Towards Bigdata in Robotics: Machine Learning Pipeline for Robot NEEMs (Narrative-Enabled Episodic Memories) in an SQL Database

Abdelrhman Bassiouny

Master Thesis
Erasmus Mundus Master in
Marine and Maritime Intelligent Robotics
Universitat Jaume I

July 17, 2023

Supervised by:

Zoe Falomir, Dr.-Ing. (Universitat Jaume I)
Tom Schierenbeck (IAI, University of Bremen)
Prof. Michael Beetz, PhD. (IAI, University of Bremen)

Co-funded by the
Erasmus+ Programme
of the European Union

To My Beloved Family

Acknowledgments

First of all, I would like to thank Dr. Zoe Falomir for her continuous support, advice, and help all this time, not just for the technical support but also for the career and personal advice. Dr. Zoe, your advice is gold, I hope you continue giving this advice and guidance going forward and I wish you luck wherever you are in the future.

Big thanks to Tom Schierenbeck for always being their guiding and helping me throughout my time at the Institute For Artificial Intelligence (IAI) in Bremen, without you the result would almost surely be a bleeding edge software. Tom, your hard work is an inspiration and I wish you all the best with your Ph.D.

I would like to thank Prof. Michael Beetz for giving me the opportunity to work under his supervision at the Institute for Artificial Intelligence (IAI) and experience a great and fruitful collaborative working environment that always pushes and motivates me to reach greater heights that I could not have reached alone.

I have learned a lot during the 6 months I spend at IAI, thank you to all my colleagues there for the technical support but also for the good times and the nice talks during the Uni-Mensa lunches.

I would like to thank all my professors at Universitat Jaume I (UJI), and a special thanks to Prof. Pedro Sanz for his great advice during my year at UJI and for always being there when I needed him.

I would also like to thank all my friends from the MIR Masters for all the fun and learning experiences we had together, it is definitely an unforgettable experience. I hope we can preserve this friendship throughout our lives while we all grow together and achieve great feats.

I cannot forget all my professors from Universite de Toulon (UTLN) as well during the first year of my master's degree. Thank you all for shaping me and all my colleagues and preparing us for what's coming ahead. A special thank you to Prof. Vincent and Claire Dune for the internship opportunity at COSMER, and thank you for all the effort from Prof. Ricard Marxer for coordinating this great Master's degree. I hope it continues to raise even more students in the coming years.

Finally, I would like to thank my family for their unconditional love and unending support without which I could not have reached this point.

Abstract

Being perfect at a single task in a specific situation is not what is ultimately sought from robots. One of the main factors of appeal towards robots is the promise to autonomously perform their tasks while adapting to the context around them. Big data and machine learning have proven their ability to produce more general solutions that are less confined to specific clear contexts (i.e. computer vision contexts where very big datasets are available to the public). Robotics has yet to enter that stage because every robot is different. Even if the robots are the same, the tasks they are doing may be different. Even if the tasks are the same, the contexts around them when performing these tasks could also be different. Narrative-Enabled Episodic Memories (NEEMs) were designed to include context in the form of ontologies within the data collected from robot sensors. The storage and joining of symbolic (context) and non-symbolic (numerical data from sensors) data in a relational database (SQL) that is available in the cloud is an important step for robotics. This is the main goal of this master's thesis. The work done here shows how the NEEMs can deal with these two types of data (Symbolic & non-Symbolic) stored and linked in an SQL database. This work also shows complete machine learning pipelines on different robotic tasks starting from querying the database to fitting the model and testing it in simulation. Two experiments have been done in this work on two different machine learning problems, the first is to *Predict Next Task*, which tries to make use of the context and the history of operation of the robot to try and predict the next task(s) in a plan and see how the context will help in that. The second experiment focuses on *Failure Recovery* in which the robot has failed to perform its task and has to recover from this failure, so the goal has been to make use of the context and the previous experiences in the memory to find the best way to recover from this failure. For each experiment, the results are reported, the benefit of the context is discussed, as well as, which information in the context has been more relevant for each task. Moreover, a simulation of the results of one of the experiments has been performed on a simulated PR2 robot in an apartment environment to see how the system performs with the complete robot software. A Jupyter Notebook that shows and tests the whole machine learning pipeline from querying the database to testing the model in simulation has been provided.

Contents

CHAPTER 1

INTRODUCTION

Contents

1.1 Work Motivation

In the last one and half decades the scientific community has concentrated on *deep learning*. This is mainly due to the advent of *big data* and the development of hardware like GPUs that enable *machine learning* to achieve its potential. But, *robotics* has not yet reached the point of managing truly *big data*. To be more specific, the part of robotics that deals with the robot's *cognitive abilities* is far behind in the usage of deep learning and *big data*. Other areas like *perception through computer vision* have benefited from *big data* due to the availability of images all over the internet and also due to the focus of the community on solving the vision problems since it was not just important for robotics but to other domains like security, surveillance, medical applications, and many others.

To be able to leverage the success of *machine learning* in the area of *cognitive robotics*, one has to start with getting *big data* for *cognitive robotics*. The reason why *big data* has not yet arrived in *cognitive robotics* is due to many reasons. Some of these reasons are the small number of robots available for research, the differences that exist in the robot hardware, the differences in the deployment environment, and the huge amount of different tasks that a robot should do. A big issue is that the robotics community also has different software and robot cognitive architectures that make the data formats differ making integration and collaboration very hard. In addition, when the robot has to be deployed in areas outside the industry, the amount of variability in the environment and the situations that the robot has to deal with is huge, and the required data increase exponentially with the increase in the variability of the situations.

The robotics community needs to have a way to share the data collected from their robots in a way that is independent of robot hardware and setup as much as possible or at least include in the data: the type of robot, hardware, environment, and task that was used. In addition, the data collection, formatting, pre-processing, and curation processes could be automated. Database management systems are an excellent solution for data storage, retrieval, and maintenance, but they do not solve data collection and formatting problems.

For this purpose, the Narrative-Enabled Episodic Memories or *NEEMs* [13] and the NEEM HUB[1] (Figure 1.1) were developed as part of *CRAM* (Cognitive Robot Abstract Machine)[3, 2] under the EASE project[2] which tries to solve most of these problems for the domain of robotic manipulation in everyday activities at home. Figure 1.2 shows the *CRAM* illustration diagram. The *NEEMs* are episodic memories of the robot while performing an activity. What is special about NEEMs is that they do not just consist of the low-level information of the robot motors, and sensors but also they add the narrative of the activity which includes the context around the activity. This context includes the plans, the goals, the states of objects and their description, the facts that the robot knew at that time, and the reasoning that the robot performed. All this information is time annotated so that it can be linked to the low-level information too by matching the time at which they occurred.

The information in the NEEMs can be categorized into symbolic data (the narrative), and non-symbolic data (the low level motor and sensor information). The symbolic data comes mainly from using ontologies like the *SOMA* ontology (The SOcio-physical Model of Activities) [4], which is an ontology that models everyday activities. This model is agent agnostic, so it works for both a robot and a human, the usage of the ontologies and the reasoners used by the robot is developed as part of Knowrob [2].

