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 July 4, 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.

Contents

1 Introduction **6**
1.0.1 old . . . 6
1.0.2 new . . . 7

2 Related Work **9**

3 Introduction to Hardware and Software components **11**
3.1 Robotic section: ROS and PyCRAM . . . 11
3.1.1 ROS . . . 11
3.1.2 Pr2 . . . 12
3.1.3 PyCram . . . 12
3.1.4 PyBullet . . . 13
3.2 Knowledge Section: Ontologies and Rules . . . 14
3.2.1 Ontology . . . 14
3.2.2 SWRL . . . 15
3.3 Software Libraries . . . 16
3.3.1 RDFLib . . . 16
3.3.2 Flask . . . 17
3.3.3 Owlready . . . 17
3.3.4 WikiHow-Instruction-Extraction . . . 18

4 Data Acquisition **19**
4.1 Task variations . . . 20
4.1.1 Hyponyms . . . 20
4.2 Task Analysis and Defintion . . . 22
4.2.1 Video Analysis Conclusion and Results . . . 23
4.3 Data Acquisition Conclusion . . . 31

5 Data Representation **32**
5.1 The Knowledgebase . . . 32
5.1.1 Ingredients . . . 34
5.1.2 Tools and containers . . . 35
5.1.3 Tasks . . . 36
5.1.4 Motions . . . 37
5.2 Rules - Grafiken anpassen . . . 38
5.2.1 Mixing . . . 39
5.2.2 Stirring . . . 40
5.2.3 Beating . . . 40

5.2.4 Whisking . . . 41
5.2.5 Folding . . . 41

6 Execution on the Robot **43**
6.1 Simulation Environment . . . 43
6.2 Control Primitives . . . 45
6.3 Simulated Motions . . . 47
6.3.1 Whirlstorm Motion . . . 47
6.3.2 Circular Motion . . . 47
6.3.3 Horizontal Eliptical . . . 48
6.3.4 Folding Motion . . . 48
6.4 Simulation to Real-World gap . . . 49
6.5 Challenges and Difficutlies, Summary of the Simulation . . . 50
6.6 Implementation in PyCram . . . 51
6.6.1 Robots Plan Execution . . . 51
6.6.2 Asserted Knowledge . . . 51
6.6.3 Inferring Knowledge . . . 51
6.6.4 MixingActionSWRL . . . 52
6.6.5 MixingActionDesignator . . . 52
6.7 Evaluation . . . 55
6.7.1 Mixing Task . . . 55
6.7.2 Beating Task . . . 56
6.7.3 Stirring Task . . . 58
6.7.4 Whisking Task . . . 59
6.7.5 Folding Task . . . 60

7 Knowledge Graph Visualization Tool **61**
7.1 Main concept . . . 61
7.2 Features . . . 62
7.2.1 Graph Visualization . . . 62
7.2.2 Graph Processing . . . 63
7.2.3 Load Ontology . . . 66
7.2.4 Search Classes . . . 67
7.2.5 Custom Inference - Mixing . . . 68
7.2.6 QueryBuilder . . . 70
7.3 Implementation . . . 73
7.3.1 BackEnd . . . 73
7.4 Fazit . . . 78
7.4.1 Weaknesses . . . 78

8 Conclusion **79**
8.1 Summary . . . 79
8.2 Future Work . . . 80
8.3 Last Words . . . 80

9 Appendix **84**
9.1 Video Analysis 84
9.2 Evaluation 86
9.2.1 Mixing Task 87
9.2.2 Stirring Task 88
9.2.3 Beating Task 89
9.2.4 Whisking Task 91
9.2.5 Folding Task 92

1 Introduction

1.0.1 old

This master's thesis addresses the issue of mixing and its variations in the context of cooking as high-level tasks, taking into account contextual knowledge, in order to derive statements about which mixing movements a robotic agent should perform using a knowledge base. Before we begin presenting the structure of the master's thesis, we would like to address five essential questions to provide readers with a better overview:

- **What is the problem?** In the world of cooking and baking, various tasks need to be performed. While a human learns over their lifetime how to prepare recipes and intuitively execute the necessary actions, a robot must have these required actions readily available. A robot's tasks in the context of cooking can be diverse, such as cutting, pouring, or mixing. In this thesis, we focus on the task of mixing and examine the issue of different mixing actions and the context in which they are performed.

- **Why is it important?** In the context of mixing, various movements need to be considered to achieve the desired results. It is not sufficient to consider a single movement as a general movement in the context of mixing, as different outcomes of a mixing action cannot be achieved otherwise. Therefore, different movements must be taken into account to achieve the desired results of a mixing action.

- **Why is it difficult?** At the current state of research, there is no systematic approach to determine which mixing movement should be performed under specific circumstances. Therefore, a systematic approach will be necessary, primarily utilizing video material. The challenge is to identify the relationships between the actions performed by humans, the ingredients used, and the associated movements. However, the biggest problem is deciding how to model this knowledge in a symbolic knowledge base so that the relevant knowledge is usable for the robot.

- **How is the problem solved?** Through the analysis of videos, we aim to determine how general principles can be inferred from individual cases, especially regarding human actions during the mixing of ingredients. We consider both the nature of the ingredients and the movements executed. The identified relationships will be modeled in a knowledge base, allowing a robotic agent to derive a possible movement with necessary parameters for execution. The knowledge base is symbolically modeled,

enabling theoretical support for execution by many robotic agents.

- **How can this work contribute to research?** By implementing our knowledge base, it is possible for robotic agents to query the required knowledge for mixing actions. Our contribution is aimed at the development of robots capable of performing non-trivial tasks autonomously in household settings without having a predefined environment.

1.0.2 new

This master's thesis addresses the issue of mixing and its variations in the context of cooking as high-level tasks, taking into account contextual knowledge, to derive statements about which mixing movements a robotic agent should perform using a knowledge base.

Before presenting the structure of the master's thesis, we address five essential aspects to provide readers with a better overview. The problem at hand involves various tasks in the world of cooking and baking. While humans learn over their lifetime how to prepare recipes and intuitively execute the necessary actions, a robot must have these required actions readily available. A robot's tasks in cooking can be diverse, such as cutting, pouring, or mixing. This thesis focuses on the task of mixing and examines the issue of different mixing actions and the context in which they are performed.

The importance of considering various movements to achieve desired results in mixing is emphasized, as a single movement cannot suffice for different outcomes. Different movements must be taken into account to achieve the desired results of a mixing action. The difficulty arises from the lack of a systematic approach to determine which mixing movement should be performed under specific circumstances. A systematic approach, primarily utilizing video material, is necessary to identify the relationships between the actions performed by humans, the ingredients used, and the associated movements. The biggest challenge lies in deciding how to model this knowledge in a symbolic knowledge base so that the relevant knowledge is usable for the robot.

To solve this problem, the analysis of videos is employed to determine how general principles can be inferred from individual cases, especially regarding human actions during the mixing of ingredients. Both the nature of the ingredients and the movements executed are considered. The identified relationships are modeled in a knowledge base, allowing a robotic agent to derive possible movements with the necessary parameters for execution. The knowledge base is symbolically modeled, enabling theoretical support for execution by many robotic agents.

This work contributes to research by enabling robotic agents to query the required knowledge for mixing actions. The implementation of our knowledge base aims at developing robots capable of performing non-trivial tasks autonomously in household settings without a predefined environment.

Our work is divided into multiple chapters that will cover every aspect of our research and practical work. 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 chapters, 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. Before we will delve into the discussion of our results and draw conclusions to wrap up our work, we want to present a newly implemented web framework for graph visualization.

The following graphic is intended to provide an overview and will be referenced in each chapter to indicate which part of the work the chapter is addressing.

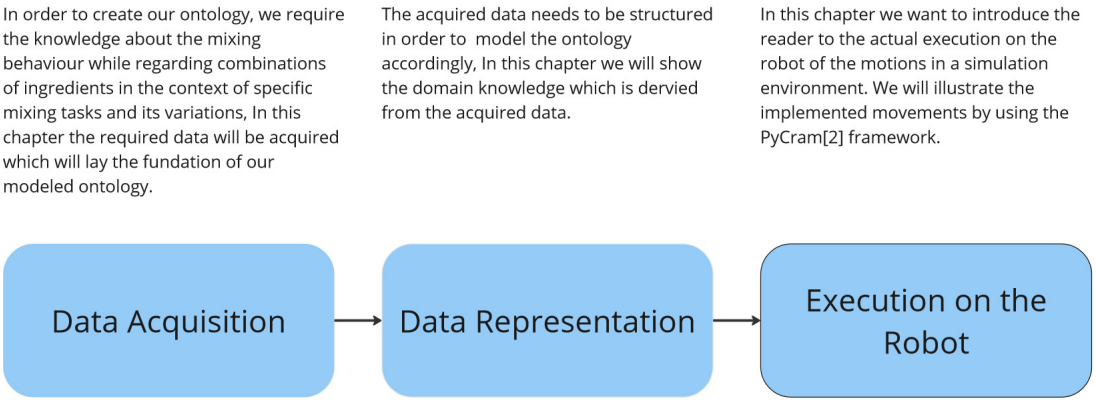


Figure 1.1: Structure overview

2 Related Work

There are some 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 [18] 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* [18] 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* [18] does not require fully available knowledge about the agents environment, instead the robot should be capable of recognizing certain objects for cutting operations.

BakeBot [10] realised on the *Pr2* robotics platform attempts to achieve baking cookies. An implementation of locating relevant things for mixing tasks 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.

Robotic roommates making pancakes [6] is another paper which shows how robot behave in a baking environment. This article is about two robots that bake pancakes through collaboration. Although the mixing motions are not explicitly mentioned in this article, it shows an approach to actions through reasoning. This approach is similar to our work. As a result, the agent is able to reason a sequence of actions based on an input.

Some interesting approaches pursue robot learning by tracking human movement [12]. By learning these movements, they can be transferred to primitive motions and parameterized based on tracking. In our work, we do not pursue learning through demonstrations; however, this paper mentions parameterizing motion primitives (such as a radius), which is similarly addressed in our work.

FluidLab [28] 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.

As we also developed a graph visualization framework, we have to consider existing frameworks that work similary.

Stardog Explorer [26] is a tool designed to help users visualize and query data in the *Stardog* graph database. It has a user-friendly interface, a drag-and-drop *SPARQL* query builder, and several options for visualizing data, like graphs, charts, and tables. These features make it easier for users to understand complex data relationships without needing advanced technical skills.

Neo4j Bloom [20] is a visualization and exploration tool designed for the *Neo4j* graph database. It provides an intuitive interface for users to query and explore their data visually. *Neo4j Bloom* [20] supports natural language search, allowing users to construct queries in plain English, which are then translated into *Cypher*, *Neo4j's* query language.

Gephi [5] is an open-source network analysis and visualization software that is widely used for exploring and understanding complex networks. It offers dynamic filtering, real-time network manipulation, and a variety of layout algorithms to help users visualize their data in meaningful ways. *Gephi* [5] is particularly popular in research and academic settings due to its flexibility and extensibility.

The tool developed in this thesis shares some features with the mentioned frameworks, such as data visualization and query building. However, it is specifically tailored to our use case. It provides customized visualization and querying capabilities that better meet the unique needs of our dataset, while also including a custom inference builder, making data analysis more efficient and effective for our particular application.

3 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.

3.1 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) [25], to communicate with the different robot components. These two frameworks are now introduced in the following.

3.1.1 ROS

ROS [25], which stands for Robot Operating System, is an open-source middleware framework designed to develop and control robots. Despite its name, *ROS* [25] 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* [25] include:

Figure 3.1: ROS [25]

- **Nodes**: *ROS* [25] 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* [25] 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* [25] network.

3.1.2 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.

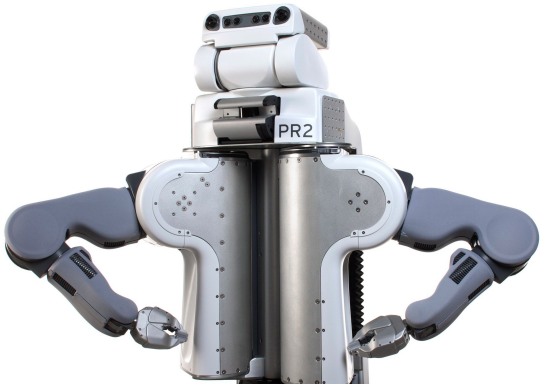


Figure 3.2: PR2 [1]

3.1.3 PyCram

PyCRAM [2] 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* [2] is developed in the *Python*-programming language with support for the *ROS* [25] middleware which is used for communication with different software components as well as the robot.

CRAM [7] (Cognitive Robot Abstract Machine) is a software toolbox for the design, the implementation, and the deployment of cognition-enabled autonomous robots performing everyday manipulation activities. *CRAM* [7] 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* [7] 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.

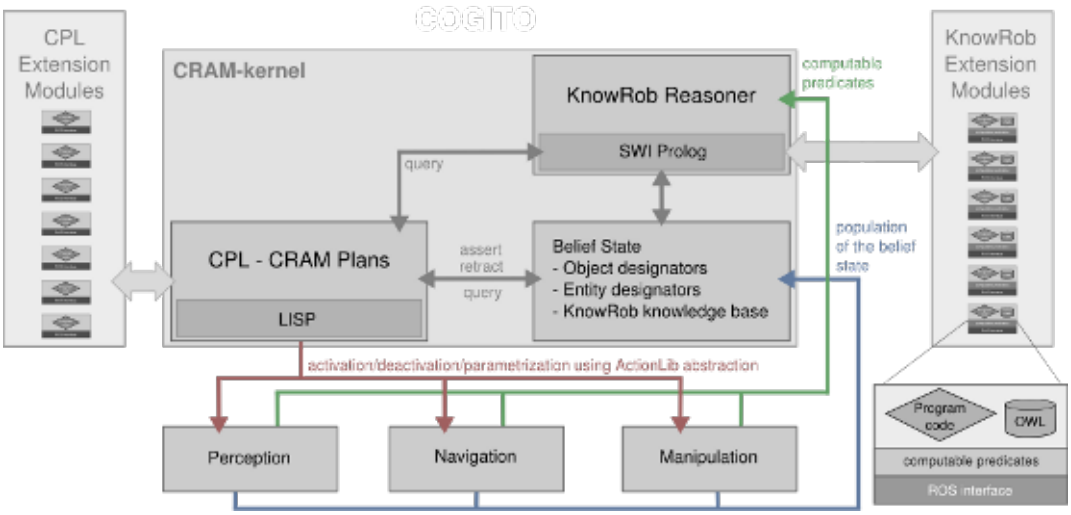


Figure 3.3: *CRAM* [7] Language Architecture

3.1.4 PyBullet

PyBullet[11] is an open-source physics engine and 3D simulation library used for robotics, machine learning, and computer graphics research. It offers accurate physics simulation, 3D rendering, robotics support, and seamless integration with machine learning frameworks. With a *Python*-API and cross-platform compatibility, it's a versatile tool for simulating complex environments and interactions.

3.2 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.

3.2.1 Ontology

Ontologies [22] 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.

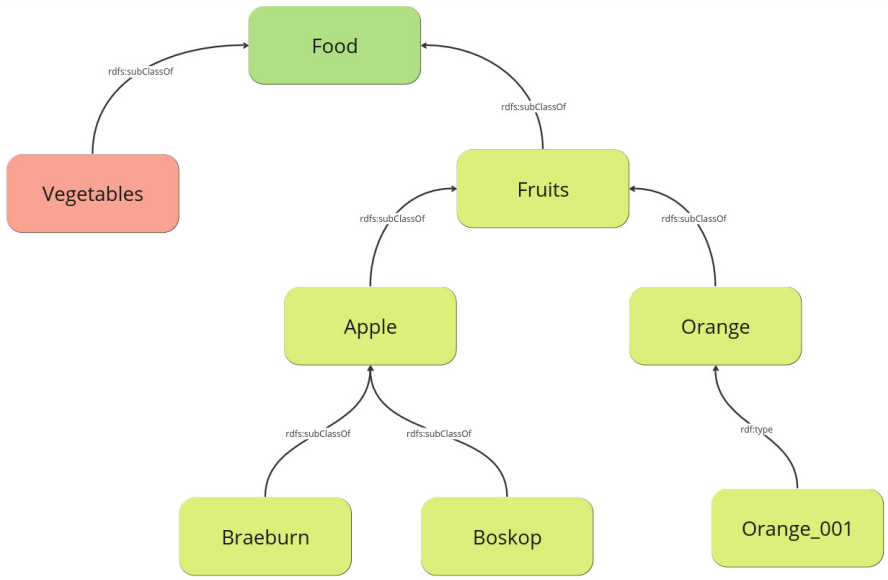


Figure 3.4: Ontology example

Key components of ontologies include:

- **Concepts/Classes**: These represent abstract or concrete entities within a domain. For example, in a food product ontology, *Apple* and *Orange* might be classes.

- **Properties/Roles**: These define the relationships between concepts. For instance, in a our example ontology, *subClassOf* could be a property connecting individuals like *Fruits* and *Apple*.

- **Instances/Individuals**: These are specific members or examples of a class, *Or-*

ange_001 can be an instance of the class *Orange*.

- **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.

3.2.2 SWRL

SWRL [16], 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* [16] is designed to be used in conjunction with *OWL* to express complex relationships and infer new information based on existing knowledge.

SWRL [16] 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 [16] 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* [16] 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* [16] rule:

$$\begin{aligned}&\text{StirringTask}(?task) \wedge \text{LiquidIngredient}(?ing) \wedge \text{hasIngredient}(?task, ?ing)\\&\wedge \text{Motion}(?motion) \wedge \text{performMotion}(?task, ?motion)\\&\rightarrow \text{CircularMotion}(?motion)\end{aligned}$$

This rule expresses the following statement: Whenever a stirring task is executed with the task having any kinds of liquid ingredients the motion it performs is going to be a circular motion. This rule effectively reclassifies the instance of *Motion* to *CircularMotion*.

SWRL [16] 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.

3.3 Software Libraries

3.3.1 RDFLib

RDFLib [17] 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 to build applications that leverage semantic web technologies.

3.3.2 Flask

Flask [15] is a lightweight and flexible web framework for *Python*, designed to make it easy to create web applications. It follows the *WSGI* standard and the minimalist approach, offering essential tools and features for building web services and APIs.

3.3.3 Owlready

One important library used for our implementation is *Owlready* [19]. *Owlready* [19] 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* [19] simplifies ontology-related tasks by providing a convenient and object-oriented interface for working with *OWL* ontologies in *Python*.

