A Knowledgebase with which you can generate robot plan for multiple mixing actions

Master Thesis

Naser Azizi, Sorin Arion

Prüfer der Master Thesis:	1. Prof. Michael Beetz PhD
	2.
Supervisor	Michaela Kümpel

Eidesstattliche Erklärung

Hiermit erklären wir, dass die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 6. Mai 2024

Naser Azizi, Sorin Arion

Motivation

Although industrial robots have been considered standard for many years, the widespread adoption of AI-based autonomous household robots is still far from being anticipated in every household. Obstacles such as unfamiliar surroundings and a limited knowledge base can significantly hinder the ability of an autonomous robot to effectively respond in a given situation, hence, the robot must precisely plan for every minor detail, even those that might appear trivial to us humans. To achieve the defined objectives, a proficient and comprehensive robotic system including Control, Perception, Navigation, and Knowledge is necessary. An illustrative goal could be cooking. To execute this task, the robot must possess various capabilities, including placing items, grasping objects, pouring liquids, cutting ingredients, and mixing components. Those things seem trivial to us humans, but for the robots it requires a lot of Implementation and information. In this thesis, we aim to introduce an approach detailing how a robotic system can execute mixing actions effectively. The challenges arise from the diverse methods of mixing, which further vary based on the specific ingredients being combined. Additionally, the consideration of compatible Mixing Tools and Mixing Containers is crucial, as not every tool can be utilized with every container. Through the incorporation of a knowledge graph containing rules related to to various actions, we aim to empower the robotic system to make informed decisions on the appropriate motions to employ. This decision-making process will take into account the specific task at hand and the involved ingredients. By incorporating this knowledge graph, we aim to advance towards autonomous robots capable of engaging in cooking activities.

Introduction

First, we intend to analyze existing research, particularly examining studies that delve into diverse mixing motions and exploring related systems that contain a queryable knowledge graph. In the following section, we will introduce the reader with the frameworks employed to accomplish our objectives. Subsequently, we will present our approach to data acquisition and data representation. In this chapter, we aim to explain the mixing tasks under consideration and articulate our methodology for representing this data to enable effective querying. In the subsequent section, we will illustrate the implementation of our rules, with the ultimate aim of determining the appropriate motion to be employed. To validate our concept, we have opted to simulate various scenarios and assess their outcomes, demonstrating the efficacy of our implemented system. Finally, we will delve into the discussion of our results and draw conclusions to wrap up our work.

1 Related Work

In diesem Kapitel möchten wir auf die verschiedenen Arbeiten hinweisen, die ähnlich zu unserer sind. Da unsere Arbeit sich auf zwei Hauptkomponente bezieht, nämlich der Mixing Motions und der dementsprechenden Wissensrepräsentation, werden wir dieses Kapitel in zwei Abschnitten unterteilen, die sich jeweils mit dem Fokus befassen.

Knowledge Representation

In diesem Abschnitt untersuchen wir ähnliche Arbeiten, die sich mit der Wissensrepräsentation befassen, welche abfragbare Parameter und Entscheidungen enthält, welche dazu führen sollen, die richtige Motion auszuführen.

FOON

Cutting

FoodCutting aims to equip robotic agents with necessary information about how to execute cutting tasks in unknown environments for the household domain. From high level plans FoodCutting breaks down the cutting task, part of some robot plan, into executable motions regarding cutting fruits and vegetables. These motions are parameterized by some technique and repetitions to achieve the agents objective. FoodCutting does not require fully available knowledge about the agents environment, instead the robot should be capable of recognizing certain objects for cutting operations.

Pouring?

Motions

Dieser Abschnitt stellt die Arbeiten vor, welche sich mit der Motion Plannung von verschiedenen Mixing Bewegungen auseinadersetzt.

BakingBot

BakeBot realised on the PR2 robotics platform attempts to achieve baking cookies. An implementation of locating relevant things for MixingTasks like a bowl and ingredients has been realised for semi-structured environments. An algorithm to perform mixing motions has been implemented as well, to mix ingredients with different characteristics into an uniform dough. These motions are limited to a simple circular and linear mixing motion. The authors follow an bottom-up approach which is inherently motion driven rather than a symbolic one which attempts to break down high level tasks into executable low level motions.

Mixing Paper

FluidLab

FluidLab is a simulation environment for different kinds of manipulation tasks regarding liquids. Its underlying engine uses differentiable physics enabling reinforcement learning and optimization techniques in manipulation to utilize the engine, to achieve several tasks including liquids and solids, like mixing tasks.

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1641754: relativ alte Quelle, dort möchte man einen Hilfsroboter vorstellen, der mit dem Menschen zusmamen Teilaufgaben im Bereich des Kochens übernehmen kann. Mixing wird als special motion erwähnt und es werden keine Beispiele einer Motionausführung im Bereich des Mixings angeboten. Quelle weiter benutzen?: Eher nicht

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8954776: Bezieht sich kaum auf Mixing Aktion, wird erwähnt dass es getan wird, jedoch nicht weiter darauf eingegangen, reinnehmen? Nein.

- https://robomechjournal.springeropen.com/articles/10.1186/s40648-021-00204-6

- https://www.researchgate.net/profile/Daniela_Rus/publication/265243176_BakeBot_Baking_Cookies_with_the_PR2/links/56d043ad08aeb52500cd34a0.pdf

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9083695: Mixing steht drin, allerdings nur high level commands sowas wie: Mix all the ingredients with a blender, es wird nicht gesagt wie man das mixen soll oder womit etc. Reinnehmen?: Vlt als Beispiel für MixingAktionen wo nicht klar wird wie das abgebildet wird auf motions etc.

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7301404

- https://ieeexplore.ieee.org/abstract/document/8460964

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9096241: Paper, dass die Cooking Roboter beschreibt und kategorisiert, gut als Anlaufstelle um Quellen zu erhalten. Reinnehmen?: Eventuell wenn wir irgendwie kategorisieren wollen wo unser system steht.

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10004056

- https://robbreport.com/gear/electronics/moley-robotics-robot-kitchen-uk-for-sale-123

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8310925

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7523919: Nein

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7523919

- https://www.researchgate.net/publication/361723447_The_use_of_Robotics_in_the_Kitchens_of_the_Future_The_example_of_Moley_Robotics/link/63beb455a99551743e5
download?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F

- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6100855

- https://cs.brown.edu/people/stellex/publications/bollini12.pdf BakeBot Paper related, Mixing wird kurz erwähnt, welche Bewegungen gemacht werden, jedoch nicht in abhängigkeit von irgendwelchen Gegebenheiten wie zb Zutaten. Reinnehmen?: Ja, wenn es sonst nichts gibt ...

There are multiple approaches for mixing tasks in the kitchen domain. From high level symbolic to low level motions and learning approaches, these different approaches, tries to tackle the challenge of solving mixing in the kitchen environment.

FoodCutting aims to equip robotic agents with necessary information about how to execute cutting tasks in unknown environments for the household domain. From high level plans FoodCutting breaks down the cutting task, part of some robot plan, into executable motions regarding cutting fruits and vegetables. These motions are parameterized by some technique and repetitions to achieve the agents objective. FoodCutting does not require fully available knowledge about the agents environment, instead the robot should be capable of recognizing certain objects for cutting operations.

BakeBot realised on the PR2 robotics platform attempts to achieve baking cookies. An implementation of locating relevant things for MixingTasks like a bowl and ingredients has been realised for semi-structured environments. An algorithm to perform mixing motions has been implemented as well, to mix ingredients with different characteristics into an uniform dough. These motions are limited to a simple circular and linear mixing motion. The authors follow an bottom-up approach which is inherently motion driven rather than a symbolic one which attempts to break down high level tasks into executable low level motions.

FluidLab is a simulation environment for different kinds of manipulation tasks regarding liquids. Its underlying engine uses differentiable physics enabling reinforcement learning and optimization techniques in manipulation to utilize the engine, to achieve several tasks including liquids and solids, like mixing tasks.

Our mixing approach will be most similar to the FoodCutting approach, in which we model symbolic knowledge about how to perform mixing tasks, which technique should be used and infering parameters for the execution of the underlying motion.

2 Introduction to Hardware and Software components

This chapter serves as an introduction to the frameworks, software, and libraries we use. The work we present can be simplified explained as a knowledge base containing parameters that a robot can use to perform specific actions.

Firstly, we would like to introduce the robotic component, which acts as the interface between the knowledge base we have created and the robot, followed by an introduction to Ontologies and Knowledgebases.

2.1 Robotic section: ROS and PyCRAM

The main framework we use is *PyCRAM* [2], which serves as an interface for various software components such as knowledge, perception, or manipulation. This framework utilizes another framework, *ROS* (Robotic Operating System) [13], to communicate with the different robot components. These two frameworks are now introduced in the following.

2.1.1 ROS

ROS, which stands for Robot Operating System, is an open-source middleware framework designed to develop and control robots. Despite its name, *ROS* is not a traditional operating system but rather a set of software libraries and tools that help in building and managing robot software. It provides a standardized and modular approach to developing robotic systems, allowing for easier collaboration and code reuse in the robotics community. Key features and components of *ROS* include:

- **Nodes**: *ROS* systems are organized into nodes, which are individual processes that

perform specific tasks. Nodes communicate with each other by passing messages over topics, creating a decentralized and modular architecture.

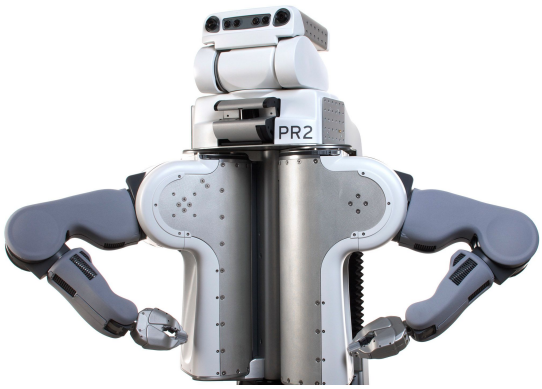
- **Topics**: Nodes exchange data through topics, which are named buses over which messages are passed. This publish/subscribe communication model allows for asynchronous and loosely coupled interactions between nodes.

- **Launch files**: *ROS* uses launch files to specify how to start multiple nodes and configure the system. This helps simplify the setup of complex robotic systems.

- **Master**: The ROS Master is responsible for managing the communication between nodes by keeping track of publishers, subscribers, and services. It facilitates the discovery and connection of nodes within a *ROS* network.

2.1.2 Pr2

Introduced in 2010 by Willow Garage, the *PR2* [1] stands as an advanced research robot. Boasting multiple joints and 20 degrees of freedom, this robot excels in autonomous navigation and the manipulation of a diverse array of objects, making it an ideal choice for our specific needs. Additionally, it is equipped with a HeadStereoCamera that can be used to perceive the surroundings.



2.1.3 PyCram

PyCRAM is a toolbox for designing, implementing and deploying software on autonomous robots. The framework provides various tools and libraries for aiding in robot software development as well as geometric reasoning and fast simulation mechanisms to develop cognition-enabled control programs that achieve high levels of robot autonomy. *PyCRAM* is developed in Python with support for the ROS middleware which is used for communication with different software components as well as the robot.

CRAM (Cognitive Robot Abstract Machine) [5] is a software toolbox for the design, the implementation, and the deployment of cognition-enabled autonomous robots performing everyday manipulation activities. *CRAM* equips autonomous robots with lightweight reasoning mechanisms that can infer control decisions rather than requiring the decisions to be pre-programmed. This way *CRAM*-programmed autonomous robots are much more flexible, reliable, and general than control programs that lack such cognitive capabilities. *CRAM* does not require the whole domain to be stated explicitly in an abstract knowledge base. Rather, it grounds symbolic expressions in the knowledge representation into the perception and actuation routines and into the essential data structures of the control programs.