The NEEMs are stored in a MongoDB[3] database, which is a NoSQL database that is less strict than SQL databases and provides flexibility for the data structure, basically allowing for schemaless data storage. This decreases the need of a structure for storing the data and lowers the engineering work on the data before storing it, also it allows for easy update of the data with time without having to conform to a specific struc-

[1]https://neemgit.informatik.uni-bremen.de/neems
[2]https://ease-crc.org/
[3]https://www.mongodb.com/

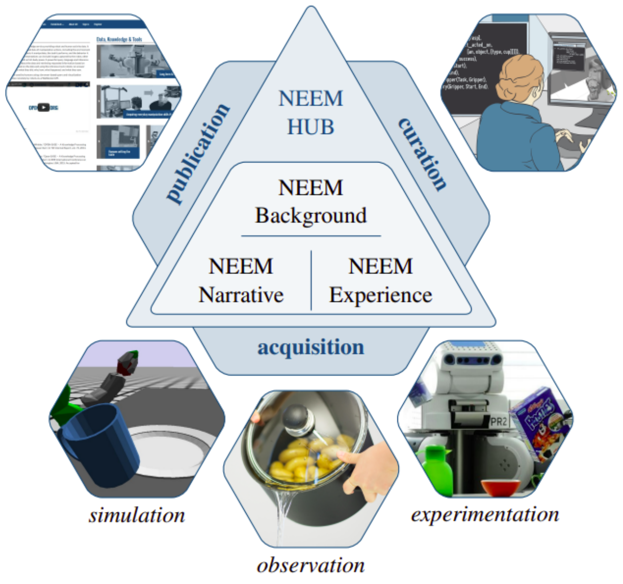


Figure 1.1: NEEM HUB overview.

ture. As machine learning algorithms and pipelines favor structured data, it provides a consistent structure which provides a consistent input to the machine learning algorithm which is crucial, also forcing the users to conform to the datatypes and structure which is important for the maintenance of the data and for the ability of old data to work with newer data on the same machine learning algorithms without having to adapt them.

The key research in master this thesis is to find the best way to store the NEEMs in an SQL database while preserving all the information and having them easily accessible, understandable, and expandable. Since the main goal is to allow for *big data for cognitive robotics*. As hypothesized in [11]: *"A memory system in a cognitive robot control architecture mediates between (i) high-level abilities, usually represented in a symbolic manner, such as language understanding, scene understanding, planning, plan execution monitoring and reasoning, and (ii) low-level abilities, such as sensor data processing, sensorimotor control"*. That is exactly what the NEEMs are trying to achieve, but they currently lack being big scale and do not support queries on multiple episodic memories at the same time, so one of the goals of this work is to mitigate this issue.

Another key research question in this master thesis is to find out which context or which parts of the narrative included in the NEEMs are relevant for which tasks. This would help improve the NEEMs by adding missing information in the context in newer NEEMs or by removing any redundant or useless information from them. This would help the robotics research community in finding the relevant information that the robot needs in order to perform everyday tasks and in inspiring algorithms that focus on these

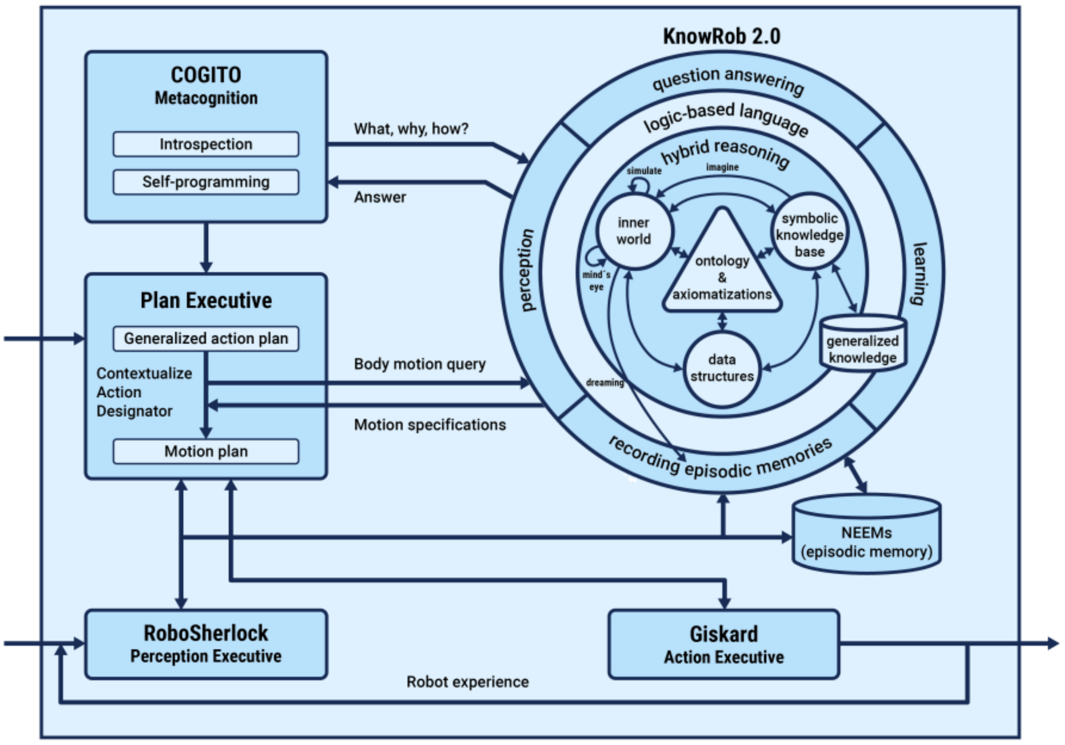


Figure 1.2: The *CRAM* cognitive architecture including the *NEEMs*.

aspects and also on techniques to acquire such information while the robot is performing its task.

1.2 Literature Review

Allowing robots to have memory is not a new concept in the literature since there are previous research works that created artificial cognitive architectures (in which memory is a crucial part) that try to mimic humans' cognitive processes [1, 12]. Kotseruba *et al.* [8] estimated that the number of artificial cognitive architectures are around 300. Most of the previous work on artificial memory focus on conceptual aspects, such as the memory including cognitive concepts like working memory (WM) [7, 9] and long-term memory (LTM) [6]. However, there are no works regarding more practical aspects like efficiency, making these memories shared among robots and allowing for big-scale data collection and storage to make use of the current technological advancement in machine learning. Although Peller-Konrad [11] *et al.* have shown in some of their diagrams the possibility of using SQL databases, there are no real details or implementation behind it.

Thus, this is the reason why this work focus is not to develop new cognitive architecture or memory concepts, but to allow for big-scale data collection, storage, and retrieval in an efficient and easy manner of an already existing memory concept, that is the NEEMs. The goal is to preserve all the benefits and the concepts behind the NEEMs, while making them work on a bigger scale and also on the cloud. And finally, to show how machine learning pipelines would benefit from this improvement.

1.3 Objectives

The first objective was to migrate the NEEMs from the NoSQL database to MariaDB[4] which is an open-source SQL database based on MySQL[5] and make it available for all users to insert and retrieve data using SQL queries.

The second objective was to use JPTs (Joint Probability Trees) [10] which is a statistically transparent machine learning algorithm based on PCs (Probabilistic Circuits)[5] to learn from the NEEMs database and show what helps improve the robot performance while doing activities similar to the ones performed in the NEEMs:

- One aspect to learn from the NEEMs is the ability to infer the next best task for the robot to do (given the environment and robot state) and see if the model can provide useful predictions which would replace the need to write rigid hand-crafted plans for the robot to perform similar activities that differ slightly (i.e. performed at a different kitchen).
- Another aspect to learn is failure recovery, since the NEEMs also include the ultimate state of the tasks performed, whether they failed or succeeded, and what was done after they failed to recover from such failure.

