBJECT-IN-HAND ?OBJ*. Since plans can call sub plans, the sub plan call relationship is represented in the log files by the *subAction* property. The order of the sub plans is specified by the *nextAction* and *previousAction* predicates. Failures are indicated by the *eventFailure* and *caughtFailure* properties referencing the failure. As with the tasks, the failure type is indicated by the OWL class name of the failure, but the *label* property of the failure yields the correct failure type in more cases. Start and end time are also represented as classes referenced by the properties *startTime* and *endTime*. Finally, the task can have a *designator* property referencing a designator. These are the most important classes and properties in the OWL file.

Then, there is the MongoDB. In a MongoDB, the data are stored as documents which might contain other documents. Similar to the designators in CRAM, these documents consist of key value pairs where the key is a string and the value can be either a string

[1]There is actually no class named *Task* in the OWL file. Instead, the properties are applied directly on the sub classes. However, the *Task* class is introduced for a better visualization.

[2]This information can also be taken from the MongoDB.

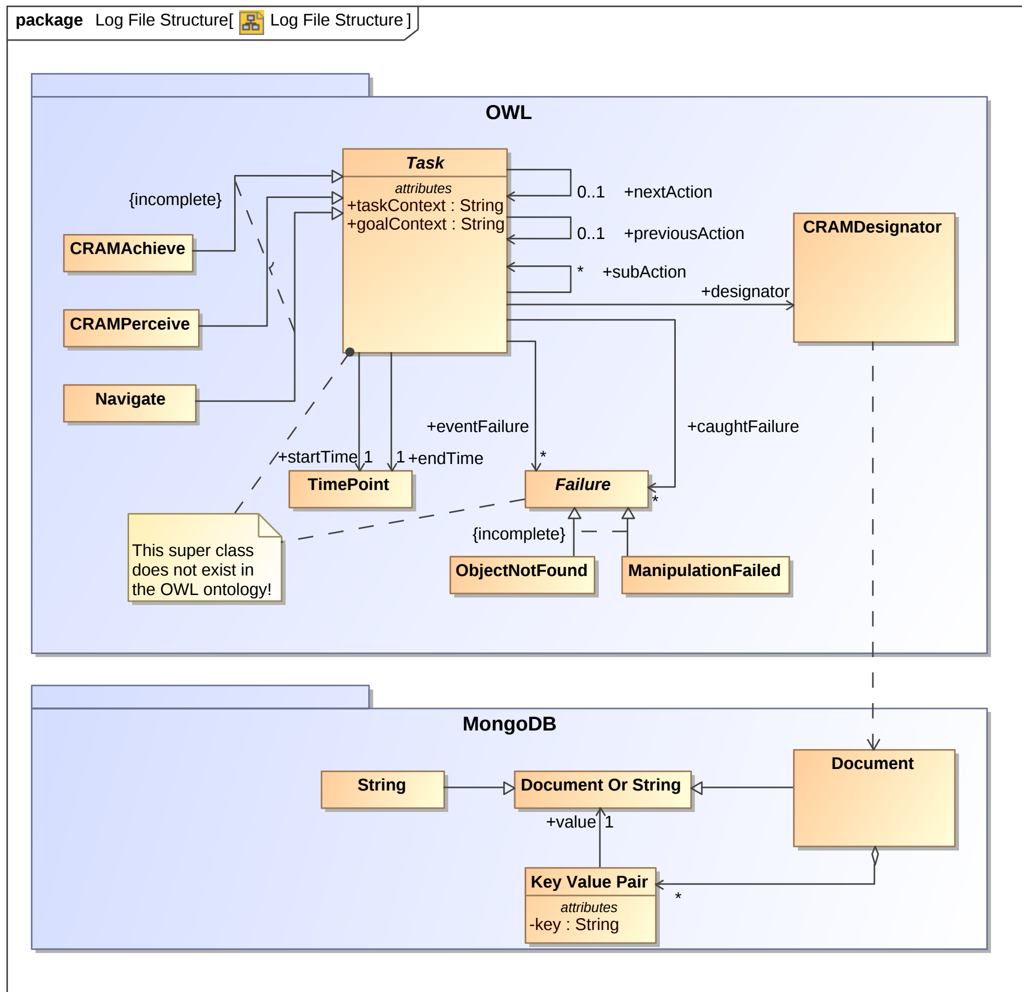


Figure 4.1.: The Structure of the CRAM Log Files

or a document. This makes it easy to store designators in a MongoDB[3], which is done in CRAM as shown by the package at the bottom of Figure 4.1. The mapping between the designator object in the OWL file containing no key value pairs and the more detailed designator in the MongoDB is done via the name of the OWL instance describing a designator: With this ID, the MongoDB can be queried for the correct designator. Apart from the designators, there is other information stored in the MongoDB such as the robot pose. However, it will not be used in this thesis. In summary, the log files consist of tasks connected by calling relationships, as well as of properties. Instead of directly describing designators, a reference to a MongoDB document is stored.

[3]Unfortunately, the designators are sometimes stored differently using empty keys, but this is an implementation detail.

An example of a log file is shown in Figure 4.2. It describes execution of a fictional plan similar to the one referenced in the last line of Listing 2.4. The instances of the OWL classes described above are depicted as a UML object diagram. As the log files represent sub plan calls, they have the form of a tree. The root node *Achieve_AbC* is the call to the *achieve* goal stating that the robot should have a *mug* standing on the *cupboard* with the name *kitchen_sink_block in* its *hand*[4] The corresponding designator describing the mug is stored in the MongoDB. It is assumed that this plan calls three sub plans: In the first sub node *Perceive_GhI*, the *mug* on the *cupboard* with the name *kitchen_sink_block* is perceived. The second sub node *Navigate_MnO* describes a plan call to a navigate goal causing the robot to move around. Finally, the robot tries to *pick* up the *mug* standing at the pose with the position $(1.5, 0.0, 0.9)$ and the orientation $(0, 0, 0, 1)$ in the node *Achieve_PqR*. This simple example of a log file will be used throughout the chapter.

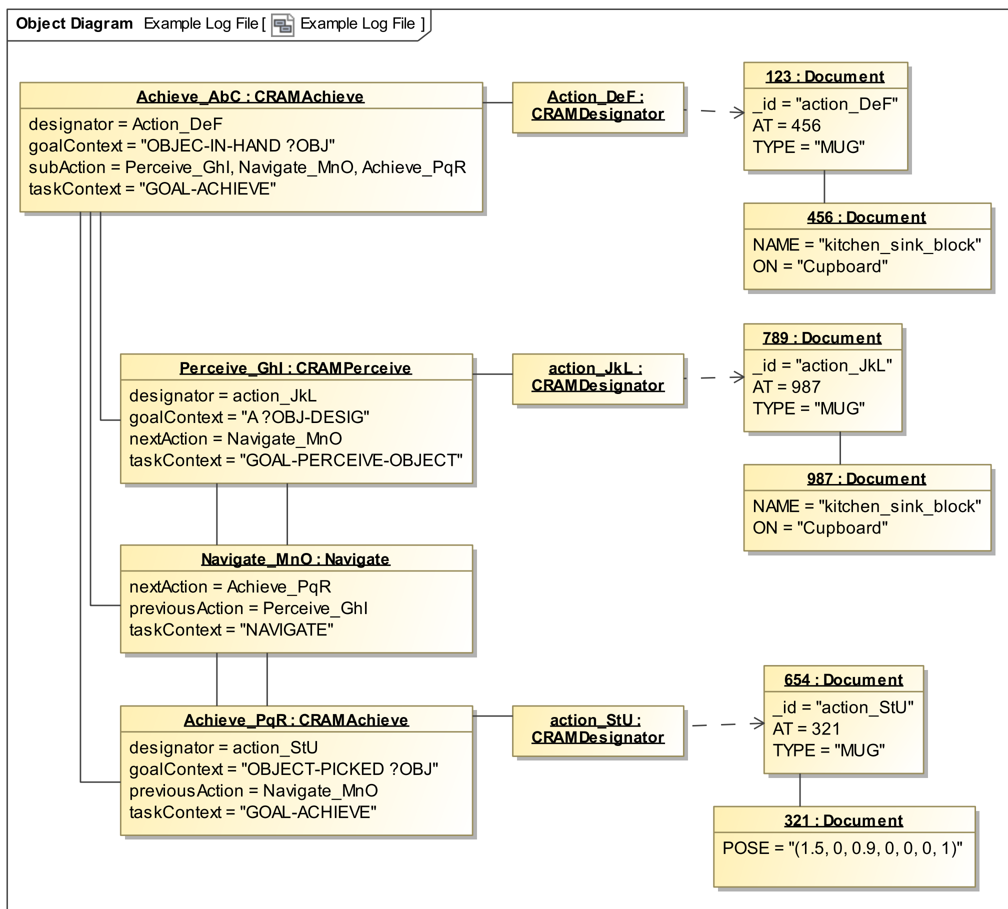


Figure 4.2.: An Example Log File as UML Object Diagram

[4] Due to a better visualization, the document objects actually do not fit to the class definition in this diagram.

The following sections will introduce different MLN designs developed in the creation process of this thesis. Moreover, a method for efficiently computing formula weights is introduced. Finally, the MLN design used in the implementation in Chapter 6 is introduced.

4.1. A Naive Approach

The perhaps first thing that comes into ones mind when thinking about a MLN design is to simply convert the OWL file structure in Figure 4.1 to first-order predicate logic. Since the concept of plans and sub plans is very generic and not only applied by CRAM, it is obvious not to model designators but to replace them by something more generic. This idea was translated to a MLN. Listing 4.1 shows the predicate declarations for this MLN.

```
1  taskType(task, taskType!)
2  goal(task, goal?)
3  parentTask(task, task?)
4  successor(task, task?)
5  error(task, error)
6  usedInTask(task, object)
7  perceives(task, object)
8  acts(task, actionType, actionParameter)
9  duration(task, abstractDuration)
10 objectType(object, objectType)
11 objectProperty(object, objectProperty)
12 objectLocation(object, abstractLocation)
13 executedAt(task, abstractLocation)
14 causedRobotPositionDifference(task, spatialRelation)
15 causedObjectPositionDifference(task, object, spatialRelation)
```

Listing 4.1: Predicate Declarations for a Naive MLN

As mentioned before, the design translates the types and relations from the OWL file to first-order logic. Hence, the type *task* conforms to the abstract task in Figure 4.1. Furthermore, the predicates are related to the relations and properties: The predicate *taskType* corresponds to the *taskContext* property, while the *goal* predicate corresponds to the *goalContext* property. Moreover, the *subAction* relation is represented by the predicate *parentTask* and the *successor* predicate models *nextAction*. *eventFailure* and *caughtFailure* are aggregated in the predicate *error*. Finally, time points are aggregated in the *duration* predicate. All in all, these predicates are very close to the log file.

The designators are very CRAM specific. To make the MLN applicable for other planning languages, the information from the designators is modeled with an agent system in mind. [RN10] defines an agent as a system able to perceive its environment and act there. Thus, there are the predicates *perceives* and *acts*. The robot is expected to perceive objects, such as mugs. Since an action does not only affect objects, the *acts* predicate is modeled without any object relation. In return, the predicate *usedInTask* additionally defines

the relation between executed tasks and objects. The objects are modeled as separate type. There are predicates specifying the *objectType* as well as *objectProperties* and an *objectLocation*. Another rather general concept is the attempt to specify the pose of the robot as well as the pose differences of the robot and the objects achieved by the execution of the tasks. This is done using the predicates *executedAt*, *causedRobotPositionDifference*, and *causedObjectPositionDifference*.

Every training database for the MLN contains a whole log file. This is necessary due to the connection of the different tasks and objects. Listing 4.2 for example shows an extract of the training database for the log shown in Figure 4.2. It shows the *object-in-hand* and the *navigate* task.

```
1           taskType("Achieve_AbC", "ACHIEVE")
2           goal("Achieve_AbC", "OBJECT-IN-HAND ?OBJ")
3           usedInTask("Achieve_AbC", mug-1)
4           duration("Achieve_AbC", medium)
5           objectType(mug1, "MUG")
6           objectLocation(mug1, "kitchen_sink_block")
7           /* ... */
8           taskType("Navigate_MnO", "NAVIGATE")
9           parentTask("Navigate_MnO", "Achieve_AbC")
10          duration("Navigate_MnO", medium)
11          /* ... */
```

Listing 4.2: Extract of the Training Database for a Naive MLN

Concerning the formula design, [Jai12] states that conditional probabilities are best modeled as conjunctions. Therefore, those atoms believed to be dependent on each other are tied together with a conjunction. An extract of the formulas for this naive MLN design is shown in Listing 4.3. The full MLN is shown in Listing B.2.

```
1  0.0 taskType(?t, +?tt) ^ goal(?t, +?g)
2  0.0 taskType(?pt, +?tt) ^ parentTask(?t, ?pt) ^ taskType(?t, +?tt2)
3  0.0 taskType(?t, +?tt1) ^ successor(?t, ?t2) ^ taskType(?t2, +?tt2)
4  /* ... */
5  0.0 taskType(?t, +?tt) ^ perceives(?t, ?o) ^ objectType(?o, +?ot)
6  0.0 goal(?t, +?g) ^ perceives(?t, ?o) ^ objectType(?o, +?ot)
7  /* ... */
```

Listing 4.3: Extract of the Template Formulas for a Naive MLN

To make the MLN independent of a single execution, object and task names are replaced by variables. However, some constants are required in the formulas in order to state relationships between them. Fortunately, pracmln offers the $+$ operator to replace a variable by the constants of its domain: If the variable domain is D, the operator replaces the formula with $|D|$ formulas using a constant from D instead of the variable. An example is depicted in Listing 4.4: It shows the formulas generated from the first formula in Figure 4.2 with replaced variables.

```
1 0.0 taskType(?t, "ACHIEVE") ^ goal(?t, "OBJECT-IN-HAND ?OBJ")
2 0.0 taskType(?t, "ACHIEVE") ^ goal(?t, "A ?OBJ-DESIG")
3 0.0 taskType(?t, "ACHIEVE") ^  goal(?t, "OBJECT-PICKED")
4 0.0 taskType(?t, "PERCEIVE-OBJECT") ^ goal(?t, "OBJECT-IN-HAND ?OBJ")
5 0.0 taskType(?t, "PERCEIVE-OBJECT") ^ goal(?t, "A ?OBJ-DESIG")
6 0.0 taskType(?t, "PERCEIVE-OBJECT") ^ goal(?t, "OBJECT-PICKED")
7 0.0 taskType(?t, "NAVIGATE") ^ goal(?t, "OBJECT-IN-HAND ?OBJ")
8 0.0 taskType(?t, "NAVIGATE") ^ goal(?t, "A ?OBJ-DESIG")
9 0.0 taskType(?t, "NAVIGATE") ^ goal(?t, "OBJECT-PICKED")
```

Listing 4.4: One Expanded Template Formula for a Naive MLN

From a semantic point of view, the first formula in Listing 4.3 states which goal exists for which task type. In the example, the combination of the *ACHIEVE* task type and the *OBJECT-IN-HAND ?OBJ* goal is shown in the training data and will thus get one of the highest weights in Listing 4.4. The second formula tries to model the relationship between task types and the parent task type, which can be useful for determining potential parent or child tasks. The third formula is intended to model the relationship between tasks and their successor tasks. Then, the formula in line 5 states which object types are perceived in which tasks. Since a plan is actually represented by the combination of task type and goal, the same relation is represented for the goal in line 6.

If the robot wants to query for the objects which are expected at the *kitchen_sink_block*, as required by the first use case in Figure 1.1, it could execute the query in Listing 4.5. To infer the success probability for the *achieve object-in-hand* task, the robot would execute the query in Listing 4.6. Finally, it would execute the query in Listing 4.7 to get the missing location of a mug to pick up.

```
P( objectType(o1, ?t) |
   taskType(T, "PERCEIVE-OBJECT")
   ^ goal(T, "A ?OBJ-DESIG")
   ^ perceives(T,O)
   ^ objectLocation(O, "kitchen_sink_block") )
```

Listing 4.5: Query for an Object Expected at a Location in the Naive MLN

```
P( !(EXIST ?e (error(T, ?e))) |
     taskType(T, "ACHIEVE")
   ^ goal(T, "OBBJECT-IN-HAND ?OBJ")
   ^ usedInTask(T,O)
   ^ objectType(O,"MUG")
   ^ objectLocation(O, "kitchen_sink_block") )
```

Listing 4.6: Query for a Success Probability in the Naive MLN

```
P( objectLocation(O, ?l) |
     !(EXIST ?e (error(T, ?e)))
   ^ taskType(T, "ACHIEVE")
   ^ goal(T, "OBBJECT-IN-HAND ?OBJ")
   ^ usedInTask(T,O)
   ^ objectType(O,"MUG") )
```

Listing 4.7: Query for a Plan Parametrization in the Naive MLN

Nevertheless, this approach causes several problems: First of all, the number of formulas gets far too large. Moreover, the size of the training databases is very large, too. This is especially important since one has to bear the computational complexity in mind when designing MLNs. However, there are more serious problems concerning the represented probability distributions. More precisely, the inferred probabilities do not conform with the expected probabilities when executing queries on a manually created example MLN. [Jai12] actually requires the conjunctions to be exhaustive to model a conditional probability distribution. As a consequence, this would require even more formulas containing the negation of each atom.

The problem here is especially that atoms created with the predicate *taskType* are present in almost every conjunction. That makes it difficult to query for those atoms since different formulas containing atoms of this predicate influence each other. This is especially a problem if the number of objects in the training database does not conform to the number of objects in the query. Then, a formula and thus its weight might not be available during the inference due to a missing object. However, this weight was used during the training and thus, the inference produces wrong probabilities. An example is the formula in line 5 in Listing 4.3: If the variable $?o$ has an empty domain since objects are uninteresting in a query for the task type of a sub task, the formula is not used during the query. However, this weight is necessary to determine the probability for *taskType* atoms. What results is a wrong probability.

In summary, this approach is too naive to work.

4.2. A Generic Approach

The experiences from Section 4.1 were used for creating an improved MLN design. It is still generic in the sense that it models an agent system rather than a CRAM specific system. Listing 4.8 shows the predicate declarations of this MLN design.

```
1  currentTask(timeStep,taskType?)
2  currentTaskFinished(timeStep)
3  currentParameter(timeStep,propertyKey,propertyValue?)
4  currentParentTask(timeStep,taskType?)
5  parentParameter(timeStep,propertyKey,propertyValue?)
6  childTask(timeStep,taskType)
7  nextTask(timeStep,taskType?)
8  nextTaskFinished(timeStep)
9  nextParameter(timeStep,propertyKey,propertyValue?)
10 duration(timeStep,duration?)
11 error(timeStep,error?)
12 objectType(object,objectType?)
13 objectProperty(object,objectPropertyKey,objectPropertyValue?)
14 objectLocation(timeStep,object,location?)
15 perceivedObject(timeStep,object)
16 usedObject(timeStep,object)
```

Listing 4.8: Predicate Declarations for the Generic MLNs

The most obvious difference to the previous MLN is the replacement of tasks by time steps. When looking at the time steps of the tasks in the OWL file, one might get the idea to view the task tree as an execution of a state machine. The start and end time can be considered as two different states of the executed plan. Thus, the MLN models a probabilistic state machine. *currentTask* defines the state of the state machine at a specific time point. Since two time points of each task are used, the predicate *currentTaskFinished* allows to distinguish start and end time. The idea to use probabilistic state machines in CRAM is not new: this has already been done in [Kir09], as well as in [BG05] for the predecessor of CRAM.

Furthermore, the semantics of *nextTask* changes: For tasks having children, the next task is the first child task instead of a parallel task. The order of the *nextTask* relationship for the tasks in Figure 4.2 is visualized in Figure 4.3. Since all but the root task have no child tasks, there is only one state for them.