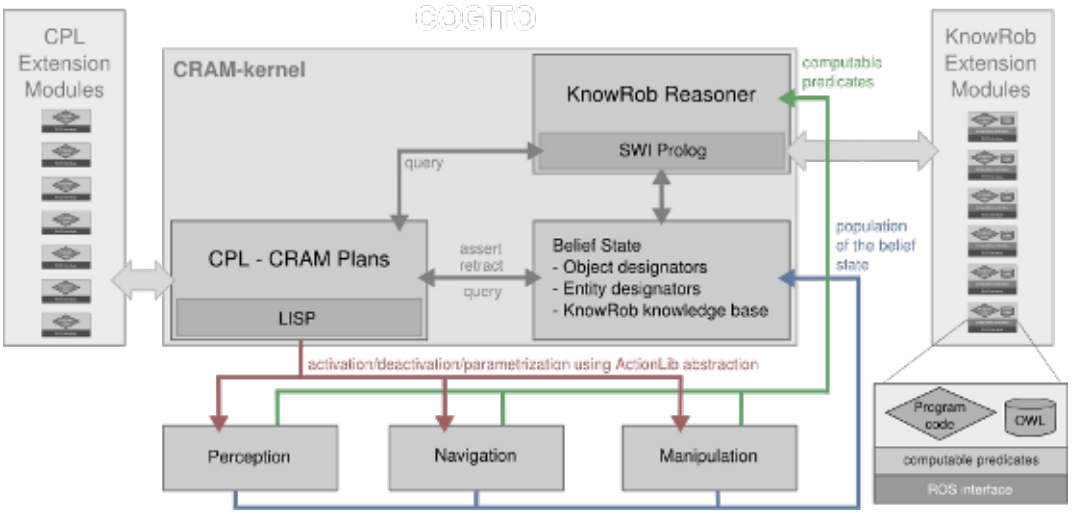


Abbildung 2.1: *CRAM* Language Architecture

2.2 Knowledge Section: Ontologies and Rules

The parameters inferred for various robot actions come from a knowledge base. In the following, the principle of an ontology, as well as the concept of rules, which play a crucial role in parameter inference, will be introduced.

2.2.1 Ontology

Ontologies are structured frameworks that provide a formal representation of knowledge within a specific domain. They play a crucial role in knowledge representation, facilitating the organization and sharing of information in a way that is both machine-readable and understandable by humans.

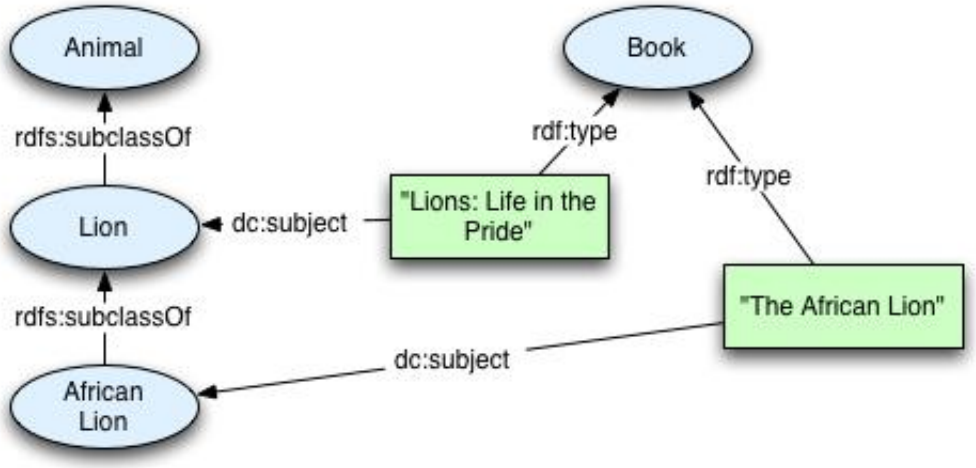


Abbildung 2.2: Ontology example [8]

Key components of ontologies include:

- **Concepts/Classes**: These represent abstract or concrete entities within a domain. For example, in a medical ontology, *Patient* and *Disease* might be classes.
- **Properties/Roles**: These define the relationships between concepts. For instance, in a social network ontology, *FriendOf* could be a property connecting individuals.
- **Instances/Individuals**: These are specific members or examples of a class. In a geographical ontology, *New York City* and *Paris* could be instances of the class *City*.
- **Axioms**: These are statements that describe the properties and relationships of the entities within the ontology. Axioms help define the logic and rules governing the domain.
- **Hierarchy**: Ontologies often organize concepts into a hierarchy, with more general concepts at the top and more specific ones below. This hierarchical structure aids in categorization and understanding.
- **Inference Rules**: These rules define how new information can be derived from existing information in the ontology. Inferences help systems reason and make deductions based on the knowledge encoded in the ontology.

We utilize ontologies in knowledge representation and reasoning systems to empower the robot with the ability to comprehend and handle information in a structured fashion for our specific objectives.

2.2.2 SWRL

SWRL [9], which stands for Semantic Web Rule Language, is a rule language that allows users to define rules about the relationships between classes and individuals in ontologies represented in the Web Ontology Language (*OWL*). *SWRL* is part of the W3C's Semantic Web technology stack and is designed to be used in conjunction with *OWL* to express complex relationships and infer new information based on existing knowledge.

SWRL Rules have a specific syntax and consist of two main components:

- **Antecedent (Body)**: This part of the rule specifies the conditions or constraints that must be satisfied for the rule to be applicable. It describes the current state of the ontology that triggers the rule.

- **Consequent (Head)**: This part defines the actions or inferences that should be taken if the conditions specified in the antecedent are satisfied. It describes the changes or additional information that should be inferred when the rule is triggered.

SWRL supports various built-in predicates and functions, and users can create their own custom rules to suit their specific ontology. Some common elements in *SWRL* rules include:

- **Individuals**: Refers to specific instances of classes in the ontology.

- **Class and Property Relationships**: Describes relationships between classes and properties in the ontology.

- **Built-in Predicates and Functions:** Includes operations such as arithmetic, string manipulation, and comparison functions that can be used in the rule conditions.

Here's a simple example of a *SWRL* rule:
$Person(?x)^h asChild(?x, ?y) -> Grandparent(?x, ?z)^h asChild(?y, ?z)$
In this example:

If an individual (?x) is a Person and has a child (?y),

Then, infer that the individual (?x) is a Grandparent of another individual (?z), and the child (?y) is the parent of (?z).

SWRL rules are useful for expressing complex relationships and constraints within ontologies, enabling automated reasoning systems to make inferences and derive new knowledge from existing data.

2.3 Libraries

2.3.1 RDFLib

RDFLib [10] is a Python package designed to work with *Resource Description Framework (RDF)* data. RDF is a standard model for data interchange on the web and is used to represent information in the form of subject-predicate-object triples. Key features of the library are:

- **Parsing RDF Data**: *RDFLib* can parse RDF data in various formats such as RDF/XML, N-Triples, Turtle, and more.
- **Creating and Modifying RDF Graphs**: It allows you to create and manipulate RDF graphs, which consist of a collection of RDF triples. You can add, remove, or modify triples in the graph.
- **Querying RDF Data**: *RDFLib* provides a *SPARQL* query engine, allowing you to perform queries on RDF graphs using the *SPARQL* query language.
- **Serializing RDF Data**: You can serialize RDF graphs back into different RDF formats using *RDFLib*, making it easy to store or share RDF data.
- **Working with Ontologies**: It supports working with RDF vocabularies and ontologies, enabling the use of predefined classes and properties from existing ontologies like RDF Schema (*RDFS*) or the Web Ontology Language (*OWL*).
- **Integration with Semantic Web Tools**: *RDFLib* can be integrated with other Semantic Web tools and frameworks, allowing you to build applications that leverage Semantic Web technologies.

2.3.2 Owlready

One important library used for our implementation is *Owlready* [11]. *Owlready* is a *Python* library designed for ontology-oriented programming. It facilitates the development, manipulation, and querying of ontologies using the Web Ontology Language (*OWL*), a standard for representing knowledge in a machine-readable format. *Owlready* simplifies ontology-related tasks by providing a convenient and object-oriented interface for working with *OWL* ontologies in *Python*.

Key features of the *Owlready* library include:

- **Object-Oriented Programming (OOP)**: *Owlready* adopts an object-oriented

approach, allowing users to interact with ontology entities as Python-objects. This makes it more intuitive for developers familiar with Python's OOP principles.

- **Ontology Loading and Parsing**: The library supports the loading and parsing of *OWL* ontologies, making it easy to access and manipulate ontology data within Python-scripts or applications.

- **Class and Individual Manipulation**: *Owlready* provides functionality for creating, modifying, and querying classes and individuals within an ontology. This allows for dynamic and programmatic management of ontology content.

- **Reasoning Support**: Depending on the version and features, *Owlready* may offer support for reasoning tasks. Reasoning involves deducing implicit information based on the logical relationships defined in the ontology.

- **Integration with *RDFLib***: *Owlready* may integrate with *RDFLib*, another Python-library commonly used for working with Resource Description Framework (*RDF*) data. This integration enhances the capabilities of handling semantic data.

2.3.3 WikiHow-Instruction-Extraction

WikiHow-Instruction-Extraction [4] is an extraction tool, that can collect informations from the WikiHow Corpus. The goal of this tool is to analyse a WikiHow corpus using basic *NLP* techniques to gather information about Everyday tasks like *Pouring*, *Cutting* or *Discarding*. These information should support cognitive robots in understanding and parameterizing these tasks to better handle unknown tasks, working in underspecified environments and handling common task-object combinations. Every verb has its own class, in which the verb and the additionally desired hyponyms/synonyms are defined. These verbs serve as keywords for the search in the WikiHow articles. Additionally, various parameters can be set, such as excluding different categories that are not relevant for the search.

Data acquisition

A part of this master's thesis involves creating a knowledge base that includes certain queryable parameters, allowing the robot to perform specific actions. The initial step in building the knowledge base is to gather data. In this chapter, we elaborate on our data acquisition strategy.

Task variations

As our main focus is to represent the knowledge about mixing, first we had to acquire the different types and variations of mixing in order to create a complete Knowldegere-presentation. The first step in acquiring the needed data, was to acknowledge which task varations of mixing are actually important. So we had to analyze the word *Mixing* and its hyponyms. But first we have to ask ourselves: **What is Mixing?**

What is Mixing?

The definition provided by the Oxford Dictionary [7] is as follows:

Definition: To put together or combine (two or more substances or things) so that the constituents or particles of each are interspersed or diffused more or less evenly among those of the rest; to unite (one or more substances or things) in this manner with another or others; to make a mixture of, to mingle, blend.

Ultimately, this definition conveys that mixing requires at least two elements or substances, which are then combined (evenly) with each other, resulting in a (new) substance. This definition is general and can be applied to various contexts. For our work, the aspect of cooking or mixing different (cooking) ingredients is important. Therefore, we consider some hyponyms of the verb *mixing* irrelevant for our cause and do not take them into account. An adapted definition for our work could be:

Definition: Mixing is the combination of various (cooking) ingredients through different motions in a container.

Hyponyms

Hyponyms are subordered words of a given word, for example one hyponym of mixing could be beating. To conduct this analysis, one can utilize tools from various websites, such as FrameNet[6] and WordNet[14]. These platforms provide users with the ability to search for specific words and obtain various associations for those words, including synonyms, acronyms, or, crucial for our case, hyponyms. A full list of the considered hyponyms can be seen in table 2.1.