Key features of the *Owlready* [19] library include:

- **Object-Oriented Programming (OOP)**: *Owlready* [19] adopts an object-oriented approach, allowing users to interact with ontology entities as *Python*-objects. This makes it more intuitive for developers familiar with *Pythons*-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* [19] 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* [19] may offer support for reasoning tasks. Reasoning involves deducing implicit information based on the logical relationships defined in the ontology.

- **Integration with *RDFLib***: *Owlready* [19] 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.

3.3.4 WikiHow-Instruction-Extraction

WikiHow-Instruction-Extraction [4] is an extraction tool, that can collect informations from the WikiHow [3] corpus. The goal of this tool is to analyse a WikiHow corpus using *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 [3] articles. Additionally, various parameters can be set, such as excluding different categories that are not relevant for the search.

4 Data Acquisition

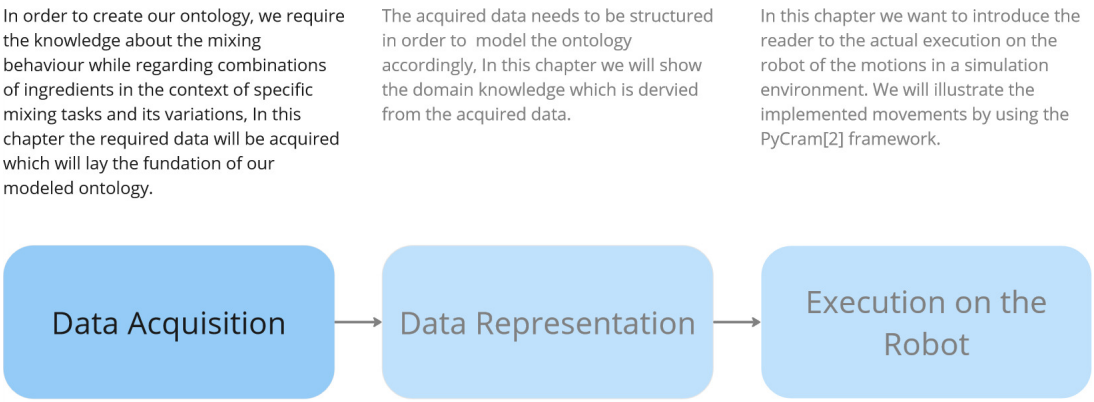


Figure 4.1: Structure Overview: *Data Acquisition*

Data acquisition involves understanding what mixing is, how it is performed by people, and how connections can be derived from video analyses. Essentially, it's about discovering the various ways mixing is done, the contexts in which it occurs, and what insights we can draw from it. This includes identifying different mixing motions, determining which contexts are relevant, and which can be disregarded. First we have to ask ourselves for our context **What is Mixing?**

What is Mixing? The definition provided by the Oxford Dictionary [13] 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.

4.1 Task variations

Now that we have defined what *Mixing* means in our context, we aim to identify the various types of *Mixing* and determine which ones are important for our work. We will also specify which types of *Mixing* we will not consider further, providing arguments for our decisions. 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 Knowldegerepresentation. 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.

4.1.1 Hyponyms

Hyponyms are subordered words of a given word, for example one hyponym of *Mixing* could be *Beating*[24]. To conduct this analysis, one can utilize tools from various websites, such as FrameNet[8] and WordNet[27]. 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 4.1.

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.

For the search, we consider each verb and a triple consisting of the verb forms. We consider three forms of the verbs: the infinitive, the past tense, and the present progressive. An example of such a search would be:

```
MIX ("mix", "mixed", "mixing")
```

After we have defined the desired verbs that we ultimately want to search for, we want to 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* [3] articles. 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

Table 4.1: *Mix*-verb 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.

The verb *Combine* is not informative regarding the executed movement of an action and is equated to the verb *Mix*.

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

4.2 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* [3] 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.

Task	Ingredients	Description
Beating	Egg yolk	circular, swirling wildly around the bowl
Stirring	Beaten Egg Yolk , Parmesan and Pepper	Circular, from the inside to the outside.
Whisk	Eggs	Circular but also straight, wildly motion.
Mixing	Eggs, melted butter	Circular, from the inside to the outside, also diving.
Folding	cooked eggs in melted butter	Gently motion from the outisde to the inside straight, then moving about 90 degree before going to the inside again.

Table 4.2: 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.

4.2.1 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. Since we do not consider electric tools, every motion can be performed with any tool.
- **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* [21]. 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 2 new sub-categories: *Liquid* and *Semi-Liquid* ingredients. This corresponds to ingredients falling under the *Wet* ingredients but with a liquid or semi-liquid state, such as milk, water, eggs, butter and oil. This is significant because some movements differ when the given ingredients are *Semi-Liquid* or *Liquid*.

We will also include 2 sub-categories for the *Dry* category, namely the *Solid* and *Powder* ingredients category. This differs from the each other in that it consists of solid components, while we define the *Powder* 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.

Our initial set of ingredients is: *Milk, Oil, Water, Vinegar, Vanilla Extract, Sauces*, *Egg White, Egg yolk, Butter, Whipped Cream*, *Flour, Salt, Sugar, Baking Soda, Cocoa Powder*, *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. In the chapter Data Representation these classes will be categorized.

Containers and Tools

Throughout the analysis of *WikiHow* videos common tools and containers were identied during the exuecution of various mixing tasks. The most dominant occuring container was a bowl, followed by a pan, cup and then lastly a mug. Those objects can be used to store and serve food for everyday meals. Thus these objects can be summarised into a single concept - *Crockery* [23]

Common occuring tools like a mixer, spatula, spoon and whisk were used. To acquire different kinds of tools and containers, we look into ontologies like *SOMA-Home* [9], where an hierarchy of containers and tools is defined. *SOMA-Home* is a taxonomy modelling concepts for a kitchen environment. *SOMA-Home* uses the class **Crockery**, the class of all kinds of containers. Via subsumption over **Crockery**, we acquire different kinds of containers for mixing.

Identified tools from the video analysis are partially represented by *SOMA-Home*, where mainly cutleries is defined in that ontology. Some tools like mixers and whisks neither belong to the class of all cutleries nor are they represented in *SOMA*. Retrieving a set of tools for these exceptions can't be achieved.

To summarize, the following tools where identified:

- Cutlery
- Mixer
- Whisk

Containers are summarized into the concept of crockery.

Motions

From these videos, we extracted information about the executed motions. In this section, we will describe what the extracted motions are. Additionally, an abstract description is provided of how each motion is implemented in pycram2, using formulas and plots for visualization. We hold the following assumptions about each motion:

- Each container has a symmetric shape in the x and y directions, considering z as pointing either upwards or downwards.
- The height of the executed motion remains constant.
- There is no collision detection.

These motions were extracted during analysis of the videos:

1. Circular Motion
2. Folding Motion
3. Horizontal Elliptical Motion
4. Whirlstorm Motion
5. Circular Diving To Inner Motion

Circular Motion The circular motion is generated on a circular plane, which effectively moves the tool in a circle. By having a center coordinate = $(centerX, centerY)$, which is the center coordinate of the container to execute the Circular Motion, we generate a set of coordinates with the following functions:

$$x = centerX + radius * cos(radian)$$
$$y = centerY + radius * sin(radian)$$

We retrieve the respective radians using the following function:

$$radian = angle * \frac{\pi}{180}$$

Generating a sequence of points lying inside a circle is accomplished by generating a sequence of angles starting from 0 to 360 degrees, converting them into radians, and using formulas to compute the circular coordinates.

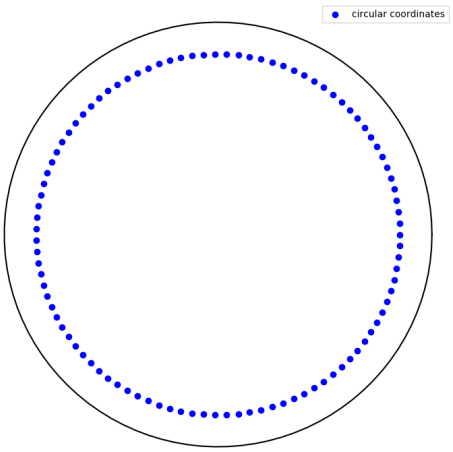


Figure 4.2: Top Down View - Circular Motion

Folding Motion The folding motion differs from circular motion; instead, it follows a linear trajectory. This movement consists of lifting part of the ingredients and gently folding them over the rest. A thorough mixing of the ingredients is possible because of this motion. Repetition of this method ensures even distribution throughout the mixture. It's a methodical and delicate approach to ingredient integration.

In its current implementation, the motion follows a singular line. One endpoint rests on the edge of the container, while the other is positioned at the center. Although the motion itself is a straight line traversing the container, it is executed by moving from one

end to the center and then repeating the motion in reverse—from the center to one of the ends, creating a folding effect. While this line covers only part of the container, its direction needs realignment to mix different sections of the container thoroughly

This can be accomplished by a 2D transformation, constructing a 2D rotation matrix for some angle θ, where θ determines the rotation of the line.

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

and applying the transformation to the line using this function:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\theta) * \begin{bmatrix} x \\ y \end{bmatrix}$$

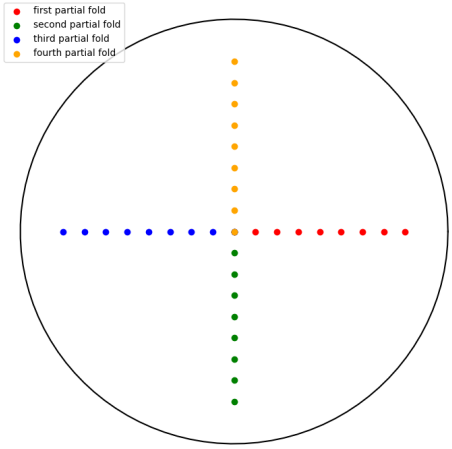


Figure 4.3: Top Down View - Folding 4 areas of a container

In this example, the folding motion is applied to four different sections of the circular container. The number of areas covered or the angle at which the folding line is adjusted is not critical. The fundamental movement remains consistent: the motion proceeds along a line and then reverses."

After executing the folding motion to cover various parts of the container, we apply a transformation with a different angle ϕ.

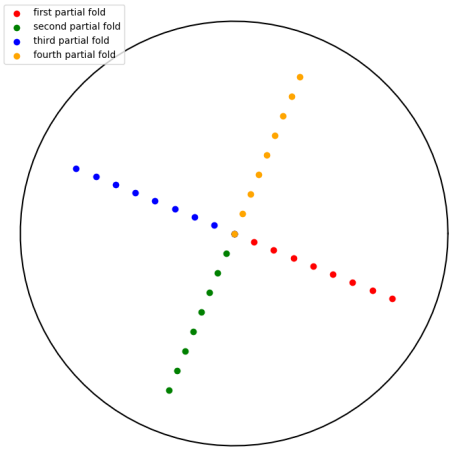


Figure 4.4: Cover different areas of the container with $\phi = 22.5$

Horizontal Elliptic Motion The horizontal elliptic motion is executed on a horizontal plane and involves multiple ellipses to cover the entire container. This motion is commonly employed in the task of beating, where ingredients within a container are thoroughly mixed through stirring or beating.

Using the function to compute x,y coordinates on a circle, we use two different radii. One radius is a small value that is not zero, to make each ellipse narrow. The smaller that radius, the narrower the ellipse becomes. The other radius is sampled from an interval. The interval is defined as

$$radii = [radius, \frac{radius}{2}]$$

meaning the sampled radius ranges from the full radius of the container to half of that radius.

A sampled radius from this interval is taken, if the following condition is true for all resulting circle coordinates:

$$\sqrt{(xCord - xCenter)^2 + (yCord - yCenter)^2} < radius$$

The end result will be ellipses that are narrow on one axis and conform to the shape of the container on the other axis. A generated ellipse considering the above conditions can look like this:

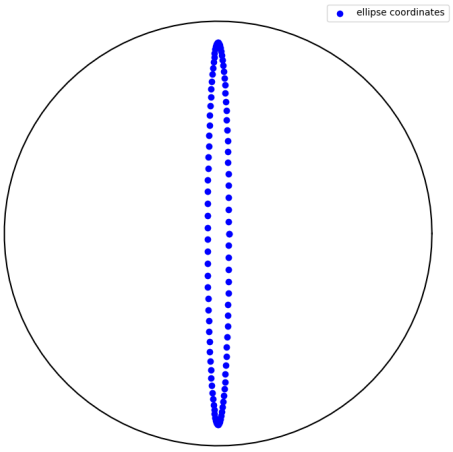


Figure 4.5: Top Down View - Single Ellipse

To cover most areas of any container and assuming that generating y-coordinates used a constant radius, all y-coordinates are shifted after generating the ellipses. To check if the y-coordinates are within the container's bounds, the following condition is checked for every y-coordinate:

$$\sqrt{(yCord)^2 + (yCenter)^2} < radius$$

If this condition is met, the direction of the shift is reversed, switching from incrementing to decrementing and vice versa.

One horizontal elliptic motion consisting of multiple ellipses could look like this:

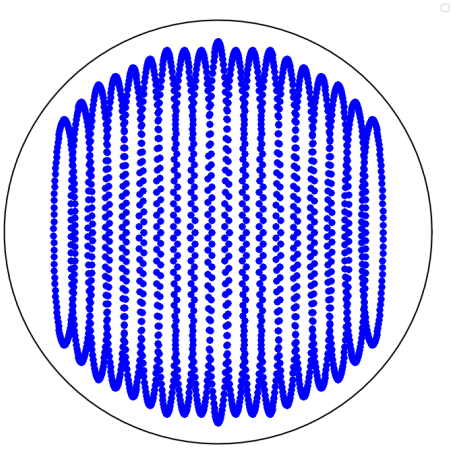


Figure 4.6: Multiple ellipses covering the containers area

Whirlstorm Motion The whirlstorm motion uses the function to generate circle coordinates with differing radii sampled from an interval:

$$[upperBoundRadius, lowerBoundRadius]$$

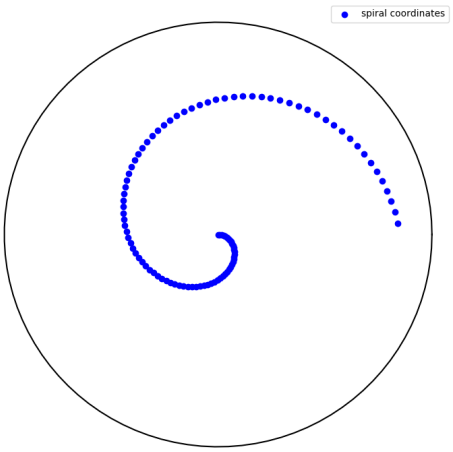


Figure 4.7: Multiple ellipses covering the containers area

The whirlstorm motion is a dynamic and complex swirling pattern that simulates a series of interconnected spiral motions. This motion starts outwardly gradually moving to the inner part of a container, creating a visual effect reminiscent of a whirlpool or tornado. The movement alternates in direction and shape periodically, ensuring a diverse and comprehensive coverage within the containers bounds. This results in a fluid, spiraling trajectory, effectively mimicking natural swirling motions.

In total four whirlstorm patterns are generated, where the first two pattern look like this:

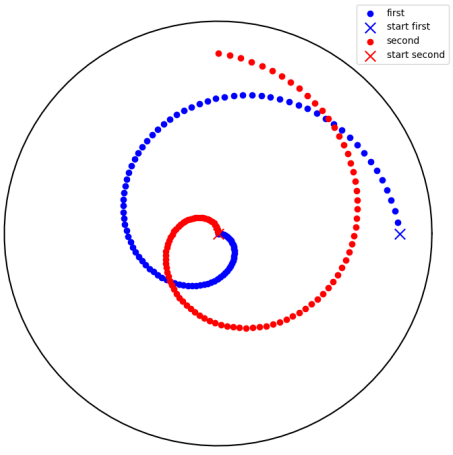


Figure 4.8: Multiple ellipses covering the containers area

The third and fourth look like this:

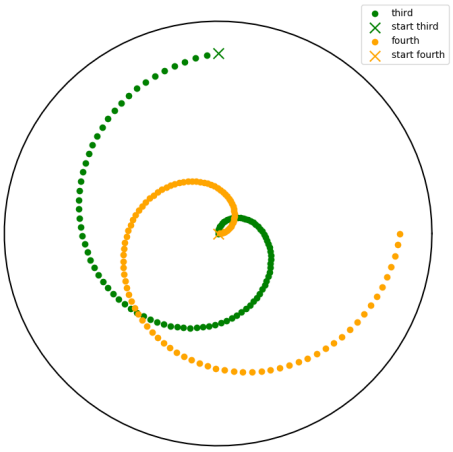


Figure 4.9: Multiple ellipses covering the containers area

Circular Diving To Inner This motion is a circular motion, where the tool is dipped into the container, causing the ingredients to be stirred up and down. This creates a consistent mixture. However, since dipping is involved, which directly contradicts our assumption that height remains constant over the course of the motion, and eventually leaves the boundaries of the used container, we don't provide an abstract implementation of this motion. This motion will have a representation in our *mixing* ontology but won't be implemented in pycram2.

4.3 Data Acquisition Conclusion

The data acquisition lays the foundation to create a model, mainly an *OWL* ontology, to define relationships between the identified actors for various mixing tasks. By identifying several tasks which were executed in the *WikiHow* videos [3], we were able to define specific motions from different kinds of mixing tasks.

The motions that need to be executed are heavily dependent on the task and the set of ingredients used for mixing. Tools and containers are more or less negligible for the execution of the motions. However, they are still considered, since each container can store different kinds of food, and tools are limited to the container being used.

In total five different motions were identified, which will be executed onto four subcategories of ingredients. In the following chapter we will present a taxonomy of mixing with all relevant actors, which enables the robot to execute various mixing tasks in simulation.

5 Data Representation

In order to create our ontology, we require the knowledge about the mixing behaviour while regarding combinations of ingredients in the context of specific mixing tasks and its variations, In this chapter the required data will be acquired which will lay the fundation of our modeled ontology.

The acquired data needs to be structured in order to model the ontology accordingly, In this chapter we will show the domain knowledge which is dervied from the acquired data.

In this chapter we want to introduce the reader to the actual execution on the robot of the motions in a simulation environment. We will illustrate the implemented movements by using the PyCram[2] framework.