1.4 Environment and Initial State

As mentioned in section 1.1, the starting point of this master thesis work is that the NEEMs are stored in a NoSQL database called MongoDB. So the migration task to a SQL database was imposed on me as my first task during my thesis internship by the research group, but the design decisions on how the migration will be performed and what changes needed to be made in the data, and what final structure the data will take in the SQL database was left for me to decide. Some of my decisions were influenced by the end users in the research group such that it worked well for their needs. Other decisions were solely influenced by me for reasons related to implementation complexity and maintainability of the SQL database.

The usage of the JPT machine learning algorithm was convenient since it was developed in the research group and helped improve the collaboration with other researchers

[4]https://mariadb.org/
[5]https://www.mysql.com/

in the group and provide feedback to and from the research group. The main goal is to show a complete machine learning pipeline from writing an SQL query to retrieve the data from the newly formed NEEM SQL database to use the data to fit a machine learning model and test it on simulation to prove the usability and the benefit of the work done on the new SQL database.

The research question of finding which parts of the narrative/context are relevant for the tasks to perform by the robot was decided by me after consultation with my supervisors and colleagues in the research group.

Finally, I have had access to all the software developed by the research group and their databases. Most notable are the *cram* organization[6], and the *Institute for Artificial Intelligence - University of Bremen* organization[**?**] on GitHub, and the *NEEM-Hub* on Gitlab[7].

[6]https://github.com/cram2
[7]https://neemgit.informatik.uni-bremen.de/neems

CHAPTER 2

Planning and resources evaluation

Contents

2.1 Planning

Table 2.1 details the time planning of the work, including all its tasks and subtasks, and the dependencies between them are shown.

An illustrated Gantt chart can be seen in Figure 2.1. The plan has been modified throughout the work.

2.2 Resource Evaluation

This work required minimal resources of a PC and a local server for maintaining the database. Most of the work is available as software components without the need for any hardware. The PCs and the server were provided by the Institute For Artificial Intelligence (IAI) in Bremen.

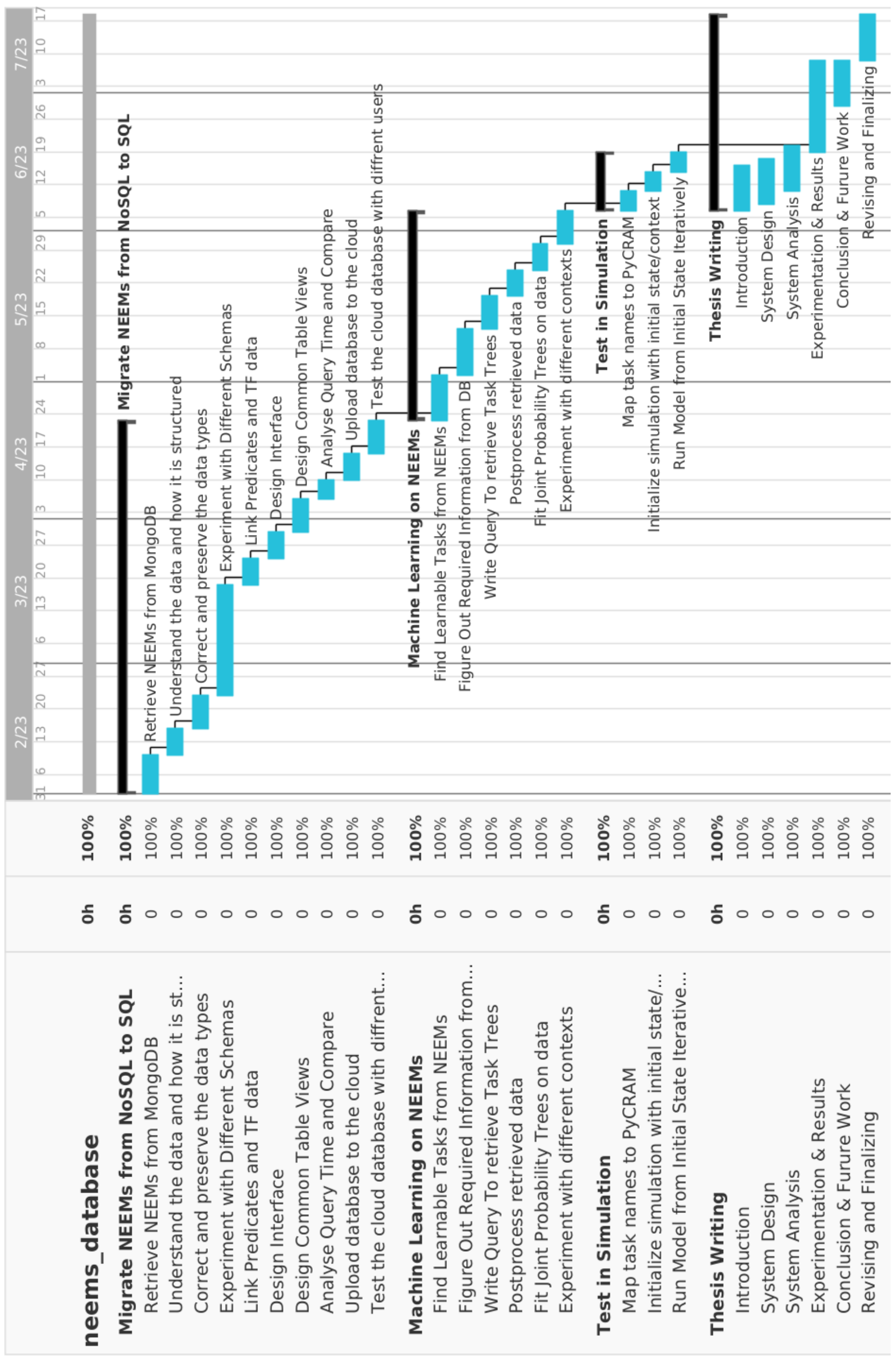


Figure 2.1: The Gantt chart for all the tasks in this work, the work break down structure is also shown in Table 2.1