WikiHow extraction

For all those hyponyms, we delegated a WikiHow extraction search (see: WikiHow-Instruction-Extraction) which should show us, how many times one of these words occur, in the context of cooking. To execute the search, we define a new Class *MixingVerb*, which represents the verb *mixing* and its hyponyms.

```
7       MIX( verb: "mix", past: "mixed", participle: "mixing"),
        no usages
8       AMALGAMATE( verb: "amalgamate", past: "amalgamated", participle: "amalgamating"),
        no usages
9       BLEND( verb: "blend", past: "blent", participle: "blending"),
        no usages
10      COALESCE( verb: "coalesce", past: "coalesced", participle: "coalescing"),
        no usages
11      COMBINE( verb: "combine", past: "combined", participle: "combining"),
        no usages
12      COMMINGLE( verb: "commingle", past: "commingled", participle: "commingling"),
        no usages
13      COMPOUND( verb: "compound", past: "compouded", participle: "compounding"),
        no usages
14      CONFLATE( verb: "conflate", past: "conflated", participle: "consolidated"),
        no usages
15      FOLD( verb: "fold", past: "folded", participle: "folding"),
        no usages
16      FUSE( verb: "fuse", past: "fused", participle: "fusing"),
        no usages
17      INTERMIX( verb: "intermix", past: "intermixed", participle: "intermixing"),
18
19      JOIN( verb: "join", past: "joined", participle: "joining"),
        no usages
20      JUMBLE( verb: "jumble", past: "jumbled", participle: "jumbling"),
        no usages
21      LUMP( verb: "lump", past: "lumped", participle: "lumping"),
        no usages
22      MERGE( verb: "merge", past: "merged", participle: "merging"),
        no usages
23      PAIR( verb: "pair", past: "paired", participle: "pairing"),
        no usages
24      UNIFY( verb: "unify", past: "unified", participle: "unifying"),
        no usages
25      UNITE( verb: "unite", past: "united", participle: "uniting"),
        no usages
26      BEAT( verb: "beat", past: "beat", participle: "beating"),
        no usages
27      WHIP( verb: "whip", past: "whipped", participle: "whipping"),
        no usages
28      WHISK( verb: "whisk", past: "whisked", participle: "whisking");
```

Abbildung 2.3: *MixingVerb* definition

Since we have now defined the desired verbs that we ultimately want to search for, let's adjust some search parameters. The crucial parameters involve filtering the categories; in our case, we want to focus on mixing in the cooking domain. Therefore, we filter out all articles that do not exist in the Food and Entertaining category.

Hyponyms occurance

For each defined verb, a search is initiated to determine how often this verb appears in the WikiHow articles [3]. This is done to ascertain which verbs are ultimately relevant for our implementation and which ones we can exclude, as they are infrequently used in everyday language. In the table below the results can be seen.

Hyponym	Occurance
Mix	5300
Amalgamate	0
Beat	956
Blend	1041
Coalesce	1
Combining	3591
Coommingle	0
Compound	0
Conflate	0
Folding	821
Fuse	17
Intermix	0
Join	53
Jumble	0
Lump	7
Merge	6
Pair	352
Stir	6027
Unify	2
Unite	2
Whip	863
Whisk	2267

Tabelle 2.1: Mix synoyms/hyponyms occurance

Further examination and conclusion

This search is ultimately intended to provide us with information on which tasks we want to represent in the knowledge base. Therefore, we decide on certain tasks based on two conditions: frequency and executability. The first condition is easy to understand; tasks that do not occur or occur very rarely are not considered. The second condition relates to the executability of the task in the context of robot movements. Additionally, some tasks with relatively high frequency are also examined more closely, as in English, the past tense is used as an adjective under certain circumstances.

Taking into account the first condition, the following tasks are not considered: *Amalgamate, Coalesce, Comingle, Compound, Conflate, Fuse, Intermix, Join, Jumble, Lump, Merge, Unify, and Unite.*

The second condition excludes another task: *Blend. Blend* is mostly used in the context of a blending machine, which is not handled by the robot. Without this machine, the blending execution cannot be performed correctly, so this task is excluded for us.

Upon closer examination, we will also not consider the verb *Whip* because it is mostly used as an adjective for ingredients, such as whipped cream. This highlights that only the verb *Whip* has relatively low frequency. The same applies to *Pair*, where the past tense is used to describe a combination of different ingredients, such as wine paired with cheese.

Thus, the tasks we consider are: *Mix, Combine, Beat, Fold, Stir, and Whisk.*

Task analysis and defintion

Now that we have selected the tasks, they need to be analyzed to understand the context in which they are ultimately used. Our goal is for the robot to perform these tasks in a manner similar to how a human would. To achieve this, in the next step, we need to closely examine these tasks. It is recommended to analyze videos on WikiHow or other sources where these tasks are presented as activities. The analysis involves observing the movements associated with each task. We present these analyses in tabular form below. The examination includes the task itself, the respective ingredients being processed, the tools used for it, and the container in which the task is carried out. The tasks will be ennumerated, and we will provide the full table with the actual video source in the appendix (HERE REFERENCE TO APPENDIX).

Task	**Tool**	**Container**	**Ingredients**	**Description**
Beating[1]	Whisk	Bowl	Egg yolk (Wet ingredient)	circular, swirling wildly around the bowl
Stirring	Whisk	Bowl	Beaten Egg Yolk (Wet), Parmesan(Powder) and Pepper (Powder)	Circular, from the inside to the outside.
Whisk	Fork	Bowl	Eggs (Wet)	Circular but also straight, wildly motion.
Mixing	Spatula	Pan	Eggs, melted butter (Wet)	Circular, from the inside to the outside, also diving.
Folding	Spatula	Pan	cooked eggs in melted butter (Wet)	Gently motion from the outisde to the inside straight, then moving about 90 degree before going to the inside again.

Tabelle 2.2: Video analysis

After a thorough analysis of the videos and the information provided in them, we conclude that the executed movement is not only related to the task but also influenced by the ingredients used. However, some tasks are deterministic in the sense that the movement is performed regardless of the specific ingredients.

In the following, we aim to structure the extracted information from the videos and present our findings.

Video analysis conclusion and results

Based on the extracted information, we conclude that for our goals, the following aspects, in addition to the task, are important and will be further considered:

- Ingredients: The ingredients play a crucial role in the movement decision associated with the tasks and will be defined more precisely.
- Tools: The tools used may not be decisive in the movement decision.
- Container: As conducted with the tools, the container wont play a crucial in the motion decision.
- Motions: The motions ultimately represent the movement of the robot for our implementation. These movements are extracted from the videos and defined in alignment with robot motions.

Ingredients

Through the video analysis , we come to the conclusion that the nature of ingredients is important. Primarily, ingredients can be divided into two main categories: *Dry* and *Wet* [12]. This division becomes crucial, especially regarding the measurement of quantities for each type of ingredient.

For our case, a somewhat finer categorization of ingredients is necessary to map the various movements sensibly to the given types. In addition to *Wet* ingredients, we introduce a new category: *Liquid* ingredients. This corresponds to ingredients falling under the *Wet* ingredients but with a liquid state, such as milk, water, and oil. This is significant because some movements differ when the given ingredients are *Wet* or *Liquid*.

Another category we include is the *Solid* ingredients category. This differs from the *Dry* category in that it consists of solid components, while we define the *Dry* category to include powdery substances. The *Solid* category encompasses foods like vegetables, fruits, and meats. Through these distinctions, we can create a definition refining the ingredient categories.

Definition: Ingredients are primarily divided into *Dry* and *Wet*, with a need for finer differentiation. An abstracted category called *Liquid* is introduced to encompass liquid states such as milk and water. An additional category, *Solid*, distinguishes itself from

Dry by including solid components like vegetables and meat. These distinctions allow for a precise definition of ingredients, enhancing the analysis of mixing motions in the context of cooking/baking and their representation. In the initial version, the ingredients we defined are for example:

- *Liquid: Milk, Oil, Water, Vinegar, Vanilla Extract, Sauces.*
- *Wet: Egg White, Egg yolk, Butter, Whipped Cream.*
- *Dry: Flour, Salt, Sugar, Baking Soda, Cocoa Powder.*
- *Solid: Onions, Pork, Chicken, Minced meat, Bacon.*

These ingredients are commonly used in the analyzed videos and are also very common ingredients for baking and cooking recipes.

Tools and Container

These come from SOMA and should be accordingly descirbed.

Motions

From this videos we extracted informations about the executed motions. The following motions can be defined:

- **Circular**: Moving the tool in a defined circular movement in the container, not changing the radius during execution
- **Whirlstorm**: Moving from the inside to the outside of the container with the tool, by circulating in an incremented radius.
- **Folding**: Gently motion, where you start from the outside, moving one straight line to the inner side, then picking the tool up and going to the initial state before moving the tool for about 90 degrees, then going back in a straight line to the inner side of the container again.
- **VerticalCircular**: Imagine a line which can be seen as the diameter of the container, from this line one can define certain regions on which you move the tool circular from side to side. This motion is used by the beating task.
- **CircularDivingToInner**: Starting from the outerside, moving the tool in the container arround its edge for about 270 degrees, before diving to the middle of the

container. This motion is used by certain tasks where is required to turn the ingredients over.

Data Acquision Conclusion

In the data acquisition phase, our focus was on creating a foundation of the necessary data and information required for a robot to perform various mixing tasks. After identifying the tasks, we were able to analyze videos to examine the movements associated with each task. We concluded that not only the task itself but also the selection of ingredients is crucial for determining the movement.

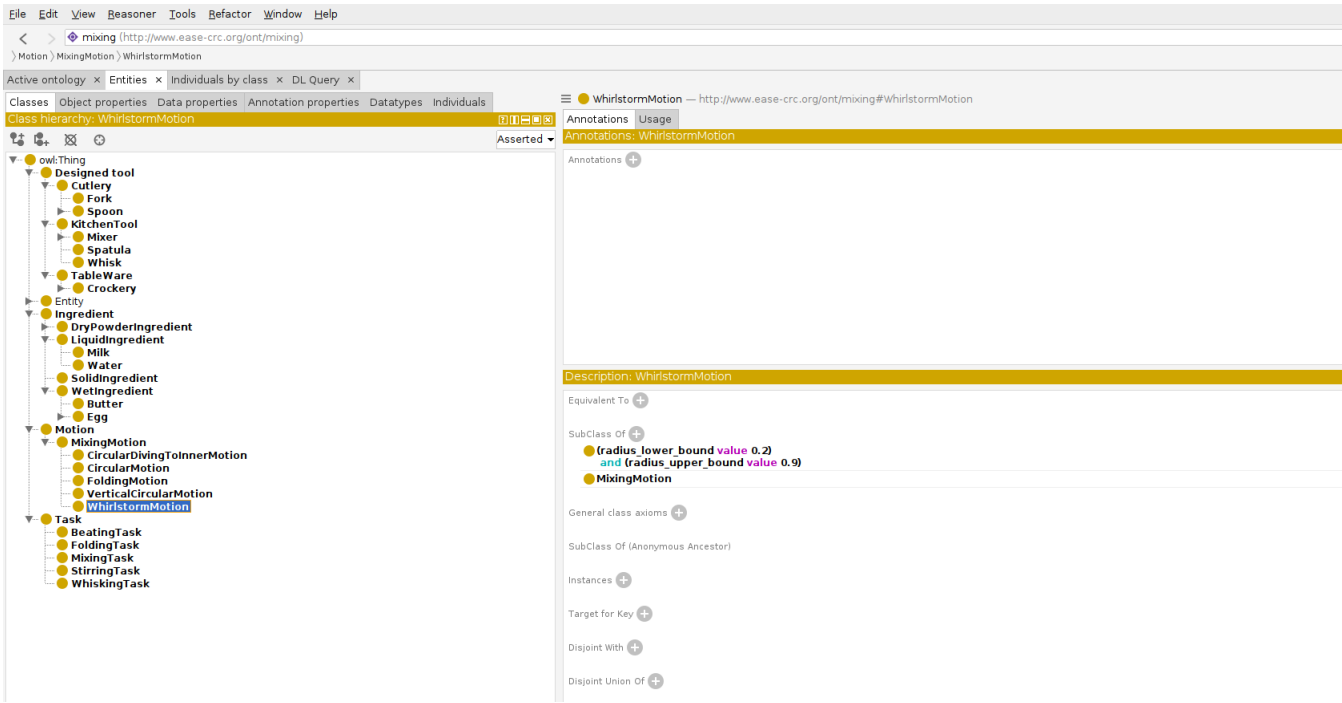
The ingredients are divided into four different categories, and we were able to define five motions that the robot should perform in a manner consistent with everyday use. In the next chapter, we will present how we intend to represent this knowledge and create a system that enables the robot to know which movement to execute based on a given task and set of ingredients.