In this chapter we will present you the mixing knowledgegraph using *OWL* as basis to formally define knowledge. This knowledge base defines domain knowledge about mixing, which encompasses several aspects derived from the data acquisition. Using assertional knowledge such as the task to execute, the set of ingredients as input for a mixing task, and additional knowledge like which container to use, the knowledge base can infer motions which should be executed via *SWRL* rules. Our goal is not to teach a robot how to execute a specific mixing motion, but rather infer a motion and relevant parameters for this motion.

5.1 The Knowledgebase

Our knowledgebase consists of the superclasses *Ingrediet, DesginedTool, Task* and *Motion*. These are the concepets which we dervied from the Data Acquisition. The following image illustrates the *Mixing* ontology hierarchy.

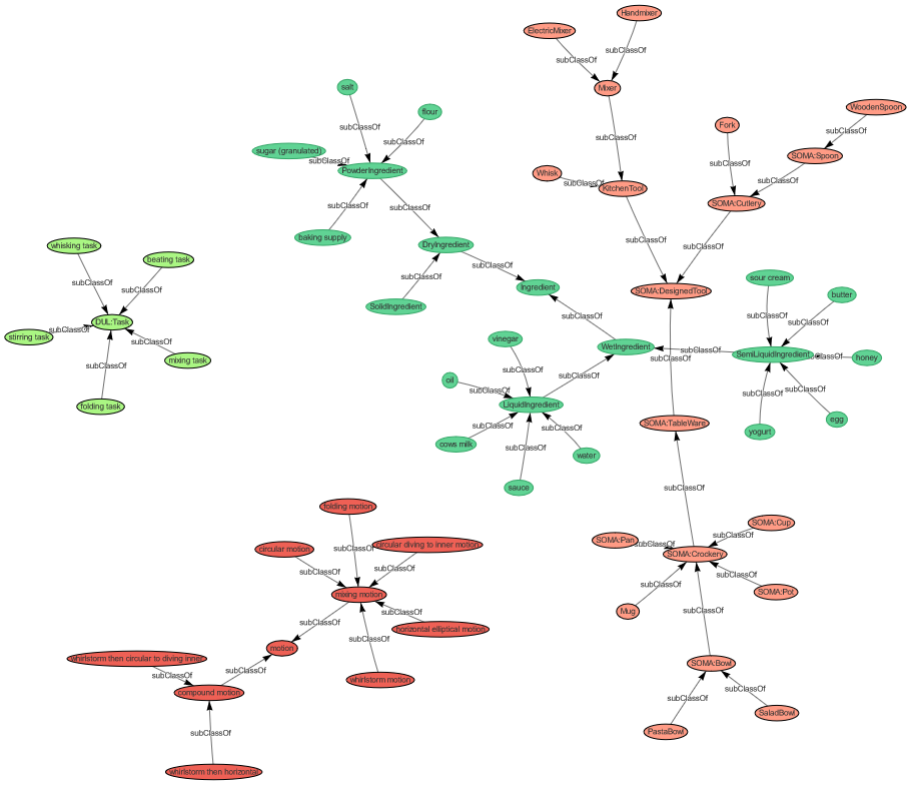


Figure 5.1: Mixing Ontology Taxonomy

5.1.1 Ingredients

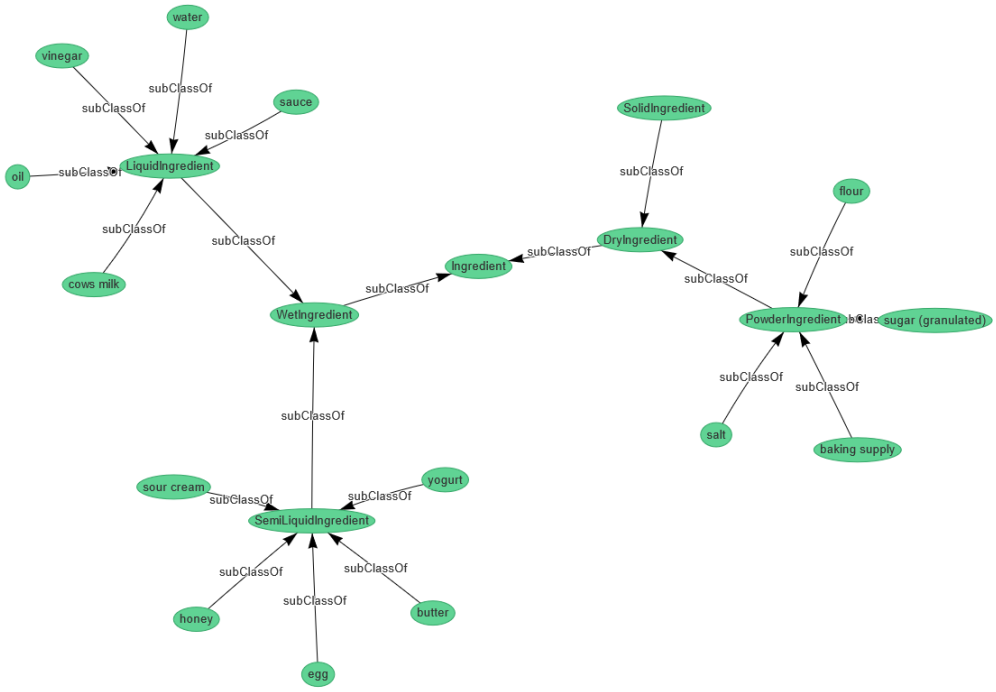


Figure 5.2: Ingredients Hierarchy in Mixing Ontology

Ingredients are primarily divided into *Dry* and *Wet*, with a need for finer differentiation. Wet ingredients can be further differentiated into *Liquid* and *Semi-Liquid*, where liquids include examples like water and semi-liquids include examples like egg whites or yolks. Dry ingredients, on the other hand, are divided into the subcategories *Solid* and *Powder*. Solid ingredients include whole vegetables and meat, while powder ingredients are not solid but rather granulated. This hierarchy of ingredients is modeled with a view towards the cooking domain and enables us to formulate rules to infer motions to perform. There does not exist a single hierarchy of ingredients, but rather many hierarchies modeling different aspects of ingredients.

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

For each specific ingredient, a respective *FoodOn*[14] class is used. By using *FoodOn*, we

directly include all available subclasses of the *FoodOn* class once the ontology is imported into the *mixing* ontology. Different variations of the same ingredient are then covered in case a finer differentiation is needed. Extending the hierarchy of ingredients can be done by for example integrating *FoodOn* classes.

5.1.2 Tools and containers

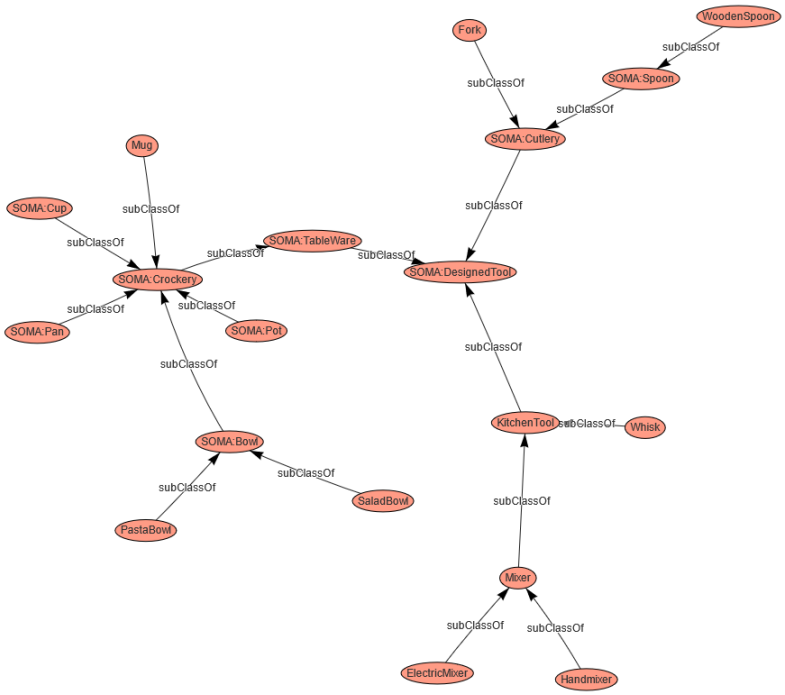


Figure 5.3: Tools and Container Hierarchy in Mixing Ontology

Using the identified tools and containers from the data acquisition, we partially use *SOMA* [9] to define parts of the containers and tools hierarchy. We use the class *DesignedTool* from *SOMA-Home*, which consists of all kinds of container and tool classes. *Crockery*, which is subsumed by *DesignedTool*, is included with each container we identified. To include all kinds of cutlery, we use the class *Cutlery*, which subsumes the classes *Fork* and *Spoon*. Mixers and whisks are not represented in *SOMA-Home*, so we introduce a new class, *KitchenTool*, which includes both of these tools.

Using *SOMA-Home* has a few advantages:

1. Using an existing hierarchy can be reused in other taxonomies.

2. If *SOMA-Home* is extended with new containers and tools, these concepts can be easily included.

3. The *mixing* ontology can be used by *Soma-Home*, because both ontologies share classes.

5.1.3 Tasks

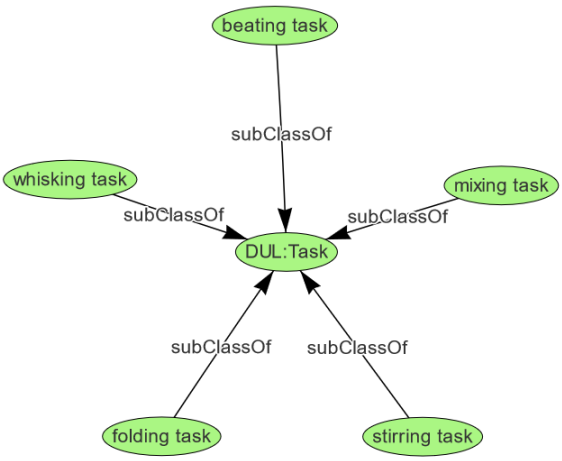


Figure 5.4: Task Hierarchy in Mixing Ontology

Mixing tasks are subclasses of the *Task* superclass, where *Task* is a *DUL* concept. All variations of mixing tasks, which were acquired during our data acquisition, are included in the superclass *Task*. These tasks include Stirring, which mostly involves a circular motion, Beating, which is mainly used in the context of eggs and other wet ingredients, Folding, which represents a gentle type of mixing, and Whisking, which is similar to beating. While some of these tasks have only one motion associated with them, like the Folding task, we discovered that the other tasks can have multiple motions associated with them. This decision is influenced by the set of ingredients on which the motion will be performed.

5.1.4 Motions

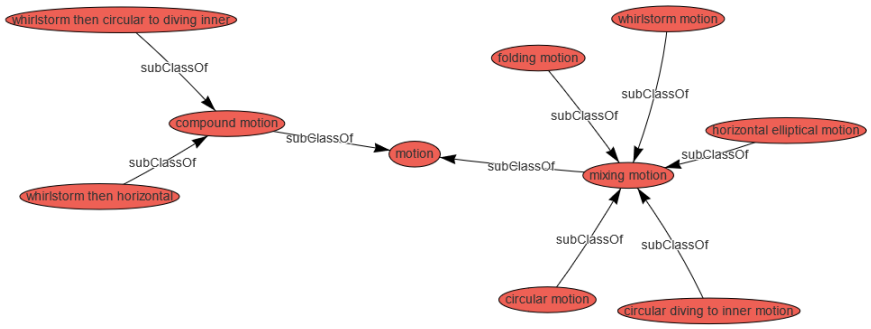


Figure 5.5: Motion Hierarchy in Mixing Ontology

The top level class *Motion* is the class of all possible motions which can be performed. MixingMotion a subclass of motion, are all mixing motions which were found during data acquisiton. To make a distinction between one singular mixing motion and motions consisting of at least two motions we created the class *CompoundMotions*, which defines a sequence of motions to be performed. *MixingMotion* is comprised of the following motions:

- *Circular Motion*: Simple Circular Motion
- *Circular Diving To Inner*: Circular Motions including diving moving to the center of a container
- *Folding Motion*: Motion executed as line with back and forth on multiple areas inside the container.
- *Horizontal Elliptical Motion*: Multiple ellipse motions around the container, which resembles mixing ‘wildly‘
- *Whirlstorm Motion*: Whirlstorm or spiral like motion

Each motion has class restrictions with different kinds of motion parameters.

The following motion parameters are modelled in the *mixing* ontology:

- *Radius lower Bound Relative*: The smallest possible radius relative to an object.
- *Radius Upper Bound Relative*: The maximum possible radius relative to an object.

- *Folding Rotation Shift*: An angle value of how much the folding line has to be rotated around .

- *Repetetive Folding Rotation Shift*: After the folding motion has been executed this parameter rotates all folding lines. to cover similar areas of a container.

- *Ellipse Shift*: How much the ellipse has to be moved in meters. For example, if the ellipse shift is 0.04, then the ellipse is shifted 4 cm in the positive or negative direction along the x or y axes.

CompoundMotion is comprised of the following classes:

- *WhirlstormThenCircularDivingToInner*: Execute whirlstorm motion followed by circular diving to inner motion.

- *WhirlstormThenHorizontal*: Execute whirlstorm motion followed by horizontal elliptical motion.

Each of these compound motions have an OWL expression stating which motion is executed and which motions if following:

```
Motion
 and (hasMotion some
 (WhirlstormMotion
 and (followedBy some HorizontalEllipticalMotion)
 and (followedBy exactly 1 Motion)))
```

This compound motion is equivalent to a *Motion*, which has a motion *WhirlstormMotion* followed by *HorizontalEllipticalMotion*, where *WhirlstormMotion* is exactly followed by one motion.

How each of these motions is related to a task and the set of ingredients will be explained in the next section.

5.2 Rules - Grafiken anpassen

In order to infer the right motion based on the ingredients and task input, rules have to be defined. This rules are SWRL-rules. In this section we want to illustrate the inference on a high level. The rules can be thoguht as 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 following section for every task and ingredient type combination available.

5.2.1 Mixing

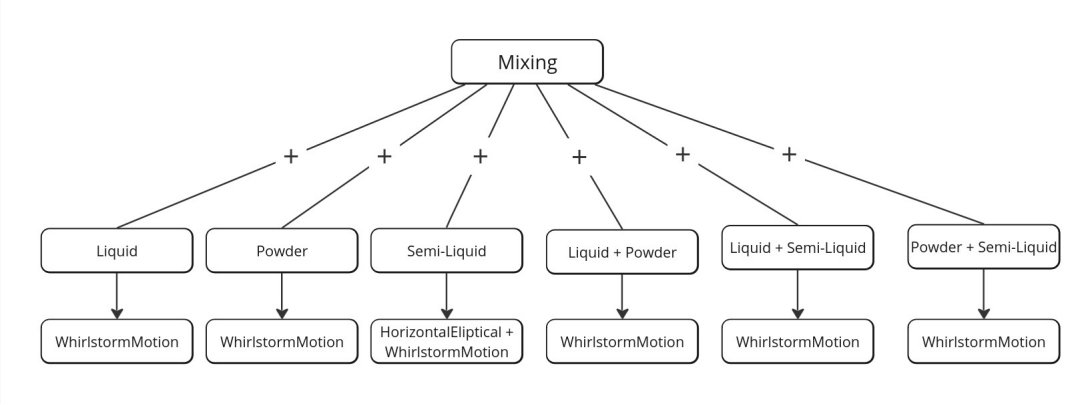
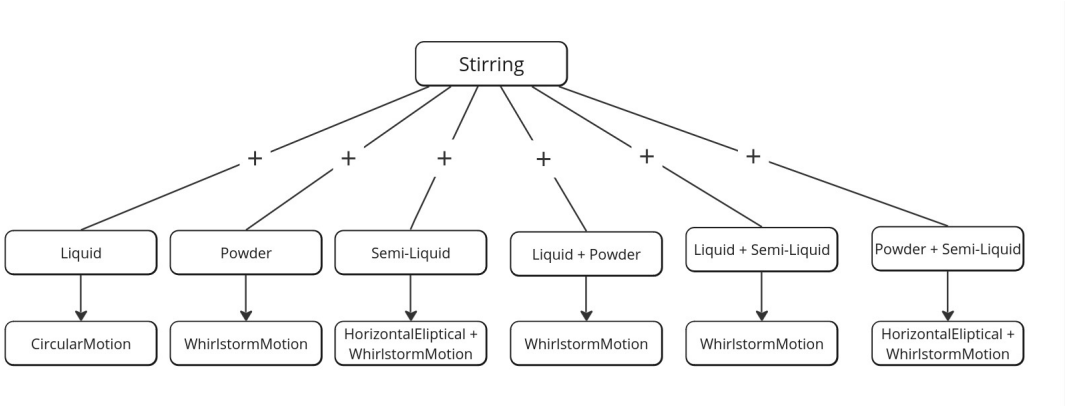


Figure 5.6: Mixing decision tree

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 (Quelle).

As 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.

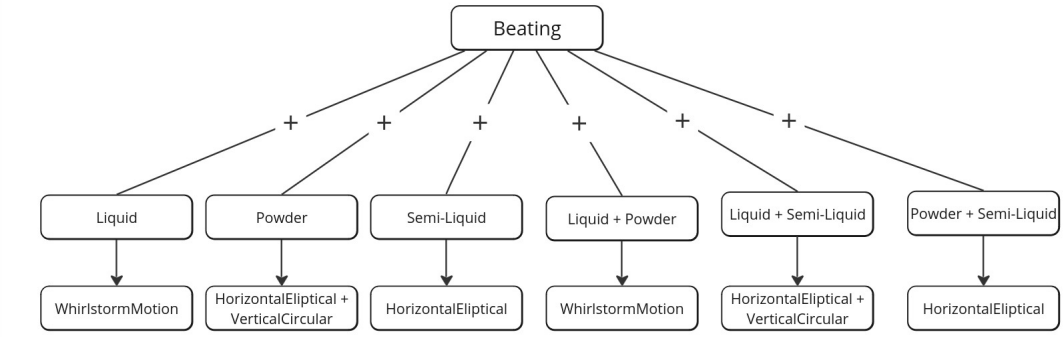
5.2.2 Stirring



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 *WhirlstormMotion*, 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, as would be the case with the *Whirlstorm Motion*, but rather just stirred.