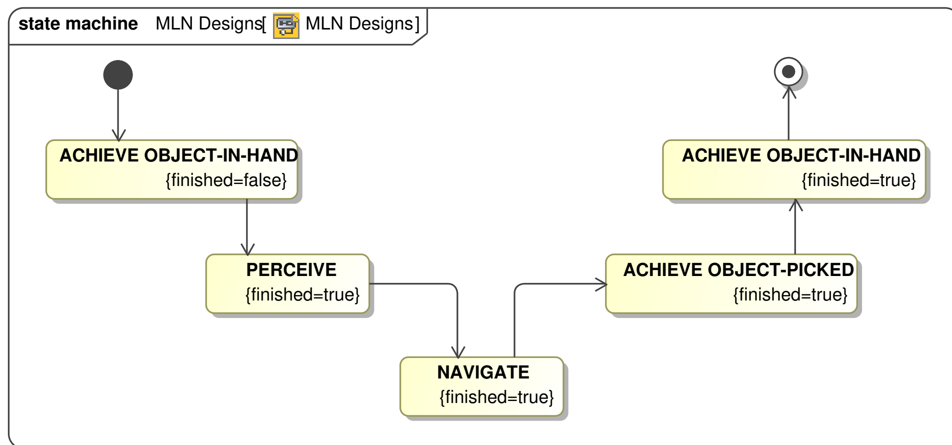


Figure 4.3.: State Machine for the Example in Figure 4.2

Another difference to the previous MLN is the disappearance of the *goal* predicate since it was too CRAM specific. Moreover, it is possible to define task parameters now. There are still predicates for modeling child and parent task relations and the objects are treated similar to the objects in the naive MLN.

The result of the new MLN structure is an increased number of smaller training databases since the objects do not depend on each other anymore. For the example in Figure 4.2, four training databases will be created. Listing 4.9 shows the training database for the *object-in-hand* task.

```
1   currentTask(0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
2   currentParameter(0,"?OBJ",
           "(:type :mug) (:at (:on Cupboard) (:name kitchen_sink_block))")
3   childTask(0, "ACHIEVE OBJECT-PICKED ?OBJ")
4   childTask(0, "PERCEIVE-OBJECT A ?OBJ-DESIG")
5   childTask(0,"NAVIGATE")
6   nextTask(0,"PERCEIVE-OBJECT A ?OBJ-DESIG")
7   nextTaskFinished(0)
8   nextParameter(0,"?OBJ-DESIG",
           "(:type :mug) (:at (:on Cupboard) (:name kitchen_sink_block))")
9   duration(0, Long)
```

Listing 4.9: One Training Database for a Generic MLN

In contrast to the previous MLN, the formulas are constructed by conjugating ground literals from the training databases instead of using the + operator. Thus, only one formula per training database is created if this formula does not already exist. Again, constants which are too special, such as object names or time steps, are replaced by variables. To avoid the influence of other formulas, three different MLNs are created for different use cases. In each MLN, every formula has the same structure.

The formulas of the *object* MLN can be used to infer object properties, types, locations or the task in which an object is used or perceived. In this MLN design, it is assumed that there is a correlation between the task type, the parent task and the used or perceived objects. Therefore, the task types in the training databases which perceive or manipulate objects are conjugated with the parent task type, the perceived or used object, as well as the object properties. Listing 4.10 shows these formulas for the example scenario depicted in Figure 4.2.

```
1   0.0 currentTask(?t0,"PERCEIVE-OBJECT A ?OBJ-DESIG")
        ^ currentParentTask(?t0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
        ^ perceivedObject(?t0,?o1) ^ objectType(?o1,"MUG")
        ^ objectLocation(?t0,?o1,"kitchen_sink_block")
        ^ !objectProperty(?o1,?k,?v)
2   0.0 currentTask(?t0,"ACHIEVE OBJECT-PICKED ?OBJ")
        ^ currentParentTask(?t0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
        ^ usedObject(?t0,?o1) ^ objectType(?o1,"MUG")
        ^ objectLocation(?t0,?o1,"(1.5,0.0,0,9,0,0,0,1)")
        ^ !objectProperty(?o1,?k,?v)
```

Listing 4.10: Example Formulas of a Generic Object MLN

Concerning the use cases in Figure 1.1, the object MLN can be used to infer what is perceived at a specific location. Listing 4.11 shows the query for the object type which is usually perceived at the *kitchen_sink_block*.

```
P ( objectType(0,?t)
    ^ currentTask(0,"PERCEIVE-OBJECT A ?OBJ-DESIG")
    ^ objectLocation(0,O,"kitchen_sink_block")
    ^ perceivedObject(0,O) )
```

Listing 4.11: Query for an Object Perceived at a Location in the Generic Object MLN

The state machine MLN is designed for inferring which will be the next executed task. Fur this purpose, the relation between the current task, the parameters of the current task, the next task and the parameters of the next task are important. Moreover, it is important whether these tasks are finished or not. Therefore, the atoms which represent this information and which appear together in a training database are conjugated to a formula. One of the formulas for the example scenario is shown in Listing 4.12. A typical query for the state machine MLN is the query for the next task which is shown in Listing 4.13.

```
1  0.0 currentTask(?t0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
       ^ currentParameter(?t0,"?OBJ",
         "(:type :mug) (:at (:on Cupboard) (:name kitchen_sink_block))")
       ^ !currentTaskFinished(?t0)
       ^ nextTask(?t0,"PERCEIVE-OBJECT A ?OBJ-DESIG")
       ^ nextTaskFinished(?t0)
       ^ nextParameter(?t0,"?OBJ-DESIG",
         "(:type :mug) (:at (:on Cupboard) (:name kitchen_sink_block))")
```

Listing 4.12: Example Formula of a Generic State Machine MLN

```
P ( nextTask(0, ?t) |
    ^ currentTask(0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
    ^ currentParameter(0,"?OBJ", "(:type :mug) (:at (:on Cupboard)
                                  (:name kitchen_sink_block))")
    ^ !currentTaskFinished(0) )
```

Listing 4.13: Query for the Next Task in the State Machine MLN

One might ask why the *nextTask* predicate is defined instead of using *currentTask* with the next time step as argument. There are two reasons: The first one is that other formulas would be true in worlds where a formula is true due to the newly created ground atom and thus have an influence on the probability. The second reason is the computational complexity: Introducing a new *currentTask* ground atom due a broader *timeStep* domain creates more worlds and thus it makes the inference harder.

The task MLN is similar to the state machine MLN. However, in contrast to the state machine MLN, the task MLN is able to infer the most probable child or parent task as well as the duration or errors, but not the next task or the next task parameters. Hence, the conjugation of atoms is retained, but instead of including the next task, its parameters and the information whether the task is finished, the duration, errors, child tasks and the parent task are included. A formula of the task MLN in the example is shown in Listing 4.14.

```
1  0.0 currentTask(?t0,"ACHIEVE OBJECT-PICKED ?OBJ")
      ^ currentParentTask(?t0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
      ^ parentParameter(?t0,"?OBJ",
        "(:type :mug) (:at (:on Cupboard) (:name kitchen_sink_block))")
      ^ currentParameter(?t0,"?OBJ",
        "(:type :mug) (:at (:pose (1.5,0.0,0,9,0,0,0,1)))")
      ^ !childTask(?t0,?tt)
      ^ duration(?t0,Medium) ^ !error(?t,?e)
```

Listing 4.14: Example Formula of a Generic Task MLN

The task MLN is able to perform inferences for the last two use cases mentioned by Figure 1.1. Listing 4.15 shows the query for the success probability of a parametrized *achieve object-in-hand* plan. The reversed query is shown in Listing 4.15, where a plan parameter is inferred given that the task is successful.

```
P ( !(EXIST ?e (error(0, ?e))) |
      currentTask(0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
    ^ currentParameter(0,"?OBJ", "(:type :mug) (:at (:on Cupboard)
                                 (:name kitchen_sink_block))") )
```

Listing 4.15: Query for a Success Probability in the Task MLN

```
P ( currentParameter(0, ?p) |
      currentTask(0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
    ^ !(EXIST ?e (error(0, ?e))) )
```

Listing 4.16: Query for a Successful Parametrization of a Task in the Task MLN

Nevertheless, there are also problems with this approach. Most importantly, the pseudo log likelihood learner returns the wrong weights in contrast to the log likelihood learner. Unfortunately, log likelihood learning is intractable for larger databases. Moreover, both learners do not allow to update weights when new training data is available. However, this is required to achieve lifelong learning. Thus, there are problems with the available learning algorithms.

Another problem arises from the type of negation that is used in the formulas: Sometimes, it is necessary to state that no other ground atom of a certain predicate is true. Otherwise, all other atoms of that predicate would get the probability 0.5 which is not desirable, for example for object properties. Therefore, the formulas contain a negation of this predicate. For instance, the literal *!objectProperty(?o1,?k,?v)* is used for negating the *objectProperty* predicate. However, the grounding process creates one ground formula for each variable assignment to the variables *?k* and *?v*. Hence, if an atom of the predicate appears non negated, there is one less ground formula for this case due to a contradiction. This results in incorrect probabilities.

4.3. Simple Weight Calculation

One problem in the MLNs presented in the previous section was the intractable learning. However, the MLNs learned with the log likelihood learner showed one useful conspicuousness: It can be observed that the weights have a special form in most cases. This form is $c + \ln(p_i)$ where c is a constant (for example 40) and p_i states how often the ground formulas created from the formula are true in the training databases.

Section 2.4.1 already mentioned that [Jai12] describes a case where weights can easily be calculated: If the formulas are mutually exclusive and if possible exhaustive, the weights w_i can be set to $\ln(p_i)$ where p_i is the probability for the formula. A set of formulas is mutually exclusive if in one world maximally one formula is true. It is exhaustive if all combinations are modeled. [Jai12] also mentions that it is possible to add arbitrary constants to those weights. While the mutual exclusiveness mentioned by [Jai12] is necessary, it turns out that a constant c, which is large enough, replaces the exhaustiveness under certain circumstances.

The claim that assigning the formula weight $c + \ln(p_i)$ replaces learning the formula weights with an optimization algorithm under certain conditions has to be proven. In other words, it has to be proven that the log likelihood is maximized. This is done in Theorem A.1. For the proof, it was necessary to formulate the conditions mentioned above. These conditions are:

- $\exp(c)$ is much larger than the number of worlds generatable from the constants in the training data and in the MLN.
- Every formula has to appear at least once in the training data.
- The formulas are mutually exclusive for all worlds generatable from the constants in the training data and in the MLN.
- Every formula must be true in the same number of worlds t in every training database.
- In every training database, there must be exactly one true ground formula.

Thus, in a MLN meeting all these conditions, there is a simple algorithm for fast and at the same time exact weight learning. Moreover, updating the MLN with new training data is easy: The weights of existing formulas just have to be increased. However, these conditions are very restrictive. Nevertheless, it is possible to create useful MLNs meeting them as the following sections will show.

4.4. A Refined Generic Approach

The weight calculation described in Section 4.3 allows tractable learning and updating the weights if new training data is available. But before it can be applied to the MLNs in Section 4.2, it is necessary to modify the MLN design to match the requirements for the easier weight calculation. Though the predicate declarations stay the same, it is necessary to get rid of the negation literal since it results in a different amount of worlds for some formulas. One solution to achieve that is to equip every formula with a negation of all but the contained atoms of the predicate to negate, if a negation of that predicate is required. A simple implementation of this solution is to hard-code all of these negated atoms. However, this causes problems as soon as the domains are enlarged. A more intelligent implementation is the usage of a negated existential quantifier. This implementation is, for instance, applied in Listing 4.17 in the example object MLN.

```
1  0.0 currentTask(?t0,"PERCEIVE-OBJECT A ?OBJ-DESIG")
       ^ currentParentTask(?t0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
       ^ perceivedObject(?t0,?o1) ^ objectType(?o1,"MUG")
       ^ objectLocation(?t0,?o1,"kitchen_sink_block")
       ^ (!(EXIST ?ol0 ((!(?ol0=kitchen_sink_block))
         ^ objectLocation(?t0,?o1,?ol0))))
       ^ (!(EXIST ?op0, ?op1 (objectProperty(?o1,?op0,?op1))))
2  0.0 currentTask(?t0,"ACHIEVE OBJECT-PICKED ?OBJ")
       ^ currentParentTask(?t0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
       ^ usedObject(?t0,?o1) ^ objectType(?o1,"MUG")
       ^ objectLocation(?t0,?o1,"(1.5,0.0,0,9,0,0,0,1)")
       ^ (!(EXIST ?ol0 ((!(?ol0="(1.5,0.0,0,9,0,0,0,1)"))
         ^ objectLocation(?t0,?o1,?ol0))))
       ^ (!(EXIST ?ol0, ?op1 (objectProperty(?o1,?ol0,?op1))))
```

Listing 4.17: Example Formulas of a Refined Generic Object MLN

These formulas use the fact that existential quantification is replaced by a large disjunction in MLNs. With the domains *timeStep={0}*, *object={mug1}*, *propertyKey={}*, as well as *location={"kitchen_sink_block", "pancake_table", "(1.5,0.0,0,9,0,0,0,1)"}*, the first formula of Listing 4.17 for example is grounded to the one shown in Listing 4.18. What the formulas actually do is restricting the set of worlds considered during the inference exactly to those seen in the training databases.

```
1  0.0 currentTask(0,"PERCEIVE-OBJECT A ?OBJ-DESIG")
       ^ currentParentTask(0,"ACHIEVE OBJECT-IN-HAND ?OBJ")
       ^ perceivedObject(0,mug1) ^ objectType(mug1,"MUG")
       ^ objectLocation(0,mug1,"kitchen_sink_block")
       ^ !objectLocation(0,mug1,"pancake\_table")
       ^ !objectLocation(0,mug1,,"(1.5,0.0,0,9,0,0,0,1)\")
```

Listing 4.18: Grounded Formula of a Refined Generic Object MLN

As one might expect, this approach is still problematic. One problem is that far more negated existential quantifiers are actually required to make the weight calculation really applicable. In Listing 4.17 for example, a negated existential quantifier for the *childTask* predicate is required. Thus, the formulas get larger and larger.

Other problems get visible if one tries to convert real CRAM logs to this MLN structure. First of all, the designators in CRAM cannot be represented well with the MLN structure: Since designators model both objects in the world and plan parameters, they are complex, recursive data structures. However, these recursive structures have to be pressed into one string to serve as plan parameter. An example is the *?OBJ* parameter in Listing 4.14.

Then, the generic agent approach for modeling the interaction of the robot within an environment is not well suited for CRAM. The reason is that the perception routines in CRAM work differently: Instead of perceiving all objects in the environment, a *perception request* is executed. This request tells the robot to perceive a mug on the counter top for example. Therefore, the perceived objects and the parameter for the perception task contain the same information. Concerning the actions the robot is executing, there is a similar problem: The actions that are passed to the process modules executing them are action designators which are recursive structures, too.

Finally, use cases for a probabilistic state machines and sub task relations are rare. Though it would be possible to learn how a plan like "prepare breakfast" is composed, this information would be useless: This plan already exists. A probabilistic state machine predicting which task might be executed next might be possible, but the bullet reasoning approach does already provide a simulation mechanism.

Hence, the concept of a generic MLN design is no good idea.

4.5. A Designator Based Approach

The problems encountered in the previous approach lead to the decision to make the MLN CRAM specific. Due to the fact that the designators carry most of the important information about objects, locations, and actions, a designator based MLN design was created. In this MLN design, reasoning about objects and actions is done by reasoning about designators. Moreover, reasoning about the relationships between tasks is omitted. Listing 4.19 shows the predicate declarations for the designator based MLN.

```
1 name(task,taskName!)
2 success(task)
3 goalPattern(task,goalPattern?)
4 goalProperty(task,goalKey,goalPropertyValue)
5 goalDesignator(task,goalKey,designatorHash)
6 designatorHash(designator,designatorHash?)
7 designatorType(designator,designatorType?)
8 designatorProperty(designator,designatorPropertyKey,
      designatorPropertyValue)
9 subDesignator(designator,subDesignatorKey,designatorHash)
```

Listing 4.19: Predicate Declarations of a Designator Based MLN

Since the relationships between tasks are not considered in the MLN, it is possible to replace the time step type by the *task* type again. To be able to reason about success probabilities as well as to restrict the search for designators to special tasks, there are predicates like *name* and *success* for this task type. As mentioned before, the designators are modeled explicitly as recursive key value structure in this approach. Only those designators which are referenced by the *goalPattern*, which is the Prolog annotation of the task, are used in the MLN. They are bound to a task by the predicate *goalDesignator*. Since the Prolog annotation can also contain values which are no designators as parameters, these parameters can be modeled as *goalProperty*. An example of such a property is a pose.

To avoid having one large formula with all the sub designators and to create smaller MLNs, each sub designator gets its own formula. Designators have an ID to be able to reference sub designators. In order to update an existing formula, the IDs of two designators must be equal if they have the same properties. Therefore, the ID is implemented as hash. This ID is represented by the predicate *designatorHash*. Apart from that, a designator can have a *designatorType*, identifying the designator as object, location or action designator. The key value pairs of the designator are modeled by the predicates *designatorProperty* for string values and *subDesignator* for sub designators.

Similar to the previous approaches, formulas are constructed by creating conjunctions from the training databases. More precisely, one formula is created for each (sub-) designator in the training data. A formula consists of the designator hash identifying the (sub-) designator, as well as the designator type, designator properties and the sub designators. This information is extended by the name and the goal pattern of the plan parametrized with the designator as well as the information whether this task was successful. The corresponding atoms are considered as a unit and thus conjugated. A *goalDesignator* atom connects the task with the designator. To be able to use the simple weight calculation presented in Section 4.3, negated existential quantifiers are used again. Again, the formulas actually restrict the set of worlds considered during the inference exactly to those seen in the training databases. As an example, Listing 4.20 shows the formulas created from the *object-in-hand* task shown in Figure 4.2

```
1  0.0 name(?t1,"ACHIEVE") ^ success(?t1)
       ^ goalPattern(?t1,"OBJECT-IN-HAND ?OBJ")
       ^ goalDesignator(?t1,"?OBJ","69d54d73b9")
       ^ designatorHash(?d0,"69d54d73b9") ^ designatorType(?d0,"object")
       ^ designatorProperty(?d0,"TYPE","mug")
       ^ subDesignator(?d0,"AT","c4eb4c6730")
       ^ !(EXIST ?p0 ((?p0=/="OBJECT-IN-HAND ?OBJ") ^ goalPattern(?t1,?p0)))
       ^ !(EXIST ?p0, ?p1 ((?p0=/="?OBJ" v ?p1=/="69d54d73b9")
          ^ goalDesignator(?t1,?p0,?p1)))
       ^ !(EXIST ?p0, ?p1 ((?p0=/="AT" v ?p1=/="c4eb4c6730")
          ^ subDesignator(?d0,?p0,?p1)))
       ^ !(EXIST ?p0, ?p1 ((?p0=/="TYPE" v ?p1=/="mug")
          ^ designatorProperty(?d0,?p0,?p1)))
       ^ !(EXIST ?p0, ?p1 (goalProperty(?t1,?p0,?p1)))
2  0.0 name(?t1,"ACHIEVE") ^ success(?t1)
       ^ goalPattern(?t1,"OBJECT-IN-HAND ?OBJ")
       ^ goalDesignator(?t1,"?OBJ","69d54d73b9")
       ^ designatorHash(?d0,"c4eb4c6730") ^ designatorType(?d0,"location")
       ^ designatorProperty(?d0,"NAME","kitchen_sink_block")
       ^ designatorProperty(?d0,"ON","Cupboard")
       ^ !(EXIST ?p0 ((?p0=/="OBJECT-IN-HAND ?OBJ") ^ goalPattern(?t1,?p0)))
       ^ !(EXIST ?p0, ?p1 ((?p0=/="?OBJ" v ?p1=/="69d54d73b9")
          ^ goalDesignator(?t1,?p0,?p1)))
       ^ !(EXIST ?p0, ?p1 ((?p0=/="NAME" v ?p1=/="kitchen_sink_block")
          ^ (?p0=/="ON" v ?p1=/="Cupboard")
          ^ designatorProperty(?d0,?p0,?p1)))
       ^ !(EXIST ?p0, ?p1 (goalProperty(?t1,?p0,?p1)))
       ^ !(EXIST ?p0, ?p1 (subDesignator(?d0,?p0,?p1)))
```

Listing 4.20: Extract of a Designator Based MLN

Compared to the previous MLN designs, one task is divided into different formulas. Therefore, there are also different databases for each task. Listing 4.21 shows the training database for the *achieve object-in-hand* task in Figure 4.2 as example.

```
1  name("Achieve_AbC","ACHIEVE")
2  success("Achieve_AbC")
3  goalPattern("Achieve_AbC","OBJECT-IN-HAND ?OBJ")
4  goalDesignator("Achieve_AbC","?OBJ","69d54d73b9")
5  designatorHash("123","69d54d73b9")
6  designatorType("123","object")
7  designatorProperty("123","TYPE","mug")
8  subDesignator("123","AT","c4eb4c6730")
9  ---
10 name("Achieve_AbC","ACHIEVE")
11 success("Achieve_AbC")
12 goalPattern("Achieve_AbC","OBJECT-IN-HAND ?OBJ")
13 goalDesignator("Achieve_AbC","?OBJ","69d54d73b9")
14 designatorHash("456","c4eb4c6730")
15 designatorProperty("456","NAME","kitchen_sink_block_counter_top")
16 designatorProperty("456","ON","Cupboard")
17 designatorType("456","location")
```