3 Data Representation

In this chapter we want to introduce our main ideology behind the Data Representation. Our main goal is to determine different motions for different given scenarios, while keeping it as simplified as possible. The main idea behind simplifying the decisions, is motivated through given situations in which the robot would be to dependent on multiple parameters in order to suceed and execute its motion. By simplifying these scenarios the robots decision will rely on less parameters thus the succes rate on making decisions will rise.

The Knowledgebase

We designed a Ontology in which we illustrated our needed classes. The Knowledgebase is divided upon the superclasses Ingredient, Tools, Tasks, Motions and Container.



Ingredients

The Ingredient has 4 subclasses: WetIngredients, SolidIngredients, LiquidIngredients and DryPowderIngredients. The WetIngredients are all the ingredients that can not be considered liquids but also not considered dry and solid, like (melted) butter or eggyolk. Liquid ingredients consists of liquids such as water and milk. Powder ingredients are the ingredients which are dry but also consists of small particles like sugar, salt and flour. Last we have the solid ingredients which consists of ingredients with solid material like vegetables, fruits and meat.

Tools and containers

The tools can be also divided in multiple categories. Cutlery consists of Fork and Spoon, while Spoon also has subclasses like a wooden spoon and tea spoon. Then we have kitchen tools where we can find different types of mixers and whisks. Last we got the superclass Crockery in which our containers we will be saved, like different bowls, pot and mugs.

Tasks

Different Tasks will be saved under the Task superclass. The tasks consists of Mixing, which can be regarded as the umbrela term of all tasks, Stirring which is mostly a task that includes a circular motion, Beating will be mostly used in context with eggs and other wet ingredients, Folding which represents a gentle type of mixing and the last task is whisking which is similar to the beating task. While some of these tasks can be regarded as deterministic when considering the motion decision, like the folding task will always rely on a folding motion, we discovered that the task itself is not as ausschlaggebend, but in combination with the ingredients, one can decide upon a better motion decision.

Motions

The motions are necessary for the robotic system to know what has to be done?. The motions are inferred from Rules that regards a task component, combined with ingredients. The motion also contain parameters, which should determine the moving space available for the robotic system. As in the mixing world, most of the container have a circular formed base, our motions are defined with a radius. The most important parameters are:

- Radius Lower Bound: This parameter describes the smallest possible radius for a motion. For example if the used container is a bowl, that has a radius of 10 cm, a radius lower bound of 0.1 would imply that the smallest possible radius on which the robot can perform its motion is 1 cm.

- Radius Upper Bound: Similar to the radius lower bound, the upper bound determines the maximum radius on which the robot can execute motions.

Our implemented motions are:

- Circular Motion: A circular motion is a motion defined on a circular with a constant radius. This motion basically sets the 2 parameters radius upper bound and radius lower bound on the same value.
  HIER BILD

- Whirlstorm Motion: The Whirlstorm motion covers multiple parts of the container, the motion starts on the center of the container and increments its radius until it reaches the radius upper bound parameter, before turning back to the center.
  HIER BILD

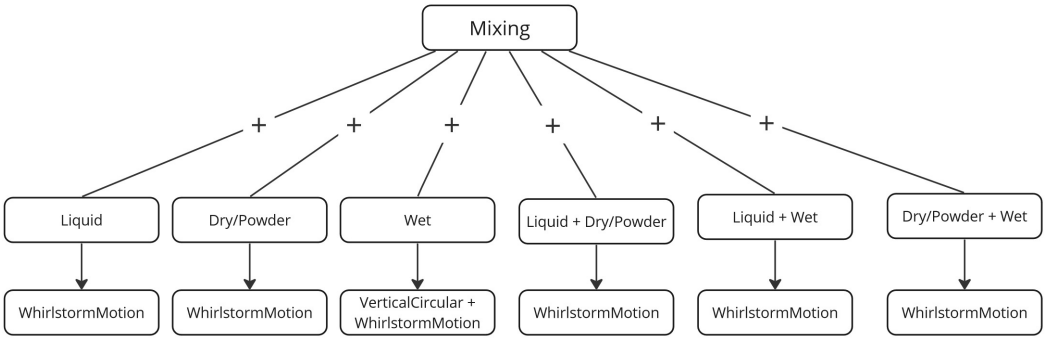
- Folding Motion: Folding is a special motion which is mostly used in the context of the folding task. The motion's goal is to mix the ingredients gently without overmixing the mixture. The motion starts on a point on the circle with the radius of the upper bound parameter, from there the motion draws a straight line to the middle point of the container, then getting back to the start point, next the tool will be moved 90 degrees on the defined circle and it moves back to the middle, this motion is repeteadet 4 times, and then you move the tool 20 degrees to cover all of the ingredients before starting the cycle again.
  HIER BILD

- Vertical Circular Motion: The last motion can also be seen as the most difficult one. The main idea of this motion is to 'wildly' mix the ingredients. Beschreibung fehlt, Bild hier rein.

Rules

In order to infer the right motion based on the ingredients and task input, rules have to be defined. This rules are written in SWRL, which was presented in the USEDLIBRARIES section. In this section we want to illustrate the inference on a high level, while the implementation of it will be shown in the implementation chapter. The rules can be thoguht was if conditions, which will then result in a motion. For example, if we regard the combination of the task Mixing and the ingredient type liquid, we infer the motion Whirlstormmotion. This can be done for every task and ingredient combination. The inference can be illustrated with decision trees which will be shown in the next images for every task and ingredient type combination available. Hier die decision trees?.

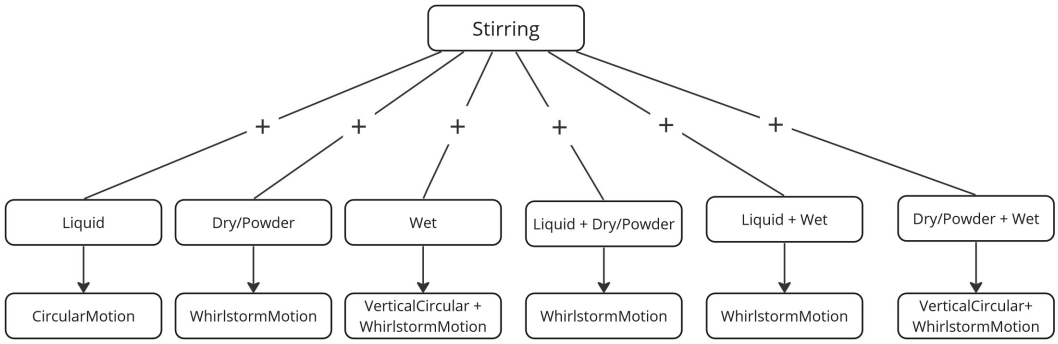
Mixing



Definition: In the context of baking or cooking, a mixing task refers to the process of combining multiple ingredients thoroughly to create a homogeneous mixture. The goal is to distribute the ingredients evenly, ensuring that each component contributes to the overall texture, flavor, and consistency of the final dish or baked good. Mixing is a fundamental step in many recipes and is essential for achieving a balanced and cohesive result.

Äs can be seen from the illustration, the mixing task can primarily be mapped to the Whirlstorm motion. This is not surprising when considering the definition, as this motion results in the uniform distribution of all ingredients in the container."

Stirring

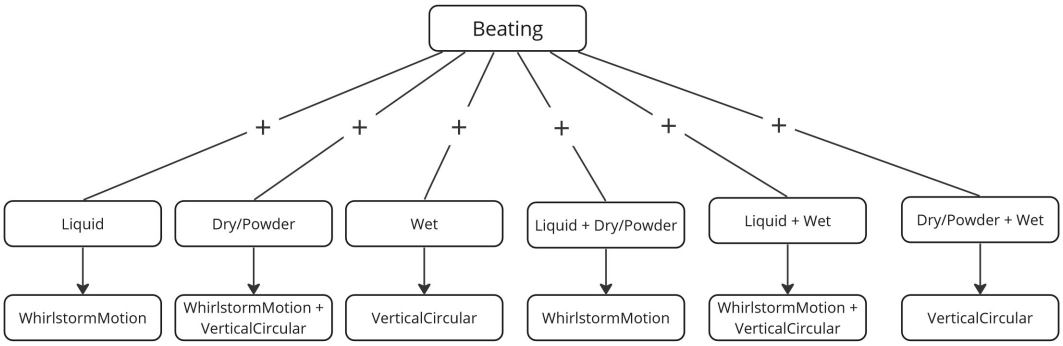


Definition: In the context of baking or cooking, a stirring task involves using a utensil, such as a spoon, spatula, or whisk, to agitate and circulate the ingredients within a mixture. The purpose of stirring is to achieve a uniform distribution of ingredients

In comparison to mixing, the stirring task, depending on the ingredients, maps to a broader range of motions. In addition to the Whirlstorm motion, the Circular Motion is used here for the first time. This is particularly important when considering the stirring task with

the ingredient type Liquid, as one does not want the ingredients to be "whirled,äs would be the case with the Whirlstorm motion, but rather just stirred.

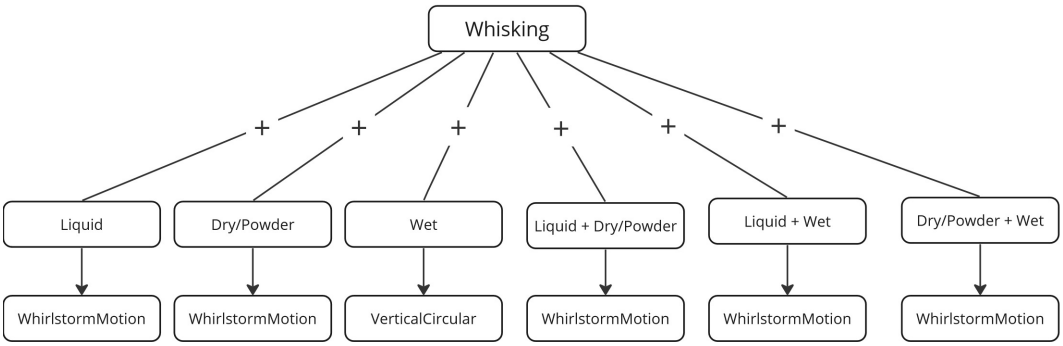
Beating



Definition: In the context of baking and cooking, a "beating"task refers to the process of vigorously stirring or mixing ingredients to achieve a specific texture or consistency. Beating is often done to incorporate air into the mixture, create smooth and uniform blends, or alter the physical properties of certain ingredients.

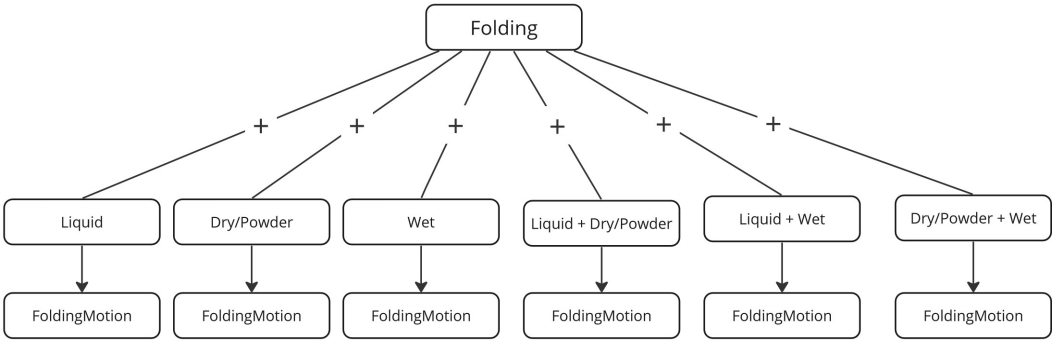
In addition to the Whirlstorm Motion, the VerticalCircular Motion also predominates here. This can be derived from the definition as well, as this motion requires a random mixing style, to which the VerticalCircular Motion aligns.