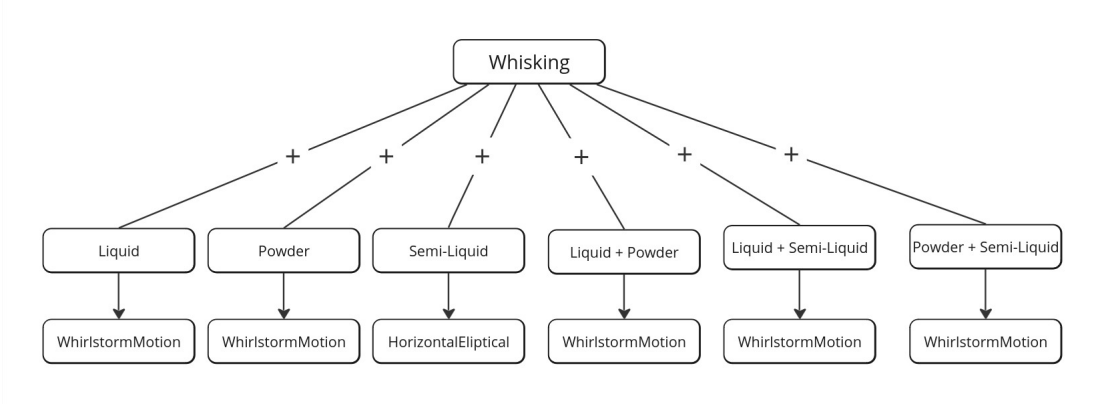
5.2.3 Beating



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.

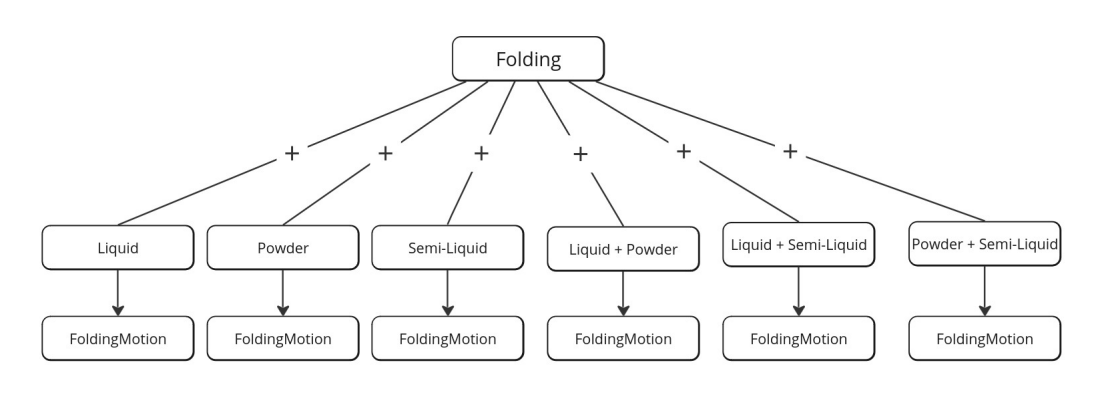
In addition to the Whirlstorm Motion, the Horizontal Eliptic Motion also predominates here. This can be derived from the definition as well, as this motion requires a wild mixing style, to which the Horizontal Eliptic Motion aligns.

5.2.4 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.

5.2.5 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.

6 Execution on the Robot

In order to create our ontology, we require the knowledge about the mixing behaviour while regarding combinations of ingredients in the context of specific mixing tasks and its variations, In this chapter the required data will be acquired which will lay the fundation of our modeled ontology.

The acquired data needs to be structured in order to model the ontology accordingly, In this chapter we will show the domain knowledge which is dervied from the acquired data.

In this chapter we want to introduce the reader to the actual execution on the robot of the motions in a simulation environment. We will illustrate the implemented movements by using the PyCram[2] framework.

In our work, we have created a knowledge base that can infer parameters to enable an agent to perform various actions. Since testing these actions on a real robot proves to be challenging, we do so in a simulation. This chapter aims to provide an overview of the Simulation Environment in which we test our work and present it as a proof of concept. Initially, we will explain the environment, including objects, followed by an overview of the motion primitives. Finally, we will showcase some of our motions (in Simulated Motions) and discuss the Simulation to Real-World gap before evaluating every motion as a proof of concept and drawing a conclusion where we discuss existing challenges.

6.1 Simulation Environment

We utilize the simulation environment *BulletWorld*, which is already integrated into the framework *PyCRAM* (PyCRAM). In this environment, physics is simulated, which is advantageous for executing the motions we have defined or customized. Additionally, leveraging the *PyCRAM* framework allows us to utilize various existing control primitives, such as *Pick and Place*. Communication between the simulation and framework is facilitated by *ROS1* (ROS), which communicates via various nodes, each representing a joint, for instance. Through these nodes, one can obtain information about the joints or transmit commands to modify the parameters of the joints accordingly.

The agent that will execute the various motions in the simulation is the *PR2* (PR2). This robot model has 2 movable arms, which is important for our work, as one hand executes

the motion while the other arm holds the container in which the action is performed. In addition to the fact that the *PR2* model is already available in the *PyCram* framework, the *PR2* model also has enough joints and degrees of freedom to execute the motions we have adapted.

The agent operates in a kitchen-like environment, the model of which is already included in the *PyCRAM*-core and therefore particularly suitable for our use case. The kitchen also features elements such as drawers that can be opened, indicating an approximation to a real kitchen. Additionally, the kitchen model includes a table on which the agent's actions are ultimately executed. Other furniture is also present in this environment, but is not relevant to our use case.

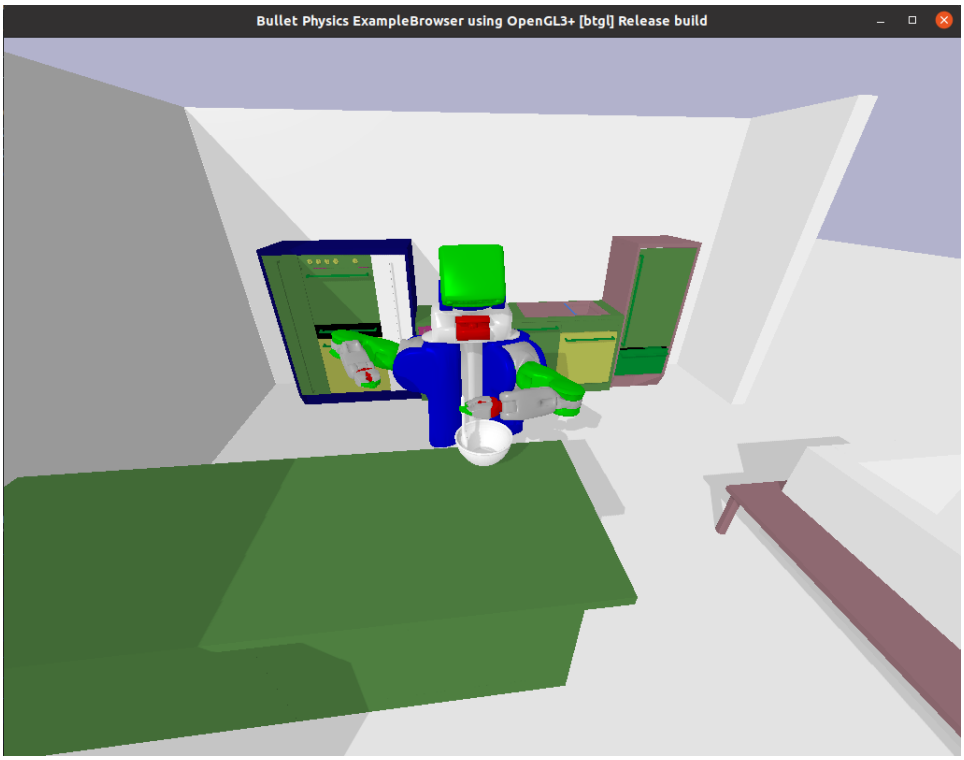


Figure 6.1: Simulation Environment containing the *PR2* and *Kitchen* models.

For the simulation, we use the objects that have also been defined in the ontology (see Containers and Tools Acquisition). The following objects are available:

- Container: small and large bowls (defined in the ontology as *SaladBowl* and *PastaBowl*), pan, pot, cup, and mug.
- Tools: whisk (even though it is not defined in the ontology, we still use it as it is a very commonly used mixing instrument), spoon, wooden spoon, and fork.

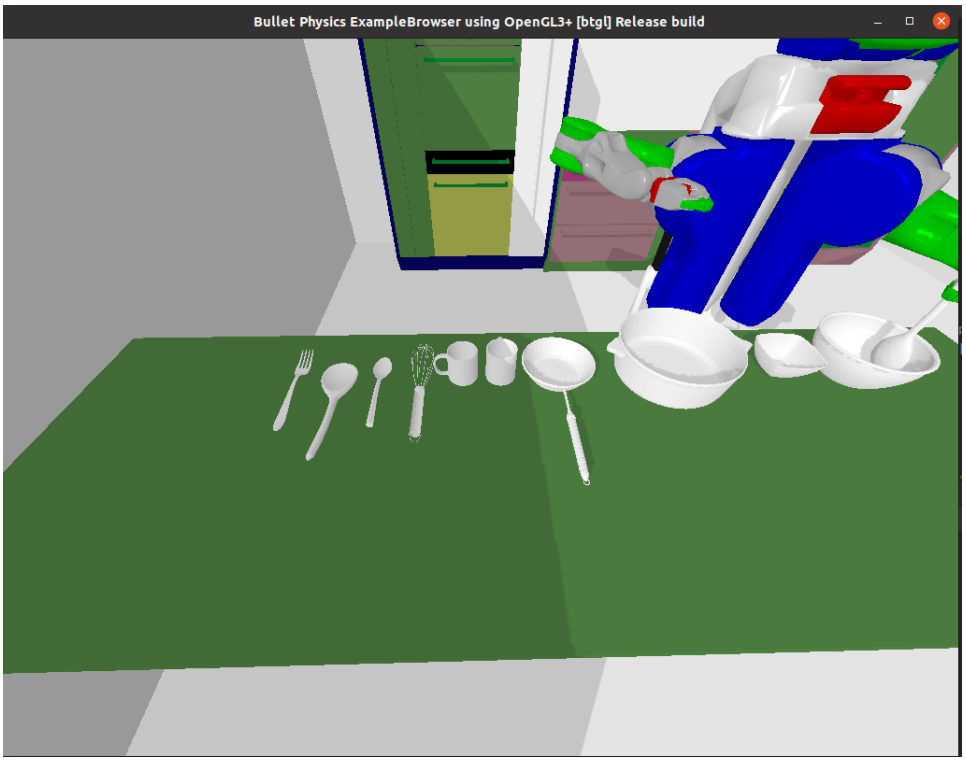


Figure 6.2: Used models for the simulation: *Fork, WoodenSpoon, Spoon, Whisk, Mug, Cup, Pan, Pot, Small Bowl, Big Bowl*

In the simulation itself, there are no ingredients. When executing the motion, we assume a set of given ingredients, which ultimately play a role in selecting the parameters (see Data Representation). Representing ingredients in a simulation proves to be challenging, in addition to the fact that modeling the creation of the mixed ingredients would also be necessary. This approach is also addressed in the section **??** (REFERENCE FUTURE WORK).

6.2 Control Primitives

The framework PyCram offers a variety of pre-implemented control primitives. These primitives can be used as building blocks to structure the entire execution of a task. These primitives are also defined as *Action Designators*, and the following ones are relevant for our use case:

```
NavigateAction(target_locations=[pickup_pose.pose]).
                                        resolve().perform()
```

```
PickUpAction(object_designator_description=tool_object,
             arms=pickup_pose.reachable_arms,
             grasps=["top"]).resolve().perform()

NavigateAction(target_locations=[nav_pose]).
                                    resolve().perform()

LookAtAction(targets=[container_object.resolve().pose]).
                                    resolve().perform()

MixingActionSWRL(
          object_designator_description=container_object,
object_tool_designator_description=tool_object,
          ingredients=[Set of Ingredients],
task="given task",
arms=["left"],
grasps=["top"]).parameters_from_owl().perform()
```

The last *Action Designator* is a modified version of the existing *Mixing Action Designator*, which has been adapted for our purposes. This designator receives as parameters a container object and a tool object (both not significant for inference), as well as a set of ingredients and a task, which are significant for inference. The performed motion results from the inference together with the applied parameters (HERE POINT OUT TO SOMETHING).

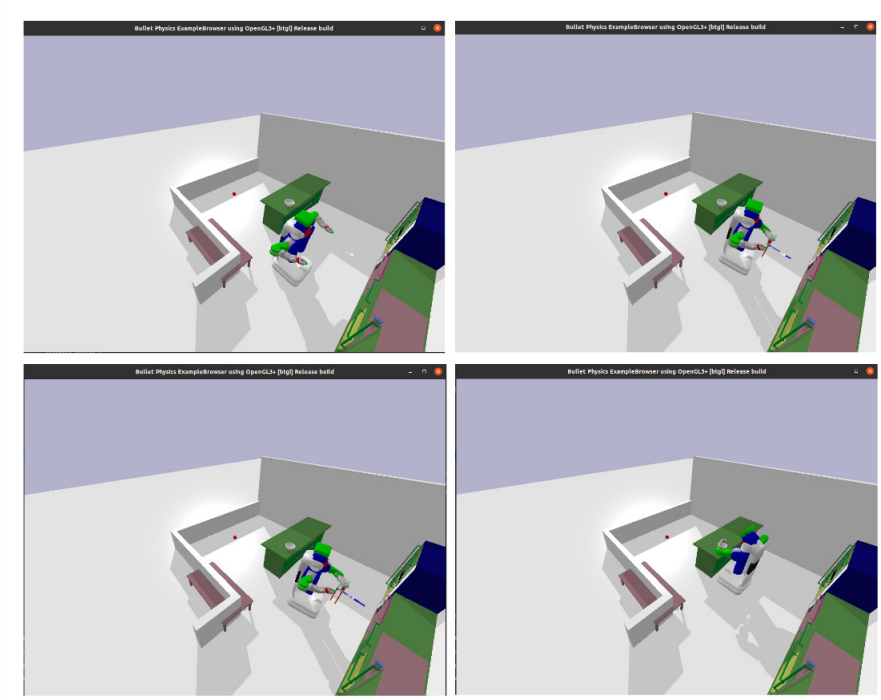


Figure 6.3: Feautured *Control Primitives*

6.3 Simulated Motions

In this section, we aim to illustrate how the motions we have implemented. We will discuss the respective movements as well as the parameters involved. In PyCram, there is the option to interpret the motions as visual axes, which simulate the process of the executed motion beforehand. In this section, we therefore showcase the various motions as they appear in the simulation with these axes, along with a sketch of the implemented movements.

6.3.1 Whirlstorm Motion

The *Whirlstorm* motion was already an implemented motion in PyCram. In our version, we have slightly modified the motion (see Chapter **??**).

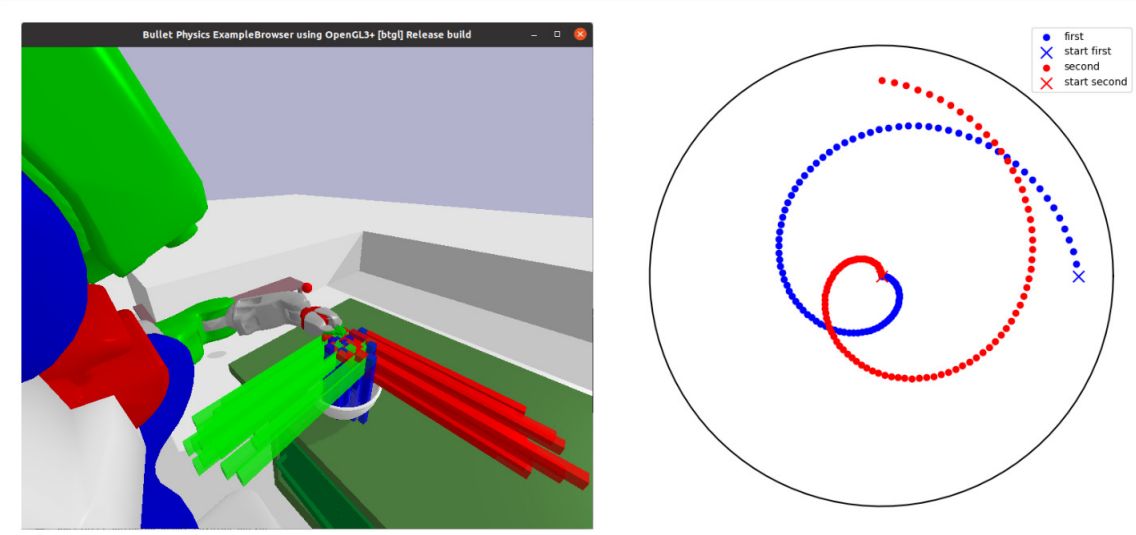


Figure 6.4: Feautured *Whirlstorm Motion*

In Figure **??**, it can be observed that the Whirlstorm motion has two starting points. Initially, the motion starts at the edge of the container and then moves counterclockwise to the second starting point, from which the motion continues analogously.

6.3.2 Circular Motion

The *Circular* motion is a newly implemented mixing variation that represents a simple circular movement along the container.

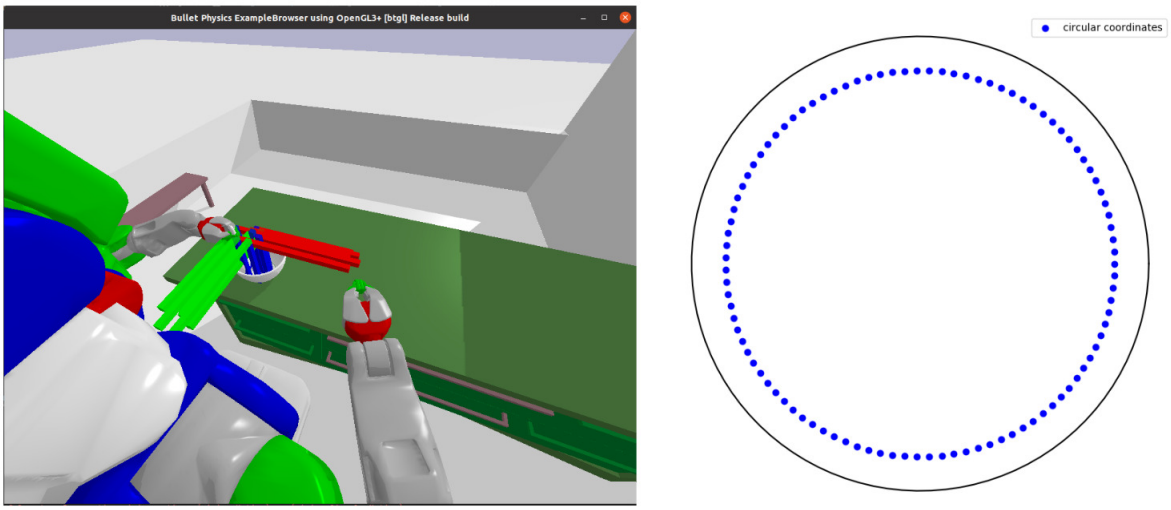


Figure 6.5: Feautured *Circular* Motion

6.3.3 Horizontal Eliptical

Same as above

6.3.4 Folding Motion

The *Folding* motion represents a gentle variation of the mixing motion. This motion starts at the edge of the container and moves toward the center. This is then repeated three more times until the cycle restarts with a slight shift.