Listing 4.21: Training Databases for one Task in the Designator Based MLN

Of course, this MLN design is suited for the use cases shown in Figure 1.1. Listing 4.22 shows the query for the type of an object perceived at the *kitchen_ sink_ block*. The calculation of a success probability for the *achieve object-in-hand* plan is shown in Listing 4.23. Finally, there is the use case to find a parametrization for a plan to be successful. This is realized by the query in Listing 4.24. There, two atoms are conjugated in the query in order to get the key for the sub designator as well as the properties of the sub designator.

```
P ( designatorProperty(D,"TYPE", ?p) |
    EXIST ?dh,?sdh (name(T, "PERCEIVE-OBJECT")
                    ^ success(T)
                    ^ goalPattern(T, "A OBJ-DESIG ?OBJ-DESIG")
                    ^ goalDesignator(T, "?OBJ-DESIG", ?dh)
                    ^ designatorType(D, "object")
                    ^ designatorHash(D, ?dh)
                    ^ subDesignator(D, "AT", ?sdh)
                    ^ designatorType(SD,"location")
                    ^ designatorHash(SD, ?sdh)
                    ^ designatorProperty(SD,"ON","Cupboard")
                    ^ designatorProperty(SD,"NAME","kitchen_sink_block"))
  )
```

Listing 4.22: Query for Objects Perceived at a Location Using the Designator Based MLN

```
P ( success(T) |
    EXIST ?dh,?sdh (name(T, "ACHIEVE")
                    ^ goalPattern(T, "OBJECT-IN-HAND ?OBJ")
                    ^ goalDesignator(T, "?OBJ", ?dh)
                    ^ designatorType(D, "object")
                    ^ designatorProperty(D, "TYPE", "mug")
                    ^ designatorHash(D, ?dh)
                    ^ subDesignator(D, "AT", ?sdh)
                    ^ designatorType(SD,"location")
                    ^ designatorHash(SD, ?sdh)
                    ^ designatorProperty(SD,"ON","Cupboard")
                    ^ designatorProperty(SD, "NAME", "kitchen_sink_block"))
  )
```

Listing 4.23: Query for a Success Probability Using the Designator Based MLN

```
P ( designatorProperty(SD,?sdk, ?sdv) ^ subDesignator(D, ?k, ?sdh) |
    EXIST ?dh, ?sdh, ?k (name(T, "ACHIEVE")
                        ^ success(T)
                        ^ goalPattern(T, "OBJECT-IN-HAND ?OBJ")
                        ^ goalDesignator(T, "?OBJ", ?dh)
                        ^ designatorType(D, "object")
                        ^ designatorProperty(D, "TYPE", "mug")
                        ^ designatorHash(D, ?dh)
                        ^ subDesignator(D, ?k, ?sdh)
                        ^ designatorHash(SD, ?sdh))
  )
```

Listing 4.24: Query for the Completion of a Designator Using the Designator Based MLN

One problem in this approach is the large number of worlds to be considered when executing a query. An example is the query in Listing 4.22. This query leaves several variables open, such as the designator hashes. Since there is almost one designator hash per formula, a large amount of worlds must be generated. Even for the MLN in Listing 4.20, the normal exact inference algorithm reports 5 435 817 984 worlds to enumerate. Thus, the inference with all existing inference methods in pracmln is intractable. Therefore, an inference algorithm able to cope with this case was developed. It is described in detail in Chapter 5.

Though this approach is already pretty usable, it has some drawbacks. First of all, the large number of worlds requires a high weight constant for the weights. This can be problematic, since exp(709.79) extends the range of double precision floating point types on normal 64 bit computers. However, this problem is solvable. A more difficult problem is that the approach does not generalize. If the robot has for example picked up a mug during the training, the approach will not be able to estimate the success probability for picking up a *green* mug. Therefore, it is not really an improvement over the Prolog reasoning described in [WTBB14].

4.6. The Final Approach

The final approach is the one that will be used in the implementation in Chapter 6. Since the mapping of the CRAM log files to the MLN design in the previous attempt works pretty well, the concept of a MLN primarily modeling designators was retained. However, it was tried to build a generalizing approach. Furthermore, the MLN was designed to work with as few worlds as possible. At the same time, the weight calculation mentioned in Section 4.3 had to be applicable to be able to update existing formulas. Listing 4.25 shows the predicate declarations of the final MLN.

```
1 name(task,taskName!)
2 failure(task,failureName!)
3 goalPattern(task,goalPattern!)
4 goalParameter(designator,task!)
5 goalParameterKey(designator,goalParameter!)
6 designatorProperty(designatorProperty,designator?)
7 propertyKey(designatorProperty,designatorPropertyKey!)
8 propertyValue(designatorProperty,designatorPropertyValue!)
```

Listing 4.25: Predicate Declarations of the Final MLN

Again, the task itself is an object. The *name* property does not specify the task name, but the plan name. Each task must have a plan name since the *name* predicate is modeled using a functional constraint which is indicated by the quotation mark. It appears strange that a task must also have a *failure*, but this is a simple method for avoiding negated existential quantifiers: If there is a failure, the failure name is referenced. Otherwise, an empty string is used as failure name. The same is true for the Prolog annotation modeled as *gaolPattern*.

Similar to the previous MLN design, the designators appearing in the Prolog annotation can be mapped to the task using the predicate *goalParameter*. More precisely, the task is mapped to the designator in order to avoid a negated existential quantifier. The key specifying the parameter name is assigned to the designator in an extra predicate named *goalParameterKey*. As the MLN should generalize over different designator properties, it seems natural to model the designator properties as objects. Therefore, properties can be assigned to a designator with the *designatorProperty* predicate. Again, the designator is actually assigned to the property to avoid a negated existential quantifier. Each property can then have a *propertyKey* and a *propertyValue*.

The most difficult part is the formula design: On the one hand, the MLN should generalize. On the other hand, it should be possible to use the weight calculation in Section 4.3. A solution is to model the co-occurrence of properties as it is done in [ZFFF14]. However, in contrast to [ZFFF14] the formulas here use conjunctions as connective instead of implications. More specifically, for each combination of two properties of one designator, one formula and thus one conjunction is created. As in the last MLN design, the task specific information such as name, goal pattern and failures for one task is part of more than one formula. Apart from the task specific information, the conjunction contains the property specific atoms. In Listing 4.26 for example, the formulas for the *object-in-hand* task in Figure 4.2 are shown.

```
1  0.0 name(?t2,"ACHIEVE") ^ failure(?t2," ")
       ^ goalPattern(?t2,"OBJECT-IN-HAND ?OBJ") ^ goalParameter(?d3,?t2)
       ^ goalParameterKey(?d3,"?OBJ") ^ designatorProperty(?dp0,?d3)
       ^ propertyKey(?dp0,"object.AT::location.ON")
       ^ propertyValue(?dp0,"Cupboard") ^ designatorProperty(?dp1,?d3)
       ^ propertyKey(?dp1,"object.TYPE") ^ propertyValue(?dp1,"MUG")
2  0.0 name(?t2,"ACHIEVE") ^ failure(?t2," ")
       ^ goalPattern(?t2,"OBJECT-IN-HAND ?OBJ") ^ goalParameter(?d3,?t2)
       ^ goalParameterKey(?d3,"?OBJ") ^ designatorProperty(?dp0,?d3)
       ^ propertyKey(?dp0,"object.AT::location.NAME")
       ^ propertyValue(?dp0,"kitchen_sink_block")
       ^ designatorProperty(?dp1,?d3) ^ propertyKey(?dp1,"object.TYPE")
       ^ propertyValue(?dp1,"MUG")
3  0.0 name(?t2,"ACHIEVE") ^ failure(?t2," ")
       ^ goalPattern(?t2,"OBJECT-IN-HAND ?OBJ") ^ goalParameter(?d3,?t2)
       ^ goalParameterKey(?d3,"?OBJ") ^ designatorProperty(?dp0,?d3)
       ^ propertyKey(?dp0,"object.AT::location.NAME")
       ^ propertyValue(?dp0,"kitchen_sink_block")
       ^ designatorProperty(?dp1,?d3)
       ^ propertyKey(?dp1,"object.AT::location.ON")
       ^ propertyValue(?dp1,"Cupboard")
```

Listing 4.26: Example Formulas of a MLN Using the Final Design

The idea of this design is that only those training designators should be taken into account by the inference which have the most properties in common with the evidence. Through the (soft) functional constraints in combination with a restricted set of properties, only those formulas are satisfiable given the evidence which contain the property keys and values specified in the evidence. In Listing 4.27 for example, only formulas containing *propertyKey(MugProperty, "object.TYPE")* and *propertyValue(MugProperty, "MUG")* are satisfiable. When the number of property objects is increased and the new property object was also part of the training data, the number of true formulas grows rapidly since one property is part of more than one formula. Due to the constant component in the formula weights as specified by Section 4.3, a world having one more true ground formula than another one is far more probable. Hence, only those worlds, and thus those training examples, which are most similar to the evidence are considered during the inference. If there is a new property which did not occur in a training example, the property is ignored and there are still worlds with true formulas since there is no negation. Therefore, the approach generalizes.

```
P ( PropertyKey(OtherProperty, ?k) |
      name(Task, "ACHIEVE")
    ^ failure(Task, " ")
    ^ goalPattern(Task, "OBJECT-IN-HAND ?OBJ")
    ^ goalParameter(Designator, Task)
    ^ goalParameterKey(Designator, "?OBJ")
    ^ designatorProperty(MugProperty, Designator)
    ^ propertyKey(MugProperty, "object.TYPE")
    ^ propertyValue(MugProperty, "MUG")
    ^ designatorProperty(OtherProperty, Designator) )
```

Listing 4.27: Query for Completing a Designator Property Key with the Final MLN

It is noticeable that the property keys, for example *object.AT::location.ON*, are different compared to the property keys in the other MLNs. Moreover, there are no predicates for the designator type or for sub designators. The reason is that all sub designators are aggregated into one large designator. In this large designator, the boundary between the different sub designators is visible by two colons in the property key. To keep the designator type of both, the parent designator and the sub designator, the designator type is also encoded in the property key: It is represented by the string ending at the period. Thus, a small workaround is applied here.

Similar to the MLN design in the previous section, splitting the formulas also requires splitting the training database. Listing 4.28 shows one of the three training databases belonging to the MLN in Listing 4.26.

```
1  name("Achieve_AbC","ACHIEVE")
2  failure("Achieve_AbC"," ")
3  goalPattern("Achieve_AbC","OBJECT-IN-HAND ?OBJ")
4  goalParameter(123,"Achieve_AbC")
5  goalParameterKey(123,"?OBJ")
6  designatorProperty(1230,123)
7  propertyKey(1230,"object.TYPE")
8  propertyValue(1230,"MUG")
9  designatorProperty(1231,123)
10 propertyKey(1231,"object.AT::location.ON")
11 propertyValue(1231,"Cupboard")
```

Listing 4.28: One Training Database for a MLN Using the Final Design

Finally, there is a MLN design which meets the conditions for learning introduced in Section 1.1, in particular the generalization, and which is suited for the use cases in Figure 1.1. The first use case is to infer what is expected to be perceived at a specific location. For this inference, the robot has to know the designator properties. Listing 4.29 shows the query that the robot can use in the example to infer what to expect at the *kitchen_sink_block*.

```
P ( propertyValue(ObjectTypeProperty, ?v) |
      name(Task, "PERCEIVE-OBJECT")
    ^ failure(Task, " ")
    ^ goalPattern(Task, "A ?OBJ-DESIG")
    ^ goalParameter(Designator, Task)
    ^ goalParameterKey(Designator, "?OBJ-DESIG")
    ^ designatorProperty(OnProperty, Designator)
    ^ propertyKey(OnProperty, "object.AT::location.ON")
    ^ propertyValue(OnProperty, "Cupboard")
    ^ designatorProperty(NameProperty, Designator)
    ^ propertyKey(NameProperty, "object.AT::location.NAME")
    ^ propertyValue(NameProperty, "kitchen_sink_block")
    ^ designatorProperty(ObjectTypeProperty, Designator)
    ^ propertyKey(ObjectTypeProperty, "object.TYPE") )
```

Listing 4.29: Query inferring what to Expect on a Cupboard Using the Final MLN

The second use case describes that the robot wants to know the success probability for the execution of a parametrized plan. For the example, this probability is inferred in Listing 4.30. There, the success probabilities for the *achieve object-in-hand* plan given a *mug* on the *kitchen_sink_block* is inferred.

```
P ( failure(Task, " ") |
      name(Task, "ACHIEVE")
    ^ goalPattern(Task, "OBJECT-IN-HAND ?OBJ")
    ^ goalParameter(Designator, Task)
    ^ goalParameterKey(Designator, "?OBJ")
    ^ designatorProperty(OnProperty, Designator)
    ^ propertyKey(OnProperty, "object.AT::location.ON")
    ^ propertyValue(OnProperty, "Cupboard")
    ^ designatorProperty(NameProperty, Designator)
    ^ propertyKey(NameProperty, "object.AT::location.NAME")
    ^ propertyValue(NameProperty, "kitchen_sink_block")
    ^ designatorProperty(ObjectTypeProperty, Designator)
    ^ propertyKey(ObjectTypeProperty, "object.TYPE") )
    ^ propertyValue(ObjectTypeProperty, "MUG") )
```

Listing 4.30: Query for Failure and Success Probabilities Using the Final MLN

Finally, there is the use case to infer how to parametrize the *object-in-hand* task for a *mug* to be successful. To get to know this, it has to extend an existing designator. Therefore, the robot first has to infer the most likely designator key, which is done in Listing 4.27. When the key, for example *object.AT::location.ON*, has been inferred, the respective ground atom *propertyKey(OtherProperty, "object.AT::location.ON")* can be appended to the evidence of the next query. The next query is used to infer the most likely property value, as shown in Listing 4.31. This procedure can be repeated until the designator is completed. The conditions for completion are explained in detail in Section 6.3. A big advantage of this designator completion is that the robot does not need to know which keys are necessary. Hence, this feature can be used when processing incomplete instructions, for example gathered by natural language processing.

```
P ( PropertyValue(OtherProperty, ?v) |
      name(Task, "ACHIEVE")
    ^ failure(Task, " ")
    ^ goalPattern(Task, "OBJECT-IN-HAND ?OBJ")
    ^ goalParameter(Designator, Task)
    ^ goalParameterKey(Designator, "?OBJ")
    ^ designatorProperty(MugProperty, Designator)
    ^ propertyKey(MugProperty, "object.TYPE")
    ^ propertyValue(MugProperty, "MUG")
    ^ designatorProperty(OtherProperty, Designator)
    ^ propertyKey(OtherProperty, "object.AT::location.ON"))
```

Listing 4.31: Query for Completing a Designator Property Value with the Final MLN

It is worth mentioning that all those queries can easily be executed with the normal exact inference algorithm. Of course, the algorithm in Chapter 5 performs also very well on these queries.

In summary, different MLN designs have been developed. The last one is a compromise between intuitive modeling, simple learning and generalization. Nevertheless, it is well suited for all the use cases presented in the use case diagram in Figure 1.1.

5. Improved Inference

One problem of the design of the designator based MLN in Section 4.5 is the intractable inference. None of the inference algorithms described in Section 2.4.4 is able to perform an inference in these MLNs. For the exact inference, the reason is the huge amount of possible worlds. The equation for the exact inference, which had already been introduced in Equation 2.5, explains that:

$$P(Q|E,L,C) = \frac{\sum\limits_{x \in \mathcal{X}_{Q,L,C} \cap \mathcal{X}_{E,L,C}} \exp\left(\sum\limits_{i=1}^{m} w_i n_i(x)\right)}{\sum\limits_{x \in \mathcal{X}_{E,L,C}} \exp\left(\sum\limits_{i=1}^{m} w_i n_i(x)\right)} \tag{2.5}$$

If this formula is implemented, every world in which the evidence ground atoms are true must be enumerated. Concerning the MLN design in Section 4.5, this is somehow frustrating since one can easily estimate the probability manually. The thoughts leading to this claim are best explained by an example.

```
1  name(task, name!)
2  success(task)
3  designatorHash(designator,designatorHash!)
4  designatorProperty(designator,propertyKey,propertyValue!)
5
6  name={"ACHIEVE", "PERFORM"}
7
8  40.0  success(?t) ^ designatorHash(?d, "a3f")
        ^ designatorProperty(?d, "TYPE", "MUG")
9  40.69 !success(?t) ^ designatorHash(?d, "a3f")
        ^ designatorProperty(?d, "TYPE", "MUG")
10 40.0  success(?t) ^ designatorHash(?d, "4e7")
        ^ designatorProperty(?d, "TYPE", "APPLE")
```

Listing 5.1: Simplified Designator Based MLN

```
P ( success(T) | designatorProperty(D, "TYPE", "MUG") )
```

Listing 5.2: Query for the Success Probability of a Plan Parametrized with a Mug

Listing 5.1 shows a simplified designator based MLN. It can be used to calculate the success probability for a plan parametrized with a designator describing a mug, as shown in Listing 5.2. The only two formulas which are satisfyable given the evidence *designatorProperty(D, "TYPE", "MUG")* are the first ones. Moreover, the only formula satisfyable given the query *success(T)* and the evidence is the first one. Thus, the only weight used in the numerator is the one for formula 1, whereas the only weights used in the denominator are the first two weights. Therefore, the probability must be mainly influenced by these two formulas. Since the weight of the second one is larger, the resulting probability will approximately be:

$$\frac{\exp(40)}{\exp(40.69) + \exp(40)} \approx 0.334$$

In the example, several interesting thoughts have been applied: First of all, worlds compatible with the evidence but incompatible with all formulas do not need to be considered explicitly. In the example, some of these worlds are those where the ground atom *designatorHash(D, "4e7")* is true. Since no formula is true in these worlds, the sum in the exponentiation is 0 and the exponentiation becomes 1. Hence, only a 1 must be added to the numerator or the denominator for each of the worlds incompatible with all formulas.

Another interesting thought is that there might be different worlds where the same weights are applied. For example, the ground atoms *name(D, "ACHIEVE")* and *name(D, "PERFORM")* do not appear in any formula. Thus, there are different worlds where the first formula is true: one for each *name* atom. However, the exponentiated sum is equal for all of these worlds. In summary, the formulas are actually used to specify a set of worlds[1]: In this case, the worlds that can be outsourced from the actual calculation are specified.

From these principles, an inference algorithm for formulas consisting of conjunctions of literals evolved. Though inference in MLNs having the final deisgn is also possible with the normal exact inference algorithm, the algorithm is explained in the following sections since it might be usable in other areas in the future.

5.1. Algorithm Description

To explain the inference algorithm descriptively, the MLN in Listing 5.3 will be used. It is assumed that the MLN is grounded with the evidence shown in Listing 5.4. Thus, the worlds shown in Table 5.1 are possible. Apart from that, Table 5.1 also shows how often a formula is true. This information can be used to validate the results later on.

[1] This idea actually origins form the lecture "Theorie reaktivier Systeme" by Jan Peleska, where system states were specified by first-order logic.

```
1  foo(x)
2  bar(y)
3  baz(y,z)
4
5  40.0 foo(?v) ^ bar(?w)
6  40.0 !foo(?v)
7  40.0 baz(?v, ?w) ^ bar(?v)
```

Listing 5.3: Example MLN Used for Explaining the Inference Algorithm

```
1  foo(a)
2  x={a,b}
3  y={0}
4  z=α
```

Listing 5.4: Evidence Database for the MLN in Listing 5.3

foo(a)	foo(b)	bar(0)	baz(0, α)	!foo(?v)	foo(?v) $\wedge$ bar(?w)	baz(?v,?w) $\wedge$ bar(?v)
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	1	0
1	0	1	1	1	1	1
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	2	0
1	1	1	1	0	2	1

Table 5.1.: Worlds for the MLN in Listing 5.3

Figure 5.1 shows the entry point of the algorithm. First of all, the evidence is converted to a hard logical formula by conjugating the ground atoms. Since there is only one ground atom in the example, the resulting formula is *foo(a)*. Moreover, the query is conjugated with the evidence. In the example, this might result in *foo(a)* $\wedge$ *foo(b)* if the query is *foo(b)*. Then, a subroutine[2] is called to efficiently calculate both the numerator and the denominator of the following equation:

$$P(Q|E,L,C) = \frac{\sum\limits_{x \in \mathcal{X}_{Q,L,C} \cap \mathcal{X}_{E,L,C}} \exp\left(\sum\limits_{i=1}^{m} w_i n_i(x)\right)}{\sum\limits_{x \in \mathcal{X}_{E,L,C}} \exp\left(\sum\limits_{i=1}^{m} w_i n_i(x)\right)} \tag{2.5}$$

This is the equation for the exact inference introduced in Equation 2.5 first. After both, the numerator and the denominator have been calculated, the numerator is divided by the denominator and returned. Up to now, there is no difference to the normal implementation of the exact inference except from the fact that the evidence is treated as hard formula.