Whisking



Definion: In the context of baking and cooking, a "whisking"task involves using a kitchen utensil called a whisk to mix, blend, or beat ingredients. A whisk typically consists of wire loops or a coil attached to a handle, and it is designed to incorporate air into mixtures, break up clumps, and create a smooth and uniform texture.

Folding



Definition: In the context of baking and cooking, a "folding"task refers to a gentle mixing technique used to incorporate ingredients without deflating or destroying the air bubbles that have been created. Folding is often employed when combining a lighter mixture (such as whipped cream or beaten egg whites) with a denser one (such as a batter or a heavier mixture). The goal is to maintain the desired texture, lightness, or fluffiness in the final dish.

Theoretically, we wouldn't need to graphically represent the Folding Task since each task and ingredient combination maps to a single motion, the Folding Motion. The graphic was nevertheless included for completeness. The Folding Task requires a specific movement that does not remove the air in the mixture, and any other movement would fail.

How to expand the Knowledgebase

Simulation

In diesem Kapitel stellen wir die Simulation vor, indem der Agent die definierten Motions ausführen kann und dadurch als Proof of Concept dient. Zunächst werden wir die Simulationsumgebung Bulletworld vorstellen, gefolgt von einigen Beispielsaktionen die dort ausgeführt werden können. Anschließend zeigen wir, wie die Parameter über Queries inferriert werden und mit der Visulasierung der Motions zeigen wir, dass die von uns definierten Motions + Parameter vom Roboter in der Simulation ausgeführt werden können.

Simulation Environment

Die Simulationsumgebung ist Bulletworld, welche als Simulationsumgebung für das Framework PyCram dient. Dies ist sehr günstig, denn somit können die in PyCram definierten Aktionen direkt simuliert werden und müssen nicht extra geportet werden. Als Schnittstelle zu den Roboter wird ROS1 genutzt, um mit den Joints zu kommunizieren. Dies geschieht über RosNodes, darüber kann man Informationen über den Zustand der Joints (oder andere Komponente) erhalten, sowie mit diesen Komponenten kommunizieren um Aktionen zu befehlen.

Der für unsere Zwecken genutzten Roboter ist der PR2, dieser ist als Modell schon in der Bulletworld implementiert und ist für unser Fall geeignet, da für die Motions, 2 Arme benötigt werden, eins um die jeweiligen Container zu halten und den anderen Arm, um das Tool, welches für die Aktionen genutzt wird, bewegt wird.

Die Umgebung, welche für unsere definierten Aktionen genutzt wird, ist eine Küche, die Möbel besteht aus einem Tisch, worauf die genutzten Container platziert werden und die Motions dementsprechend ausgeführt werden, sowie weitere Küchenmöbel, welche für unsere Fälle nicht relevant sind. BILDER KÜCHESIMULATION

Die von uns genutzten Objekte sind:

- Container: Bowl (klein und groß), Pfanne, Topf und Tasse. BILDER
- Tools: Whisk, Löffel (klein und groß), Holzlöffel. BILDER

PyCram bietet schon eine betrachtliche Anzahl an implementierten Aktionen, womit der Roboter manevriert werden kann. Unter diesen Aktionen befinden sich notwendige Na-

vigationaktionen, sowie manipulative Aktionen wie Greifen und Platzieren. Außerdem bietet PyCram schon Schnittstellen bereit womit die Joints des Roboters bewegt werden können, diese Funktionen heißen dann zum Beispiel moveTorso(). BILDER CODE FUNKtIONEN

HIER STELLEN WIR UNSERE MOTIONS VOR

Hier Vanessa fragen, wie wir ihre Motion referenzieren sollen, einfahc sagen es war vorhanden oder es detaillierten angeben?

Simulation to RealWorld gap

- Big Problem: Uncertaininty
- Perception Solution as first approach: RoboKudo
- Data acquisition: Blenderproc
- Model Training: YoloV8
- Results showing the real world Perception.

Motions

In this chapter we'll describe on an abstract level how the motions are implemented in pycram2. To simplify the implementation of motions, we hold the following assumptions about the motions which are performed on containers. Firstly we assume, the container is symmetric in the x,y direction. We don't consider the descent of the container, we keep the height for the tool constant over the performed motion. We don't have collision detection.

Circular Motion

Be $container\_center = (container\_center\_x, container\_center\_y)$ Generating points lying on a circle: $x\_coordinates = container\_center + radius\_bounds * cos(radian)$ $y\_coordinates = container\_center + radius\_bounds * sin(radian)$ Where an array of angle values is created and converted to radian values.

INSERT Plot

3.0.1 Folding Motion

Be l an array of x,y coordinates lying on a line, where the starting point of the line is not the center of the container but rather its endpoint. The line is turned via 2D transformation via the rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

The transformation $\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\theta) * \begin{bmatrix} x \\ y \end{bmatrix}$, where $\theta = 90$ in degrees.

This results in the tool being moved 90 degrees from the containers center point, to perform on another area inside the container. The folding motion is repeated 4 times.

Afterwards we transform the line $\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\theta) * \begin{bmatrix} x \\ y \end{bmatrix}$, where $\theta = 22.5$ in degrees, to cover a similar area of the container.

INSERT Plot

Vertical Circular Motion

The vertical circular motion is implemented in the following way. By using the formular to compute points on a circle, we adjust the radius for x,y, where one of the values is set to 0.01, the other one is sampled evenly from the interval $semi\_major\_radius = (radius\_upper\_bound, radius\_upper\_bound/2)$. We take the maximum sampled radius which fulfills the following condition. $\sqrt{(x\_i - center\_x)^2 + (y\_i - center\_y)^2} < radius_upper_bound$ Additionally we decrement and increment y coordinates. If the condition: $\sqrt{(y\_i - center\_y)^2} > radius_upper_bound$ is met we change from decrementing to incrementing and vice versa until this condition is fulfilled again.

Whirlstorm Motion

Using the formular to generate coordinates lying inside a circle we sample radiuses from the interval $radiuses = (radius\_upper\_bound, radius\_lower\_bound)$ This causes the motion to go from outer part of the container to the inner.

Knowledge Graph Visualization Tool

During the process of working on the master's thesis, we were searching for a framework that could visually represent our implemented knowledge graph. Our aim was to clarify the connections and relationships between individual classes. In our search, we found several interesting frameworks that seemed suitable for our needs, which are also mentioned in the Related Work chapter.

However, these solutions weren't entirely satisfactory for us because they lacked some features that we deemed important. Therefore, we decided to develop our own framwework, which should serve as a good visualization framework for our and similar use cases.

In this chapter, we present the framework we developed, from the initial idea to the implementation and the functionalities of our tool.

3.1 Main concept

We decided on 3 main components that the new framework should include:

- Clear Visualization: Our first goal was to facilitate navigation through the knowledge graph by highlighting the relations and classes of the graph as clearly as possible. This could be achieved, for example, by using different colors for different classes or by highlighting a class and its associated classes to which a relation exists. Additionally, we aim to make the graph as clear as possible and reduce the number of nodes to those that are ultimately essential for visualization.

- To clarify the relations between the queries, we implement a query builder that, given a class, can point to other classes through the relation, which in turn can point to further classes through another relation, as long as the user desires or there are no further relations. This should be done without having to use the SPARQL query language, making it much easier for the user. The Query Builder outputs a filtered graph with the selected triples.

- Inference Query: Since a part of our work relies on inferences, we want to implement an inference query that can indicate inferred parameters based on the input of certain classes. This use case is quite specific to our scenario, but it should also be able to

map to other ontologies. The output should be an action tree with the inferred parameters and a visualized graph with the associated classes that play a role in the inference.

3.2 Architecture

The architecture consists of a full-stack framework, utilizing *flask* as the web framework in the backend and *HTML* and *JavaScript* for the frontend. We use a modified version of *bootstrap* for styling the *HTML*-pages. Additionally, in the backend, we utilize the *rdfLib*(section not yet written, 2.3) library to process data from the knowledge base. The *JavaScript* functions are responsible for visualizing the graph, which is done using the *vis.js*(section not yet written, 2.3) library. The following graphics are intended to visually represent a simple description of the architecture, as well as display the folder structure.

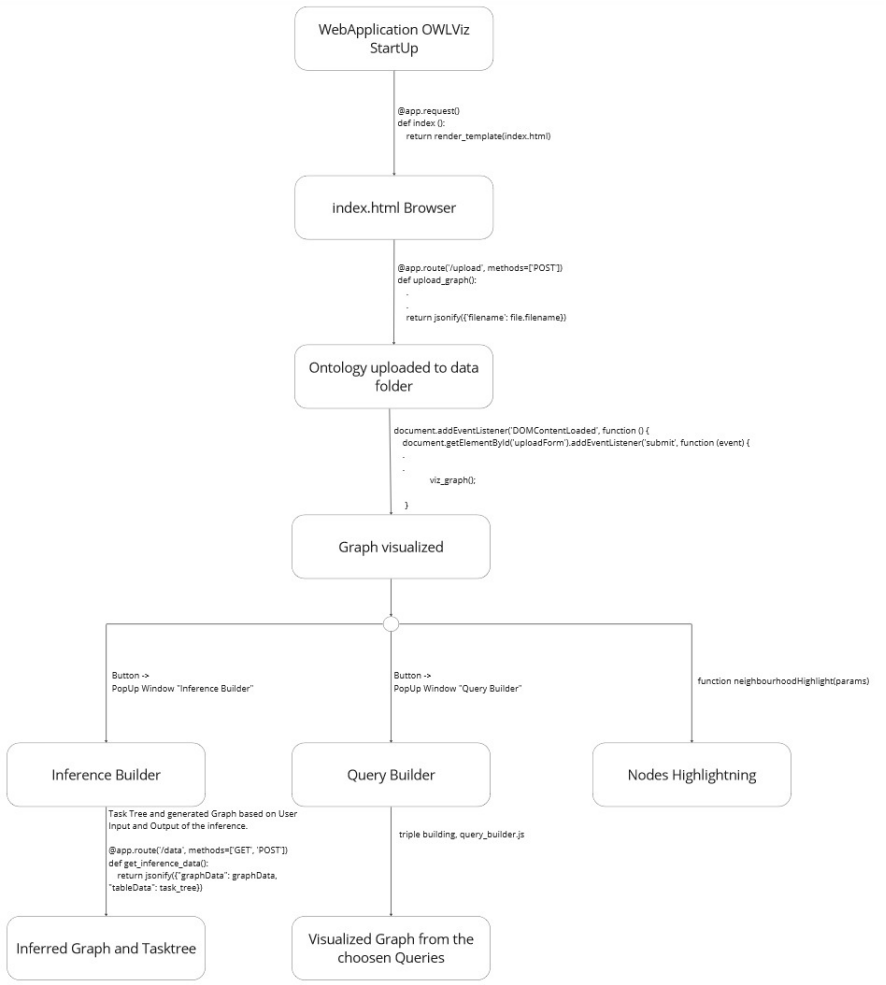


Abbildung 3.1: Architecture chart for the OWLVisualizer framework

```
main.py
templates
    index.html
data
static
    graphviz.js
    query_builder.js
src
    graph
        graph.py
        coloring.py
        graph_utility.py
    inference_builder.py
    query_builder.py
```

Below, we want to take a closer look at the individual points of the architecture from a top-level perspective.