WBS #	Name / Title	Type	Start Date	End Date
1.1	**Migrate NEEMs from NoSQL to SQL**	**group**	**2023-02-01**	**2023-04-20**
1.1.1	Retrieve NEEMs from MongoDB	task	2023-02-01	2023-02-08
1.1.2	Understand the data and how it is structured	task	2023-02-09	2023-02-14
1.1.3	Correct and preserve the data types	task	2023-02-15	2023-02-21
1.1.4	Experiment with Different Schemas	task	2023-02-22	2023-03-16
1.1.5	Link Predicates and TF data	task	2023-03-17	2023-03-22
1.1.6	Design Interface	task	2023-03-23	2023-03-28
1.1.7	Design Common Table Views	task	2023-03-29	2023-04-04
1.1.8	Analyse Query Time and Compare	task	2023-04-05	2023-04-07
1.1.9	Upload database to the cloud	task	2023-04-10	2023-04-13
1.1.10	Test the cloud database with different users	task	2023-04-14	2023-04-20
1.2	**Machine Learning on NEEMs**	**group**	**2023-04-21**	**2023-06-05**
1.2.1	Find Learnable Tasks from NEEMs	task	2023-04-21	2023-05-01
1.2.2	Figure Out Required Information from DB	task	2023-05-02	2023-05-10
1.2.3	Write Query To retrieve Task Trees	task	2023-05-11	2023-05-17
1.2.4	Postprocess retrieved data	task	2023-05-18	2023-05-23
1.2.5	Fit Joint Probability Trees on data	task	2023-05-24	2023-05-29
1.2.6	Experiment with different contexts	task	2023-05-30	2023-06-05
1.3	**Test in Simulation**	**group**	**2023-06-06**	**2023-06-16**
1.3.1	Map task names to PyCRAM	task	2023-06-06	2023-06-08
1.3.2	Initialize simulation with initial state/context	task	2023-06-09	2023-06-13
1.3.3	Run Model from Initial State Iteratively	task	2023-06-14	2023-06-16
1.4	**Thesis Writing**	**group**	**2023-06-06**	**2023-07-17**
1.4.1	Introduction	task	2023-06-06	2023-06-14
1.4.2	System Design	task	2023-06-07	2023-06-15
1.4.3	System Analysis	task	2023-06-09	2023-06-19
1.4.4	Experimentation & Results	task	2023-06-19	2023-07-06
1.4.5	Conclusion & Furure Work	task	2023-06-28	2023-07-06
1.4.6	Revising and Finalizing	task	2023-07-07	2023-07-17

Table 2.1: Project Schedule

Chapter 3

System Analysis and Design

Contents

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

3.1 Requirement Analysis

As mentioned in the previous chapter, machine learning pipelines work best when given with structured data. The problem is that the current NEEMs database is based on MongoDB[1] which is a NoSQL database so for easily querying is required to convert this to an SQL database which has a more maintainable structure for machine learning purposes.

Since the question “How to convert between a NoSQL database to an SQL database?” has no definitive or general answer that works for all cases, this leaves us with some design choices that will affect how the final SQL database will look like. To be able to answer this question, in this section, I will start by listing the requirements and dividing them into both functional and non-functional requirements.

[1]https://www.mongodb.com/

3.1.1 Functional Requirements

The proposed system can be divided into two main functionalities which have their own functional requirements. The first is to prove a service for the user that coverts NEEMs from a MongoDB server to a MariaDB server. The second main functionality is to provide the ability to query this database and this query must have certain functional requirements as well.

The data migration to SQL database should:

- Convert all current usable NEEMs from the MongoDB to the MariaDB.
- Provide the functionality to update the MariaDB with new NEEMs that were added to the MongoDB.
- Provide a user interface for the conversion functionality where the user can provide the inputs like the MongoDB and MariaDB server URIs.
- Provide correct datatypes for all the retrieved data.

The SQL query should:

- be queried by a user for retrieving user-needed information in a table that will be used for machine learning as the primary purpose.
- have the ability to query information from multiple NEEMs at the same time.
- be able to queried from anywhere, and be accessible as a server to all allowed users both locally and remotely.

The plan to implement the functional requirements will be detailed in the system design section (see Section 3.2). And the two main functional requirements of the system are shown in the Tables 3.1 and 3.2.

Table 3.1: Functional requirement «Migrate NEEMs from NoSQL to SQL».

Input:	A MongoDB server URI with NEEMs, A MariaDB server URI
Output:	Migrated NEEMs to the provided MariaDB server
Given a MongoDB server that contains NEEMs, and a MariaDB server, the code will migrate all found NEEMS to the given MariaDB server	

3.1.2 Non-functional Requirements

The non-functional requirements for the proposed NEEMs conversion from NoSQL to SQL can be summarized as follows:

Table 3.2: Functional requirement «SQL Query».

Input:	SQL Query from a user
Output:	Retrieved data from multiple NEEMs as a table
The user queries the database using an SQL query to retrieve data as a table from multiple NEEMs to be used for machine learning purposes from anywhere	

- Joining of robot TF data (i.e. the time stamped Transformation Frames or poses of all robot frames during its operation) and the triples (i.e. semantic data that constitutes the robot knowledge and reasoning).
- Structuring the data such that it is fairly understandable by a new user and can easily find and write down the query that will retrieve the required information.
- Structuring and indexing the tables in the database such that the most common queries do not take much time to provide the results.
- Making the triple data more readable for the user by removing any unnecessary information from the names of objects like big ontology links.
- Preserving the relationship between the tables/data structures from the MongoDB when converting to the MariaDB.
- Structuring the database such that it is stable and maintainable and easily updated with new data.
- Enabling the database updating directly using SQL commands instead of having to migrate from MongoDB.
- Providing another interface that enables users to utilize the developed APIs for another custom usage other than the command line interface which only provides conversion from MongoDB to MariaDB.
- Providing a way to filter the NEEMs to be converted, such that only specific NEEMs are converted from MongoDB to MariaDB.
- Providing loading bars that show the conversion progress as feedback to users.
- Allowing users to specify if certain bugs in the data are skipped or should the conversion stop when these bugs are encountered.

The non-functional requirements for the querying of the created MariaDB are the following:

- Provide most common queries as table views which make things much faster for the most common use cases.

The plan to implement the non-functional requirements will be detailed in the system architecture section (see Section **??**).

3.2 System Design

Figure 3.1 shows an illustration of the system design. The system is the complete machine learning pipeline starting from collecting the data from the robots. The data can be one of two types symbolic data which is stored in ontologies, and non-symbolic data including the robot sensory and sensorimotor control data. Then this data is transferred to the *NEEM-Hub* and stored in a no-SQL database, namely MongoDB. Then the data is given structure and linked together in an SQL database, namely MariaDB. After that, the database can be queried for required data needed by the machine learning algorithms the learn specific tasks from the *NEEMs* SQL database. Finally, after fitting the machine learning models, the models can be used online during robot operation to extract evidence which includes the context and answer the robot queries instantly.

3.3 Interface Design

The designed interface is a command line interface for the software that migrates the *NEEMs* from no-SQL to SQL. This interface allows users to provide the credentials to connect to the NoSQL database and the SQL database and provide some additional arguments that are needed to complete the migration process, the arguments are described as shown in Table 3.3.

Moreover, the program provides loading bars for each step in the migration process as shown in Figure 3.2, these steps are:

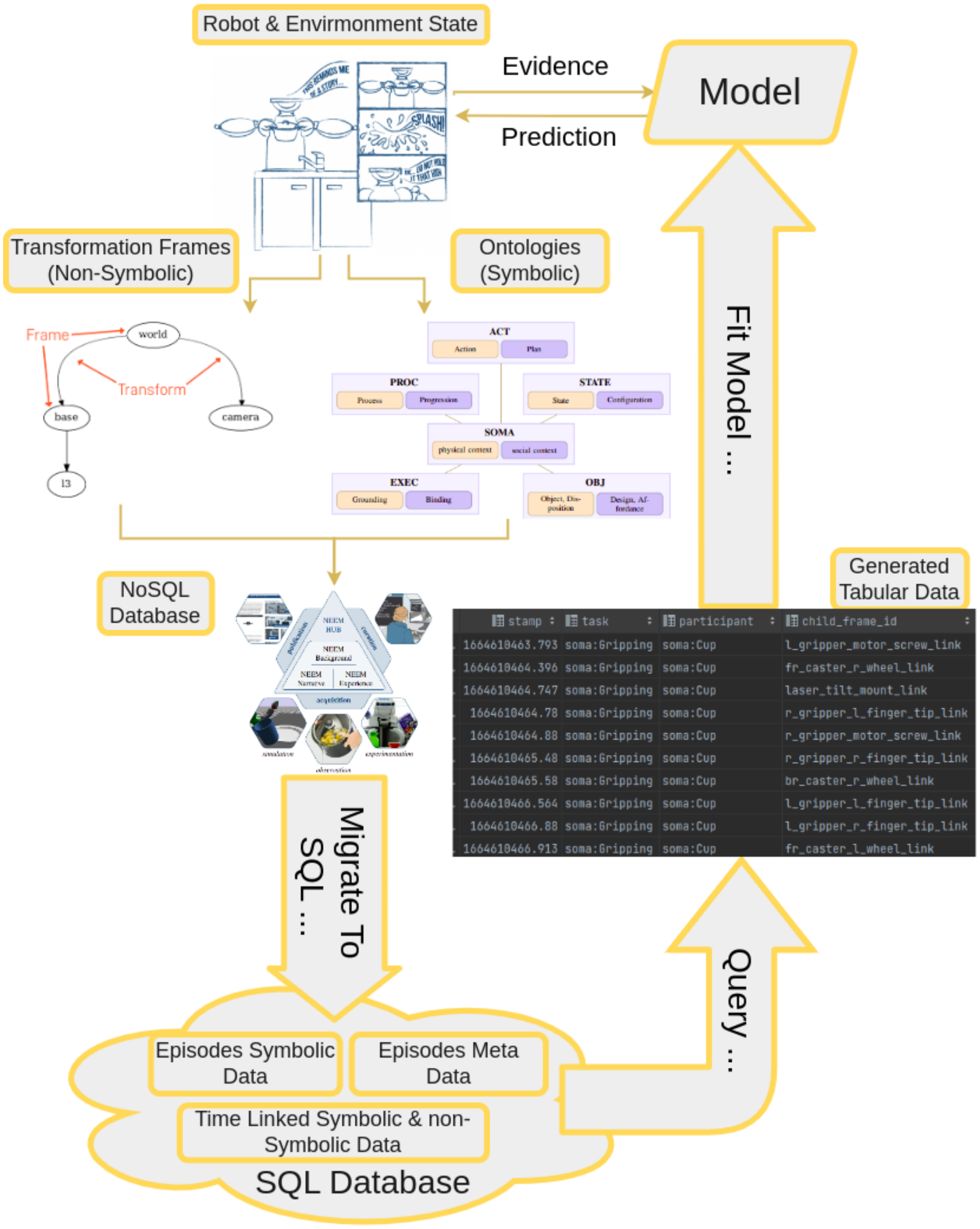
1. **Verifying the data:** ensuring the data is coherent and there is no corruption or inconsistency.

2. **Collecting and restructuring the data:** to provide the unstructured data with a tabular structure.

3. **Generating SQL commands:** producing the commands that will create the database schema and upload the data to the SQL database.

4. **Linking predicate tables:** indexing the predicate tables and linking them by using foreign keys, so that each predicate has a NEEM that it belongs to, and should be linked to it using the NEEM_id.

5. **Linking TF and triples:** linking the transformation frames to the predicates or triples using the time they occurred, this will make it easy to query both the symbolic and the non-symbolic data together.

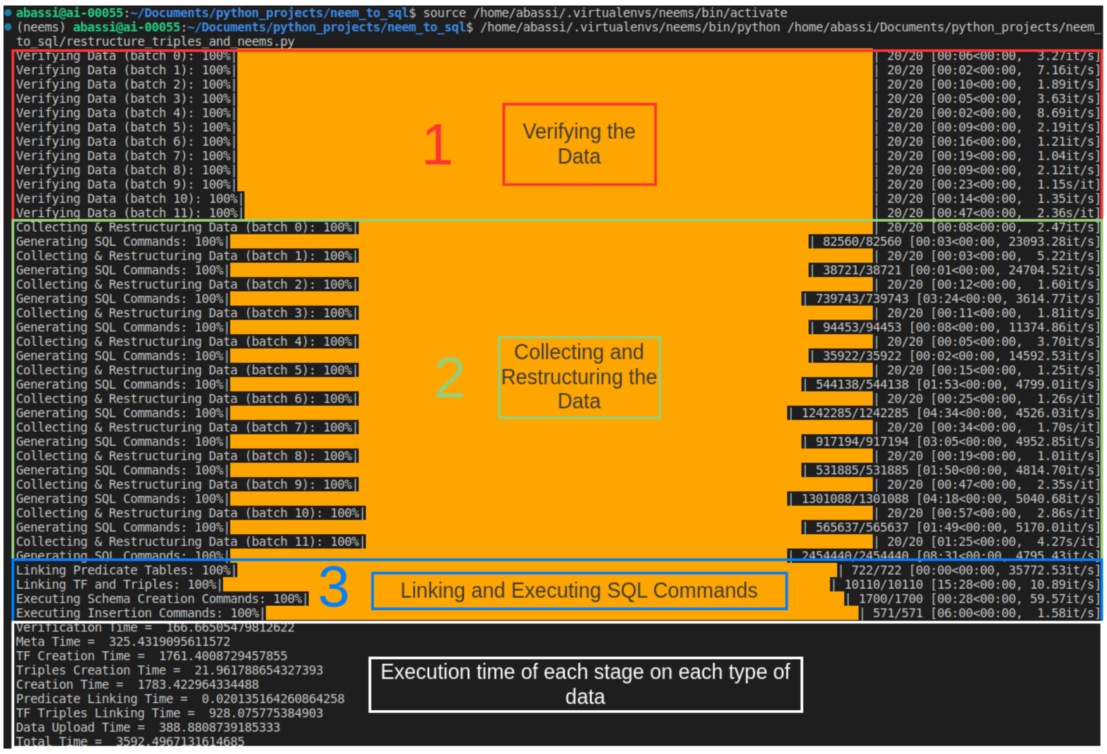


stamp	task	participant	child_frame_id
1664610463.793	soma:Gripping	soma:Cup	l_gripper_motor_screw_link
1664610464.396	soma:Gripping	soma:Cup	fr_caster_r_wheel_link
1664610464.747	soma:Gripping	soma:Cup	laser_tilt_mount_link
1664610464.78	soma:Gripping	soma:Cup	r_gripper_l_finger_tip_link
1664610464.88	soma:Gripping	soma:Cup	r_gripper_motor_screw_link
1664610465.48	soma:Gripping	soma:Cup	r_gripper_r_finger_tip_link
1664610465.58	soma:Gripping	soma:Cup	br_caster_r_wheel_link
1664610466.564	soma:Gripping	soma:Cup	l_gripper_l_finger_tip_link
1664610466.88	soma:Gripping	soma:Cup	l_gripper_r_finger_tip_link
1664610466.913	soma:Gripping	soma:Cup	fr_caster_l_wheel_link

Figure 3.1: Overview of the system design.

6. **Executing Schema Creation Commands:** this executes first the commands that will create the tables with their respective columns, primary keys, and foreign keys.