[2] Of course, in the real implementation the numerators for the different queries and the denominator are calculated together in one routine for efficiency reasons.

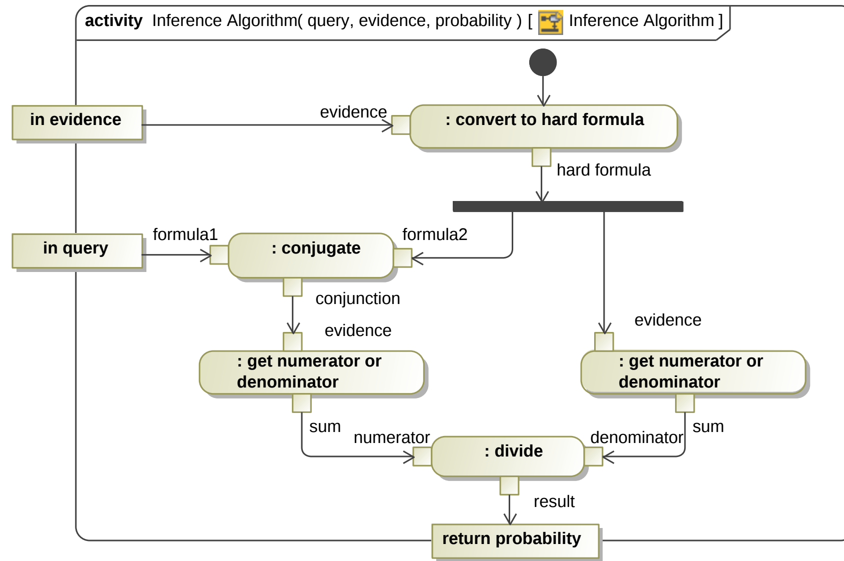


Figure 5.1.: Top Level Activities in the Inference Algorithm

Figure 5.2 shows how the numerator or the denominator is calculated. The steps shown in this diagram will be introduced in the following sections. Considering the example from Listing 5.3, it will be referred to the calculation of the denominator in the following. However, this calculation can easily be transferred to the numerator as well.

5.1.1. Get Ground Formulas and Conjugate

As in other inference algorithms, the first step is to generate ground formulas from the MLN formulas. This process is known as *grounding*. In the example, the following ground formulas are generated:

a) $!foo(a)$

b) $!foo(b)$

c) $foo(a) \wedge bar(0)$

d) $foo(b) \wedge bar(0)$

e) $baz(0, \alpha) \wedge bar(0)$

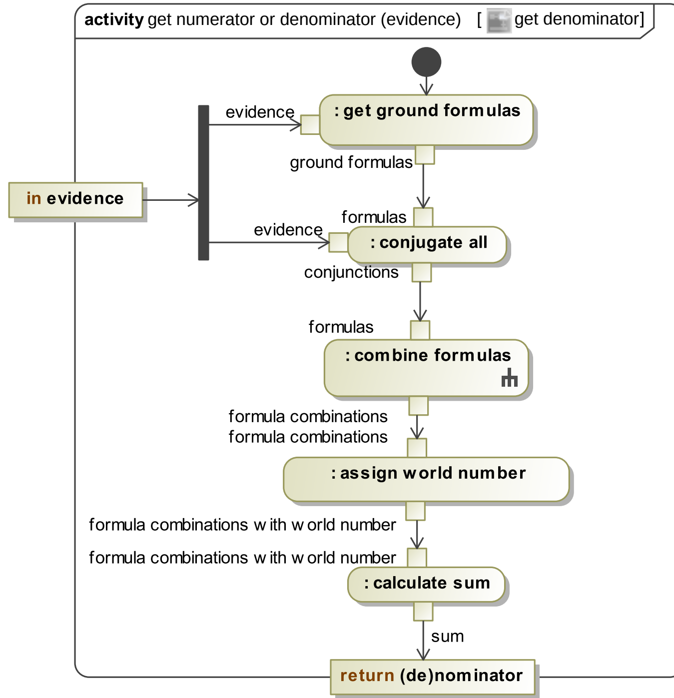


Figure 5.2.: Calculating the Numerator or Denominator in the Inference Algorithm

In general, the formula grounding is one of the bottle necks in the MLN inference. Therefore, specialized grounding algorithms are suited to pass this bottle neck. For the designator based MLN design from Section 4.5, the negated existential quantifiers are causing problems: They are actually expanded to a large disjunction. Through the equality comparison in the disjunction, too many values are generated. Moreover, the result needs to be simplified to get formulas consisting of conjunctions. Hence, a specialized grounder is implemented for negated existential quantifiers. This grounder simply determines the atoms not to negate and negates all other ground atoms of the given predicate. The results are a faster grounding and a simplified formula structure.

The final MLN design developed in Section 4.6 does not contain negated existential quantifiers. However, the grounding process is still slow. The reason are formulas with several designator properties. A naive grounding algorithm assigns one of the properties in the domain to each property variable. This assignment is not necessarily satisfiable given the evidence. If, for example, the evidence states that *propertyKey(MugProp, "MUG")* is true and the literal *propertyKey(?dp1, "APPLE")* is grounded, a naive algorithm would try *propertyKey(MugProp, "APLLE")* although the functional constraint in the *propertyKey* predicate forbids that. Thus, the grounding process is non trivial even for conjunctions.

The pracmln framework already contains an improvement for conjunctions: It grounds (soft) functional literals first and it returns only those groundings which are true given the evidence. However, it still tries too many possibilities since it does not take variables occurring in two literals into account. An example is the formula in line 8 of Listing 5.3: If the ground formulas given the constants in Listing 5.5 should be determined, the normal grounding algorithm would first select a literal, e.g. *designatorHash(?d, "a3f")* and assign a value to the variable *?d*, e.g. *D3*. Then, it would select the next literal, e.g. *designatorProperty(?d, "TYPE", "MUG")*, detect that the variable value *D3* is incompatible to the evidence and backtrack. This can be very inefficient, especially for the final MLN design. Therefore, a further improved variant applicable for conjunctions consisting of atoms has been developed.

```
P ( success(T) |
      designatorHash(D1, "4e7")
    ^ designatorProperty(D2, "TYPE", "MUG")
    ^ designatorHash(D3, "a3f")
    ^ designatorProperty(D3, "TYPE", "APPLE") )
```

Listing 5.5: Advanced Success Query in Listing 5.1

In a first step, all possible ground atoms of a predicate given the evidence are determined. Table 5.2 shows the results for the MLN in Listing 5.3 and the constants in Listing 5.5.

designatorHash
designatorHash(D1, "4e7")
designatorHash(D2, "a3f")
designatorHash(D2, "4e7")
designatorHash(D3, "a3f")

designatorProperty
designatorProperty(D1, "TYPE", "MUG")
designatorProperty(D1, "TYPE", "APPLE")
designatorProperty(D2, "TYPE", "MUG")
designatorProperty(D3, "TYPE", "APPLE")

success
success(T)

Table 5.2.: Possible Ground Atoms of the Predicates from Listing 5.1

Then, the atoms of the conjunction are sorted by the number of ground atoms. In Table 5.2, the *success* atom would be grounded first. Since the variable T is only contained in this atom, the atom is grounded normally. The other two predicates both have four possibly true ground atoms. Thus, their order is not changed. Finally, the possible variable values are determined by an intersection of the possible variable values of the different atoms. They are in turn determined by the ground atoms. In the *designatorHash(?d, "a3f")* atom, the variable values for *?d* are determined by the left table of Table 5.2: Only *D2* and *D3* are possible. The possible variable values for *?d* in the *designatorProperty(?d, "TYPE", "MUG")* atom are *D1* and *D2*, as the right table shows. Thus, the only possible assignment for *?d* is $\{D2, D3\} \cap \{D1, D2\} = \{D2\}$. This leads to an improved performance, especially in the final MLN design from Section 4.6.

Now that the ground formulas have been generated, they must be conjugated with the evidence. Since the evidence is also in the form of a logical formula, the conjugation process is easy. It is important that hard formulas are treated like evidence formulas and thus they are also part of this conjunction. In the example in Listing 5.3, the following formulas are the result of the conjugation process:

a) $!foo(b) \wedge foo(a)$

b) $foo(a) \wedge bar(0) \wedge foo(a)$

c) $foo(b) \wedge bar(0) \wedge foo(a)$

d) $baz(0, \alpha) \wedge bar(0) \wedge foo(a)$

This enumeration contains one formula less than the enumeration of the ground formulas above. The reason is that $!foo(a) \wedge foo(a)$ contains a contradiction. Thus, it is omitted. Another special case are hard formulas which are no conjunctions. They have to be in the disjunctive normal form. Then, one conjunction is created from each formula and each part of the disjunction. In the end, the conjunctions created here will lead to a decreased number of worlds to consider.

5.1.2. Combine Formulas

The most important part in the algorithm is the combination of the formulas with each other. In a nutshell, all combinations of an arbitrary number of formulas generated in the step before are generated until a fixed point is reached. Table 5.3 lists the combination creatable from the example.

Index	Formula	Maximum Number of Worlds
a)	$!foo(b) \wedge foo(a)$	4
b)	$foo(a) \wedge bar(0)$	4
c)	$foo(b) \wedge bar(0) \wedge foo(a)$	2
d)	$baz(0, \alpha) \wedge bar(0) \wedge foo(a)$	2
e)	$foo(a) \wedge bar(0) \wedge !foo(b)$	2
f)	$baz(0, \alpha) \wedge bar(0) \wedge foo(a) \wedge foo(b)$	1
g)	$baz(0, \alpha) \wedge bar(0) \wedge foo(a) \wedge !foo(b)$	1

Table 5.3.: Combined Ground Formulas of the MLN in Listing 5.3

One can imagine that a naive implementation is computationally very expensive. Therefore, special algorithms are needed again. Since the development of these algorithms is quite complex, the problem is reduced to another problem instead. Figure 5.3 shows how the reduction works.

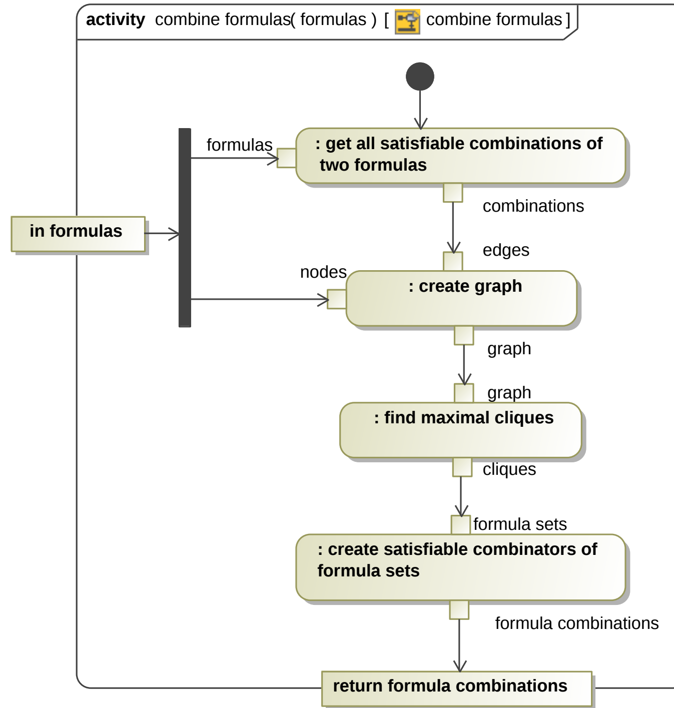


Figure 5.3.: Combining Formulas in the Inference Algorithm

Initially, all combinations of two formulas are generated. From these combinations, a graph is generated: The nodes are the formulas and an edge is created if two formulas can be combined. Now, existing algorithms[3] can be used to find maximal cliques in this graph. Afterwards, only the combinations of the maximal cliques have to be generated. The graph for the example is shown in Figure 5.4. It's cliques are $\{\{a, b\}, \{a, d\}, \{b, c, d\}\}$ and thus only the combinations of these cliques have to be generated. This is an improvement, since the check whether a combined formula is satisfiable is computationally expensive, at least in pracmln. Some of these checks are removed by the algorithm.

5.1.3. Assign World Number

One thing that has not been discussed previously is the column *Maximum Number of Worlds* in Table 5.3: As mentioned before, formulas can be used to specify sets of worlds in which they are true. Hence, the column specifies the number of worlds in which the formula combination is true. However, one world might be referenced by two formula combinations. Since the formula combinations are used to specify sets of worlds later on, a unique assignment of worlds to formulas is needed.

[3]In the implementation, the networkx package is used: https://networkx.github.io/

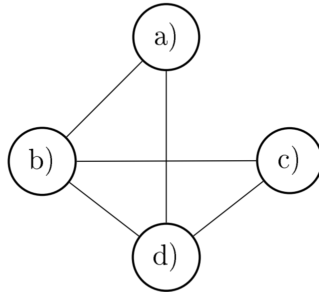


Figure 5.4.: Formula Graph for the Ground Formulas of Listing 5.3

The solution is to determine which combination contains worlds of which other combination. In Table 5.3 for example, the combination $baz(0, \alpha) \wedge bar(0) \wedge foo(a)$ contains the combination $baz(0, \alpha) \wedge bar(0) \wedge foo(a) \wedge !foo(b)$ since the former is also true in the only world in which the latter is true (Line 4 in Table 5.1). The purpose of the containment relation is to assign a world to the most special combination which is true in this world. Starting at those formulas containing no other formulas, the containment can be calculated. Afterwards, the number of worlds can be calculated by subtracting the number of worlds of the contained formulas from the maximum number of worlds. Table 5.4 shows the result for the example. It is important to notice that this process guarantees that no world is assigned to two combinations[4].

Index	Contains	Generated Worlds
g)	-	1=1
f)	-	1=1
e)	f	2-1=1
d)	f,g	2-1-1=0
c)	g	2-1=1
b)	c,d,e,f,g	4-4=0
a)	e,f	4-2=2
Sum		6

Table 5.4.: Containment of the Formula Combinations in Table 5.3

[4]Assume there is a world assigned to two combinations. Then, there would be a combination of these combinations to which the world would be assigned. Thus, the world would not be assigned to both of the formulas, which is a contradiction.

5.1.4. Calculate Sum

Now that there is a unique assignment of world numbers to formulas, the denominator can be calculated. One trick that will be used to do so is shown in the following equation:

$$\sum_{x \in \mathcal{X}} \exp(\sum_{i=0}^{|F|} w_i n_i(x)) = |X_f| + \sum_{x \in X_t} \exp(\sum_{i=0}^{|F|} w_i n_i(x)) \tag{5.1}$$

$$\begin{aligned} &\text{with } X_f = \{x' | x' \in \mathcal{X} \wedge \nexists f \in F : x' \models f\} \\ &\text{and } \;\; X_t = \{x' | x' \in \mathcal{X} \wedge \exists f \in F : x' \models f\} \end{aligned} \tag{5.2}$$

All the worlds for which the exponentiated sum is 0 are represented by X_f. Thus, the sum has to be calculated over less worlds X_t. To use the formula combinations C instead of the worlds $\mathcal{X}$, this expression can be further refined:

$$|X_f| + \sum_{x \in X_t} \exp(\sum_{i=0}^{|F|} w_i n_i(x)) = |X_f| + \sum_{c \in C} a(c) \cdot \exp(\sum_{g(c)} g) \tag{5.3}$$

In this case, $g(c)$ represents the set of weights of the ground formulas which are part of the formula combination c. $a(c)$ represents the actual world counts calculated above. Hence, only the formula combinations must be enumerated. In the example, the following calculation results:

$$\begin{aligned} \sum_{x \in \mathcal{X}_{E,L,C}} \exp(\sum_{i=0}^{|F|} w_i n_i(x)) &= |X_f| + \sum_{c \in C} a(c) \cdot \exp(\sum_{g(c)} g) \\ &= (2^3 - 6) \\ &+ 1 \cdot \exp(w_b + w_c + w_e) \\ &+ 1 \cdot \exp(w_c + w_d + w_e) \\ &+ 1 \cdot \exp(w_b + w_c) \\ &+ 0 \cdot \exp(w_c + w_e) \\ &+ 1 \cdot \exp(w_c + w_d) \\ &+ 0 \cdot \exp(w_c) \\ &+ 2 \cdot \exp(w_b) \end{aligned}$$

w_b, w_c, w_d and w_e represent the weights of the ground formulas. They are actually all 40 in Listing 5.3. If one compares the calculation above with the true ground formulas in Table 5.1, it gets clear that the calculation is correct. Afterwards, the sum is returned and can then be used to calculate the actual probability, as described in Figure 5.1.

5.2. Applicability

Though the algorithm is applicable in any MLN that uses formulas consisting of conjunctions, it might work either better or worse than the normal exact inference. However, this is highly dependent on the MLN. The algorithm is especially efficient if there are only few ground formulas which are true given the evidence and if there are only few combinations of true formulas. This is the case for the MLNs in Section 4.5, where it is the only applicable inference method compared to the algorithms mentioned in Section 2.4.4.

However, the algorithm reduces some of its work to the search for maximal cliques. The clique problem in turn is known to be NP complete [Sch08]. Thus, the algorithm is at least NP complete in the number of formula combinations. In practice, there are other bottlenecks which are computationally even more expensive, such as the formula consistency check.

To sum it up, a new inference algorithm was developed. It works by generating world counts from formula combinations. This algorithm is especially useful for the designator based MLN design developed in Section 4.5, but also applicable in the final MLN design described in Section 4.5.

6. Implemented Software

A good MLN design and an efficient inference algorithm are very important steps towards a lifelong learning robot. However, this is not enough: First of all, the logs must be converted to a MLN automatically. Then, MLN queries have to be executable from within a CRAM plan. To keep the plans as simple and maintainable as possible, the code accessing the MLNs should be on a high level. That means that routines to convert CRAM constructs, such as designators, to a MLN database are needed. Furthermore, it is desirable to encapsulate common queries, such as the queries for the designator completion. Therefore, software has been developed. This chapter is dedicated to give an overview of this software in order to use and maintain it later on. Before some interesting algorithms are explained, an overview of the components needed to create a lifelong learning robot is presented.

6.1. Overview

Figure 6.1 and Figure 6.2 show a refined version of the activity diagram from Figure 2.10. It depicts which components, implemented mainly in the form of ROS packages, are involved in a lifelong learning robot. The communication relationships among these components are shown in Figure 6.3. Files which are written by one component and read by another component, as well as ROS messages exchanged between two components, are displayed as IO flows. ROS services are displayed as provided or required interfaces in the lollipop notation.

The CRAM plans do not actually trigger the logging of function calls explicitly. Instead, macros such as *def-cram-function*, which are used to define the plan functions, call hooks. These hooks trigger Lisp functions located in the ROS package *cram_beliefstate*, which are sending the logging information to *semrec*. They communicate using the *operate* service of type *DesignatorCommunication* offered by *semrec*. Then, a running node in the *semrec* ROS package creates the OWL file and sends the designators as ROS message to a running *mongodb_log* node. This node stores the designators in the MongoDB via its C++ API. Therefore, the logging mechanism is actually composed of the components *semrec*, *cram_beliefstate* and *mongodb_log*.