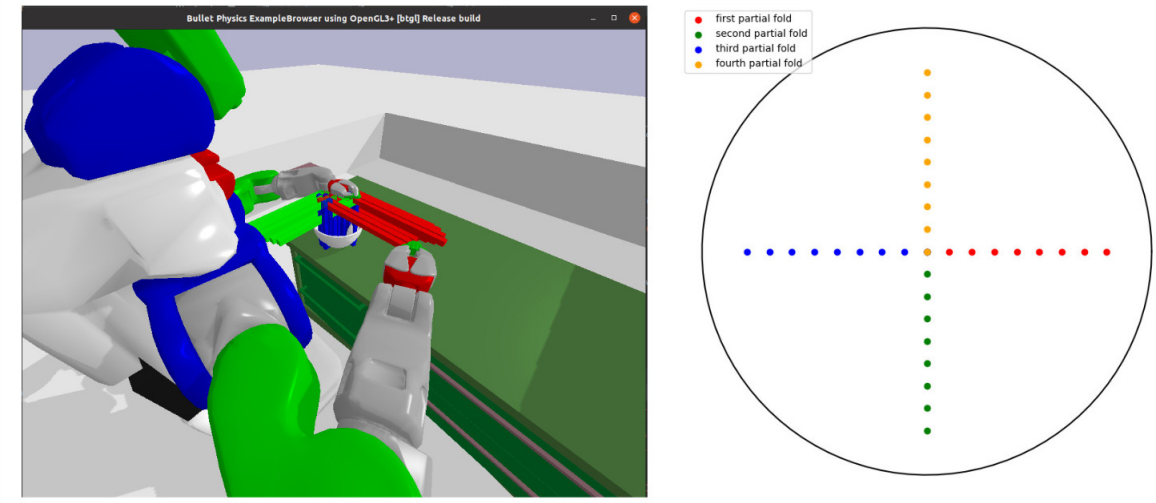


Figure 6.6: Feautured *Folding* Motion

6.4 Simulation to Real-World gap

In a kitchen setting, a robot faces many challenges when it comes to MixingMotion, especially in the real world compared to a simulation. Unlike simulations where movements are precise and predictable, real-world scenarios throw unexpected obstacles, different surface textures, and possible interruptions from human interactions at the robot. Additionally, environmental factors can affect the robot's sensors, leading to misinterpretations or inaccurate readings. Dealing with the complexity and unpredictability of a real kitchen environment requires sophisticated planning and control algorithms that go beyond relying solely on simulation data.

In simulation the tools have a predefined orientation, so that the robot can pick the tool up in the same way every time. Going into real world, this can cause a problem, because the tool might not be picked up at the handle, resulting a mixing task to be executed with the handle instead of the tip of the tool.

When it comes to mixing various ingredients in the real world as opposed to a simulation, several factors can go awry. Firstly, the physical properties of ingredients can vary widely, such as viscosity, density, and texture, which may not be accurately represented in a simulation. This can lead to unexpected interactions between ingredients, resulting in inconsistent mixing patterns or clumping. Additionally, the dynamics of the mixing process may be influenced by external factors such as temperature variations, air currents, and uneven distribution of ingredients within the mixing vessel, all of which are challenging to simulate accurately. Moreover, the mechanical limitations of the robot, such as its speed,

precision, and ability to adapt to changing conditions, can further complicate the mixing process in the real world. These discrepancies between simulation and reality highlight the need for robust control strategies that can adapt to the complexities of real-world mixing scenarios.

From a perception standpoint, uncertainty in the real world during mixing tasks can stem from various sources. Jumping from simulation to real world will immediately introduce the lack of knowledge, where objects are located, what kinds of objects the robots is seeing, what its sizes are. The robot will be in need of a perception framework, which can detect and localize objects in a scene, is able to approxiamate bounds of an object. Another significant factor is the limited accuracy and reliability of sensors in capturing detailed information about the environment and the ingredients being mixed. Lighting conditions, reflections, and occlusions can obscure the robot's view, leading to incomplete or distorted perception of the environment. Additionally, the inherent variability in appearance and shape of ingredients can pose challenges for object recognition algorithms, resulting in misidentification or ambiguity. Furthermore, sensory noise and drift may affect the consistency of sensor readings over time, further complicating perception. In contrast to the controlled conditions of simulation where sensor inputs are pristine, real-world perception introduces a level of uncertainty that necessitates robust perception algorithms capable of handling noise, ambiguity, and environmental variability to ensure accurate understanding and representation of the mixing task.

6.5 Challenges and Difficutlies, Summary of the Simulation

Overall, we are satisfied with the results of the simulation. In another chapter (see Chapter EVALUATION), the various motions are tested and evaluated with different combinations of tasks, ingredients, tools, and containers. One point we would like to address is the initially planned motions, which have a height increment, meaning the motions would not only be executed in 2 dimensions but in 3. Implementing this in the simulation proved to be challenging, so we decided against further consideration of these motions due to the effort involved.

At the current state the robot in simulation either succesfully executes actions or fails to do so. In case of failues there is no recovery to succesfully execute a mixing task. The robot is not able critically reflect upon its made failures, which will

6.6 Implementation in PyCram

6.6.1 Robots Plan Execution

The robots plan used for implementing the MixingActionDesignator and its evaluation is a simple plan, including the following steps:

- Park Arms
- Elevate the robots torso
- Navigate to the tool used for mixing
- Pick up tool
- Park arms
- Move to kitchen counter
- Look at the container to mix the ingredients
- Mix ingredients

In the following subsections we will explain, how the robot uses its knowledge during task execution, to mix.

6.6.2 Asserted Knowledge

Any robot capable of executing mixing motions need to have knowledge before executing a mixing task. The robot needs to know where the container and tools are located, for picking up and for navigation to Theoretically respective places. It needs to know which ingredients have to be mixed, in simulation only their existence have to be known. It needs to know which task to execute, since we differentiate between different mixing tasks as discussed in this section.

6.6.3 Inferring Knowledge

As of now, the robot doesn't know what kind of mixing motion it has to perform. Its associated parameters are also unknown. Therefore, the robot needs to reason about its assertional knowledge. Once it has access to all relevant information, it can infer the

appropriate mixing motion or motions, in case a compound motion has been inferred, with its corresponding parameters through logical reasoning. Logical reasoning is performed inside the custom resolver *MixingActionSWRL*.

6.6.4 MixingActionSWRL

MixingActionSWRL is a custom resolver using our *mixing* ontology to infer motions from the robots assertional knowledge. The resolvers task is to fill the missing descriptions of which motions should be executed with its associated parameters.

Initialization The resolver loads the mixing ontology, initializes the task to execute and initializes all ingredients which are being mixed. All names need to be available for later instantiation. The high level class *Motion* is instantiated, representing the individual of all possible motions. The resolver attempts to reclassify this motion into a specific one that will be executed on the robot.

First container, tool and ingredients are instantiated, additionally a task instance is created and its relations with the other individuals are assigned. To find the proper corresponding class, to the names of the container, ingredients, task and tool we perform fuzzy string matching to find a matching class to the name.

Fuzzy String Matching Using the levenshtein distance, syntactic similarity of two different strings is computed. A low distance score indicates a higher similarity than a high score. However, this metric does not account for semantic similarities, as the words meanings are not considered in the computation of distance.

Inference Once the relations have been assigned, using *OWLReady* with builtin functions calling reasoners like *Pellet* or *HermIt*, reclassifies the motion instance using the defined *SWRL* rules inside the *mixing* ontology. Since the individual instantiated from the class *Motion* has been reclassified the motion parameters can be extracted from the class.

Finally, the resolver completes the sequence of motions to be executed with its associated parameters and forwards it to the MixingAction.

6.6.5 MixingActionDesignator

The *MixingActionDesignator* is a newly implemented action designator designed to perform various types of mixing motions or sequences of motions within a specific container using a specific tool. This designator generates poses that follow a trajectory, simulating

the process of mixing in the bulletworld environment. To effectively break down the action into executable motions, certain designator descriptions are required.

Designator Description

1. *Container Object Designator*:

    - This object designator contains information about its name and its bulletword internal represnation. Its bulletworld representations holds information like the current objects pose and its 3D axis aligned bounding box. This bounding box can be used to compute the dimensions of the container along each axis. Knowing the bounds and the pose of the container is crucial, because the motion has to performed inside the container within its boundaries.

2. *Tool Object Designator*:

    - Analogous to the container this designator contains the same type of information. Knowing the dimensions of the tool is crucial in avoiding collision with the container.

3. *Arm*:

    - Which arm should be used to perform the mixing action

4. *Motions*:

    - A list of motions as strings to be performed.

5. *Motion Parameters*:

    - Each motion has its own key value pairs of parameters. These parameters alongside the motion are resolved by the MixingActionSWRL

Sequential Processing in MixingActionDesignator Once all descriptions are available for the *MixingActionDesignator*, it attempts the following steps in sequence: Retrieve BulletWorld Objects: The BulletWorld objects for both the container and the tool are retrieved from the object designator. Each BulletWorld object has a 3D axis-aligned bounding box. Using this bounding box, the dimensions can be computed. The implemented function *get_ object_ dimensions()* returns the three dimensions of each BulletWorld object.

Calculate Radius Bounds: The upper and lower radius bounds, which are relative values, are accessed to compute the absolute radius for the respective mixing motion.

Determine Container Pose: The container's pose is retrieved and transformed into its own coordinate system from the world coordinate system to compute mixing poses fitted to the container in its coordinate system.

Now that *MixingActionDesignator* has everything relevant to generate mixing poses dependent on the inferred motion from the *MixingActionSWRL* resolver, with its impelmentation described in this section

Generating 3D coordinates is done using the functions explained in the section HIER REFERENZIEREN: MOTIONS, where the necessary paramteres are retrieved from the motion parameters to compute these coordiantes.

The rotations of the folding motions and the shift of the ellipses for horizontal elliptical motion are retrieved from the motion parameters.

Failure Handling

6.7 Evaluation

In this section, we aim to present our implemented/adapted motions as a proof of concept. We consider all possible combinations of tasks and ingredients applied to every possible combination of tools and containers. Each task will showcase only one example in this chapter, with the rest available and more detailed in the appendix. The respective task trees are located in Chapter Data Representation, which illustrates which motions result from combinations of tasks and ingredients.

Through this evaluation, we aim to demonstrate that the knowledge base we have developed can aid in inferring dynamic motions along with parameters that adapt to both the tool and container. Our parameters are relative to the dimensions of the container, specifically the radius within which the motion is executed, as well as the dimensions of the tools used to execute the motion. It is crucial that the motion in the simulation does not exceed the predetermined dimensions, as this could have potentially serious consequences when applied in the real world.

6.7.1 Mixing Task

Nr.	Ingredients	Tool	Container	Inferred Motion
1	Liquid	WoodenSpoon	Salad Bowl	Whirlstorm Motion
2	Powder	WoodenSpoon	Salad Bowl	Whirlstorm Motion
3	Liquid + Dry	WoodenSpoon	Salad Bowl	Whirlstorm Motion
4	Liquid + SemiLiquid	WoodenSpoon	Salad Bowl	Whirlstorm Motion
5	Powder + Wet	WoodenSpoon	Salad Bowl	Whirlstorm Motion
6	SemiLiquid	WoodenSpoon	Salad Bowl	VerticalCircular + Whirlstorm Motion

Table 6.1: Mixing Task

- Inferred Parameters for **1-5**:

```
=> radius_lower_bound_relative = 0.0,
radius_upper_bound_relative = 0.7
```

- Inferred Parameters for **6**:

```
=> radius_lower_bound_relative = 0.0,
radius_upper_bound_relative = 0.7,
vertical_increment = 0.1
```

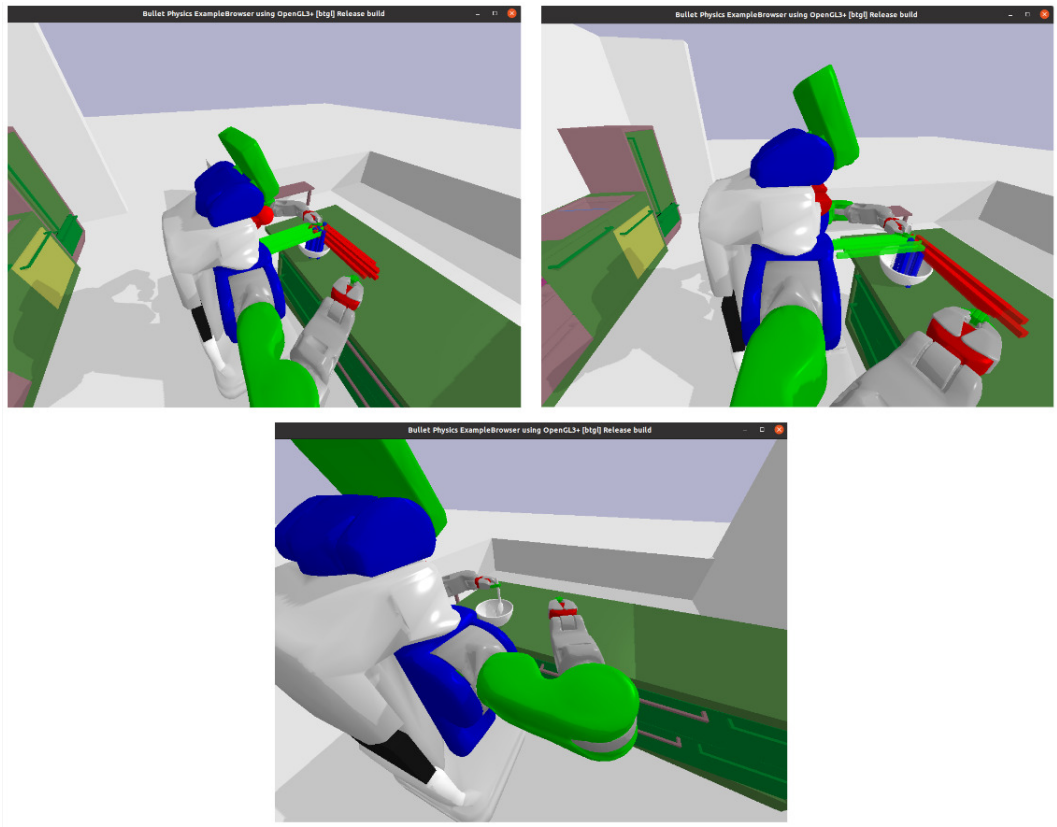


Figure 6.7: Inferred *Mixing* Motions

6.7.2 Beating Task

Nr.	Ingredients	Tool	Container	Inferred Motion
1	Liquid	Fork	Salad Bowl	Whirlstorm Motion
2	Powder	Fork	Salad Bowl	Whirlstorm Motion + Horizontal Eliptical Motion
3	Liquid + Powder	Fork	Salad Bowl	Whirlstorm Motion
4	Liquid + SemiLiquid	Fork	Salad Bowl	Whirlstorm Motion + Horizontal Eliptical Motion
5	Powder + SemiLiquid	Fork	Salad Bowl	Horizontal Eliptical Motion
6	SemiLiquid	Fork	Salad Bowl	Horizontal Eliptical Motion

Table 6.2: Beating Task

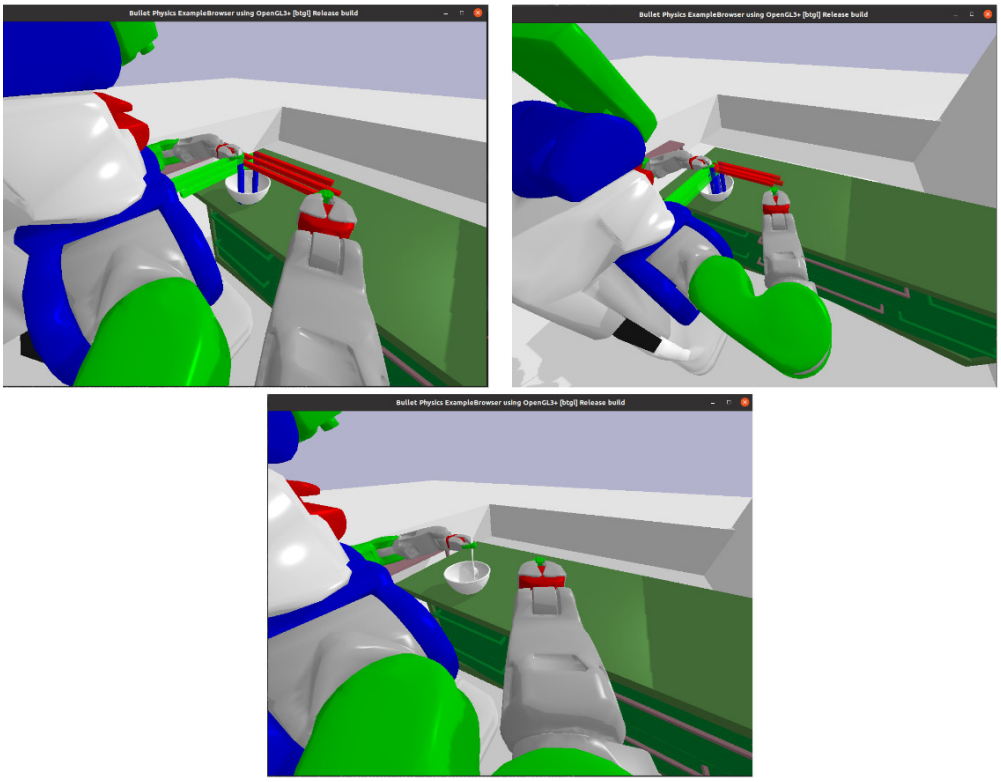


Figure 6.8: Inferred *Beating* Motions

- Inferred Parameters for **1,3**:

  ```
  => radius_lower_bound_relative = 0.0,
  radius_upper_bound_relative = 0.7
  ```