7. **Executing Insertion Commands:** this executes the commands that will insert the data to the created tables in the correct order obeying the foreign keys and the order of the tables.



```
● abassi@ai-00055:~/Documents/python_projects/neem_to_sql$ source /home/abassi/.virtualenvs/neems/bin/activate
● (neems) abassi@ai-00055:~/Documents/python_projects/neem_to_sql$ /home/abassi/.virtualenvs/neems/bin/python /home/abassi/Documents/python_projects/neem_
to_sql/restructure_triples_and_neems.py
Verifying Data (batch 0): 100%|                                  | 20/20 [00:06<00:00,  3.27it/s]
Verifying Data (batch 1): 100%|                                  | 20/20 [00:02<00:00,  7.16it/s]
Verifying Data (batch 2): 100%|                                  | 20/20 [00:10<00:00,  1.89it/s]
Verifying Data (batch 3): 100%|                                  | 20/20 [00:05<00:00,  3.63it/s]
Verifying Data (batch 4): 100%|                                  | 20/20 [00:02<00:00,  8.69it/s]
Verifying Data (batch 5): 100%|                                  | 20/20 [00:09<00:00,  2.19it/s]
Verifying Data (batch 6): 100%|                                  | 20/20 [00:16<00:00,  1.21it/s]
Verifying Data (batch 7): 100%|                                  | 20/20 [00:19<00:00,  1.04it/s]
Verifying Data (batch 8): 100%|                                  | 20/20 [00:09<00:00,  2.12it/s]
Verifying Data (batch 9): 100%|                                  | 20/20 [00:23<00:00,  1.15s/it]
Verifying Data (batch 10): 100%|                                 | 20/20 [00:14<00:00,  1.35it/s]
Verifying Data (batch 11): 100%|                                 | 20/20 [00:47<00:00,  2.36s/it]
Collecting & Restructuring Data (batch 0): 100%|                 | 20/20 [00:08<00:00,  2.47it/s]
Generating SQL Commands: 100%|                   | 82560/82560 [00:03<00:00, 23093.28it/s]
Collecting & Restructuring Data (batch 1): 100%|                 | 20/20 [00:03<00:00,  5.22it/s]
Generating SQL Commands: 100%|                   | 38721/38721 [00:01<00:00, 24704.52it/s]
Collecting & Restructuring Data (batch 2): 100%|                 | 20/20 [00:12<00:00,  1.60it/s]
Generating SQL Commands: 100%|                 | 739743/739743 [03:24<00:00, 3614.77it/s]
Collecting & Restructuring Data (batch 3): 100%|                 | 20/20 [00:11<00:00,  1.81it/s]
Generating SQL Commands: 100%|                   | 94453/94453 [00:08<00:00, 11374.86it/s]
Collecting & Restructuring Data (batch 4): 100%|                 | 20/20 [00:05<00:00,  3.70it/s]
Generating SQL Commands: 100%|                   | 35922/35922 [00:02<00:00, 14592.53it/s]
Collecting & Restructuring Data (batch 5): 100%|                 | 20/20 [00:15<00:00,  1.25it/s]
Generating SQL Commands: 100%|                 | 544138/544138 [01:53<00:00, 4799.01it/s]
Collecting & Restructuring Data (batch 6): 100%|                 | 20/20 [00:25<00:00,  1.26s/it]
Generating SQL Commands: 100%|               | 1242285/1242285 [04:34<00:00, 4526.03it/s]
Collecting & Restructuring Data (batch 7): 100%|                 | 20/20 [00:34<00:00,  1.70s/it]
Generating SQL Commands: 100%|                 | 917194/917194 [03:05<00:00, 4952.85it/s]
Collecting & Restructuring Data (batch 8): 100%|                 | 20/20 [00:19<00:00,  1.01it/s]
Generating SQL Commands: 100%|                 | 531885/531885 [01:50<00:00, 4814.70it/s]
Collecting & Restructuring Data (batch 9): 100%|                 | 20/20 [00:47<00:00,  2.35s/it]
Generating SQL Commands: 100%|               | 1301088/1301088 [04:18<00:00, 5040.68it/s]
Collecting & Restructuring Data (batch 10): 100%|                | 20/20 [00:57<00:00,  2.86s/it]
Generating SQL Commands: 100%|                 | 565637/565637 [01:49<00:00, 5170.01it/s]
Collecting & Restructuring Data (batch 11): 100%|                | 20/20 [01:25<00:00,  4.27s/it]
Generating SQL Commands: 100%|               | 2454440/2454440 [08:31<00:00, 4795.43it/s]
Linking Predicate Tables: 100%|                    | 722/722 [00:00<00:00, 35772.53it/s]
Linking TF and Triples: 100%|                  | 10110/10110 [15:28<00:00, 10.89it/s]
Executing Schema Creation Commands: 100%|            | 1700/1700 [00:28<00:00, 59.57it/s]
Executing Insertion Commands: 100%|                    | 571/571 [06:00<00:00,  1.58it/s]
Verification Time =  166.66505479812622
Meta Time =  325.4319095611572
TF Creation Time =  1761.4008729457855
Triples Creation Time =  21.961788654327393
Creation Time =  1783.422964334488
Predicate Linking Time =  0.020135164260864258
TF Triples Linking Time =  928.075775384903
Data Upload Time =  388.8808739185333
Total Time =  3592.4967131614685
```

1
Verifying the Data
2
Collecting and Restructuring the Data
3
Linking and Executing SQL Commands
Execution time of each stage on each type of data

Figure 3.2: The NoSQL to SQL Migration Process.

Figure 3.2 shows the loading bars for each step of the migration process to inform the user of the current progress and provide feedback on which step the system takes time and where it encountered a problem. The first stage is the verification of the data, the second is the creation of structure by creating the tables, and the third is linking the tables and executing the SQL commands to upload the data to the database. Finally, the execution time of each part is printed.

Table 3.3: Command Line Arguments for the Migration of the NEEMs for NoSQL database to SQL database.

Argument	**Description**
`-drop`, `-d`	Drop the tables that will be inserted first
`-skip_bad_triples`, `-sbt`	Skip triples that are missing one of subject, predicate or object
`-allow_increasing_sz`, `-ais`	Allow increasing the size of the original data type of a column
`-allow_text_indexing`, `-ati`	Allow indexing text type columns
`-max_null_percentage`, `-mnp`	Maximum percentage of null values allowed in a column otherwise it will be put in a separate table
`-batch_size`, `-bs`	Batch size (number of neems per batch) for uploading data to the database, this is important for memory issues, if you encounter a memory problem try to reduce that number
`-num_batches`, `-nb`	Number of batches to upload the data to the database
`-start_batch`, `-sb`	Start uploading from this batch
`-dump_data_stats`, `-dds`	Dump the data statistics like the sizes and time taken for each operation to a file
`-sql_username`, `-su`	SQL username
`-sql_password`, `-sp`	SQL password
`-sql_database`, `-sd`	SQL database name
`-sql_host`, `-sh`	SQL host name
`-sql_uri`, `-suri`	SQL URI this replaces the other SQL arguments
`-mongo_username`, `-mu`	MongoDB username
`-mongo_password`, `-mp`	MongoDB password
`-mongo_database`, `-md`	MongoDB database name
`-mongo_host`, `-mh`	MongoDB host name
`-mongo_port`, `-mpt`	MongoDB port number
`-mongo_uri`, `-muri`	MongoDB URI this replaces the other MongoDB arguments
`-log_level`, `-logl`	Log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
`-neem_filters_yaml`, `-nfy`	YAML file containing the neem filters

Chapter 4

Work Development and Results

Contents

The developed work and the obtained results are shown in this chapter. Also the deviations from the initial planning are and justified.