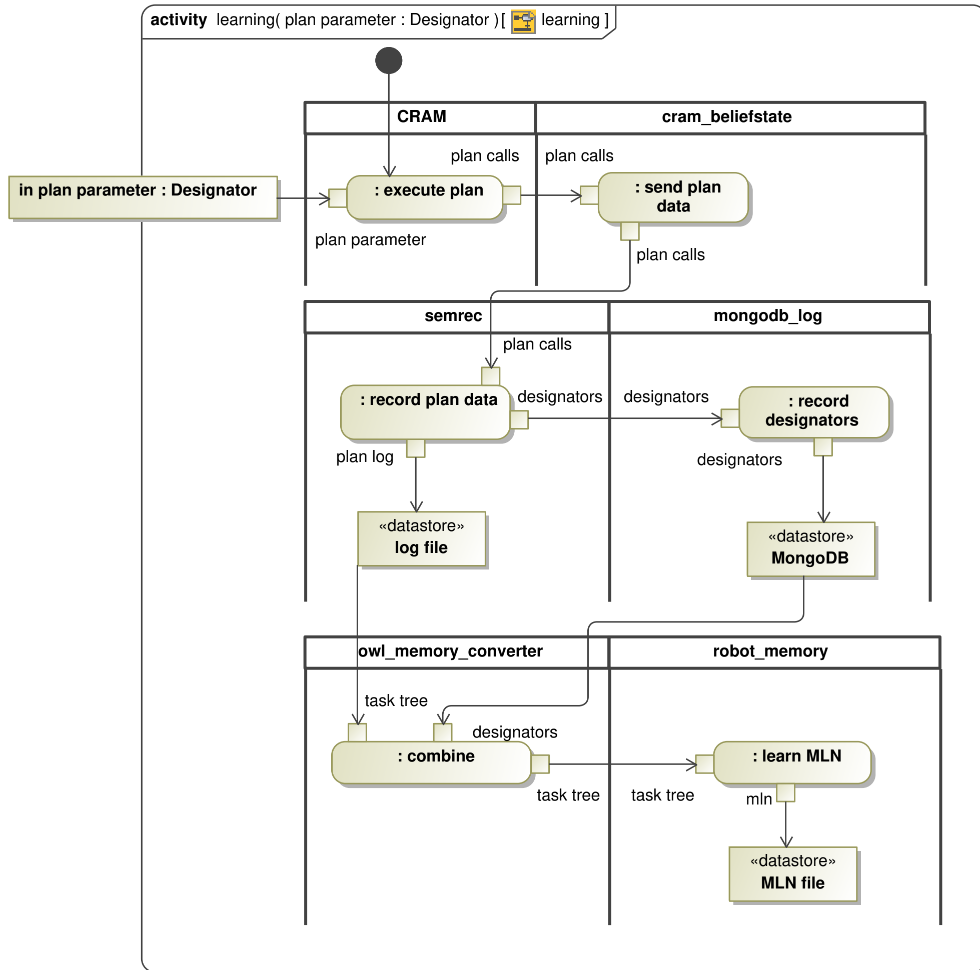


Figure 6.1.: Assignment of ROS Packages to the Learning Part of the Thesis Concept

The ROS package *owl_memory_converter* is responsible for combining log files and designators to a task tree again. It uses the *OWL API*[1] for parsing the OWL files. Though the *owl_memory_converter* is implemented in Java, the *MongoDB C API*[2] is used for extracting the designators from the MongoDB. This is required since the MongoDB Java API[3] is unable to cope with duplicate empty keys used to store designators in some cases.

[1] http://owlapi.sourceforge.net/
[2] https://api.mongodb.com/c/current/
[3] https://api.mongodb.com/java/

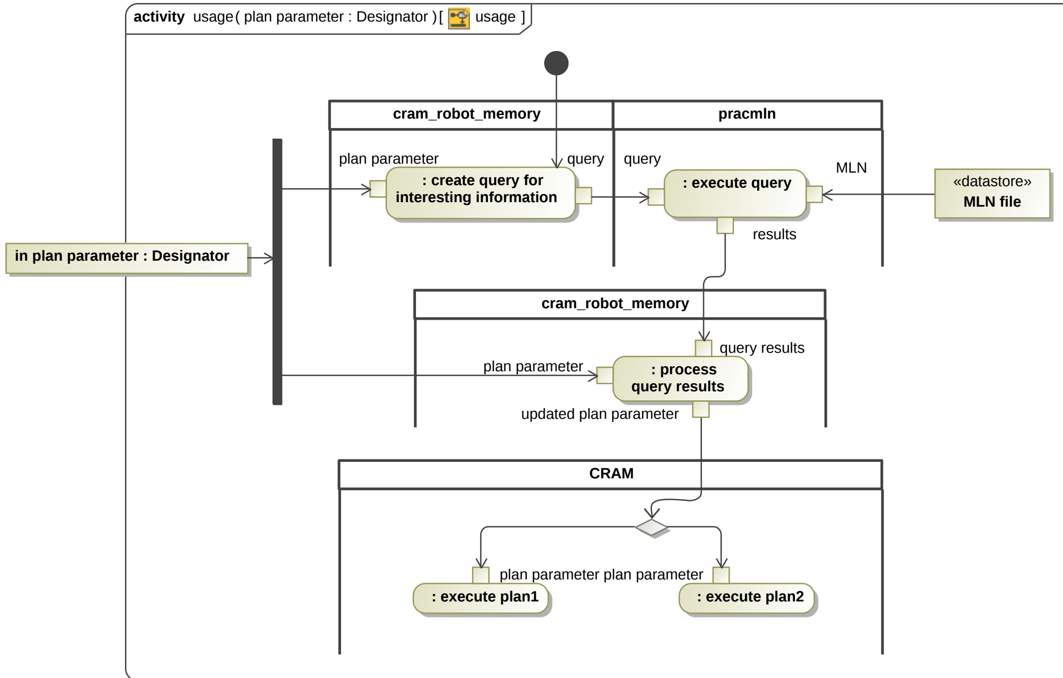


Figure 6.2.: Assignment of ROS Packages to the Inference Part of the Thesis Concept

The *owl_memory_converter* provides a ROS node able to publish the tasks and designators in the form of *RobotState* messages to a ROS topic. Before this is done, a *Trigger* service is called to signalize that messages will be sent. Afterwards, a *LearningTrigger* service is called to signalize that the learning can start. The mentioned messages are actually located in the *task_tree_messages* ROS package. To start the extraction of the OWL file, the file name can be passed to the *owl_memory_converter* via the command line or via the *Conversion* ROS service. Apart from that, the *owl_memory_converter* contains a utility to visualize the task tree as shown in Figure 2.6. Thus, the *owl_memory_converter* encapsulates the calls to the MongoDB and the OWL file.

The conversion of the task tree from the *owl_memory_converter* to a MLN is done in a node from the *robot_memory* ROS package. Its functionality is explained in Section 6.2 in greater detail.

After the MLN has been learned or updated by the *robot_memory* node, it can be queried. For this purpose, the *cram_robot_memory* ROS package provides a comfortable Lisp API. It encapsulates the MLN queries. Section 6.3 describes the *cram_robot_memory* package in detail.

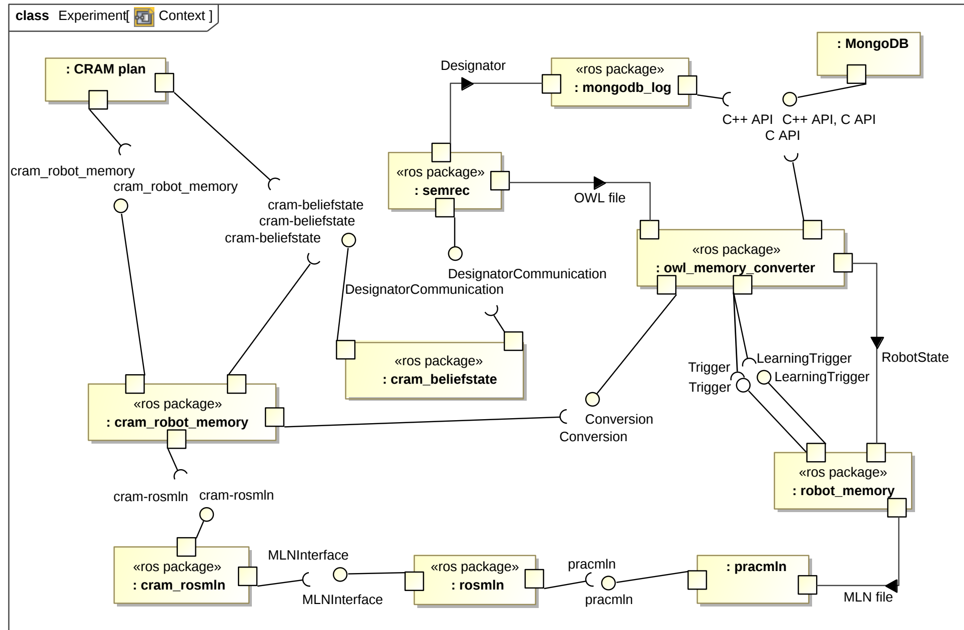


Figure 6.3.: Communication Relationships Between the ROS Packages

The *pracmln* framework introduced in Section 2.4.7 is used to execute MLN queries. In the version used here, an implementation of the inference algorithm developed in Chapter 5 is included. It is named *FastExact* in the GUI. For querying *pracmln* from ROS, *pracmln* provides a ROS interface in the ROS package *rosmln*. It is available in the form of a ROS service of the type *MLNInterface*, provided by nodes of the type *mln_server*. To make the use of this service easier, a Lisp API for accessing this service is provided in the ROS package *cram_rosmln*.

Figure 6.4 depicts the dependencies between the different ROS packages and explains which software is implemented in the scope of this thesis. Noticeably, there are ROS packages not shown in other diagrams. The reason is that *cram_robot_memory_demo* and *cram_robot_memory_evaluation* provide CRAM plans using the APIs provided by the *cram_robot_memory* package. Hence, they are represented by the *CRAM plan* in these diagrams. *cram_robot_memory_demo* actually provides a small number of demonstrations of the *cram_robot_memory* functionality while *cram_robot_memory_evaluation* contains the code for the evaluation conducted in Section 7.1. Both use utilities, e.g. to spawn a kitchen in the bullet environment, from the *cram_robot_memory_test_utils* package. They are adapted from the utilities in the *spatial_relations_demo*.

The following chapters will describe the most important components in greater detail.

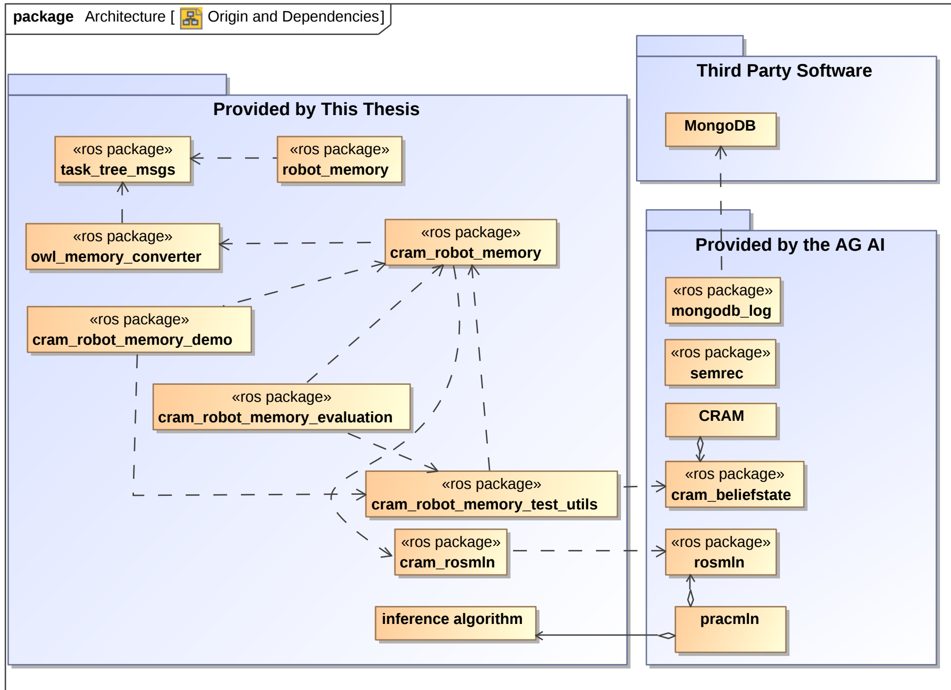


Figure 6.4.: Origin of the Different ROS Packages

6.2. robot_memory

As mentioned above, the *robot_memory* ROS package is responsible for creating or updating a MLN given a task tree. To achieve this, the package contains a python executable providing a service of the type *Trigger*. As soon as the service is called, the ROS node listens for *RobotState* ROS messages. When the calling node has finished sending the *RobotState* messages, it calls the *LearningTrigger* service which is also advertised by the *robot_memory* node. Then, the robot starts learning MLNs.

The learning process is illustrated in Figure 6.5. First of all, the task tree has to be assembled from the ROS messages. Afterwards, unneeded task tree nodes are filtered out. On the one hand, designators with multi lined parameters are filtered out since these parameters are usually not useful. Then, nodes having the task context *with-failure-handling*, *with-designators*, *perform-action-designator*, *anonymous-top-level*, *motion-planning*, *find-objects*, *at-location*, *goal-monitor-action* or *goal-perform* are filtered out. The reason is that these nodes are either redundant (e.g. *with-designators*), they contain no useful information (e.g. *anonymous-top-level*) concerning the use cases or they are too detailed (e.g. *goal-perform*). Moreover, poses which are absolute regarding the *map* tf frame are replaced by the string *<hard-coded-pose>* since a hard coded pose is too special. These first steps result in the task tree that will be used for learning.

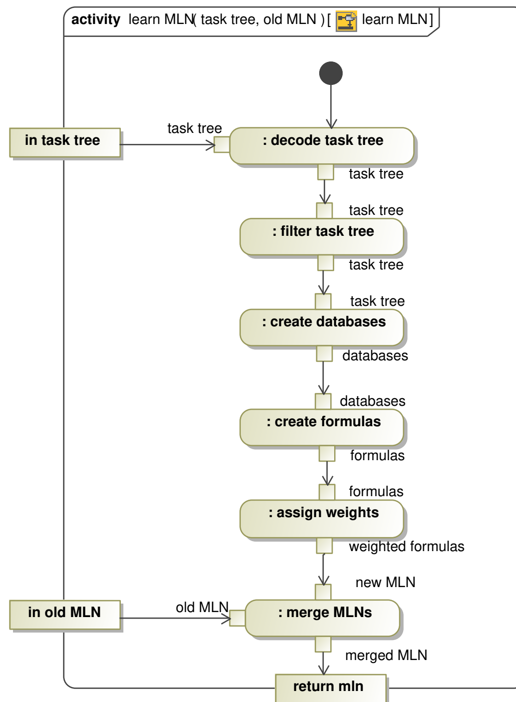


Figure 6.5.: Conversion of Task Trees to MLNs

Now, a MLN having the design described in Section 4.6 is created[4]. Therefore, the task tree is converted to a set of MLN training databases. Then, the databases are converted to formulas by conjugating the ground atoms and replacing the *task*, *designator*, and *designatorProperty* objects by variables. During the whole process, it is assured that one formula is created for each combination of two designator properties. Thus, more than one formula with the same task name might be created.

The next step is to calculate the formula weights. In the final MLN, duplicate formulas will be removed. However, the number of duplicates is used for the weight calculation. Based on the number of duplicates, the weight is assigned as described in Section 4.3. Finally, if an old MLN exists, both MLNs are merged: If the merged MLN contains duplicate formulas with the weights w_1 and w_2 using c as constant, the weight of the merged formula becomes $w = c + \ln(\exp(w_1 - c) + \exp(w_2 - c))$. The final MLN is then written to a file and the control flow returns to the caller of the *LearningTrigger*.

This process allows a lifelong learning through incrementally updated formula weights, even if the expressiveness of the MLN formulas is limited.

[4]Actually, two MLNs are created: One does not contain task information which is better suited if the task information is neither present in the query nor in the evidence.

6.3. cram_robot_memory

One important requirement for the software is that it is easily usable from CRAM. Therefore, the ROS package *cram_robot_memory* provides Lisp code encapsulating most of the MLN queries for the use cases mentioned in Figure 1.1. The most basic function provided by the package is one allowing different queries. To be usable for every query, MLN queries in the form of ground atoms or formulas still have to be passed as arguments to this function. However, it converts task names and designators to ground atoms for the evidence database automatically, if needed. It can for example be used to query which object type has been perceived at a specific location. Based on that function, there is a function for comfortably inferring a probability distribution for the task success without providing any literal. Hence, there are functions usable in two of the three use cases mentioned in Figure 1.1.

The third use case mentioned in the use case diagram is the inference of parameters causing a plan to be successful. For this inference, it is not sufficient to simply encapsulate one query. Instead, the designator completion broached in Section 6.3 has to be implemented. Figure 6.6 shows how this designator completion works. It will be explained using an example.

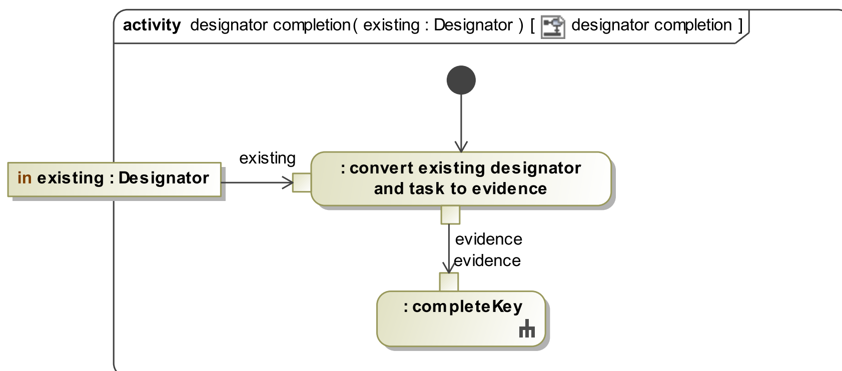


Figure 6.6.: The Designator Completion Algorithm

In the beginning, task name, parameter name and the existing designator properties are converted to a MLN database. As example, the database in Listing 6.1 will be used. Afterwards, this evidence is used to complete the key as shown in Figure 6.7. This is achieved with the query *propertyKey(OtherProperty, ?p)*, which has already been shown in Listing 4.27. The result is a probability distribution. This distribution could for example state that the key is *object.AT::location.ON* with probability $\frac{2}{3}$ or that the key is *object.AT::location.NAME* with probability $\frac{1}{3}$. Obviously, there are different probabilities for different keys and none has been tried yet, so the execution is neither aborted nor complete. Thus, the first key from this probability distribution is taken. Together with the evidence, it is used to call the *completeValue* function depicted in Figure 6.8

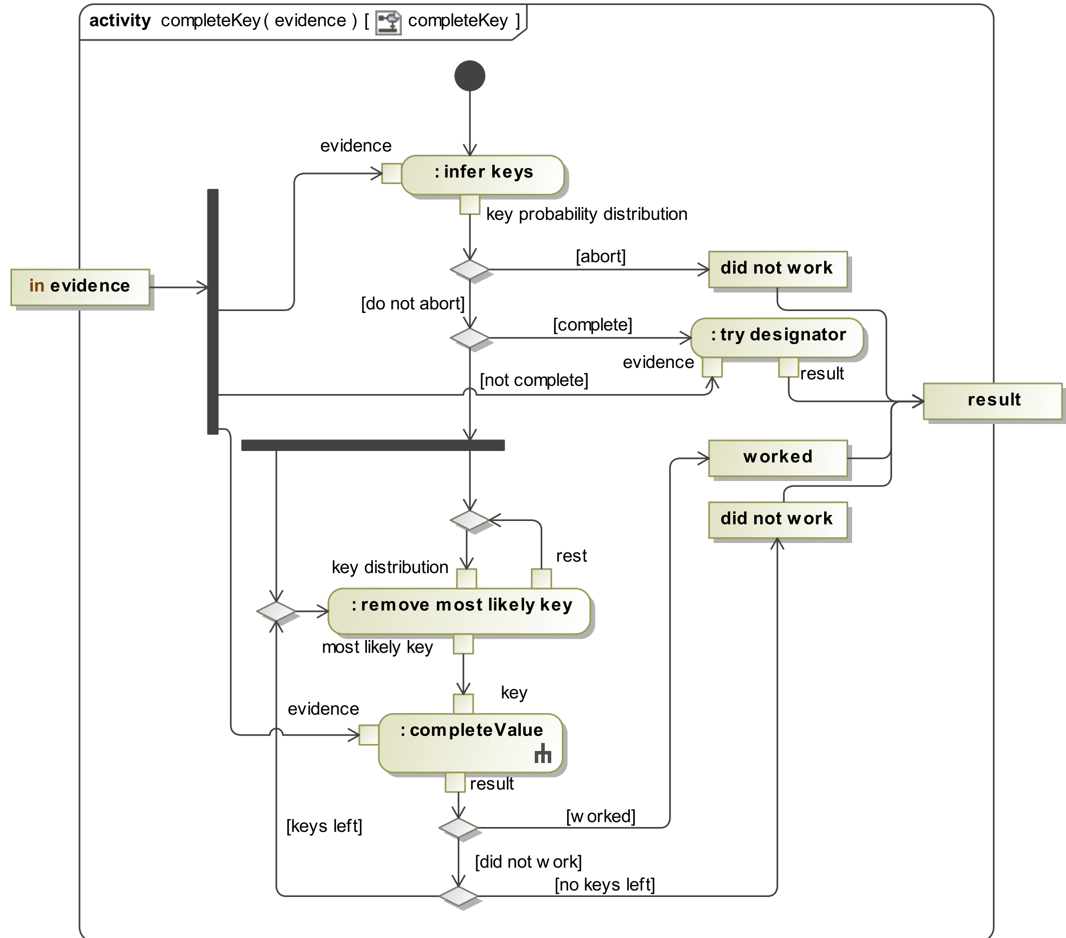


Figure 6.7.: The completeKey Function of the Designator Completion

```
1  name(Task, "ACHIEVE")
2  failure(Task, " ")
3  goalPattern(Task, "OBJECT-IN-HAND ?OBJ")
4  goalParameter(Designator, Task)
5  goalParameterKey(Designator, "?OBJ")
6  designatorProperty(MugProperty, Designator)
7  propertyKey(MugProperty, "object.TYPE")
8  propertyValue(MugProperty, "MUG")
9  designatorProperty(OtherProperty, Designator) )
```