WebApp StartUp

To start the application, the Python script *main.py* has to be executed. This can be done using the following command, provided that the necessary libraries have been installed: *$: python main.py*

This will start the *flask* web application and open the homepage *index.html*.

```python
@app.route('/')
def index():
    return render_template("index.html")
```

Abbildung 3.2: *index()* - function.

- *@app.route(/")* indicates which function is executed first when the framework is started.
- *render_ template(ïndex.html")* indicates which *HTML* page is being called, in this case, *index.html*.

Upload Ontology

The web app initially starts with a (almost) blank *HTML* page, which only contains a navigation bar.

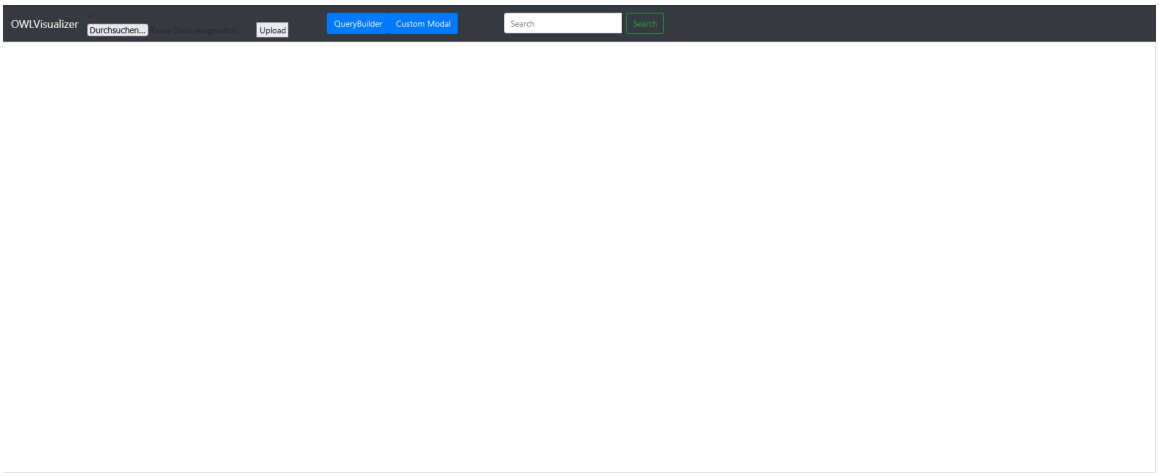


Abbildung 3.3: Startpage of the framework

Now, the user has the option to upload an ontology to work with. The function *upload_graph()* saves the ontology in the *data* folder.

```
1 @app.route('/upload', methods=['POST'])
2 def upload_graph():
3     // check if an ontology file is already present
4     // check if the given file is valid
5     // save the file in the data folder
6     return jsonify({'filename': file.filename})
```

Abbildung 3.4: *upload_graph()* - function.

Visualized Graph

Once the ontology has been uploaded, the *JavaScript* function *viz_graph()* is executed, which ultimately visualizes the graph. This function fetches the required sets of nodes and edges from the backend function *get_graph_data()*.

```
function viz_graph() {
    fetch('/get_graph_data')
        .then(response => response.json())
        .then(data => {
            const container = document.getElementById('mynetwork');
            const options = {};
            network = new vis.Network(container, data, options);
```

Abbildung 3.5: *viz_ graph()* - function.

```
@app.route('/get_graph_data')
def get_graph_data():
    // file_path = path_to_data
    knowledge_graph = Graph()
    knowledge_graph.parse(file_path)
    graph_visualize = graph.get_graph_to_visualize(knowledge_graph)
    graph_data = {'nodes': graph_visualize.get("nodes"),
      'edges': graph_visualize.get("edges")}
    return jsonify(graph_data)
```

Abbildung 3.6: *get_ graph_ data()* - function.

The data processing is complex and will be further elaborated in the Implementation section. Additionally, we have written a section intended to provide a simple introduction to our framework, which can be found in the section Simple Example of our framework . This section aims to illustrate the functionality through a trivial example.

Nodes Highlightning

For large graphs, it can become difficult to maintain an overview of the nodes and their associated relations. Therefore, we are implementing a function that highlights a node as well as its neighboring nodes. This is intended to make it easier to understand the relationships between a class and its associated nodes.

The required function is a *JavaScript* function that takes a clicked node as a parameter and calculates its neighboring nodes based on that. Subsequently, all nodes are grayed out, and the selected node and its neighbors are restored to their original color. This makes the selected nodes appear "highlighted".

```
function neighbourhoodHighlight(params) {
    if (params.nodes.length > 0) {
        // Mark and highlight neighborhood nodes
    } else if (highlightActive === true) {
        // Reset node colors and labels
    }
    // Update node dataset with changes
}
```

Abbildung 3.7: *neighbourhoodHighlight()*-function

Here is an example of a graph with highlighted nodes:

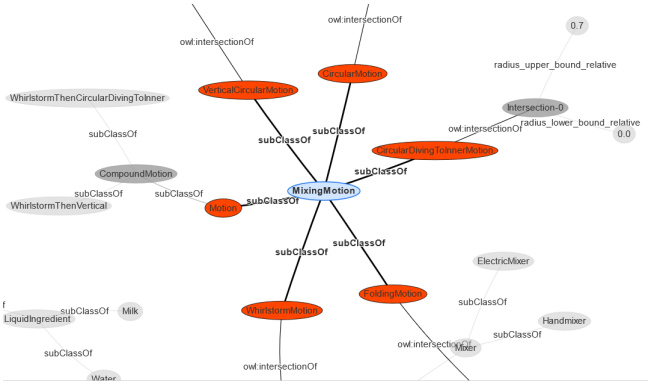


Abbildung 3.8: graph with highlighted nodes

Query Builder - Naser

The *QueryBuilder* is another feature of the OWLVisualizer. Its core feature is making triple matching possible without possesing knowledge of constructing SPARQL queries. Next to triple matching, visualizing selected triples is made possible, to further visualize relationships between concepts and restrictions and instances. Class restrictions in particular are easily queried using the QueryBuilder, without knowing how to actually query them.

Another functionality is to construct a SPARQL query from the matched triples for further usage of the user.

In this section the requirements for this feature are explained.

Requirements The *QueryBuilder* needs a processable graph to apply the pattern matching on. Two different sources of data can be considered. The parsed graph from rdflib or the graph model for visualization. Choosing the graph model for visualization is rational since we introduce consistency between graph visualization and triple matching in the *QueryBuilder*. The main drawback of choosing the graph visualization model is, constructing a SPARQL query from this model instead of the rdflib parsed graph is made difficult since the graph has to be detransformed into the rdflib parsed graph, to then construct the SPARQL query. How this is handled will be discussed in a seperate paragraph.

Triple Matching The concept of triple matching for the *QueryBuilder* is rather simple. Consider a triple consisting of subject, predicate and object. A subject is connected via a set of predicates to a set of objects. For example we have a subject A which has predicates *subClassOf*, *equivalentClass* available objects are B, C, D, where A is a subClassOf

B and A is equivalent to C and D. Suppose A is picked, then available predicates are *subClassOf*, *equivalentClass*. If *subClassOf* is chosen available objects are *B*. In essence the QueryBuilder is doing exactly that.

Initial Triple Matching There is no complete triple matching available if no triples have been selected yet. For example class restrictions are not queryable, they will be queryable once a class with restrictions has been selected. Initial triple matching is applied to OWL-Classes and instances.

Base Triple Matching Once a triple has been selected more options are available. If a OWL-Class has been chosen, the following triples become available: Relationships with classes connected with subClassOf or equivalentClass. Classes with class restriction. Classes and their instances.

If an instance has been picked, relationships between other instances are available between attributes and to which class this instance belongs is available too.

Graphfiltering Based on the matched triples provided by the user, the model for graph visualization filtered based upon the matched triples to visualize it.

SPARQL To generate a SPARQL query based on the selected triples, triples containing restrictions have to be transformed back into rdf graph containing blank nodes. This is relevant for querying in SPARQL, since matching against blanknodes is mandatory to retrieve properties and values/classes out of a restriction.

The resulting query

Inference Builder

The last option for the user to execute functions on a graph is the Inference Builder. It's important to note that the Inference Builder is not a generic use case and in this version, it's tailored to the Mixing Graph (see Data Representation). In the Implementation section, the implementation is further explained, demonstrating how it could potentially be implemented for other ontologies as well. In our case, we infer a motion and its corresponding parameters based on a given task and a potentially long list of ingredients. The Inference Builder then generates a graph that only displays the corresponding nodes for clarity, as well as a task tree that can be followed by an agent. In the first step, the task and the ingredients are sent to the backend for inference, which in turn sends back the inferred motion and parameters to the frontend.

```
$.ajax({
            type: "POST",
            url: "/data",
            data: JSON.stringify(data),
            contentType: "application/json",
            success: function (response) {
                // Handle the response from the server
                const graphData = response.graphData;
                const tableData = response.tableData;
                updateGraphAndTable(graphData, tableData);
            }

  function updateGraphAndTable(graphData, tableData) {
    .
    .
    const data = {
            nodes: graphData.nodes,
            edges: graphData.edges
        };
    .
    .
    const network = new vis.Network(container, data, options);
    .
    .
    tableBody.appendChild(row);

```

Abbildung 3.9: data processing for the inference builder

The backend function processes the inference with the received parameters, which are then used for visualization.

```
@app.route('/data', methods=['GET', 'POST'])
def get_data():
    data = request.json
    task = data.get('task')
    ingredients = data.get('ingredients')
    task_tree, graphData = inference_builder.generate_task_tree_and_graphdata(task, ingredients)
    return jsonify({"graphData": graphData, "tableData": task_tree})

```

Abbildung 3.10: *get_ data()*-function

After processing the data, the graph and the task tree are visualized based on the user input.

Graph and Table ×

Step	Action	Inference
1	Pick up any Tool	Asserted: MixingTool: ['WoodenSpoon', 'Fork', 'ElectricMixer', 'Handmixer', 'Whisk']
2.	Go to the Container	Asserted: Container: ['PastaBowl', 'SaladBowl', 'Cup', 'Mug', 'Pan', 'Pot']
3	Hold the container with the left arm	Asserted Container: ['PastaBowl', 'SaladBowl', 'Cup', 'Mug', 'Pan', 'Pot']
4	Go on the start position with the right arm for the Motion: WhirlstormMotion	Infered Motion: WhirlstormMotion
5	Execute the Motion: WhirlstormMotion with the infered Parameters	Infered Parameters: {'Parameter': 'radius_lower_bound_relative', 'Value': '0.0'}, {'Parameter': 'radius_upper_bound_relative', 'Value': '0.7'}
6.	Put the Tool down left to the container	Asserted: MixingTool: Any, Container: Any
7.	Finish	

Abbildung 3.11: inferred graph and task tree.

The data processing and the detailed implementation of the Inference Builder will be further explained in the Implementation section.

Simple Example of our framework

In this section, we present a simple example of our framework, from processing the ontology to visualization in the web application. This is intended to facilitate the reader's understanding of our framework. For this purpose, we create a small and simple ontology to explain the program flow in a straightforward manner. Additionally, we aim to demonstrate how this ontology is processed and what the interface between the frontend and backend looks like.

Simple Ontology

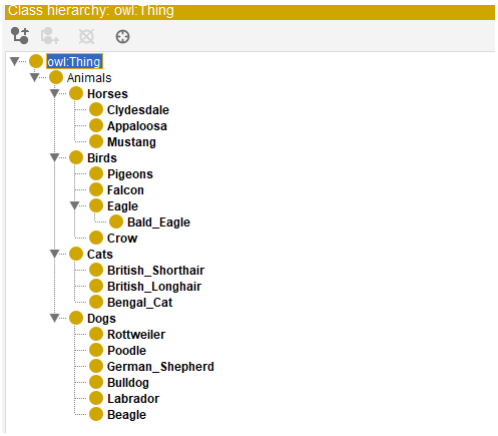


Abbildung 3.12: Simple example of an ontology

The ontology created for this purpose aims to provide a simple representation of various animal species. It includes superclass categories such as *Horses, Birds, Cats, and Dogs*, as well as subclasses like *Labrador* and *Beagle*, which are subclasses of the *Dog* superclass. This ontology does not depict complicated relations; rather, the individual classes are only connected to each other through the *subClassOf* relation.