- Inferred Parameters for **2,4,5,6**:

  ```
  => radius_lower_bound_relative = 0.0,
  radius_upper_bound_relative = 0.7,
  vertical_increment = 0.1
  ```

6.7.3 Stirring Task

Nr.	Ingredients	Tool	Container	Inferred Motion
1	Liquid	Wooden Spoon	Pot	Circular Motion
2	Powder	Wooden Spoon	Pot	Whirlstorm Motion
3	Liquid + Powder	Wooden Spoon	Pot	Whirlstorm Motion
4	Liquid + Semi-Liquid	Wooden Spoon	Pot	Whirlstorm Motion
5	Powder + Semi-Liquid	Wooden Spoon	Pot	Whirlstorm Motion + Horizontal Eliptical Motion
6	Semi-Liquid	Wooden Spoon	Pot	Whirlstorm Motion + Horizontal Eliptical Motion

Table 6.3: Stirring Task

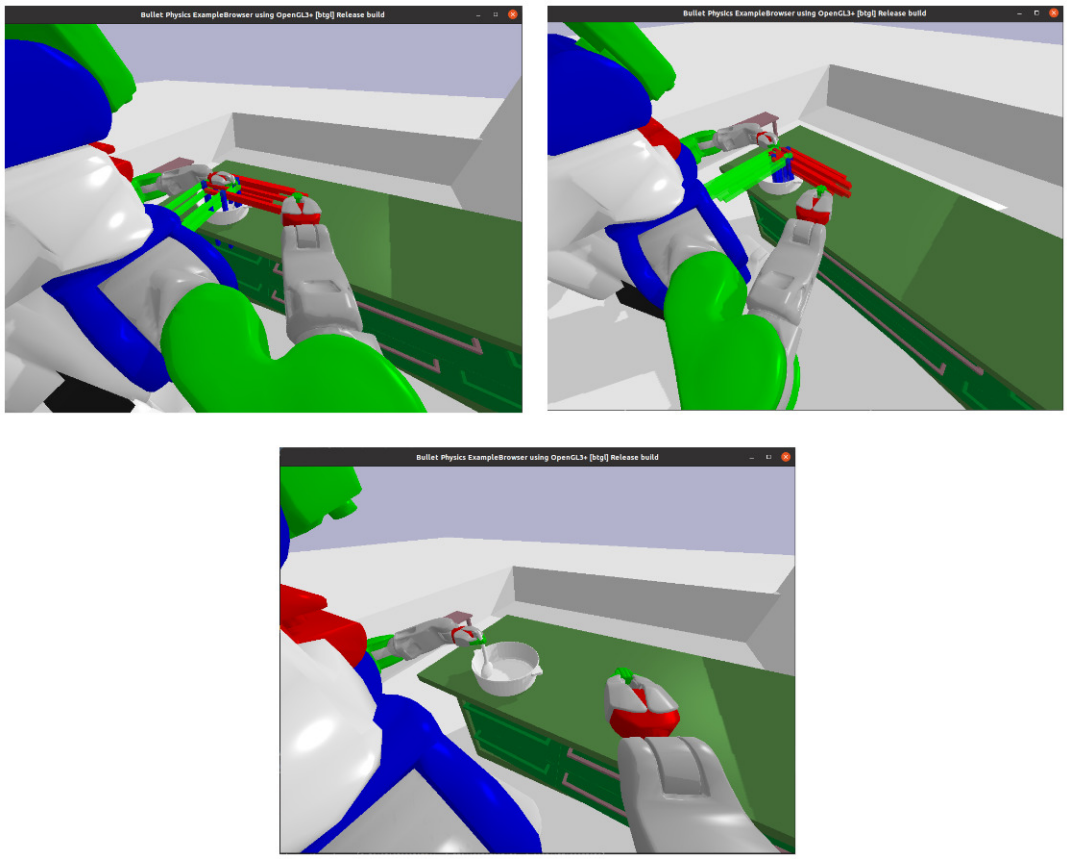


Figure 6.9: Inferred *Stirring* Motions

- Inferred Parameters for **1**:

  ```
  => radius_lower_bound_relative = 0.7,
  radius_upper_bound_relative = 0.8
  ```

- Inferred Parameters for **2,3,4**:

```
=> radius_lower_bound_relative = 0.0,
radius_upper_bound_relative = 0.7,
vertical_increment = 0.1
```

- Inferred Parameters for **5,6**:

```
=> radius_lower_bound_relative = 0.0,
radius_upper_bound_relative = 0.7,
vertical_increment = 0.1
```

6.7.4 Whisking Task

Nr.	Ingredients	Tool	Container	Inferred Motion
1	Liquid	Whisk	Bowl	Whirlstorm Motion
2	Powder	Whisk	Bowl	Whirlstorm Motion
3	Liquid + Powder	Whisk	Bowl	Whirlstorm Motion
4	Liquid + Semi-Liquid	Whisk	Bowl	Whirlstorm Motion
5	Powder + Semi-Liquid	Whisk	Bowl	Whirlstorm Motion
6	Semi-Liquid	Whisk	Bowl	Horizontal Eliptical Motion

Table 6.4: Whisking Task

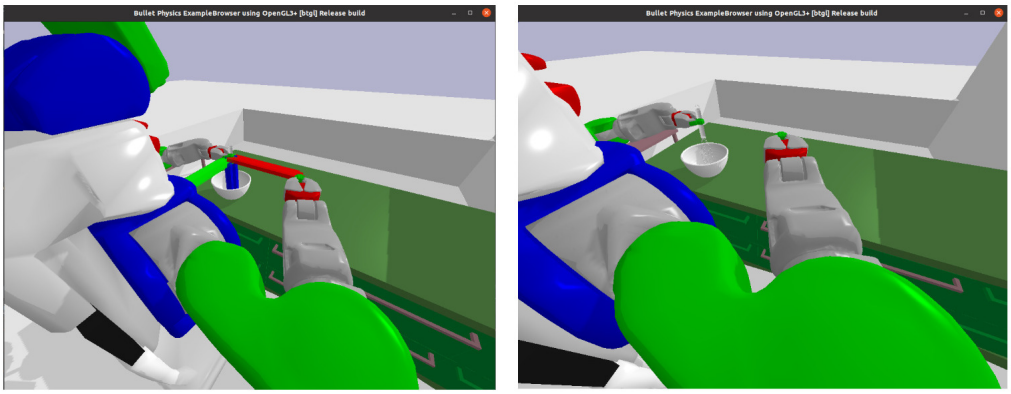


Figure 6.10: Inferred *Whisking* Motions

- Inferred Parameters for **1,2,3,4,5**:

```
=> radius_lower_bound_relative = 0.0,
radius_upper_bound_relative = 0.7
```

- Inferred Parameters for **6**:

```
=> radius_lower_bound_relative = 0.0,
radius_upper_bound_relative = 0.7,
vertical_increment = 0.1
```

6.7.5 Folding Task

Nr.	Ingredients	Tool	Container	Inferred Motion
1	Liquid	Fork	Pot	Folding Motion
2	Powder	Fork	Pot	Folding Motion
3	Liquid + Powder	Fork	Pot	Folding Motion
4	Liquid + Semi-Liquid	Fork	Pot	Folding Motion
5	Powder + Semi-Liquid	Fork	Pot	Folding Motion
6	Semi-Liquid	Fork	Pot	Folding Motion

Table 6.5: Folding Task

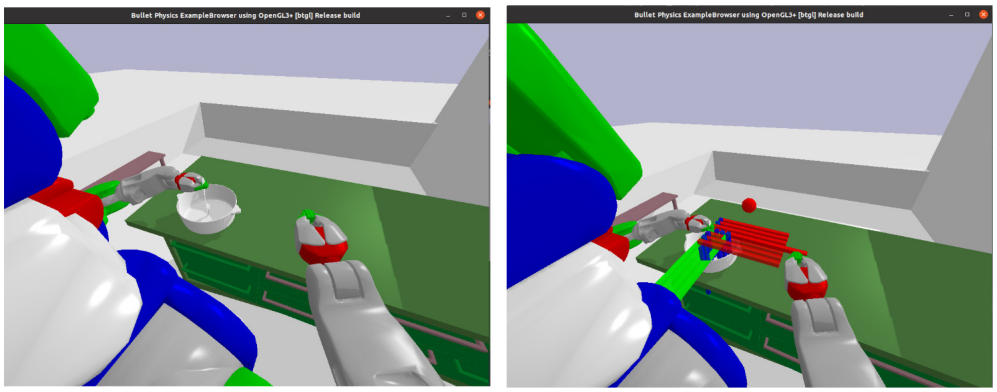


Figure 6.11: Inferred *Folding* Motions

- Inferred Parameters for **1-6**:

```
=> radius_lower_bound_relative = 0.0,
radius_upper_bound_relative = 0.7,
folding_rotation_shift = 90,
repetitive_folding_rotation_shift = 22.5,
```

7 Knowledge Graph Visualization Tool

Many existing knowledge graph visualizations are restricted to displaying certain types of relationships. Most commonly, a class hierarchy of an ontology is visualized for the user. Other graph visualization tools are capable of visualizing data. A few of these graph visualizers are mentioned in Related Work.

What we didn't find is a graph visualizer capable of displaying complex class constructs, and a few features were missing as well. Therefore, we decided to develop our own graph visualizer, which is capable of displaying class hierarchies with complex constructs.

This chapter will present the main features of our *OWLVisualizer* and explain its capabilities and limitations.

7.1 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.

- Rule Based Inference: 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.

7.2 Features

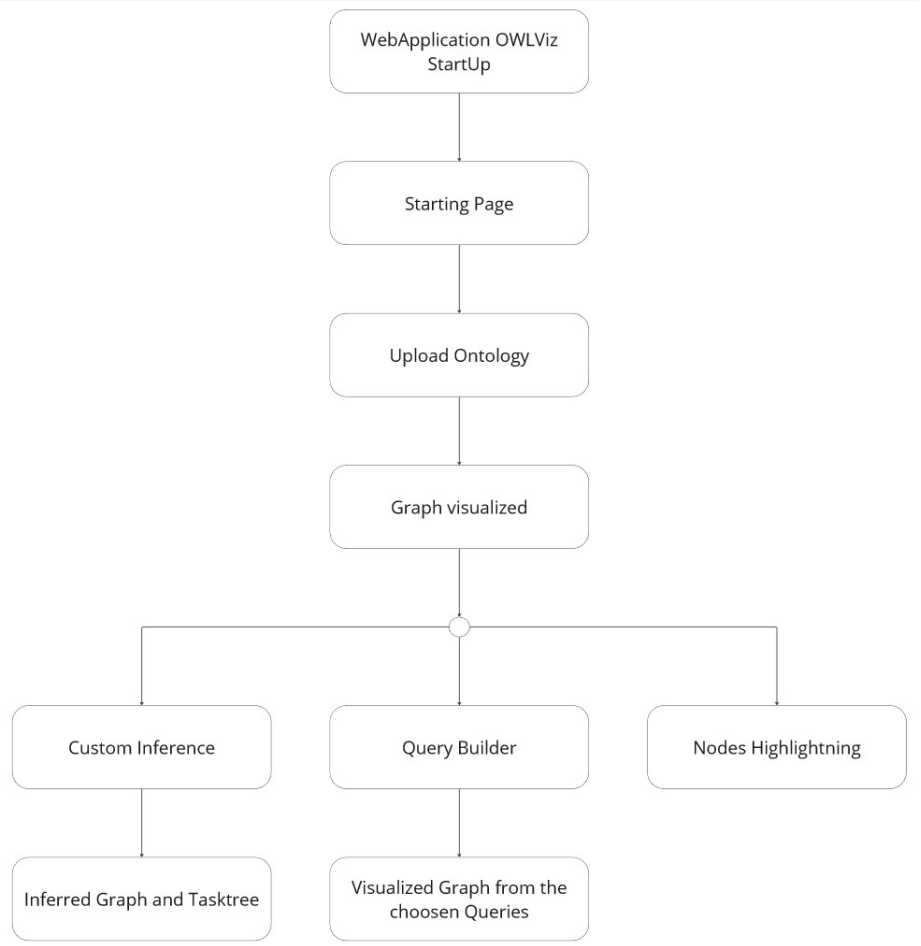


Figure 7.1: Architecture chart for the OWLVisualizer framework

7.2.1 Graph Visualization

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.

The visualization library [**?**] has been used to create graph visualization in web-browsers.

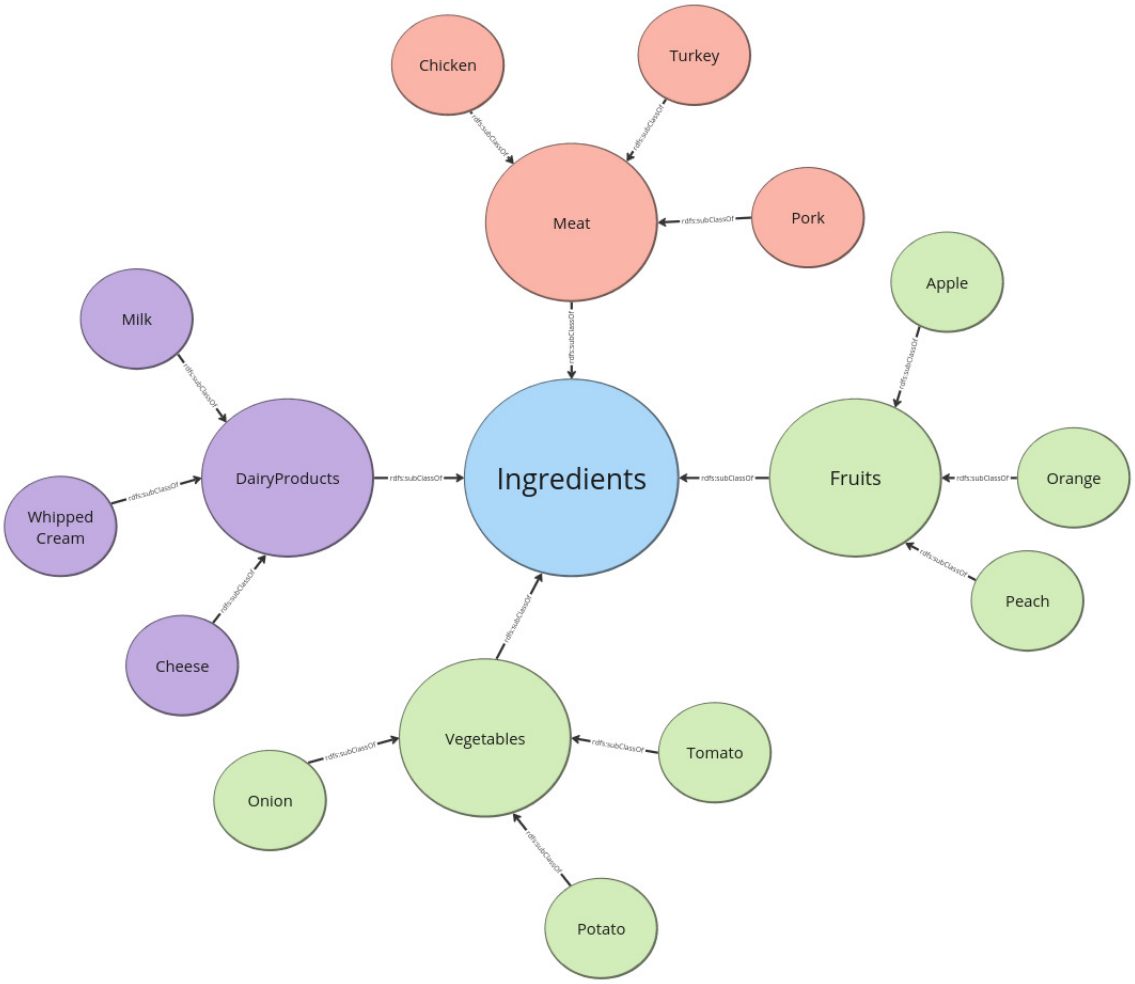


Figure 7.2: Trivial example of an ontology

The ontology created for this purpose aims to provide a simple representation of various animal species. It includes superclass categories such as *Fruits, Vegetables, Meat, and Dairy Products*, as well as subclasses like *Apple* and *Orange*, which are subclasses of the *Fruits* superclass. This ontology does not depict complicated relations; rather, the individual classes are only connected to each other through the *subClassOf* relation.

7.2.2 Graph Processing

The core component of the graph visualizer is the visualization of a hierarchy of classes. In this hierarchy, two nodes are connected by a directed edge, where the source node is

the child node and the target node is the parent node. All edges are labeled either as *subClassOf* or *equivalentClass*.

To create the visualization from the graph, the ontology is parsed, and a triple matching process is applied to identify all classes that use *subClassOf* or *equivalentClass* as predicates.

By default, all nodes are uniformly colored green. However, in the *mixing* and *FoodCutting* ontologies, additional colors are used to highlight differences.