4.1 Conversion of NEEMs from NoSQL to SQL

4.1.1 The Good, The Bad, and The Ugly in Existing NEEMs

As mentioned in Section 3.2, the NEEMs consist of symbolic and non-symbolic data. the symbolic data being the triples, and the non-symbolic data being the robot Frames Transformations (FT).

The Triples represent the ontologies that contain the robot's knowledge and on which the robot performs its logical reasoning while it was performing its task. The triples are stored in the MongoDB as collections of three variables: the subject (S), the predicate (P), and the object (O).

The triples' objects have different datatypes depending on the property/predicate but, in the MongoDB, they are all stored as strings. The datatypes can be recovered though from the predicate “rdf:Range” which takes as a subject the property/predicate name and gives as object the datatype.

The datatypes found in the NEEMs are mostly “XSD” [1] datatypes, but also some “SOMA” [2] defined datatypes. For the XSD datatypes, an automatic conversion exists using the “RDFLib” [3] python library. While the “SOMA” datatypes conversion like the “soma_array” datatype had to be implemented in the code.

The Transformation Frame data (TF data) are the coordinates of the robot frames represented as transformations which are ROS messages logged over time from the TF package[4] as seen from the example in Figure 4.1[5].

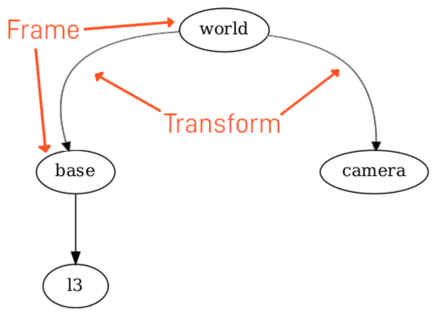


Figure 4.1: ROS TF frames and corresponding transformations example

The *TransformStamped* ROS message has a hierarchy that is represented as seen in Figure 4.2. This same structure is found in the database.

4.1.2 Finding a Rigid Structure from a Flexible One

The relation between data in the NEEMS included: (i) ONE to ONE relations (1-1); (ii) ONE to MANY relations (1-N) and (iii) MANY to MANY relations (N-N).

One to one (1-1) relationships are modeled as columns in the same table or as separate tables linked together by their primary key. The simplest example is in the TF data where each TF must have one *header* and each *header* must have one time *stamp*, etc.

To fins out the type of relationship between the data a preliminary pass through all the data is done. For example, for this pass one can deduce that a NEEM can have only one creator, thus the *created_by* variable can be stored as a column in the *neems*

[1]https://www.w3schools.com/xml/schema$_{d}types_{n}umeric.asp$
[2]https://ease-crc.github.io/soma/
[3]https://rdflib.readthedocs.io/en/stable/
[4]http://wiki.ros.org/tf
[5]https://articulatedrobotics.xyz/ready-for-ros-6-tf/

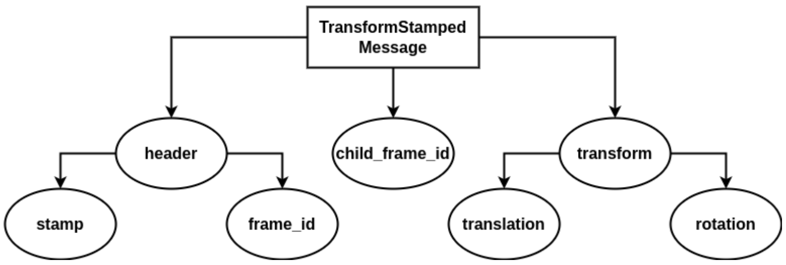


Figure 4.2: The data included in the ROS TransformStamped message. This includes the header which includes the time stamp and the frame id which is the name of the parent or reference frame. The child frame id is the name of the current frame. The transformation description including the translation and rotation

table. On the other hand, a NEEM can include more than one *activity*, thus it is a ONE (for the NEEMs) to MANY (for the *activity*), and similarly for the MANY to MANY relationships.

The ONE to MANY and the MANY to MANY are handled by storing the data in separate tables and having a third table linking them together by referencing both. This is illustrated in Figure 4.3 where the *neems table* is linked to the *neems_activity* table by another third table called *neems_activity_index* table that references data from both tables.

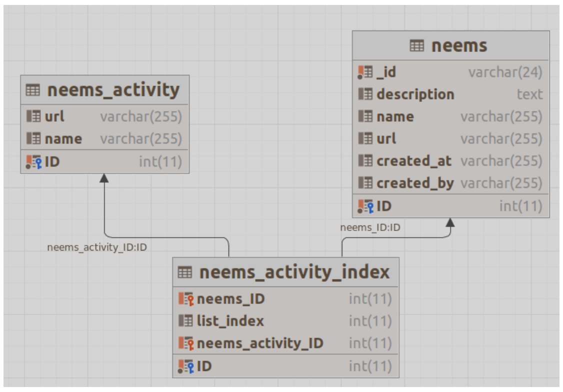


Figure 4.3: Overview of *neems* table, the *neems_activity* table, and the *neems_activity_index* table which acts as the mediator that links the *neems* table with the *neems_activity* table. This is an example of how ONE to MANY and MANY to MANY relationships are handled in the database.

4.1.3 How to handle Triples?

Predicates as Tables. The fact that each predicate had a different datatype for its object suggests that each predicate should represent a table in the SQL database. This table would have two columns, a column for the subject and a column for the object each with its own datatype since columns in SQL tables can have only one datatype. Although there is the BLOB datatype which can convert every datatype to a bytes datatype, it is not recommended to do so, because it will lose the identity of each datatype and will result in loss of information, also it would increase the size of the database and will increase the time a query gets resolved. In addition, it is more user-friendly and more informative to have a specific datatype for each column because it helps in inferring the type of data in that column.

Ontology Classes as Tables. The classes defined in the ontology like *Action*, *TimeInterval*, *Task*, etc. will be tables, these tables will have the predicates as column titles and the columns will contain the objects of these predicates, while the *Class Name* column will be the instances of this class. This structure will make the database content easier to understand by users since these classes are what user are interested in. For example, a user could easily find his way through the database and could easily determine what tables (s)he needs to join together to generate the table (s)he needs for their project or task. But, this is much harder to maintain because the classes of the ontology are susceptible to change or increase with new classes all the time, this makes the database unstable and will result in thousands of tables in a single database. Also, some predicates would have many objects for the same class instance, this requires the creation of a new table that is referenced by this column, this also increases the number of tables and the number of required joins and even nested joins to generate a table that contains these data. This would increase the complexity of the SQL query and could increase the time to resolve the query.

Triples to RDF Graph. Using the RDFLib python package, the loaded triples from the MongoDB can be loaded as an RDF Graph, this makes it easy to resolve ontology URIs to their shorter versions, and help in resolving datatypes like the XSD datatypes to python datatypes, also the RDF Graph can be queried easily for filtering or searching for specific triples.

RDF Graph to Dictionary. The RDF graph can be easily converted to a Python dictionary which can then be converted to an SQL Schema. The main algorithm that was developed converts JSON or Python dictionaries into SQL schema thus converting the triples from the RDF graph to the Python dictionary makes it very convenient because one can now re-use the algorithm (JSON/Dictionary to SQL tables) for converting the triples into SQL tables.