Listing 6.1: Evidence for the propertyKey Query in Listing 4.27

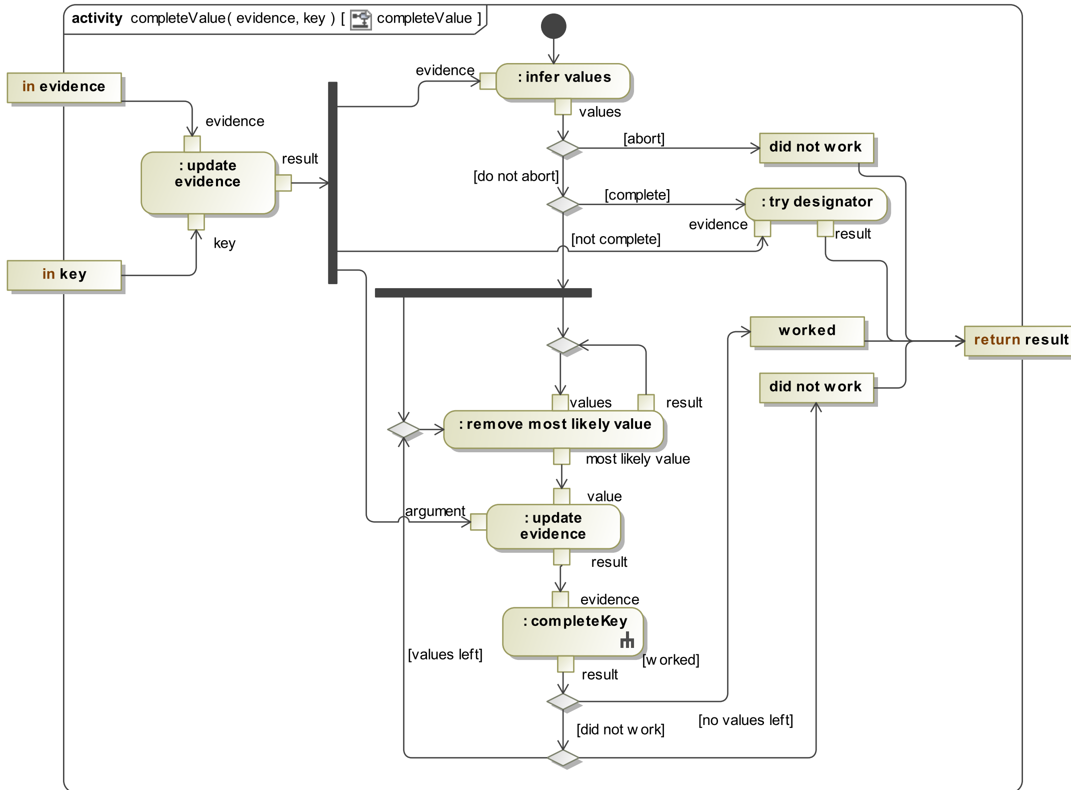


Figure 6.8.: The completeValue Function of the Designator Completion

The first thing the *completeValue* function does is merging the evidence with the ground atom *propertyKey(OtherProperty, "object.AT::location.ON")*. Then, it infers the most probable value for this key with the query *propertyValue(OtherProperty, ?v)*. This query has already been shown in Listing 4.31. Again, the result is a probability distribution. For the example, it is assumed that the most probable value in this distribution is *Cupboard* with probability $\frac{3}{4}$ and the second most probable is *CounterTop* with probability $\frac{1}{4}$. As in the key completion, there are different probabilities for different values and none has been tried yet, so the execution is neither aborted nor complete. Hence, *propertyValue(OtherProperty, "Cupboard")* is appended to the evidence. Then, the algorithm recurses and calls the *completeKey* function again.

In the next step, it is assumed that *completeKey* infers a probability of 1 for the key *object.AT::location.NAME*. Afterwards, *completeValue* is called and infers a probability of $\frac{3}{5}$ for *kitchen_sink_block* and a probability $\frac{2}{5}$ for *pancake_table*. This results in a recursive call to *completeKey* with an updated evidence. This time, the most probable key is *object.TYPE*. Again, there is a recursive call to *CompleteValue*. Finally, *completeValue* infers a probability of 1 for *MUG*. Since this property is already present in the designator, the completion seems to be finished.

Therefore, the evidence is converted to a designator. This designator is used to call a given function, for example to pick up the object. It is assumed now that this did not work. Thus, the algorithm tracks back and assigns the *pancake_table* instead of the *kitchen_sink_block* as property value. Again, the completion is continued and detects that the designator is complete. This time, picking up the object is successful and the algorithm terminates. This example showed how a completion works in principle.

What has been left out are the conditions for aborting or completion. Among other conditions, a designator is considered complete if:

- a property is inferred that is already contained in the designator.
- no more keys or values are available.
- there is a uniform distribution of the remaining keys or values.

The recursion is backtracking ("aborted") if:

- the designator seems to be completed, but a designator with the same properties or a superset of the properties has already been tried before.
- The result would be a hard coded pose.

This completes the description of the designator completion algorithm.

In summary, with the general query, the query for the success probability and the designator completion, the robot can easily cope with the situations depicted in the use case diagram in Figure 1.1. Finally, the designator completion shows one advantage of probabilistic models: It is possible to use the next probable value if the previous failed.

7. Evaluation

This chapter assesses how well the software mentioned in the previous chapter and thus the MLN design and the inference algorithm work in practice. Therefore, the suitability for the use cases in Figure 1.1 is examined. Furthermore, the opportunities of the algorithms and the MLN design, as well as their limitations are discussed.

7.1. Experiments

To check whether the software described in Chapter 6 and thus the MLN design in Section 4.6 is suited for the use cases in Figure 1.1, an experimental evaluation inspired by the one in [WB15] has been executed. The scenario in the experiments consists of a simulated PR2 robot operating in different kitchen environments. There, the PR2 is learning and updating a MLN from log files recorded while it is told to displace objects in the kitchens. Afterwards, different tests representing the use cases are executed.

7.1.1. Experiment Description

More specifically, the robot operates in the bullet reasoning environment introduced in Section 2.3.2. It is supposed to learn a model for the storage location of different objects. Therefore, 25 different kitchens are generated. Actually, the kitchen itself stays the same: It is the kitchen representing the IAI Lab at the University of Bremen provided by CRAM. Instead, the kitchens differ in the objects in the kitchen, as well as their location.

One can distinguish the 25 kitchens between 20 training kitchens and 5 test kitchens as shown in Figure 7.1. As mentioned before, each of these kitchens consists of 10 objects. Each object in the kitchen has a pose where it is spawned in the simulation environment. Moreover, there is a training object designator for each of the objects. This training object designator consists of a type property describing the object type (e.g. *:type :mug*) and of a location designator. The location designator in turn consists of a location property (e.g. *:on "Cupboard"*) and, if there is more than one location of this type, a name property (e.g. *:name "kitchen_sink_block"*). In the test kitchens, the object designator also contains a dummy property (e.g. *:dummy-key-1 "dummy-value-1"*) used to show that the MLN generalizes. Apart from the training object designator, the test kitchens contain a test object designator which is equal to the training object designator except that it does not contain the location designator.

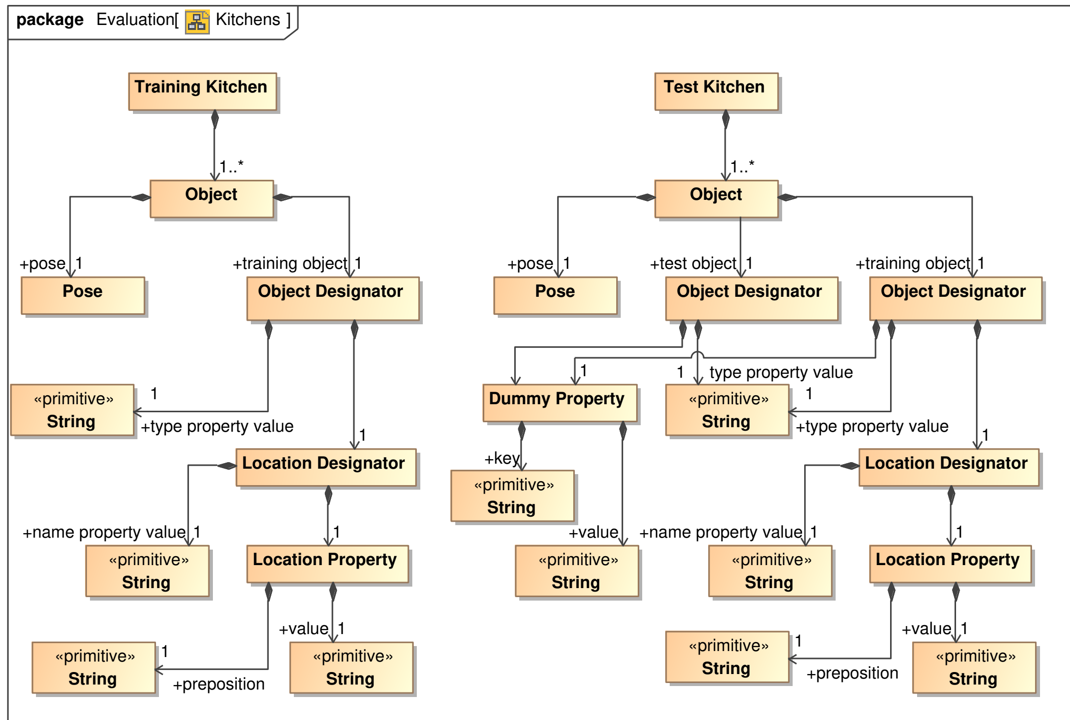


Figure 7.1.: Training and Test Kitchens Used in the Evaluation

Examples for training and test kitchens are provided in Listing 7.1 and Listing 7.2. These examples also show why the dummy property shows the generalization: When the MLN is used to execute a query given the test object at the bottom of Listing 7.2 as evidence, useful results are returned, even if an object with the dummy key and the dummy value is not part of the training data. The objects of the first training kitchen in the bullet environment are shown in Figure 7.2.

```
'(((1.4477 0.8845 0.9500)(0 0 0 1))
  ,(cram-designators::make-designator
      :object '((:type :apple)
                (:at ,(cram-designators::make-designator
                         :location
                         '((:on "CounterTop")
                           (:name "kitchen_sink_block_counter_top")))))))
```

Listing 7.1: Example Object of a Training Kitchen

```
`(((1.4044 -0.1155 0.9500)(0 0 0 1))
  ,(cram-designators::make-designator
      :object `((:type :apple)
                (:dummy-key-0 "dummy-val-3")
                (:at ,(cram-designators::make-designator
                         :location
                         '((:on "CounterTop")
                          (:name "kitchen_sink_block_counter_top"))))))
  ,(cram-designators::make-designator
      :object `((:type :apple)
                (:dummy-key-0 "dummy-val-3"))))
```

Listing 7.2: Example Object of a Test Kitchen

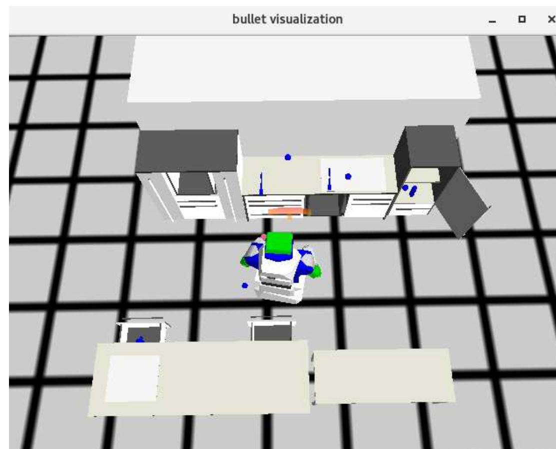


Figure 7.2.: PR2 Robot in the first Training Kitchen

The kitchens described above are sampled randomly[1]: During the generation, the object type is sampled uniformly from the set of objects in Table 7.1. Then, the dummy key, as well as the dummy value, are sampled randomly from a set of 5 keys and a set of 5 values. Moreover, the location type is sampled uniformly from the location types available for this object in the table. The actual location name is then sampled uniformly from the subset of location names in Table 7.2 having the correct location type. Finally, the position part of the pose is sampled uniformly from the coordinate range for the location name specified by Table 7.2. The orientation is always specified by the quaternion $(0, 0, 0, 1)$.

Since the CRAM plans for opening doors were not working when the evaluation was conducted, all doors and drawers are opened in the bullet environment initially. Nevertheless, the dishwasher is always unreachable for the robot. Furthermore, there are some poses at other locations which are unreachable. Thus, the robot will not always succeed during training and test.

[1]The random number generator is seeded with 0 to achieve reproducible results.

Type	Location Type 1	Location Type 2	Location Type 3
plate	Oven	Drawer	Dishwasher
fork	Drawer	Dishwasher	
knife	Drawer	Dishwasher	
mug	Drawer	Dishwasher	
pot	Oven	Refrigerator	
bowl	Oven	Refrigerator	
mondamin	Refrigerator	CounterTop	
spatula	Drawer	CounterTop	
pancake-maker	Drawer	CounterTop	
orange	CounterTop	Refrigerator	
apple	CounterTop	Refrigerator	
sugar-box	CounterTop		
cereal	CounterTop		

Table 7.1.: Objects Used in the Evaluation

Prep.	Type	Name	x_{min}	x_{max}	y_{min}	y_{max}	z
in	Dishwasher	drawer_sinkblock_dishwasher	1.40	1.60	0.10	0.25	0.25
in	Drawer	drawer_island_left_upper	-0.75	-0.55	0.85	1.00	0.55
in	Drawer	drawer_island_right_upper	-0.75	-0.55	2.45	2.60	0.55
in	Oven	drawer_oven_upper	1.30	1.50	1.80	2.00	0.95
in	Refrigerator	drawer_fridge_upper	1.35	1.60	-1.10	-0.95	0.60
on	CounterTop	kitchen_island_counter_top	-1.30	-1.00	0.60	2.80	0.95
on	CounterTop	kitchen_sink_block_counter_top	1.40	1.50	-0.50	1.10	0.95

Table 7.2.: Locations Used in the Evaluation

After the kitchens have been generated, the robot performs 20 experiments, one in each training kitchen. Before each experiment, the needed ROS nodes are started and the objects in the kitchen are spawned at the sampled pose. Then, the robot is told to displace one object after another to the *pancake_table* by calling the *achieve loc ?obj ?loc* plan with the training object designator and a location designator describing the *pancake_table*. After all objects have been displaced, the robot exports a log file and updates the MLN using this log file. This simulates the lifelong learning since the log files are learned incrementally. Finally, all the ROS nodes shut down and the next kitchen is processed. Thus, the robot tries to displace 200 objects to the *pancake_table* in total during the training phase.

When the training is finished, different tests corresponding to the use cases in Figure 1.1 are executed in each of the test kitchens. In general, the result of each test is a CSV file. Before each test is executed, the needed ROS nodes are started and the objects in the kitchen are spawned at the sampled pose in the bullet environment. After each test, the ROS nodes are shut down. The four different tests are explained in detail now.

First of all, there is the *informed test*: There, the robot displaces the training objects just like it did during the training. The generated CSV file contains the information for which object the robot succeeded. This test will serve as reference to check whether the inference of the success probability produces correct values.

Then, there is the *naive completion test*: Again, the robot displaces objects to the *pancake_table*. However, this time the test objects are used. Since the objects need a location designator to be found, the test object is completed with a location designator constructed from Table 7.2. More specifically, the robot tries each line of Table 7.2 until it successfully displaced the object. This time, the CSV file contains the information whether the robot succeeded and how many locations were tried. The test will serve as reference to check how good the designator completion works.

Moreover, there is a *MLN completion test*: Now, the robot tries to complete the test objects using the designator completion described in Section 6.3. As in the naive test, the completed designator will be used as argument for the *achieve loc ?obj ?loc* function in order to displace the object to the *pancake_table*. If this does not work, it tries the next designator proposed by the completion until no completion is left or until it succeeded. Again, the number of tries and the information whether the robot succeeded are written to the CSV file.

Finally, there is the *theoretical test*: As the name implies, the robot does not execute any plan that modifies the environment. Instead, some inferences are executed. First of all, the success probability for the *achieve loc ?obj ?loc* plan given the training object is inferred. Then, the most likely object type of a parameter of the *perceive a ?obj-desig* plan is inferred. The designator used as *?obj-deisg* parameter only consists of the location of the training object. Moreover, the test designator is completed using the designator completion described in Section 6.3. Instead of trying to execute the *achieve loc ?obj ?loc* plan, the success of the completion is determined by comparing the completed designator with the training object. The same is done with the naive completion. Hence, the CSV file contains the inferred success rate, the rank of the actual object type in the probability distribution for the object types, as well as the success and the number of tries for the designator completion using a MLN and using the naive completion.

In the end, the three CSV files containing information about 50 displace attempts are used to calculate statistics.

7.1.2. Results

Table 7.3 shows the statistics calculated from the CSV files generated during the executed evaluation.

Fully Informed Object Displacement	
Successful grasps (%):	62
Average estimated success rate (%):	64
Correctly guessed success assuming threshold 0.5 (%):	78
Successfully guessed objects for a location (%):	98
Average location rank:	3.16
MLN Completion	
Successful grasps (%):	54
Average tries for successful grasps:	1.41
Average tries for unsuccessful grasps:	1.83
Naive Completion	
Successful grasps (%):	66
Average tries for successful grasps:	2.45
Average tries for unsuccessful grasps:	7.0
MLN Completion (in Theory)	
Successful grasps (%):	80
Average tries for successful grasps:	1.63
Average tries for unsuccessful grasps:	1.50
Naive Completion (in Theory)	
Successful grasps (%):	100
Average tries for successful grasps:	4.58
Average tries for unsuccessful grasps:	None

Table 7.3.: Evaluation Results

The use case that the robot wants to infer what is expected to be perceived at a location has been tested in the theoretical test. In Table 7.3, the results for this task are shown in the last two lines of the first part. With 98%, the current training object is expected almost every time at the given location. However, the actual type is the third-most probable type inferred in average. With said that there are in average 5.42 possible objects at a location, the rank 3.16 is not the best but acceptable. Thus, the approach is suited for the first use case.

The second use case in Figure 1.1 is the success estimation. The result for the success estimation using the MLN based approach is shown in the second and third line of the first part of Table 7.3. At the first glance, the result for the success estimation looks pretty good: With 64%, the average estimated success rate is very close to the real average success rate of the informed test shown in the first line of Table 7.3.

However, the result gets a little worse if one takes a closer look: To get a more precise value of the quality of the success estimation, the estimated value for each object was binarized with the threshold 0.5. All values below this threshold are considered as unsuccessful and all values above the threshold are considered as successful. Then, the actual results were compared with this estimate. The comparison shows that the result of 78% of the displace tasks were successfully predicted. Of course, this value is not perfect but better than guessing. Hence, the approach is also suited for the second use case, even if the results are not perfect.

Finally, there is the designator completion as third use case. The results of the corresponding tests are shown in the second and third part of Table 7.3. If the displace operation is successful, the MLN completion needs approximately one less try to find an object compared to the naive completion. More importantly, the MLN completion saves time when the object cannot be found: The naive completion has to look at all 7 locations while the MLN completion can exclude some locations. Thus, the robot saves time if an object is not found.