Ontology processing

Next, the ontology is parsed. Using the *rdfLib*(section not yet written, 2.3) library , the ontology is read and made queryable. The goal is to retrieve all classes of this ontology along with their associated relations. These are then stored in a list, which will be important for visualizing the graph in the frontend.

```python
knowledge_graph = Graph()
knowledge_graph.parse("C:\Dev\simple_ontology_project\data\simple_ontology2.rdf")


def init_graph_to_visualize():
    graph_to_visualize = {'nodes': [], 'edges': []}
    return graph_to_visualize


def get_all_classes(kg: knowledge_graph, graph_to_visualize):
    nodes = set()
    for subclass, superclass in kg.subject_objects(RDFS.subClassOf):
        nodes.add(subclass)
        nodes.add(superclass)
        graph_to_visualize.get("edges").append({'from': str(subclass), 'to': str(superclass), 'label': 'subClassOf'})
    for node in nodes:
        graph_to_visualize.get("nodes").append({'id': str(node), 'label': str(node).split("#")[1]})


def get_graph_to_visualize(kg=knowledge_graph):
    graph_to_visualize = init_graph_to_visualize()
    get_all_classes(kg, graph_to_visualize)
    return graph_to_visualize
```

Abbildung 3.13: Graph initialisation

- Lines 1 and 2: A graph is created, and the ontology is parsed and inserted into this graph.
- Lines 5 to 7: This function serves to pass on the set of nodes and edges for the visualization of the graph. The nodes represent the classes, while the edges represent the relations.
- Lines 10 to 17: This function searches the graph for all classes, which are distinguished between superclass and subclass. The edges then point from a superclass to its corresponding subclass. Both classes are also added to a list of nodes.
- Lines 20 to 23: The set of nodes and edges is initialized for the graph and is now ready for visualization.

With that, the processing of the ontology for the set of nodes and edges for visualization has been completed. Now, we want to visualize these sets.

From Backend to FrontEnd: Visualization

As a reminder: The framework we are using is *flask*, which facilitates communication between the frontend, consisting of *HTML*, *CSS*, and *JavaScript*, and our backend, where the graph is prepared for visualization. First, we'll explain the communication with the frontend.

```python
app = Flask(__name__)


@app.route('/')
def index():
    return render_template("index.html")


@app.route('/get_graph_data_rdf')
def get_graph_data_rdf():
    graph_visualize = graph.get_graph_to_visualize()
    graph_data = {'nodes': graph_visualize.get("nodes"), 'edges': graph_visualize.get("edges")}
    return jsonify(graph_data)


if __name__ == '__main__':
    app.run()
```

Abbildung 3.14: Interface between Front and BackEnd

- Lines 1 to 6: When the application starts, the *HTML* page *index.html* is called.
- Lines 9 to 17: When the function *get_graph_data_rdf* is called, the set of nodes and edges is retrieved in a format that can be processed by the *JavaScript* library *vis.js*.

The function *get_graph_data_rdf* is called in the frontend *JavaScript*.

```javascript
<script>
function renderGraph() {
      fetch('/get_graph_data_rdf')
        .then(response => response.json())
        .then(graphData => {
          var container = document.getElementById('mynetwork');
          var options = {};
          var network = new vis.Network(container, graphData, options);
        })
        .catch(error => {
          console.error('Error fetching graph data:', error);
        });
    }
    window.onload = renderGraph;
  </script>
```

Abbildung 3.15: trivialized example of the FrontEnd

The graph is created using the data extracted from the output of the function *get_graph_data_rdf*. In line 14, it is specified that the function for rendering the graph is called directly upon loading the *HTML* page.

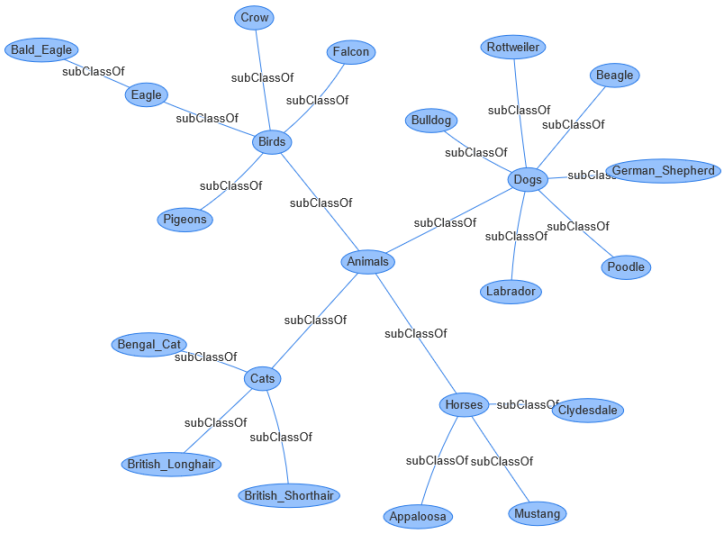


Abbildung 3.16: generatd example-graph

This example illustrated the process of visualizing a simple ontology. However, the ontologies we tested during the implementation were much more complex and required various specific implementations to visualize them. Additionally, we made several improvements to the visualization itself compared to the standard graph. The implementation of the backend, as well as the three different functions (see Main concept), will be described in more detail in the next section.

3.3 Implementation

This section explains the detailed implementation of the framework and highlights the challenges and solutions encountered during implementation. In the previous section, we provided a trivial example intended to serve as an introduction for users to understand the framework's basics. However, since ontologies can vary in structure and some relations can be complex, a simple implementation may not suffice in most cases.

We will begin by explaining the implementation of the backend, and towards the end of the section, we will also address some aspects of the frontend. In the conclusion, we will touch upon points that, in our opinion, could be implemented in the future and discuss any problems/limitations this framework might have.

BackEnd

As explained in the Architecture section and demonstrated in the Introductory Example section (see: Simple Example of our framework), the framework uses *flask* as its backend interface. This ultimately serves to send processed data to the frontend or receive data from it. Now, we'd like to explain how this data is processed and how it has been adequately prepared.

Graph Verarbeitung - Work in Progress

To visualize a graph using the JavaScript library *vis.js* (NOTE), we need a set of nodes and edges. An example of how these could be processed was explained here (NOTE). The existing classes of an ontology are not always connected only by the *subClassOf* relation. Some relations pose challenges in the graph because lists of possible classes related to a superclass can also appear. This poses a problem for visualization because in such cases, so-called *Blank Nodes* appear. These nodes do not represent any information but merely serve as connection nodes between 2 or more classes. These nodes are connected by relations such as *owl:intersectionOf*, *owl:first*, or *owl:rest* and have no added value for our visualization. The first challenge was to adequately process these nodes to remove *Blank Nodes* from the visualized graph and to represent the correct relation between the classes between which these *Blank Nodes* existed

Implementation of the Query Builder - Work in Progress

Implementation of the Inference Builder

As described in the Architecture, the Inference Builder is intended to visualize and organize the results of inference within the ontology. The inference refers to defined *SWRL* rules (see: SWRL), which can then be queried using the *OWLReady2* library (see: Owlready). Therefore, the implementation assumes this type of rules. The rules implemented by us were explained in the Simulation to RealWorld gap chapter. Ultimately, the goal is to execute the inference based on user input and visualize the results.

User Input The user selects a task and any number of ingredients.

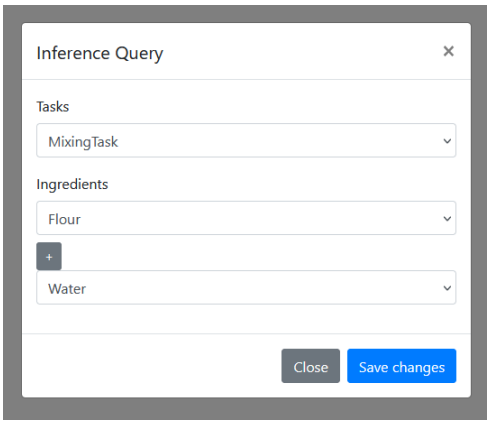


Abbildung 3.17: Select fields for the inference.

These data are extracted from the ontology using the *rdfLib*(section not yet written, 2.3). The following code will explain this further.

```python
def get_tasks():
    task_list = []
    for subj, obj in knowledge_graph.subject_objects(predicate=rdflib.RDFS.subClassOf):
        if "http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Task" in obj:
            task_list.append(str(subj))

    return task_list


def get_ingredients(superclass, knowledge_graph=knowledge_graph):
    ingredients_list = []

    for subj, obj in knowledge_graph.subject_objects(predicate=rdflib.RDFS.subClassOf):
        if superclass in obj:
            ingredients_list.append(str(subj))
            ingredients_list.extend(get_ingredients(subj))

    return ingredients_list


def check_if_leaf(kg, cls):
    is_superclass = False

    for subj, obj in kg.subject_objects(predicate=rdflib.RDFS.subClassOf):
        if cls in obj:
            is_superclass = True
            break

    if not is_superclass:
        return cls


def get_ingredients_leaf():
    leaf = []
    for item in get_ingredients(superclass="http://www.ease-crc.org/ont/mixing#Ingredient"):
        leaf.append(check_if_leaf(knowledge_graph, item))

    filtered_leaf = list(filter(lambda x: x is not None, leaf))

    return filtered_leaf
```

Abbildung 3.18: Code snippet for querying the tasks and ingredients from the ontology

- Lines 1 - 7: Each triple where the predicate of the relation matches *subClassOf* is queried. Additionally, it checks whether the object corresponds to the class *Task*. These elements are then added to a list, which is also returned.

- Lines 10 - 18: Similar to the above function, the subclasses of ingredients are returned. The only difference is that the function is called recursively since the class *Ingredients* has subclasses at multiple levels. Therefore, it also checks which class has no more subclasses.

- Lines 21 - 30: This function checks whether a given class corresponds to a leaf in a graph, meaning this class itself has no subclasses.

- Lines 33 - 40: This function ultimately returns the list of ingredients available for the user's selection.

These data are sent to the frontend using the following function:

```python
@app.route("/task_ingredients")
def get_tasks_and_ingredients():
    return jsonify({"tasks": inference_builder.get_tasks(),
                    "ingredients": inference_builder.get_ingredients_leaf()})
```

Abbildung 3.19: *get_ tasks_ and_ ingredients()*-function

The previously described functions are called, and the data is sent to the frontend in *JSON* format. In the frontend, this data is fetched and processed.

```javascript
jQuery(document).ready(function ($) {
        // Function to fetch and populate tasks
        $.ajax({
            url: '/task_ingredients',
            type: 'GET',
            success: function (data) {
                var taskSelect = $('#taskSelect');
                $.each(data.tasks, function (index, task) {
                    taskSelect.append($('<option>', {
                        value: task,
                        text: task.split("#")[1]
                    }));
                });
                var ingredientSelect = $('#ingredientSelect');
                $.each(data.ingredients, function (index, ingredient) {
                    ingredientSelect.append($("<option>", {
                        value: ingredient,
                        text: ingredient.split("#")[1]

                    }));
                });
            }
        });
    });
```

Abbildung 3.20: Data is fetched in the FrontEnd

This function populates the entries of the select fields for task and ingredient selection with the data from the backend. By doing so dynamically, it ensures that even if the ontology has new entries in these classes, they will always be available for the user's selection.