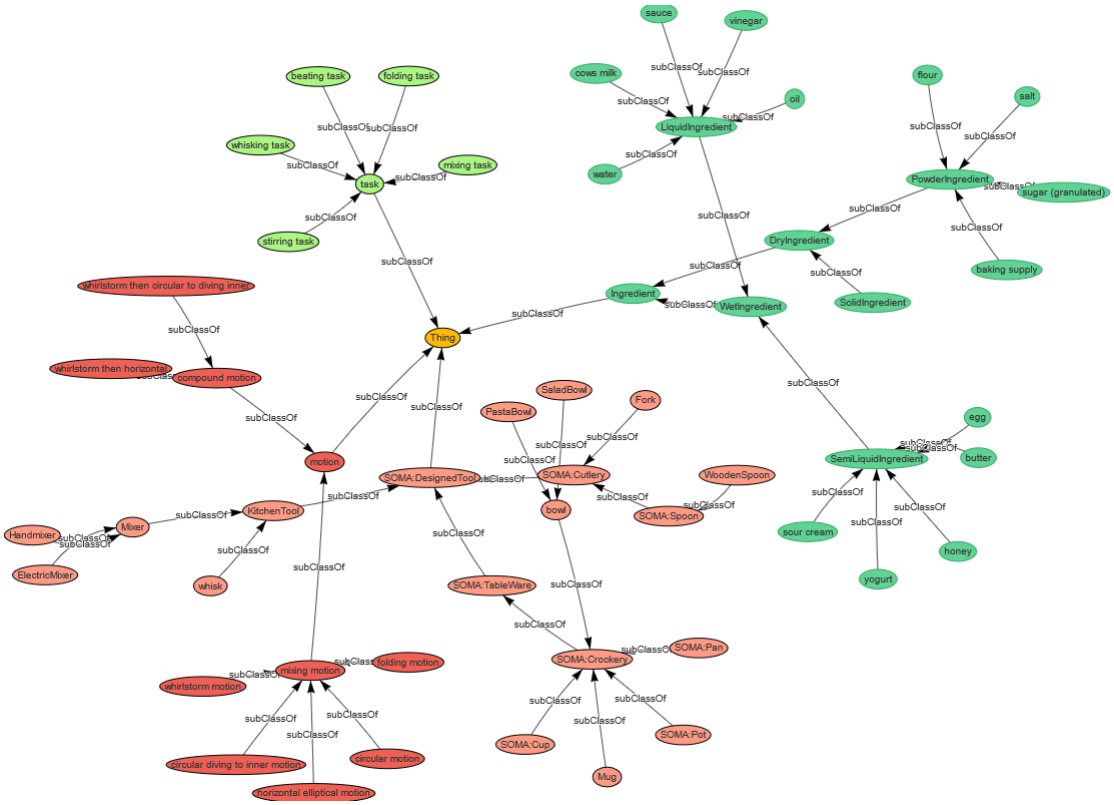


Figure 7.3: Processed Class Hierarchy

The graph visualizer is also capable of visualizing complex class expressions. However, it is essential to understand why the parsed expressions should not be used directly, as illustrated in figure 7.4. Blank nodes and meta information associated with these expressions are generally useless to the user. This type of visualization is not compact, and users unfamiliar with ontology parsing will find it confusing and uninformative. Additionally, visualizing each parsed node increases the number of nodes and edges, which decreases performance of creating the graph visualization. Therefore, a more compact visualization of complex restrictions is necessary to ensure clarity and maintain optimal performance.

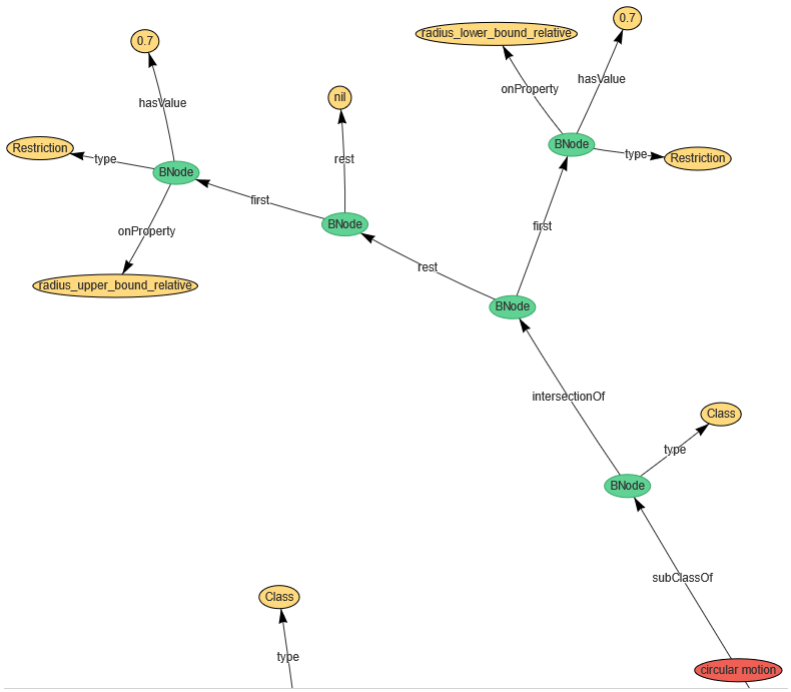


Figure 7.4: Visualize parsed class restrictions

To achieve a compact representation of class expressions, we recursively traverse the parsed class expression, searching for essential information such as the relations used and their corresponding values. All blank nodes are implicitly skipped and not visualized. Additionally, meta information is excluded.

The visualizer can also handle nested expressions, as well as intersections and unions. Each **AND** node indicates that all connected elements belong to the intersection. Similarly, each **OR** node indicates that all connected elements belong to the union.

In the figure 7.5, the following expression is visualized:

CircularMotion *subClassOf* (radius upper bound relative value 0.7) and (radius lower bound relative value 0.7)

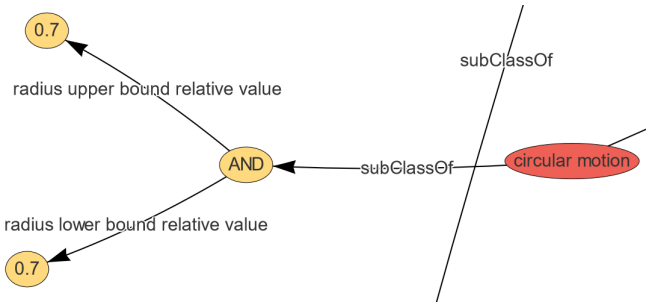


Figure 7.5: Compact representation of class expression

7.2.3 Load Ontology

By pressing the *Load Ontology* button, users can load a wide range of ontologies that have the *.owl* file extension. Upon selecting the ontology, the *OwlVisualizer* begins parsing the ontology to process the ontology. Once the processing is complete, a visual representation of the ontology is created, which can be explored by the users freely.

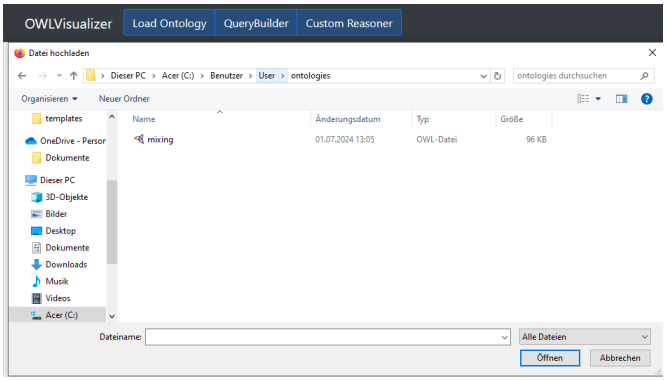


Figure 7.6: Loading Ontology

Attempting to load any other file types results in an error message being displayed on the website. When a user tries to upload a file that does not have the *.owl* extension, the *OWLVisualizer* notifies the user that the file extension is not supported by this tool, via an error message.

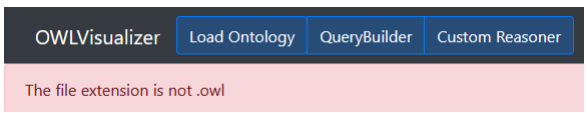


Figure 7.7: **Error**: Unsupported File Type

Additionally, if an ontology can't be parsed, an error message is thrown that ontology

couldn't be loaded properly. Whenever this issue is encountered during the parsing of the selected ontology, it immediately stops and triggers an error notification. This message informs the user that there was a problem with loading the ontology, possibly due to a formatting error or an unsupported structure within the file. This error message informs the user of this particular issue; the *OWLVisualizer* doesn't attempt to deal with this problem.

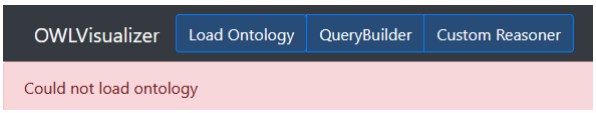


Figure 7.8: **Error**: Ontology Load Failure

7.2.4 Search Classes

The *OwlVisualizer* includes a feature that suggests classes, which are visualized as nodes, for users to focus on, improving their ability to navigate and understand complex networks.

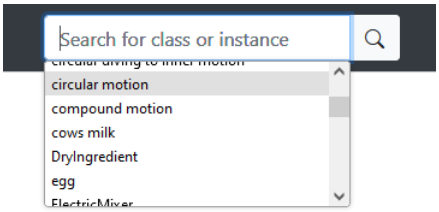


Figure 7.9: Search bar suggesting available classes

The key implementation for this feature is a search bar that suggests all visualized classes. Any nodes belonging to a class restriction are not suggested. The available suggestions are sorted in descending order, allowing users to easily search for classes and explore the class hierarchy without assuming prior familiarity with the ontology.

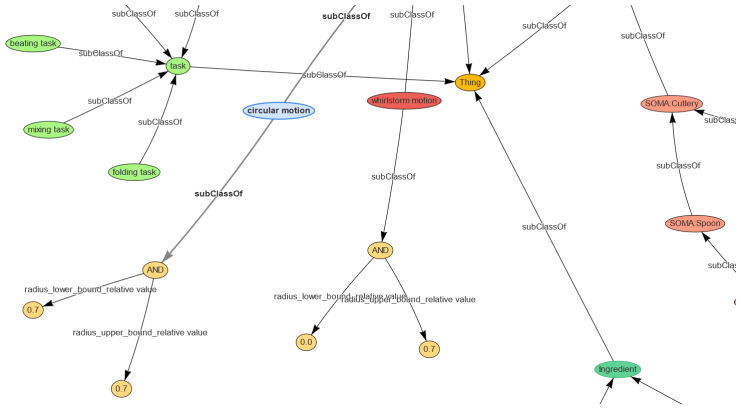


Figure 7.10: Focus node

Users can press Enter inside the search bar or click the search button to immediately focus on the chosen class. Focusing on the class is achieved by zooming to the respective node and highlighting it. This allows users to quickly identify the corresponding node along with all of its parent and child nodes.

7.2.5 Custom Inference - Mixing

One option for the user to execute functions on a graph is the Inference Builder. It's important to note that the Custom Inference Builder is not a generic use case and in this version, it's tailored to the Mixing Graph (see Data Representation). In our case, we infer a motion and its corresponding parameters based on a given task and a list of ingredients.

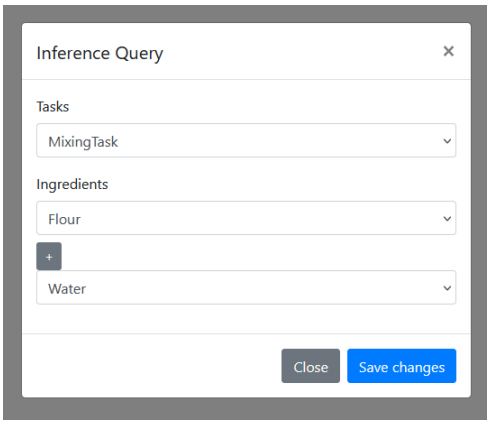


Figure 7.11: Select fields for the inference.

The Custom 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. The backend function processes the inference with the received parameters, which results in a task tree as well as a filtered graph, that contains only the classes that are used in the inference, these are then used for visualization. In the frontend, the data is processed using an *AJAX* request. After sending the task and the list of ingredients, the response, which includes the generated graph and task tree, is returned to the frontend and processed. A *JavaScript*-function then creates a graph with the nodes and edges received from the backend and generates a table with a variable number of rows based on the backend's output. After processing the data, the graph and the task tree are visualized based on the user input.

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	

Figure 7.12: inferred graph and task tree.

The data processing and the detailed implementation of the Inference Builder will be

further explained in the Implementation section.

7.2.6 QueryBuilder

The QueryBuilder is a feature that provides users with suggested triples, allowing them to explore the graph in an intuitive manner. No SPARQL knowledge is required. This feature generates a visualization of the selected triples and constructs a SPARQL query, which only needs slight modification to be queryable.

Triple Matching Triple Matching allows users to select a subject, predicate, and object to explore the graph. At the beginning of the triple matching process, all classes are available as subjects.

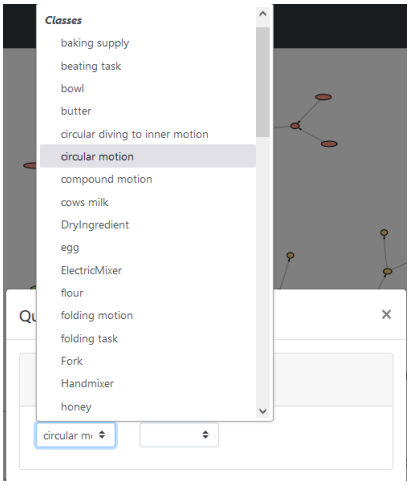


Figure 7.13: Focus node

Only predicates that appear as edge labels connected to the subject in the graph are suggested.

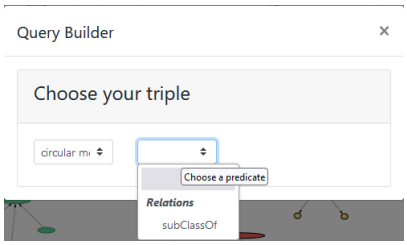


Figure 7.14: Focus node

Similarly, only objects connected to the subject via the selected edge are proposed.

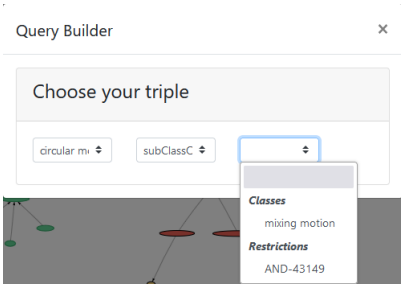


Figure 7.15: Focus node

Once a triple is selected, the user can continue to explore the graph based on the previously chosen triples.

All complex class expressions are processed in a single triple matching step, whereas navigating through the class hierarchy occurs incrementally.

Whenever a triple is selected, the QueryBuilder creates a tab with two items, one item visualizes the graph, the other item contains a SPARQL query.

View Graph In the View Graph, users can inspect the graph based on their selection of triples. This view is a subgraph, which makes it easier to inspect the relationships between classes and classes with complex class expressions. Each class expression is displayed in a separate view, allowing users to focus on each class expression independently, which also reduces clutter in the graph. The class hierarchy has its own view, which can be incrementally updated with new classes to view hierarchical relationships. Separating the views ensures that users can navigate and explore different aspects of the graph, while also improving their comprehension of the ontology.

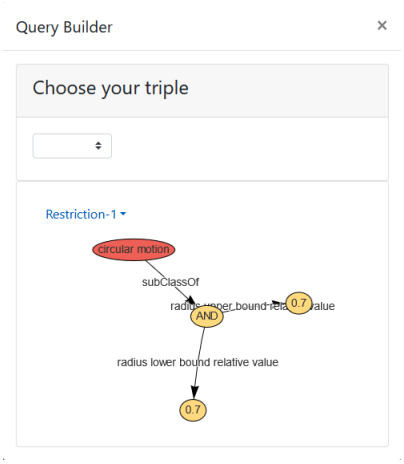


Figure 7.16: Focus node

SPARQL Query A SPARQL ASK query is generated for each chosen triple. If the triple is querying the hierarchy, the SPARQL query for querying the hierarchy is updated with the newest triples. For each complex class expression, a separate query corresponds to the view of the subgraph.

This query is a queryable triple pattern that asks if this query pattern has a solution or not. Users can then independently modify and adapt this pattern to suit their specific needs. This feature will help users unfamiliar with querying ontologies by providing them with the triple pattern.

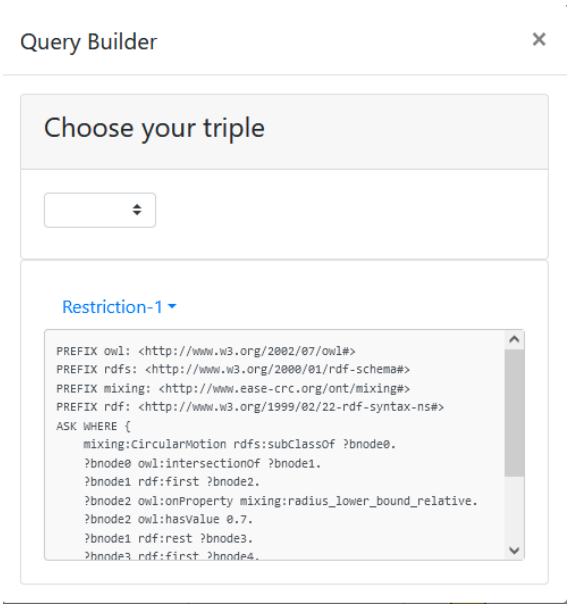


Figure 7.17: Focus node

7.3 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.

7.3.1 BackEnd

As explained in the **??** section and demonstrated in the Introductory Example section (see: **??**), the framework uses *flask* (see 3.3.2) 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 **??**, 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 **??** 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.

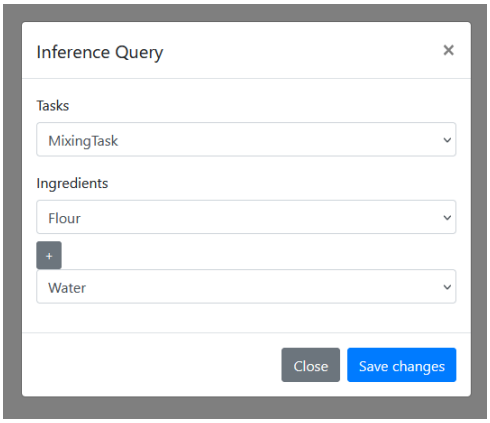


Figure 7.18: Select fields for the inference.

These data are extracted from the ontology using the *rdfLib*(see 3.3.1). In the following it will be explained further.

```
def get_tasks():
    iterate over classes that are related by rdfs:subClassOf
    check for subclasses of Task and appent to task_list
    return task_list

def get_ingredients():
    analogously to get_tasks()
    filter for ingredients that are only subclasses.
    return ingredients_list
```

Listing 7.1: Task and ingredient extraction

- 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.

- Similar to the *get_task()* 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.

These data can be sent to the frontend by calling the corresponding *@app.route*

```
@app.route('/task_ingredients')
def get_tasks_and_ingredients():
    return {tasks: get_tasks(),
        ingredients: get_ingredients()}
```

Listing 7.2: Sending task and ingredients list to the frontend

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.

```
function ($) {
    url: '/task_ingredients'
    success: function(data)
    {
        taskSelect = #taskSelectElement
        iterate through the list of tasks
        appent each task as an option to the taskSelectField
    }
    {
        analogously for ingredients
    }
}
```

Listing 7.3: Fetch and populate select fields

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: 7.12), we have the input task: *MixingTask* and the ingredients: *Flour* and *Water*. These two ingredients correspond to the superclasses *PowderIngredient* 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) ^ PowderIngredient(?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.

```
def get_inference(task, ingredients):
    load the ontology
    initialize a task instance
    iterate over the list of ingredients
        initialize an instance for every ingredient
        relate the ingredients to the task instance

    initialize the motion
    relate the motion to the task instance

    extract the rules from the ontology

    start the reasoner
    infer the motion and parameters based on the task
    and ingredients input

    return motion, parameters
```

Listing 7.4: Inference

- The classes are instantiated. The function takes parameters *task*, corresponding to a task, and *ingredients*, corresponding to a list of ingredients. The ontology for the *OWLReady2* library is instantiated to utilize the reasoner. Next the *task* is instantiated, since the task comes as input in the full *IRI* format, it needs to be processed first. 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*.