Unfortunately, the robot is less successful when using the MLN based completion approach. One thing that makes one wondering is that the success rate of the naive completion is higher than the one of the informed test. The reason is that the semantic map of the robot seems to be imperfect. Thus, the robot takes the wrong object in some cases. If it is supposed to displace an apple on the *kitchen_island_counter_top* and there is an apple on the *pancake_table*, it will displace the one at the *pancake_table*. This behavior also explains the low number of tries needed by the naive completion.

Therefore, the completion was also executed in the theoretical test. The result of this test stresses the theoretical power of the MLN based designator completion: It needs 1.63 tries in average when it successfully displaces an object, while the naive completion needs 4.58 tries. This is a difference of almost 3 tries. However, the naive completion is still more successful. If one has a closer look at the objects that could not be completed, it becomes clear why this is the case: The combination of these objects with the expected locations was tried during the training, but it always failed. Since the designator completion tries to make the plan successful, these values were not considered. Thus, this result is acceptable. This means that the learning approach developed in this thesis is also suitable for the last use case.

In summary, the lifelong learning approach developed in this thesis is not perfect, but it works in most of the cases. On the other hand, humans are also not perfect and since the environment is usually not fully observable, a perfect result is often not achievable.

7.1.3. Performance

It remains to show that the inference algorithm developed in Chapter 5 is applicable. A complete evaluation of the algorithm would require a comparison of the performance of the algorithm with the performance of the algorithms presented in Section 3.2. However, these algorithms are not implemented in pracmln. Thus, they would have to be implemented to make the comparison possible. Unfortunately, this is beyond the scope of this thesis.

As mentioned throughout the last chapters, the algorithm performs especially well in the designator based MLN design where it is the only working inference algorithm in pracmln. However, the designator based MLN design has not been used in the end. Therefore, the performance of the new inference algorithm containing the improved grounding was evaluated in the final MLN design. It was compared to the performance of the normal exact inference algorithm with the normal grounding in pracmln. The test was to complete an object designator with the property *:type :mug* given an *achieve object-in-hand* plan using the MLN learned in the previous section. Section 6.2 mentioned that the *goal-perform* nodes and thus the action designators for the process modules are not part of the MLN. In order to show what a broader variable domain of the MLNs causes, the same completion was also executed on a MLN containing these action designators. This causes the number of formulas to grow from 547 to 5169.

Inference Algorithm	Without GOAL-PERFORM	With GOAL-PERFORM
Algorithm from Chapter 5	$3s$	$128s$
Normal Exact Inference	$8s$	$256s$

Table 7.4.: Performance for one Designator Completion

Table 7.4 shows the results of the experiment. First of all, the new inference algorithm is twice as fast as the normal exact inference algorithm for the final MLN design from Section 4.6. Moreover, it becomes clear that the performance of the designator completion becomes worse if the domains get very large. The time needed for a designator completion does not grow linearly in the size of formulas. Thus, the new inference algorithm is an improvement, but it does not solve all problems.

The experiments were executed on a machine with Intel Core i5-3320M CPU and 8GB of RAM. Since the *materialization* process replacing template formulas takes very long in pracmln when *goal-perform* nodes are included, the materialization process was commented. It is not needed here, anyway. For a better comparability, the multiprocessor CPU support of the normal exact inference was switched off.

However, the new inference algorithm is not the best in every case: For the *taxonomies* example supplied by pracmln, it needs 93 seconds to get an inference result. The normal exact inference algorithm needs only 2 seconds. Hence, the performance of the algorithm is highly dependent on the MLN.

7.2. Opportunities and Limitations

The previous section showed the general suitability of the developed software and thus the MLN design for the use cases in Figure 1.1. Apart from that, the question arises in which other cases the software might be used and what limitations are to be considered when using it. Therefore, this section looks at some interesting cases to discuss these opportunities and limitations.

First of all, there is the case that a plan requires two designators as parameter. An example is the *achieve loc ?obj ?loc* plan for displacing the object *?obj* to *?loc*. Since the MLN handles these designators independently, there is no formula modeling the connection between both designators. On the one hand, this is good for this plan since it results in less, more general formulas. This in turn provides a better generalization and a faster inference. On the other hand, there might be some cases where this special knowledge is required, e.g. when the robot tries to place a bottle in a small drawer. However, the first arguments were chosen to be more important in this case.

Another detail worth looking at is the functionality of the designator completion. If the robot was, for example, trained to perceive objects with designators like the one in Listing 7.3 including a symbolic location and if it was also trained to grasp objects like the one in Listing 7.4 including a handle[2], then it is possible to merge these designators: The robot can first execute the designator completion for the object in Listing 7.5 given the perceive task. After it perceived the object correctly, it can further complete the designator resulting from the last completion given the grasp task. Subsequently, the designator completion will return a designator containing both, the symbolic location and the handle.

However, this example only works since the location of the designator in Listing 7.4 has been specified by a pose. If it was a symbolic location as in the designator in Listing 7.3 and if no grasp attempt of the object has been executed at this location, this wold not work: The handle would be assigned based on the highest number of true formulas. Thus, it might be possible that the handle would be assigned based on an equal location rather than based on the type property. Even if the correct handle would be assigned based on the type property since no matching location was available, the location of the grasp designator would additionally be added. This has not been done with the pose since hard coded poses are excluded. Thus, the designator completion provides the ability to combine two completely different designators, but only under certain conditions.

```
(cram-designators:make-designator
  :object `((:type :mug)
            (:at ,(cram-designators:make-designator
                    :location '((:on "Cupboard")
                                (:name "kitchen_sink_block"))))))
```

Listing 7.3: Designator Describing a Mug on the Kitchen Sink Block

[2]One might wonder how it was possible to grasp an object in the previous section without a handle: Handles are not required in the bullet environment.

```
(cram-designators:make-designator
  :object `((:type :mug)
            (:at ,(cram-designators:make-designator
                     :location
                     `((:pose
                          ,(cl-transforms-stamped:make-pose-stamped
                              "map"
                              0
                              (cl-tf:make-3d-vector 1.7 0.6 0.9)
                              (cl-tf:make-quaternion 0 0 0 1))))))
            (:handle
              ,(cram-designators:make-designator
                 :object
                 `((:type :handle)
                   (:at ,(cram-designators:make-designator
                           :location
                           `((:pose ,(cl-tf:make-pose
                                        (cl-tf:make-3d-vector
                                          -0.005 0 0)
                                        (cl-tf:make-quaternion
                                           0 0 0 1)))))))))))
```

Listing 7.4: Designator Describing a Mug with a Handle

```
(cram-designators:make-designator :object `((:type :mug)))
```

Listing 7.5: Designator Describing a Mug

The MLN design also allows to add further properties for a better generalization. For instance, the designator in Listing 7.6 has a *category* property which is actually not required by CRAM. This allows to transfer knowledge. If such a designator is used as an argument for a perception task during the MLN training, it can be used to complete the glass designator in Listing 7.7. Again, there are some limitations: The first limitation is that the glass must not be included in the log files. Otherwise, the existing designator for the glass will be inferred. Then, the algorithm must be modified not to accept duplicate keys. If it remained unchanged, the glass would also have the type *:mug*. Thus, there is the possibility to introduce category properties to generalize on unseen object types.

```
(cram-designators:make-designator
  :object `((:type :mug)
            (:category :drinking-vessel)
            (:at ,(cram-designators:make-designator
                     :location '((:on "Cupboard")
                                 (:name "pancake_table"))))))
```

Listing 7.6: Designator Describing a Mug with a Category Property

```
(cram-designators:make-designator
  :object `((:type :glass)
            (:category :drinking-vessel)))
```

Listing 7.7: Designator Describing a Glass with a Category Property

However, all the generalization described above has one downside: Different designators influence each other. This is, for example, the case if the robot picks up one mug as specified in Listing 7.3, one mug as specified in Listing 7.8 and one plate as specified in Listing 7.9 during the training. Normally, one would expect a probability of $\frac{1}{2}$ both for the *pancake_table* and for the *kitchen_sink_block* when inferring the location name given that there is a pickup action of a mug on a cupboard. However, the result will be a probability of $\frac{2}{3}$ for the *pancake_table* and a probability of $\frac{1}{3}$ for *kitchen_island*. The reason is that the weight for the formula connecting the *:on "Cupboard"* property with the *:name "pancake_table"* property is also increased by the plate designator. This influence can be desirable on the one hand if objects placed together at one location are usually also placed together at another location, but it can also be disturbing if the exact probability is needed.

```
(cram-designators:make-designator
  :object `((:type :mug)
            (:at ,(cram-designators:make-designator
                   :location '((:on "Cupboard")
                               (:name "pancake_table"))))))
```

Listing 7.8: Designator Describing a Mug on the Pancake Table

```
(cram-designators:make-designator
  :object `((:type :plate)
            (:at ,(cram-designators:make-designator
                   :location '((:on "Cupboard")
                               (:name "pancake_table"))))))
```

Listing 7.9: Designator Describing a Plate on the Pancake Table

One thing that has not yet been addressed by this thesis is learning actions. At the moment, actions are exclusively represented by plans and thus, it is not possible to complete actions. However, there are the action designators in CRAM. Instead of using them only for commands to the process modules, one could also use the action designators for more abstract actions. Listing 7.10 shows such a designator. It describes an extract of a table setting action. If CRAM was able to interpret such an action designator, one could first train a MLN by executing a designator as shown in Listing 7.10. Afterwards, it would be enough to pass the action designator in Listing 7.11 to the robot and it could use the designator completion to get the one in Listing 7.10. Unfortunately, it is not directly possible to further complete this designator, e.g. to infer the mug location automatically. The reason is that the property keys of all sub designators are merged together into one key, e.g. *object.AT::location.ON*. However, it is possible to call the designator completion separately for the sub designators, i.e. for the mug and the pancake table. Thus, there is the opportunity to also use the learning approach developed here for learning actions.

```
(cram-designators:make-designator
  :action `((:action "set-table")
            (:occasion "breakfast")
            (:do ,(cram-designators:make-designator
                     :action `((:goal achieve)
                               (:context loc)
                               (:argument1
                                  ,(cram-designators:make-designator
                                       :object `((:type :mug))))
                               (:argument2
                                  ,(cram-designators:make-designator
                                       :location
                                       `((:name "pancake_table")))))))))
```

Listing 7.10: Designator Describing a Table Setting Action in Detail

```
(cram-designators:make-designator
  :action `((:action "set-table")
            (:occasion "breakfast")))
```

Listing 7.11: Designator Describing a Part of the Table Setting Action

In the end, the approach developed in this thesis provides many opportunities, especially through the provided generalization. However, not everything is possible and there are undesired side effects on the probability distribution in some cases, which is the price to pay for the generalization.

8. Conclusions

This thesis presented one approach to make household robots learn models about plan parameters from their experience and to use these models in everyday activities. The developed approach is grounded on a MLN design representing a compromise between intuitive modeling, simple learning and generalization. In particular, this MLN design allows updating the weights of the MLN and thus a lifelong learning. Apart from that, an exact inference algorithm for MLNs has been developed which is faster in some of the MLN designs presented in this work. Then, software was implemented to create MLNs from log files and to use these MLNs from CRAM. A system consisting of this software, pracmln and CRAM was eventually evaluated.

All in all, the evaluation showed that the approach is already usable, even if it does not work perfectly. Moreover, the evaluation showed that there is the opportunity to apply the approach to more advanced problems. To achieve this, the MLN design and especially the generalization provided by the MLNs could be further optimized. It would be worth trying to get rid of the influence of different designators on each other while keeping the possibility to generalize. Then, one could try to get even more generalization by using the FuzzyMLNs provided by pracmln as mentioned in [NB15]. On the one hand, they could be useful to define a similarity between different plans. If an object can, for example, be picked, it should also be possible to place it somewhere. On the other hand, they could define a similarity between object types. However, this has not been used yet since it would make the *:type* designator property special and thus the implementation would be less generic. Another interesting possibility would be the combination of the designator based MLN design in Section 4.5 with the final MLN design in Section 4.6 to get a better generalization for nested designators. Moreover, the tasks with no parameter or designators with only one property are not considered yet. The handling of numeric values would also be interesting, either by clustering or by modifying the MLN formalism. Thus, more research can be done on the MLN design.

Furthermore, the performance could be improved to be able to include more information in the MLNs, e.g. the action designators for the process module calls. This can in turn be achieved by improving the performance of the inference algorithm developed in this thesis. Its implementation in pracmln would benefit the most from a more efficient formula consistency check. Of course, the performance could also be improved dramatically if the inference would be executed in parallel on a GPU. Then, an approximate inference algorithm based on the idea of the algorithm developed here would be worth thinking of. Moreover, a comparison of the algorithm with other inference algorithms would be interesting. Hence, one could also dig deeper in the field of MLN inference algorithms.

Finally, the implemented software could be more efficient if a *semrec* plugin would communicate directly with the *robot_memory* service in order to bypass the log files. The decision to use the log files instead has been made to be able to learn from older experiences as well. However, this improvement can easily be implemented due to the ROS based architecture. Considering the evaluation, it would be interesting to implement the handling of higher level action designators in CRAM, as mentioned Section 7.2, in order to use the designator completion there.

In the end, one can say that the system consisting of CRAM, pracmln and the MLN creation software developed in this thesis is already able to provide a lifelong learning for everyday robot manipulation, even if there is the possibility to improve some parts.

A. Proofs

Theorem A.1. *Let $\mathcal{D}$ be a set of training databases, F the set of formulas, $n_i(x)$ the number of true ground formulas of formula $f_i \in F$ in world x, w_i the weight of formula $f_i \in F$, C_x the set of constants in the training database $x \in \mathcal{D}$ and let $\mathcal{X}_{L,C}$ be the worlds creatable from the MLN L and the set of constants C.*
The (log) likelihood function is maximized if:

1. *The weights are determined by the number of true groundings in the training data:* $\forall i \in \{1, ..., |F|\} : w_i = c + \ln\left(\sum_{x' \in \mathcal{D}} n_i(x')\right)$ *with* $\exp(c) >> |\mathcal{X}_{L,C_x}|$ *for all Databases* x *used during the learning process*

2. *Every formula has to appear at least once in the training data (*$\ln(0)$ *is undefined):* $\forall i \in \{1, ..., |F|\} \exists x \in \mathcal{D} : n_i(x) > 0$

3. *The formulas are mutually exclusive for all worlds generatable from the constants in the training data and in the MLN:*
$\forall x_d \in \mathcal{D} \forall x \in \mathcal{X}_{L,C_{x_d}} \left(\sum_{i=1}^{|F|} n_i(x)\right) \leq 1$

4. *Every formula must be true in the same number of worlds t:*
$\exists t \forall x_d \in \mathcal{D} \forall i \in \{1, ..., |F|\} : |\{x | n_i(x) > 0 \wedge x \in \mathcal{X}_{L,C_{x_d}}\}| = t$

5. *In every database there must be exactly one true ground formula:*
$\forall x \in \mathcal{D} : \left(\sum_{i=1}^{|F|} n_i(x)\right) = 1$

To Show.

For calculating the maximum of a function, one has to determine the candidates by calculating the zero of the first gradient. Then, one can use the second gradient to determine whether the candidate is a maximum [Har06]. The optimization problem for Markov Logic Networks is convex [LD07]. This means that the log likelihood function must be convex since a maximum should be found. In other words, every candidate is a maximum and the second gradient does not need to be evaluated. Thus, it has to be shown that the following condition is true for each weight w_i:

$$0 = \frac{\partial}{\partial w_i} \ln(\prod_{x \in \mathcal{D}} P(X = x | L, C)) = \sum_{x \in \mathcal{D}} \left(n_i(x) - \sum_{x' \in \mathcal{X}_{L,C_x}} n_i(x') \cdot P(X = x' | L, C)\right) \text{[Jai12]}$$

Proof.

$$
\begin{aligned}
0 &= \sum_{x \in \mathcal{D}} \left(n_i(x) - \sum_{x' \in \mathcal{X}_{L,C_x}} n_i(x') \cdot P(X = x'|L, C) \right) && \text{(A.1)} \\
&= \sum_{x \in \mathcal{D}} n_i(x) - \sum_{x \in \mathcal{D}} \sum_{x' \in \mathcal{X}_{L,C_x}} n_i(x') \cdot P(X = x'|L, C) && \text{(A.2)} \\
&= p_i - \sum_{x \in \mathcal{D}} \sum_{x' \in \mathcal{X}_{L,C_x}} n_i(x') \cdot P(X = x'|L, C) \text{ with } p_i = \sum_{x \in \mathcal{D}} n_i(x) && \text{(A.3)} \\
&= p_i - \sum_{x \in \mathcal{D}} \frac{\sum\limits_{x' \in \mathcal{X}_{L,C_x}} n_i(x') \cdot \exp\left(\sum\limits_{j=1}^{|F|} w_j \cdot n_j(x')\right)}{\sum\limits_{x' \in \mathcal{X}_{L,C_x}} \exp\left(\sum\limits_{j=1}^{|F|} w_j \cdot n_j(x')\right)} && \text{(A.4)} \\
&= p_i - \sum_{x \in \mathcal{D}} \frac{|X_{L,C_x,f}| + \sum\limits_{x' \in X_{L,C_x,t}} n_i(x') \cdot \exp\left(\sum\limits_{j=1}^{|F|} w_j \cdot n_j(x')\right)}{|X_{L,C_x,f}| + \sum\limits_{x' \in X_{L,C_x,t}} \exp\left(\sum\limits_{j=1}^{|F|} w_j \cdot n_j(x')\right)} && \text{(A.5)} \\
&\quad \text{with } X_{L,C,f} = \{x'|x' \in \mathcal{X}_{L,C} \wedge \nexists i \in \{1, ..., |F|\} : n_i(x') > 0\} \\
&\quad \text{and } \; X_{L,C,t} = \{x'|x' \in \mathcal{X}_{L,C} \wedge \exists i \in \{1, ..., |F|\} : n_i(x') > 0\} \\
&\approx p_i - \sum_{x \in \mathcal{D}} \frac{\sum\limits_{x' \in X_{L,C_x,t}} n_i(x') \cdot \exp\left(\sum\limits_{j=1}^{|F|} w_j \cdot n_j(x')\right)}{\sum\limits_{x' \in X_{L,C_x,t}} \exp\left(\sum\limits_{j=1}^{|F|} w_j \cdot n_j(x')\right)} && \text{(A.6)} \\
&= p_i - \sum_{x \in \mathcal{D}} \frac{t \cdot \exp(w_i)}{\sum\limits_{j=1}^{|F|} t \cdot \exp(w_j)} && \text{(A.7)} \\
&= p_i - \sum_{x \in \mathcal{D}} \frac{t \cdot \exp(c + \ln(p_i))}{\sum\limits_{j=1}^{|F|} t \cdot \exp(c + \ln(p_j))} && \text{(A.8)} \\
&= p_i - \sum_{x \in \mathcal{D}} \frac{t \cdot \exp(c) \cdot p_i}{t \cdot \exp(c) \cdot \sum\limits_{j=1}^{|F|} p_j} && \text{(A.9)} \\
&= p_i - p_i \frac{|\mathcal{D}|}{\sum\limits_{j=1}^{|F|} p_j} = p_i - p_i \frac{1}{1} = 0 && \text{(A.10)}
\end{aligned}
$$

□

Justification.

The proof starts with the ith (i is representing any formula index) gradient of the log likelihood function in Equation A.1. It is rewritten in Equation A.2. Due to Assumption 5, $\sum_{x \in \mathcal{D}} n_i(x)$ can be replaced by the number of true ground formulas p_i of the formula i in Equation A.3. In Equation A.4 the formula for $P(X = x'|L, C)$ is plugged in. In Equation A.5, the worlds $X_{L,C_x,f}$ where the weighted sum of the applied formulas is zero are extracted from the sum. Afterwards, Equation A.6 shows an approximation: Since $\exp(c)$ (and thus the weighted sum) is much larger than $|\mathcal{X}_{L,C_x}|$ (see Assumption 1), the sum representing the worlds with weighted sum zero affects the whole equation only marginally. Thus, it is removed. In Equation A.7, different assumptions are used: In the sum inside the exponentiation, maximally one n_j can be 1 due to Assumption 3. All others are 0. Since the sum runs over $X_{L,C_x,t}$, exactly one n_j is 1. Finally, the total number of worlds for each ground formula is restricted to t as specified by Assumption 4. Hence, the term can be rewritten. Equation A.8 plugs in the definition of w_j respective w_i as defined in Assumption 1. After some simplifications in Equation A.9, the fraction is canceled and rearranged in Equation A.10. Finally, Assumption 5 asserts that the total number of true ground formulas $\sum_{j=1}^{F} p_j$ is equal to the number of databases.