Inference An example can illustrate the process of inference calculation. Taking the above example (see: 3.11), we have the input task: *MixingTask* and the ingredients: *Flour* and *Water*. These two ingredients correspond to the superclasses *DryPowderIngredient* and *LiquidIngredient*, respectively (see: Data Representation for more information about the structure of the ontology.). This combination ultimately corresponds to the SWRL-Rule:

```
MixingTask(?x) ^ DryPowderIngredient(?ing1) ^ hasIngredient(?x, ?ing1)
^ LiquidIngredient(?ing2) ^ hasIngredient(?x, ?ing2) ^ Motion(?motion)
^ performMotion(?x, ?motion) -> WhirlstormMotion(?motion)
```

From this, the motion *WhirlstormMotion* is inferred with the corresponding parameters:

```
radius_lower_bound_relative   0.0
radius_upper_bound_relative   0.7
```

To calculate this inference, we use the library *OWLReady2* (see: Owlready), which enables us to utilize a reasoner for the inference.

```
1  def get_inference(task, ingredients):
2      onto = get_ontology("C:\Dev\OWLVisualizer\static\ontologies\mixing.owl").load()
3      task_name = task.split("#")[1]
4      task_instance = onto[task_name](f"{task_name}-1")
5      ingredients_list = []
6      for i in ingredients:
7          ingredient = i.split("#")[1]
8          ingredient_instance = onto[ingredient](f"{ingredient}-1")
9          ingredients_list.append(ingredient_instance)
10         task_instance.hasIngredient.append(ingredient_instance)
11
12     motion = onto.Motion("motiontop")
13     task_instance.performMotion.append(motion)
14     union = set()
15     for ingredient_instance in ingredients_list:
16         ing_ancestors = set(ingredient_instance.is_a[0].ancestors())
17         union = union.union(ing_ancestors)
18     intersection1 = set(onto.Ingredient.subclasses()).intersection(union)
19     rules = set()
20     for r in onto.rules():
21         body_classes = {pred.class_predicate for pred in r.body}
22
23         if len(body_classes.intersection(intersection1)) == len(intersection1) \
24                 and task_instance.is_a[0] in body_classes:
25             rules.add(r)
26
27     sync_reasoner(infer_property_values=True)
28     a = motion.is_a[0]
29     motion = a.name
30     triples = []
31     get_restrictions_recursive(rdflib.URIRef(a.iri), triples)
32     parameters = []
33
34     temp_dict = {}
35     for item in triples:
36         if isinstance(item[1], rdflib.term.URIRef) and str(item[1]).endswith("onProperty"):
37             temp_dict["Parameter"] = str(item[2]).split("#")[-1]
38         elif isinstance(item[1], rdflib.term.URIRef) and str(item[1]).endswith("hasValue"):
39             temp_dict["Value"] = str(item[2])
40             parameters.append(temp_dict.copy())
41             temp_dict.clear()
42
43
44
45     return motion, parameters, task_instance
```

Abbildung 3.21: Inference function

- Lines 1 - 10: The classes are instantiated. The function takes parameters *task*, corresponding to a task, and *ingredients*, corresponding to a list of ingredients. In line 2, the ontology for the *OWLReady2* library is instantiated to utilize the reasoner. In lines 3 and 4, the *task* is instantiated. Since the task comes as input in the full *IRI* format, it needs to be processed first. From lines 5 to 10, the same is done analogously for the ingredients, this time only for a list of ingredients. Subsequently, the instances of the *ingredients* are added to the instance of *task*.

- Lines 12 - 13: A top-level *Motion* is instantiated to indicate that the *Motion* is inferred during reasoning. ???

- Lines 14 - 18: Since the superclasses of the *ingredients* are important for the rules, they are added to a set which will be used later for inference.

- Lines 19 - 25: The existing rules are examined and matched with the input to infer the correct *SWRL* rules.

- Lines 27 - 45: Here, the reasoner is first started. The inferred motion is stored, and now it is about determining the required parameters based on the motion. These parameters are then stored in a dictionary. The function returns the motion, the parameters, and the instance of *task*.

Preparing the Output For clarity, we have decided to reduce the graph to only the classes used in the inference. The middle node of the graph represents the task instance selected by the user. From this node, you can reach the other classes that are important for the inference. These classes correspond to the classes of the ingredients, i.e., the ingredient instances selected by the user, the inferred motion along with parameters, and each instance for the tools and containers, each of which can represent any possible combination if no selection has been made.

```
1  def generate_task_tree_and_graphdata(task, ingredients):
2
3      motion, parameters, task_instance = get_inference(task, ingredients)
4
5      graph = init_graph_to_visualize()
6      nodes = set()
7      for property in task_instance.get_properties():
8          for value in property[task_instance]:
9              nodes.add(value)
10             graph.get("edges").append({'from': task_instance.name, 'to': value.name, 'label': property.name})
11             for cls in value.is_a:
12                 if property.name == "hasIngredient":
13                     graph.get("edges").append({"from": cls.name, "to": cls.is_a[0].name, 'label': "subClassOf"})
14                     nodes.add(cls.is_a[0])
15                 graph.get("edges").append({'from': cls.name, 'to': "Thing", 'label': "subClassOf"})
16                 graph.get("edges").append({'from': value.name, 'to': cls.name, 'label': "is a"})
17                 nodes.add(cls)
18
19     graph.get("edges").append({'from': task_instance.name, 'to': task_instance.is_a[0].name, 'label': "is a" })
20     nodes.add(task_instance.is_a[0])
21     nodes.add(task_instance)
22
23     for node in nodes:
24         graph.get("nodes").append({'id': node.name, 'label': node.name})
25
26     for para in parameters:
27         graph.get("nodes").append({'id': str(para["Parameter"]), 'label': str(para["Parameter"]),"color": "yellow"})
28         graph.get("edges").append({'from': str(para["Parameter"]), 'to': str(motion), 'label': "subClassOf"})
29         graph.get("nodes").append({'id': str(para["Value"]), 'label': str(para["Value"]), "color": "orange"})
30         graph.get("edges").append({'from': str(para["Parameter"]), 'to':str(para["Value"]), 'label': "value"})
31
```

Abbildung 3.22: generating the visualized graph and task tree

- Line 3: The results of the inference function (see: Inference) are stored and used for further processing.

- Lines 5 and 6: We initialize an empty graph, which will be filled with nodes and edges throughout the function. Additionally, a set is initialized for the set of nodes.

- Lines 7 to 17: Based on the task instance, the graph is structured starting from this node. It checks which relations exist for this instance and accordingly inserts into the

set of edges and nodes. Additionally, for the *hasIngredient* relation, the superclass of the ingredient is considered. Finally, the edge labeling is chosen based on the relation, and the classes are added to the graph.

- Lines 19 to 21: The task instance itself is processed and added to the graph.
- Lines 23 and 24: Iterating over the set of nodes, they are added to the graph.
- Lines 26 to 30: The inferred parameters are processed and added to the graph in the end.

For the task tree, we create a 3-column table. The first column represents the step, the second column represents the action, and the third column represents the parameters used to illustrate them. The task tree is a list of entries, which can also contain dynamic data such as motions and ingredients.

```
32      first_entry = {"col1": "1", "col2": "Pick up any Tool", "col3": "Asserted: MixingTool: " + str(get_tool_leaf()
    [1])}
33      second_entry = { 'col1': '2.', 'col2': 'Go to the Container', 'col3': "Asserted: Container: " +
    str(get_container_leaf()[1])}
34      third_entry = {'col1': '3', 'col2': 'Hold the container with the left arm', 'col3': "Asserted Container: " +
    str(get_container_leaf()[1])}
35      fourth_entry = {'col1': '4', 'col2': 'Go on the start position with the right arm for the Motion: ' + motion,
36                      'col3' : "Infered Motion: " + motion}
37
38      fifth_entry = {'col1': '5', 'col2': 'Execute the Motion: ' + motion + ' with the infered Parameters', 'col3':
    "Infered Parameters: " +
39
    str(parameters).replace("]","").replace("[", "")}
40
41      sixth_entry = {'col1': '6.', 'col2': 'Put the Tool down left to the container', 'col3':"Asserted: MixingTool:
    Any, Container: Any"}
42
43      seventh_entry ={'col1': '7.', 'col2': 'Finish', 'col3': ""}
44
45      task_list = [first_entry, second_entry, third_entry, fourth_entry, fifth_entry, sixth_entry, seventh_entry]
46
47      return task_list, graph
48
```

Abbildung 3.23: generating the visualized graph and task tree

- 1: The first entry pertains to the robot's action, which is to choose a tool for the upcoming actions. The list of *Tools* corresponds to the subclasses of *Tools* from the ontology. (REFERENCE TO GET TOOLS LEAF)
- 2 and 3: Analogous to the first entry, the container is chosen and must be held with an arm for the upcoming motion.
- 4: This action describes the starting point of the motion; each motion has its own starting point (see: Simulation to RealWorld gap).
- 5: In this step, it is explained with which parameters the motion must be executed.
- 6: Once the motion execution is complete, the used tool is set aside.

- 7: The final step does nothing but announce the end of the task tree.

These data are forwarded to the frontend via the *flask* interface, where ultimately the graph and the task tree are visualized (see: 3.11)

Fazit -

Implementation

- OWLReady
- SWRL
- Inferring parameters for the actual plan

Evaluation

- Evaluation of multiple motions multiple times
- Results of that

Summary / Fazit

avc

Literaturverzeichnis

[1] *PR2*. https://robotsguide.com/robots/pr2 [Accessed: 03.05.2024].

[2] *PyCram*. https://github.com/cram2/pycram [Accessed: 03.05.2024].

[3] *wikiHow*. https://www.wikihow.com/Main-Page [Accessed: 03.05.2024].

[4] *WikiHow Instruction Analysis for Robot Manipulation*. https://github.com/Food-Ninja/WikiHow-Instruction-Extraction.

[5] Beetz, Michael, Lorenz Mösenlechner und Moritz Tenorth: *CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments*. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Seiten 1012–1017, Taipei, Taiwan, October 18-22 2010.

[6] Berkely, University of California: *FrameNet*. https://framenet.icsi.berkeley.edu/ [Accessed: 03.05.2024].

[7] Dictionary, Oxford English: *mix VERB, Meaning and use*, 2002. https://www.oed.com/dictionary/mix_v?tab=meaning_and_use [Accessed: 03.05.2024].

[8] Group, W3C Working: *Representing Classes As Property Values on the Semantic Web*, 2005. https://www.w3.org/TR/2005/NOTE-swbp-classes-as-values-20050405/ [Accessed: 03.05.2024].

[9] Horrocks, Ian, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosofand und Mike Dean: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. W3C Member Submission, Mai 2004. http://www.w3.org/Submission/SWRL/, Last access on Dez 2008 at: http://www.w3.org/Submission/SWRL/.

[10] Krech, Daniel, Gunnar AAstrand Grimnes, Graham Higgins, Jörn Hees, Iwan Aucamp, Niklas Lindström, Natanael Arndt, Ashley Sommer, Edmond Chuc, Ivan Herman, Alex Nelson, Jamie McCusker, Tom Gillespie, Thomas Kluyver, Florian Ludwig, Pierre Antoine Champin, Mark Watts, Urs Holzer, Ed Summers, Whit Morriss, Donny Winston, Drew Perttula, Filip Kovacevic, Remi Chateauneu, Harold Solbrig, Benjamin Cogrel und Veyndan Stuart: *RDFLib*, August 2023. https://github.com/RDFLib/rdflib.

[11] Lamy, Jean Baptiste: *Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies*. Artificial Intelligence in Medicine, 80:11–28, 2017, ISSN 0933-3657. https://www.sciencedirect.com/science/article/pii/S0933365717300271.

[12] Ohene, Michael: *A Proposed General Formula to Create and Analyze Baking Recipes*. 2017.

[13] Stanford Artificial Intelligence Laboratory et al.: *Robotic Operating System*. https://www.ros.org.

[14] University, Princeton: *WordNet, A Lexical Database for English*. https://wordnet.princeton.edu/ [Accessed: 03.05.2024].