- A top-level *Motion* is instantiated to indicate that the *Motion* is inferred during reasoning.

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

- Lastly, the reasoner is executed. 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 and the parameters.

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.

```
def generate_task_tree_and_graph_data(task, ingredients):
    motions, parameters = get_inference(task, ingredients)

    initialize a graph
    process through the inlcuded classes in the inference
    and add them to a set.
    append the elements of this set to the graph and
    inlcude the edges by considering the corresponding
    relations
    process the inferred parameters
```

Listing 7.5: generating the graph

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

- 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.

- 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.

- Iterating over the set of nodes, they are added to the graph.

- 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.

- 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: **??**).

- 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: 7.12)

7.4 Fazit

7.4.1 Weaknesses

Using vis.js Using the graph visualization library vis.js yields poor performance on large graphs. This reason is known to the developers of vis.js (See reference) and the cause of the problem is coming from the physics simulation. The visualized graph is a force directed graph and the physics simulation attempts to make a user readable graph by seperating and clustering nodes who don't or belong to each other. This is done iteratively and consumes a lot of time for large sets of nodes connected via many edges. The more interconnected the graph is the more terrible the performance gets.

Data Visualizer This framework in its core is not a graph data visualizer. While it is absolutely able to display relationships and attributes of instance data, visualizing large datasets defined as ontology is not feasible. It is limited by vis.js which is poorly optimised for large sets of nodes and edges. Thus this framework can't be a graph data visualizer.

8 Conclusion

8.1 Summary

This master's thesis serves as a proof of concept to demonstrate how a robotic system like the PR2, utilizing the cognitive architecture CRAM, can make knowledge-based decisions and perform mixing actions in a simulation.

In order to implement the motions in CRAM, one must first understand the relationship of the motions to the overall context of the task the robot needs to solve. What we have discovered, primarily conducted through video analyses of various cooking videos, is that the motions are primarily dependent on the ingredients in the context of mixing task, which variations we highlighted in our work. The containers and tools used do not directly influence the motion itself, but they do modify the action range of the motions.

The model we use—a knowledgebase—captures various important aspects of a mixing action. It includes information about the ingredients being mixed, the types of motions that can be performed, the type of container used for storing and mixing ingredients, and the tools needed for mixing. By keeping the model simple, we ensure that the robot can infer motions using the rule-based language SWRL.

Our modeled motions do not specify how a motion is executed but provide capabilities via parameters to adapt to different containers and tools. The implementation of each motion—**Circular**, **Folding**, **Horizontal Elliptical**, and **Whirlstorm**—is available in pycram, and any robotic system that can use *Cram* can perform these motions.

We also transitioned from theory to practice by utilizing our knowledgebase in simulation, where we also tested our newly implemented *Mixing Action Designator*, which considers every theoretical knowledge we have acquired in the previeous sections. As a proof of concept, we tested as many scenarios as possible to demonstrate how the knowledge base we created could be applied to a robot.

In addition to this proof of concept, we developed a new way to visualize ontologies by focusing on complex class expressions, which hasn't been done before. With the *OWLVisualizer*, a user can query complex class expressions without needing to know how to write *SPARQL* queries. The *OWLVisualizer* generates a *SPARQL* query for the user, using triple matching to get the desired graph. This query can be customized by changing its contents without altering the triple matching. Another feature of the OWLVisualizer is

the Custom Inference Builder, which serves as an inference tool for our knowledge base, creating a task tree based on its input.

8.2 Future Work

This proof of concept was established in *Cram* in simulation. In simulations many things are simple. The robot has ground truth for each available object, the ingredients to mix, the tools to perform mixing, the container to mix in. To move ahead, our proof of concept should be implemented on a real robot, to see how much more difficult it will be in real world scenarios.

With regards to future work many limitations arose during implementation of our proof of concept. Our motions are effectively 2D motions, where one axis is not changing over the course of the motion. Moving from simulation to real world, it will be necessary to include dives into each motion, so that the substances are properly mixed.

Another limiting aspect is the lack of collision detection. As of now, we ensure that the tool is not colliding with objects only with respect to 2 out of 3 axes. If the tool is placed to low, and the container has a spherical shape and not cuboid shape, the tool will eventually collide with the object. To tackle this limitation, we require depth-based perception and various techniques to find out optimal height for the tool during execution of a mixing action.

Once we move to real world we loose ground truth on all relevant objects required for mixing. Here we will need perception frameworks capable of detecting and locating objects in a scene, picking and placing objects and different kinds of actions for example pouring, to fill a container with ingredients.

In simulation the robot will not fail unless the inference fails. As long as a motion is inferred, mixing will be executed. This has to be adjusted

- Diving motions - Prediction substances - Deployment on real robot - Uncertainity. -> Object Detection - Failure Handling

8.3 Last Words

- GEOMETRIE IN EINLEITUNG BETRACHTEN. keine fragen in EINLEITUNG

Bibliography

[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] Bastian, Mathieu, Sebastien Heymann, and Mathieu Jacomy: *Gephi: The open graph viz platform*, 2009. https://gephi.org/, Accessed: 2024-05-15.

[6] Beetz, Michael, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth: *Robotic roommates making pancakes*. pages 529–536, October 2011.

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

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

[9] Beßler, Daniel, Robert Porzel, Mihai Pomarlan, Abhijit Vyas, Sebastian Höffner, Michael Beetz, Rainer Malaka, and John Bateman: *Foundations of the socio-physical model of activities (soma) for autonomous robotic agents*. April 2021.

[10] Bollini, Mario, Jennifer K Barry, and Daniela Rus: *Bakebot: Baking cookies with the pr2*. 2011. https://api.semanticscholar.org/CorpusID:18098545.

[11] Coumans, Erwin and Yunfei Bai: *Pybullet, a python module for physics simulation for games, robotics and machine learning*. http://pybullet.org, 2016–2021.

[12] Danno, Dylan, Simon Hauser, and Fumiya Iida: *Robotic cooking through pose extraction from human natural cooking using openpose*. In Ang Jr, Marcelo

H., Hajime Asama, Wei Lin, and Shaohui Foong (editors): *Intelligent Autonomous Systems 16*, pages 288–298, Cham, 2022. Springer International Publishing, ISBN 978-3-030-95892-3.

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

[14] Dooley, D. M., E. J. Griffiths, G. S. Gosal, P. L. Buttigieg, R. Hoehndorf, M. C. Lange, L. M. Schriml, F. S. L. Brinkman, and W. W. L. Hsiao: *Foodon: a harmonized food ontology to increase global food traceability, quality control and data integration*. NPJ Science of Food, 2:23, Dec 2018.

[15] Grinberg, Miguel: *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[16] Horrocks, Ian, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosofand, and Mike Dean: *SWRL: A semantic web rule language combining OWL and RuleML*. W3C Member Submission, May 2004. http://www.w3.org/Submission/SWRL/, Last access on Dez 2008 at: http://www.w3.org/Submission/SWRL/.

[17] 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, and Veyndan Stuart: *RDFLib*, August 2023. https://github.com/RDFLib/rdflib.

[18] Kuempel, Michaela and Vanessa Hassouna: *Using web knowledge for flexible meal preperation*. https://github.com/Food-Ninja/WebKat-MealRobot, 2024.

[19] 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.

[20] Neo4j, Inc.: *Neo4j graph platform*, 2024. https://neo4j.com/, Accessed: 2024-05-15.

[21] Ohene, Michael: *A proposed general formula to create and analyze baking recipes*. 2017.

[22] ontotext: *What are ontologies?* https://www.ontotext.com/knowledgehub/fundamentals/what-are-ontologies/, visited on 2024-06-06.

[23] Oxford Learner's Dictionaries: *Crockery*, 2024. https://www.

oxfordlearnersdictionaries.com/definition/english/crockery, Accessed: 2024-06-12.

[24] Oxford Learner's Dictionaries: *Hyponym*, 2024. https://www.oxfordlearnersdictionaries.com/definition/american_english/hyponym, Accessed: 2024-06-05.

[25] Stanford Artificial Intelligence Laboratory et al.: *Robotic operating system*. https://www.ros.org.

[26] Stardog Union: *Stardog Explorer*, 2024. https://www.stardog.com/products/explorer/, Accessed: 2024-05-15.

[27] University, Princeton: *Wordnet, a lexical database for english*. https://wordnet.princeton.edu/ [Accessed: 03.05.2024].

[28] Xian, Zhou, Bo Zhu, Zhenjia Xu, Hsiao Yu Tung, Antonio Torralba, Katerina Fragkiadaki, and Chuang Gan: *Fluidlab: A differentiable environment for benchmarking complex fluid manipulation*, 2023.

9 Appendix

9.1 Video Analysis

In the section Task Analysis and Defintion, a simplified table was presented, which is intended to represent a video analysis. From this analysis, we extract important information that is used for modeling the knowledge base. In the appendix, we present more video analyses and provide the corresponding sources.

Note:

- We provide each table entry with a number, and the corresponding sources are mentioned after the table.

Nr.	Task	Ingredients	Description
1	Mixing	Eggs, melted butter	Circular, from the inside to the outside.
2	Mixing	dry yeast, water	Whirlstorm-like motion.
3	Mixing	dry yeast, water, flour, salt	Whirlstorm-like motion.
4	Mixing	milk, egg, flour	Whirlstorm-like motion.
5	Whisking	milk, egg, flour, salt	Whirlstorm-like motion.
6	Stirring	Semi-Liquid batter, milk	Whirlstorm-like motion.
7	Whisking	Powder ingredients	Whirlstorm-like motion.
8	Beating	Semi-Liquid ingredients, Powder ingredients	Horizontal Eliptical motion
9	Mixing	milk, Powder ingredients	Whirlstorm-like motion.
10	Whisking	flour, salt, baking soda	Whirlstorm-like motion.
11	Beating	egg yolk	Horizontal Eliptical motion.
12	Stirring	egg yolk, parmesan, pepper	Horizontal Eliptical and Whirlstorm motion.
13	Mixing	brown sugar, butter	Whirlstorm-like motion.

Table 9.1: Video analysis

14	Stirring	eggs, smashed bananas, Powder ingredients	Horizontal Eliptical and Whirlstorm motion.
15	Beating	egg yolk, sugar	Horizontal Eliptical motion.
16	Stirring	Liquid mixture, vanilla extract	Circular motion.
17	Beating	egg white	Horizontal Eliptical motion.
18	Folding	Semi-Liquid mixture	Folding motion.
19	Whisking	Liquid mixture	Whirlstorm-like motion.
20	Whisking	Semi-Liquid mixture	Whirlstorm-like motion.
21	Folding	Semi-Liquid, powder ingredients	Folding motion.
22	Stirring	sugar, liquid	Whirlstorm-like motion.
23	Beating	butter, sugar	Horizontal Eliptical motion.
24	Beating	Semi-Liquid mixture, liquid	Horizontal Eliptical motion and Whirlstorm-like motion.
25	Mixing	butter, cream cheese	Whirlstorm and Horizontal Eliptical Motion.

Table 9.2: Video analysis

Source

- **1**: https://www.youtube.com/watch?v=s9r-CxnCXkg
- **2,3**: https://www.youtube.com/watch?v=1-SJGQ2HLp8
- **4**: https://www.wikihow.com/Make-Pancakes
- **5,6**: https://www.youtube.com/watch?v=iwxJTIxIoFo
- **7,8,9**: https://www.youtube.com/watch?v=vkcHmpKxFwg
- **10**: https://www.youtube.com/watch?v=loqCY9b7aec
- **11,12**: https://www.youtube.com/watch?v=3AAdKl1UYZs
- **13,14**: https://www.wikihow.com/Make-Banana-Bread
- **15-20**:https://www.wikihow.life/Make-a-Sponge-Cake
- **21**: https://www.wikihow.com/Make-an-Easy-Sponge-Cake
- **22-25**: https://www.wikihow.com/Make-Cake-Icing

9.2 Evaluation

In the appendix, we would like to build on the concept presented in Evaluation. Here we will present the tables in more detail and cover as many scenarios as possible. As a reminder: A combination of a task and a set of ingredients infers motions, which are in turn associated with parameters. These parameters are used to adjust the motion depending on the chosen tool and container. Ultimately, it must be calculated in which range the motion can be executed. The calculation is as follows:

```
radius_upper_bound = ((dim[0] * radius_bounds[0]) -
    max(dim2[0], dim2[1])) / 2

radius_lower_bound = max(0, ((dim[0] * radius_bounds[1]) -
    max(dim2[0], dim2[1])) / 2)
```

where the dimensional parameters are defined as follows:

```
dim = [max(obj_dim[0], obj_dim[1]),
    min(obj_dim[0], obj_dim[1]), obj_dim[2]]

dim2 = [max(tool_dim[0], tool_dim[1]),
    min(tool_dim[0], tool_dim[1]), tool_dim[2]]
```

tool_ dim and *obj_ dim* correspond to the dimensions of the tool and the container, respectively. With this calculation, it is then possible to define an action radius to perform the motion safely, meaning without the tool, for example, hitting the edge of the container.

In the following, we would like to prove that our system adapts to the environment and calculates the action radii depending on the given dimensions.

Notes:

- The radius bounds are a list of two elements where the first element corresponds to the radius of the upper bound and the second element corresponds to the radius of the lower bound.
- Some combinations of task and ingredient result in the same motion and are therefore summarized in this case.
- It can happen that the action radius is negative; in this case, fine-tuning is necessary. Generally, a motion should not be performed with a negative action radius. However, for relatively small radii, an exception could be made.
- The tools spoon and fork are not considered separately as they are very similar in relevant form and size.

- The folding task infers the same parameters regardless of the ingredients.
- Every value for the radii is in meters.
- **Whisk Dimensions:** *0.11, 0.08, 0.31*
- **Wooden Spoon Dimensions:** *0.09, 0.095, 0.29*
- **Fork Dimensions:** *0.06, 0.04, 0.25*
- **Salad Bowl Dimensions:** *0.25, 0.25, 0.11*
- **Pot Dimensions:** *0.31, 0.37, 0.11*
- **Small Bowl Dimensions:** *0.14, 0.14, 0.07*

9.2.1 Mixing Task

Liquid, Powder, Liquid and Powder, Liquid and Semi-Liquid

- **Inferred Motions:** *Whirlstorm Motion*
- **Inferred Parmaters:** *0.7, 0.0*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.3: Mixing Task which infer Whirlstorm Motion

Semi-Liquid, Powder and Semi-Liquid

- **Inferred Motions:** *Whirlstorm Motion, Horizontal Eliptical Motion*
- **Inferred Parmaters:** *0.7, 0.0* and *ellipse shift: 0.04*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.4: Mixing Task that infer Whirlstorm and Horizontal Eliptical Motion

9.2.2 Stirring Task

Liquid, Liquid and Powder

- **Inferred Motion:** *Circular Motion*
- **Inferred Parameters:** *0.7, 0.7*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0.033]
Whisk	Pot	[0.075, 0.075]
Whisk	Small Bowl	[-0.004, -0.004]
Wooden Spoon	Salad Bowl	[0.04, 0.04]
Wooden Spoon	Pot	[0.082, 0.082]
Wooden Spoon	Small Bowl	[0.003, 0.003]
Fork	Salad Bowl	[0.058, 0.058]
Fork	Pot	[0.1, 0.1]
Fork	Small Bowl	[0.03, 0.03]

Table 9.5: Stirring Task which infer Circular Motion

Powder

- **Inferred Motions:** *Whirlstorm Motion*
- **Inferred Parmaters:** *0.7, 0.0*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.6: Stirring Task which infer Whirlstorm Motion

Semi-Liquid, Semi-Liquid and Liquid, Semi-Liquid and Powder

- **Inferred Motions:** *Whirlstorm Motion, Horizontal Eliptical Motion*
- **Inferred Parmaters:** *0.7, 0.0* and *ellipse shift: 0.04*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.7: Stirring Task that infer Whirlstorm and Horizontal Eliptical Motion

9.2.3 Beating Task

Liquid, Liquid and Powder

- **Inferred Motions:** *Whirlstorm Motion*
- **Inferred Parmaters:** *0.7, 0.0*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.8: Beating Task which infer Whirlstorm Motion

Powder, Semi-Liquid, Liquid and Semi-Liquid

- **Inferred Motions:** *Whirlstorm Motion, Horizontal Eliptical Motion*
- **Inferred Parmaters:** *0.7, 0.0* and *ellipse shift: 0.04*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.9: Beating Task which infer Whirlstorm and Horizontal Eliptical Motion

Powder and Semi-Liquid

- **Inferred Motions:** *Horizontal Eliptical Motion*
- **Inferred Parmaters:** *0.7, 0.0* and *ellipse shift: 0.04*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.10: Beating Task which infer Horizontal Eliptical Motion

9.2.4 Whisking Task

Liquid, Powder, Liquid and Powder, Liquid and Semi-Liquid, Semi-Liquid and Powder

- **Inferred Motions:** *Whirlstorm Motion*
- **Inferred Parmaters:** *0.7, 0.0*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.11: Whisking Task which infer Whirlstorm Motion

Semi-Liquid

- **Inferred Motions:** *Horizontal Eliptical Motion*
- **Inferred Parmaters:** *0.7, 0.0* and *ellipse shift: 0.04*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.12: Whisking Task which infer Horizontal Eliptical Motion

9.2.5 Folding Task

Every Ingredient

- **Inferred Motions:** *Folding Motion*
- **Inferred Parmaters:** *0.7, 0.0*; *repetitive folding rotation shift: 22.5* and *folding rotation shift: 90*

Tool	Container	Action Radius
Whisk	Salad Bowl	[0.033, 0]
Whisk	Pot	[0.075, 0]
Whisk	Small Bowl	[-0.004, 0]
Wooden Spoon	Salad Bowl	[0.04, 0]
Wooden Spoon	Pot	[0.082, 0]
Wooden Spoon	Small Bowl	[0.003, 0]
Fork	Salad Bowl	[0.058, 0]
Fork	Pot	[0.1, 0]
Fork	Small Bowl	[0.03, 0]

Table 9.13: Folding Task