B. Code Listings

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY owl "http://www.w3.org/2002/07/owl#" >
                   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
                   <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
                   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
                   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
                   <!ENTITY bike "http://localhost/bicycle.owl#" >
                  ]>
<rdf:RDF xmlns="http://localhost/bicycle.owl#"
         xml:base="http://localhost/bicycle.owl"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:bike="http://localhost/bicycle.owl#">
    <owl:Ontology rdf:about="http://localhost/bicycle.owl"/>
    <owl:Class rdf:about="&bike;Human"/>
    <owl:Class rdf:about="&bike;Bicycle"/>
    <owl:Class rdf:about="&bike;RoadBike">
        <rdfs:subClassOf rdf:resource="&bike;Bicycle"/>
    </owl:Class>
    <owl:ObjectProperty rdf:about="&bike;owner">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:range rdf:resource="&bike;Human"/>
        <rdfs:domain rdf:resource="&bike;Bicycle"/>
    </owl:ObjectProperty>
    <owl:DatatypeProperty rdf:about="&bike;color">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="&bike;Bicycle"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:NamedIndividual rdf:about="&bike;Marc">
        <rdf:type rdf:resource="&bike;Human"/>
    </owl:NamedIndividual>
    <owl:NamedIndividual rdf:about="&bike;MarcsRoadBike">
        <rdf:type rdf:resource="&bike;RoadBike"/>
        <bike:owner rdf:resource="&bike;Marc"/>
        <bike:color rdf:datatype="&xsd;string">black</bike:color>
    </owl:NamedIndividual>
</rdf:RDF>
```

Listing B.1: OWL File Describing Bicycles

```
1  parentTask(?t, ?pt) ^ successor(?t, ?st) => parentTask(?st, ?pt).
2  !parentTask(?t, ?pt) v !successor(?t, ?pt).
3  !parentTask(?t, ?pt) v !successor(?pt, ?t).
4  0.0 taskType(?t, +?tt) ^ goal(?t, +?g)
5  0.0 taskType(?pt, +?tt) ^ parentTask(?t, ?pt) ^ taskType(?t, +?tt2)
6  0.0 taskType(?t, +?tt1) ^ successor(?t, ?t2) ^ taskType(?t2, +?tt2)
7  0.0 taskType(?t, +?taskType) ^ error(?t, +?error)
8  0.0 goal(?t, +?goal) ^ error(?t, +?error)
9  0.0 error(?t1, ?e1) ^ successor(?t1, ?t2) ^ error(?t2, ?e2)
10 0.0 successor(?t1,?t2) ^ taskType(?t1,?tt) ^ taskType(?t2,?tt) ^ error(?
       t1,?e)
11 0.0 taskType(?t, +?taskType) ^ usedInTask(?t, ?o) ^ objectType(o, +?
       objectType)
12 0.0 goal(?t, +?goal) ^ usedInTask(?t, ?o) ^ objectType(?o, +?objectType)
13 0.0 taskType(?t, +?taskType) ^ executedAt(?t, +?abstractLocation)
14 0.0 objectType(?o, +?objectType) ^ usedInTask(?t,?o) ^ executedAt(?t, +?
       abstractLocation)
15 0.0 taskType(?t, +?taskType) ^ duration(?t, +?abstractDuration)
16 0.0 executedAt(?t, +?abstractLocation) ^ duration(?t, +?abstractDuration)
17 0.0 usedInTask(?t, ?o) ^ objectType(?o, +?objectType) ^ duration(?t, +?
       abstractDuration)
18 0.0 goal(?t, +?goal) ^ duration(?t, +?abstractDuration)
19 0.0 objectType(?o, +?objectType) ^ objectProperty(?o, +?objectProperty)
20 0.0 objectType(?o, +?objectType) ^ objectLocation(?o, +?abstractLocation)
21 0.0 taskType(?t, +?taskType) ^ usedInTask(?t, ?o) ^ objectLocation(?o, +?
       abstractLocation)
22 0.0 taskType(?t, +?taskType) ^ causedRobotPositionDifference(?t, +?
       spatialRelation)
23 0.0 goal(?t, +?goal) ^ causedRobotPositionDifference(?t, +?
       spatialRelation)
24 0.0 causedRobotPositionDifference(?t, +?spatialRelation) ^ duration(?t,
       +?abstractDuration)
25 0.0 taskType(?t, +?taskType) ^ usedInTask(?t, ?o) ^
       causedObjectPositionDifference(?t, ?o, +?spatialRelation)
26 0.0 goal(?t, +?goal) ^ usedInTask(?t, ?o) ^
       causedObjectPositionDifference(?t, ?o, +?spatialRelation)
27 0.0 causedObjectPositionDifference(?t, ?o, +?spatialRelation) ^ duration
       (?t, +?abstractDuration)
28 0.0 taskType(?t, +?taskType) ^ perceives(?t, ?o) ^ objectType(?o, +?
       objectType)
29 0.0 goal(?t, +?goal) ^ perceives(?t, ?o) ^ objectType(?o, +?objectType)
30 0.0 perceives(?t, ?o) ^ objectType(?o, +?objectType) ^ error(?t, +?error)
31 0.0 perceives(?t, ?o) ^ objectType(?o, +?objectType) ^ duration(?t, +?
       abstractDuration)
32 0.0 taskType(?t, +?taskType) ^ perceives(?t, ?o) ^ objectType(?o, +?
       objectType) ^ duration(?t, +?abstractDuration)
33 0.0 goal(?t, +?goal) ^ perceives(?t, ?o) ^ objectType(?o, +?objectType) ^
        duration(?t, +?abstractDuration)
34 0.0 taskType(?t, +?taskType) ^ acts(?t, +?actionType, +?actionParameter)
35 0.0 goal(?t, +?taskType) ^ acts(?t, +?actionType, +?actionParameter)
36 0.0 acts(?t, +?actionType, +?actionParameter) ^ error(?t, +?error)
37 0.0 acts(?t, +?actionType, +?actionParameter) ^ duration(?t, +?dur)
```

Listing B.2: Template Formulas for a Naive MLN

C. Affirmation

Herewith I affirm that I composed this work single-handed and that I did not use any sources or facilities other than listed here. Especially, I did not use any internet sources which are not listed in the bibliography. I did not submit this work for another examination. All passages which have been taken from other works literally or whose gist have been taken from other works are marked in combination with the source.[1]

Syke, ____

[1] http://www.uni-bremen.de/fileadmin/user_upload/single_sites/zpa/pdf/pruefungsordnungen/allgemeiner_teil/master/AT_MA_2010-01-27.pdf

D. CD

The CD contained in the paper envelope at the bottom contains the source code of the implementation. Furthermore, it contains this document and the evaluation data.

Bibliography

[AMPSQ14] Al-Moadhen, Ahmed ; Packianather, Michael ; Setchi, Rossi ; Qiu, Renxi: Automation in Handling Uncertainty in Semantic-knowledge based Robotic Task-planning by Using Markov Logic Networks. In: *Procedia Computer Science* 35 (2014), S. 1023–1032. – http://www.sciencedirect.com/science/article/pii/S1877050914011533/pdf?md5=de2c1419bde6ba933a17ce3cc9888918&pid=1-s2.0-S1877050914011533-main.pdf, accessed on 2016-06-24

[BG05] Beetz, Michael ; Grosskreutz, Henrik: Probabilistic hybrid action models for predicting concurrent percept-driven robot behavior. In: *Journal of Artificial Intelligence Research* 24 (2005), S. 799–849. – http://www.jair.org/media/1565/live-1565-2552-jair.pdf, accessed on 2016-07-15

[BMT10] Beetz, Michael ; Mösenlechner, Lorenz ; Tenorth, Moriz: CRAM - A Cognitive Robot Abstract Machine for everyday manipulation in human environments. In: *The 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* IEEE, 2010, S. 1012–1017. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5650146, accessed on 2016-03-13

[Cou10] Cousins, Steve: ROS on the PR2. In: *Robotics & Automation Magazine, IEEE* 17 (2010), Nr. 3, S. 23–25. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5569012, accessed on 2016-03-10

[cra16] *Using Prolog for reasoning.* : *Using Prolog for reasoning.* http://www.cram-system.org/tutorials/beginner/cram_prolog, accessed on 2016-03-22, 2016

[DSBAR05] De Salvo Braz, Rodrigo ; Amir, Eyal ; Roth, Dan: Lifted First-order Probabilistic Inference. In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2005 (IJCAI'05), 1319–1325. – http://www.ai.sri.com/~braz/papers/ijcai05.pdf, accessed on 2016-05-29

[GD10] Gogate, Vibhav ; Domingos, Pedro M.: Exploiting Logical Structure in Lifted Probabilistic Inference. In: *Statistical Relational Artificial Intelligence*, 2010. – http://www.cs.washington.edu/sites/default/files/ai/papers/starai10.pdf, accessed on 2016-05-29

[GD16] Gogate, Vibhav ; Domingos, Pedro: Probabilistic Theorem Proving. In: *Commun. ACM* 59 (2016), Juni, Nr. 7, S. 107–115. http://dx.doi.org/10.1145/2936726. – DOI 10.1145/2936726. – ISSN 0001–0782. – http://dl.acm.org/citation.cfm?id=2936726, accessed on 2016-07-30

[Gib14] Gibson, J.J.: *The Ecological Approach to Visual Perception: Classic Edition*. Taylor & Francis, 2014 (Psychology Press & Routledge Classic Editions). – ISBN 9781317579380. – https://books.google.de/books?id=8BSLBQAAQBAJ, accessed on 2016-05-24

[Har06] Hartmann, Peter: *Mathematik für Informatiker. Ein praxisbezogenes Lehrbuch*. 4. Vieweg & Sohn Verlag, 2006

[HLRB13] Hermans, Tucker ; Li, Fuxin ; Rehg, James M. ; Bobick, Aaron F.: Learning stable pushing locations. In: *Development and Learning and Epigenetic Robotics (ICDL), 2013 IEEE Third Joint International Conference on* IEEE, 2013, S. 1–7. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6652539&tag=1, accessed on 2016-05-23

[HPK+12] Herzog, Alexander ; Pastor, Peter ; Kalakrishnan, Mrinal ; Righetti, Ludovic ; Asfour, Tamim ; Schaal, Stefan: Template-based learning of grasp selection. In: *2012 IEEE International Conference on Robotics and Automation (ICRA)* IEEE, 2012, S. 2379–2384. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6225271, accessed on 2016-05-23

[Jai12] Jain, Dominik: *Probabilistic Cognition for Technical Systems*, Technische Universität München, Diss., 2012. – http://mediatum.ub.tum.de/doc/1096684/1096684.pdf, accessed on 2016-04-03

[JGMS10] Jha, Abhay ; Gogate, Vibhav ; Meliou, Alexandra ; Suciu, Dan: Lifted inference seen from the other side: The tractable features. In: *Advances in Neural Information Processing Systems*, 2010, S. 973–981. – http://www.cs.washington.edu/sites/default/files/ai/papers/nips10-b.pdf, accessed on 2016-05-29

[JMW09] Jain, Dominik ; Maier, Paul ; Wylezich, Gregor: Markov Logic as a Modelling Language for Weighted Constraint Satisfaction Problems. In: *Eighth International Workshop on Constraint Modelling and Reformulation, in conjunction with CP2009*, 2009. – https://ias.cs.tum.edu/_media/spezial/bib/jain09modref.pdf, accessed on 2016-04-01

[Kaz16] Kazhoyan, Gayane: *Bullet world demonstration*. http://www.cram-system.org/tutorials/advanced/bullet_world, accessed on 2016-07-13, 2016

[Kir09] Kirsch, Alexandra: Robot learning language - integrating programming and learning for cognitive systems. In: *Robotics and Autonomous Systems*

57 (2009), Nr. 9, S. 943–954. – http://www.sciencedirect.com/science/article/pii/S0921889009000785, accessed on 2016-05-22

[LD07] Lowd, Daniel ; Domingos, Pedro: Efficient Weight Learning for Markov Logic Networks. In: *Knowledge discovery in databases: PKDD 2007* (2007)

[MB11] Mösenlechner, Lorenz ; Beetz, Michael: Parameterizing Actions to have the Appropriate Effects. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. San Francisco, CA, USA, September 25–30 2011. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6094883, accessed on 2016-03-18

[MB13] Mösenlechner, Lorenz ; Beetz, Michael: Fast Temporal Projection Using Accurate Physics-Based Geometric Reasoning. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Karlsruhe, Germany, May 6–10 2013, S. 1821–1827. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6630817, accessed on 2016-03-18

[MDR14a] Moldovan, Bogdan ; De Raedt, Luc: Learning relational affordance models for two-arm robots. In: *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on* IEEE, 2014, S. 2916–2922. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6942964, accessed on 2016-05-28

[MDR14b] Moldovan, Bogdan ; De Raedt, Luc: Occluded object search by relational affordances. In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on* IEEE, 2014, S. 169–174. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6906605, accessed on 2016-05-28

[MLBSV07] Montesano, Luis ; Lopes, Manuel ; Bernardino, Alexandre ; Santos-Victor, Jose: Modeling affordances using bayesian networks. In: *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on* IEEE, 2007, S. 4102–4107. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4399511, accessed on 2016-05-24

[MMO+12] Moldovan, Bogdan ; Moreno, Pablo ; Otterlo, Martijn van ; Santos-Victor, José ; De Raedt, Luc: Learning relational affordance models for robots in multi-object manipulation tasks. In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on* IEEE, 2012, S. 4373–4378. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6225042, accessed on 2016-05-28

[Moo90] Moore, Andrew W.: *Efficient memory-based learning for robot control*, University of Cambridge, Diss., 1990. – http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-209.pdf, accessed on 2016-05-20

[MZK+08] Milch, Brian ; Zettlemoyer, Luke S. ; Kersting, Kristian ; Haimes, Michael ; Kaelbling, Leslie P.: Lifted Probabilistic Inference with Counting Formulas. In: *Aaai* Bd. 8, 2008, S. 1062–1068. – https://www.aaai.org/Papers/AAAI/2008/AAAI08-168.pdf, accessed on 2016-05-29

[NB15] Nyga, Daniel ; Beetz, Michael: Reasoning about Unmodelled Concepts – Incorporating Class Taxonomies in Probabilistic Relational Models. In: *Arxiv.org*, 2015. – http://arxiv.org/pdf/1504.05411v1.pdf, accessed on 2016-04-02

[NHRL14] Neumann, Bernd ; Hotz, Lothar ; Rost, Pascal ; Lehmann, Jos: A robot waiter learning from experiences. In: *Machine Learning and Data Mining in Pattern Recognition*. Springer, 2014, S. 285–299. – http://link.springer.com/chapter/10.1007/978-3-319-08979-9_22, accessed on 2016-05-22

[Nyg16] Nyga, Daniel: *pracmln: Markov Logic Networks in Python*. http://pracmln.org, accessed on 2016-04-12, 2016

[OCC+09] Oyama, Akihisa ; Chitta, Sachin ; Conley, Ken ; Bradski, Gary ; Konolige, Kurt ; Cousins, Steve: Come on in, our community is wide open for robotics research! In: *The 27th annual conference of the robotics society of Japan* Bd. 9, 2009, S. 2009. – http://www.willowgarage.com/sites/default/files/RSJ2009_AkihisaOyama_ComeOnIn_En.pdf, accessed on 2016-03-10

[PD06] Poon, Hoifung ; Domingos, Pedro: Sound and efficient inference with probabilistic and deterministic dependencies. In: *AAAI* Bd. 6, 2006, S. 458–463. – http://homes.cs.washington.edu/~pedrod/papers/aaai06a.pdf, accessed on 2016-04-01

[PG15] Pandey, Amit K. ; Gelin, Rodolphe: Human robot interaction can boost robot's affordance learning: A proof of concept. In: *Advanced Robotics (ICAR), 2015 International Conference on* IEEE, 2015, S. 642–648. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7251524&tag=1, accessed on 2016-05-26

[QGC+09] Quigley, Morgan ; Gerkey, Brian ; Conley, Ken ; Faust, Josh ; Foote, Tully ; Leibs, Jeremy ; Berger, Eric ; Wheeler, Rob ; Ng, Andrew: ROS: an open-source Robot Operating System. In: *ICRA workshop on open source software* Bd. 3, 2009, S. 5. – https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf, accessed on 2016-03-10

[RD06] Richardson, Matthew ; Domingos, Pedro: Markov logic networks. In: *Machine Learning* 62 (2006), Nr. 1-2, S. 107–136. – http://link.springer.com/content/pdf/10.1007%2Fs10994-006-5833-1.pdf, accessed on 2016-03-28

[RN10] Russell, Stuart ; Norvig, Peter: *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010

[RNZ+13] Rockel, Sebastian ; Neumann, Bernd ; Zhang, Jianwei ; Dubba, Krishna Sandeep R. ; Cohn, Anthony G. ; Konečný, Š ; Mansouri, Masoumeh ; Pecora, Federico ; Saffiotti, Alessandro ; Günther, Martin u. a.: An ontology-based multi-level robot architecture for learning from experiences. In: *Designing Intelligent robots: Reintegrating AI II. AAAI Spring Symposium-Technical Report* AAAI Press, 2013, S. 52–57. – http://eprints.whiterose.ac.uk/81158/7/AAAI-SS-2013-RACE-final-no-copyright.pdf, accessed on 2016-05-22

[Sch08] Schöning, Uwe: *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag, 2008 https://books.google.de/books?id=eFqeJAAACAAJ. – ISBN 9783827418241

[SHKK10] Song, Dan ; Huebner, Kai ; Kyrki, Ville ; Kragic, Danica: Learning task constraints for robot grasping using graphical models. In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on* IEEE, 2010, S. 1579–1585. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5649406, accessed on 2016-05-24

[SN09] Shavlik, Jude W. ; Natarajan, Sriraam: Speeding Up Inference in Markov Logic Networks by Preprocessing to Reduce the Size of the Resulting Grounded Network. In: *IJCAI* Bd. 9, 2009, S. 1951–1956. – https://alchemy.cs.washington.edu/papers/pdfs/shavlik-natarajan09.pdf, accessed on 2016-05-29

[TB13] Tenorth, Moritz ; Beetz, Michael: KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots. In: *International Journal of Robotics Research (IJRR)* 32 (2013), April, Nr. 5, S. 566 – 590. – http://ijr.sagepub.com/content/32/5/566.full.pdf, accessed on 2016-03-22

[TM95] Thrun, Sebastian ; Mitchell, Tom M.: Lifelong robot learning. In: *Robotics and Autonomous Systems* 1 (1995), Nr. 15, S. 25–46. – http://www.sciencedirect.com/science/article/pii/092188909500004Y, accessed on 2016-05-22

[WB15] Winkler, Jan ; Beetz, Michael: Robot Action Plans that Form and Maintain Expectations. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Hamburg, Germany, 2015. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7354106, accessed on 2016-03-13

[WBMB12] Winkler, Jan ; Bartels, Georg ; Mösenlechner, Lorenz ; Beetz, Michael: Knowledge Enabled High-Level Task Abstraction and Execution. In: *First Annual Conference on Advances in Cognitive Systems* 2 (2012).

– http://www.cogsys.org/pdf/paper-3-2-135.pdf, accessed on 2016-03-13

[WHB14] Wang, Chang ; Hindriks, Koen V. ; Babuska, Robert: Active learning of affordances for robot use of household objects. In: *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on* IEEE, 2014, S. 566–572. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7041419&tag=1, accessed on 2016-05-26

[Win13] Winkler, Jan: *Lesson 01 - Primer to Pick and Place.* http://ai.uni-bremen.de/wiki/software/cram/tutorials/pick-and-place-primer, accessed on 2016-07-13, 2013

[WTBB14] Winkler, Jan ; Tenorth, Moritz ; Bozcuoglu, Asil K. ; Beetz, Michael: CRAMm – Memories for Robots Performing Everyday Manipulation Activities. In: *Advances in Cognitive Systems* 3 (2014), S. 47–66. – http://www.cogsys.org/papers/2013conference22.pdf, accessed on 2016-03-13

[ZFFF14] Zhu, Yuke ; Fathi, Alireza ; Fei-Fei, Li: Reasoning about object affordances in a knowledge base representation. In: *Computer Vision–ECCV 2014*. Springer, 2014, S. 408–424. – http://link.springer.com/chapter/10.1007/978-3-319-10605-2_27, accessed on 2016-05-27