4.1.4 Linking Triples

In the case of **Predicates as Tables**, it was found that no linking is needed, since linking in SQL is done using a parent-child relationship between two tables, most predicates do not have this type of relationship, and the user can infer the relationship between the data in the tables by looking at the ontology itself instead of the SQL tables and then the user can perform the required joining between the tables using SQL. Indexing the columns of the predicate tables was very important because it made the queries of joining the tables resolve much faster than without indexing, and this is one of the key benefits of using SQL databases.

In the case of **Classes as Tables**, each Class relates to other Classes by a Predicate, these predicates are columns in the tables of each class, and since the classes usually have a hierarchy, they can be linked together with a child-parent relationship, and thus the foreign key of the SQL tables can be used, for example, and action is part of a task, which means one cannot have an action without first having a task, this means that a row must first exist in the *Task* table for an instance of a task before having a row in the *Action* table that uses this task. Figures 4.4 and 4.5 show the **Classes as Tables** schema *TimeInterval* and *Action* tables and their links respectively. This is how foreign keys work as well that is why they can be used in this type of schema but not in the **Predicates as Tables** schema. This creates many foreign keys and the schema gets very complicated, but the user can make use of the foreign keys to infer the relations of the tables without needing to look at the ontology.

A table summarizing the main differences between the two different schemas of **Classes as Tables** and **Predicates as Tables** is shown in Table 4.1. Looking at the table it is clear that **Predicates as Tables** is the clear winner here, that is why it was chosen as the schema type to use for the NEEMs SQL Database. The only drawbacks of **Predicates as Tables** is that there is no hierarchy (i.e. no parent and child relationship) between the predicates and the relationships between the predicate tables are not explicitly represented in the database, but this issue can be mitigated by looking at the ontology and visualizing it using software like Protege[6].

Table 4.1: Comparison of Classes and Predicates as Tables showing the advantages (+) and disadvantages (-) of each.

Classes as Tables	**Predicates as Tables**
More Vague (-)	More Understandable (+)
More Joins (-)	Less Joins (+)
1000+ tables (-)	∼200 tables (+)
Clear Hierarchy (+)	No Hierarchy (-)
Less Stable (-)	More Stable (+)
Explicit Relations (+)	Implicit Relations (from ontology) (-)

[6]https://protegeproject.github.io/protege/getting-started/

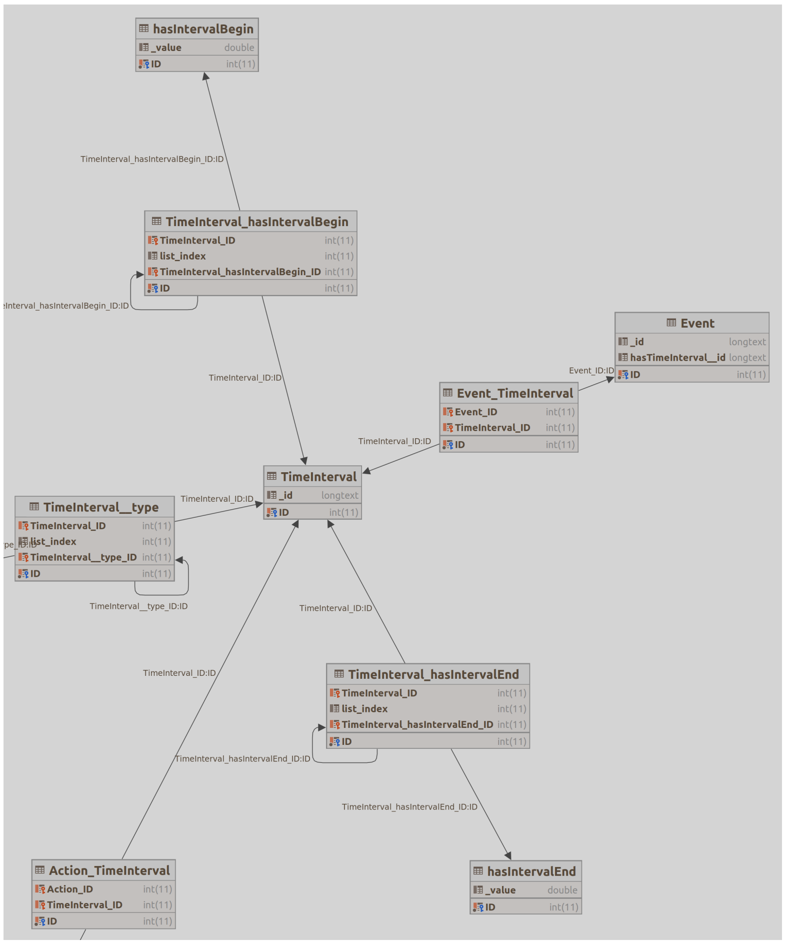


Figure 4.4: Overview of the **TimeInterval** table and some of its related tables in the **Classes as tables** schema. The links represent the relationships between the ontology classes. The links are explicitly represented as foreign keys in the database.

4.1.5 Joining TFs and Triples

The key that is used to link between the TF data and the triples is the time stamp, the stamp in the TF data that is found in the header part of the message can be used and compared with the time stamp of the symbolic actions in the triples that can be found using the predicate/property *hasTimeInterval* which has action name/instance as subject and gives an instance of a *TimeInterval* class as object, every *TimeInterval* has a start and end that can be found using the predicates *hasIntervalBegin* and *hasIntervalEnd* respectively. Both of these predicates take as subject a *TimeInterval* instance and gives a float as object which represents the time stamp. Figures 4.6 and 4.7 show a visual explanation of the tree *hasTimeInterval*, *hasIntervalBegin*, and *hasIntervalEnd* respectively.

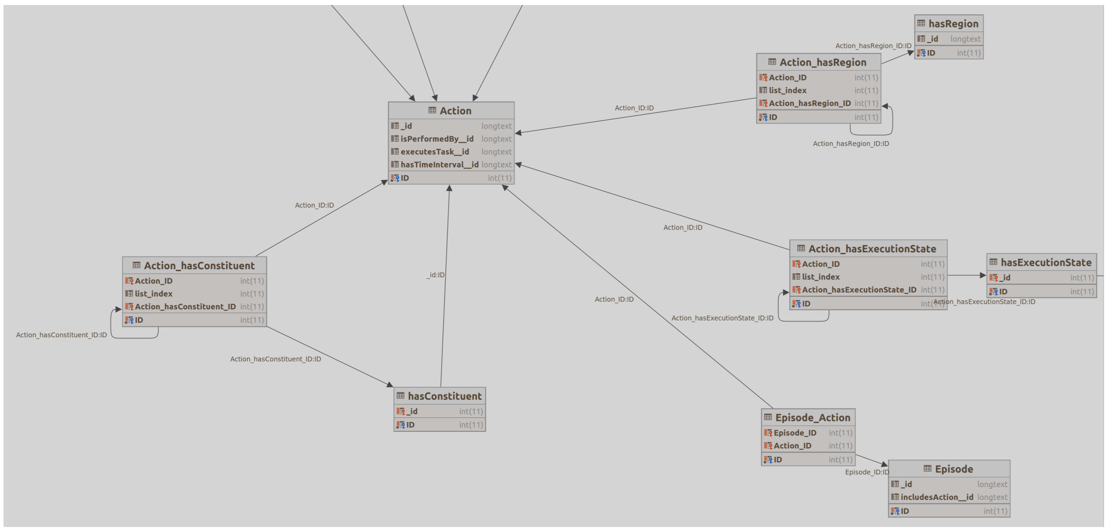
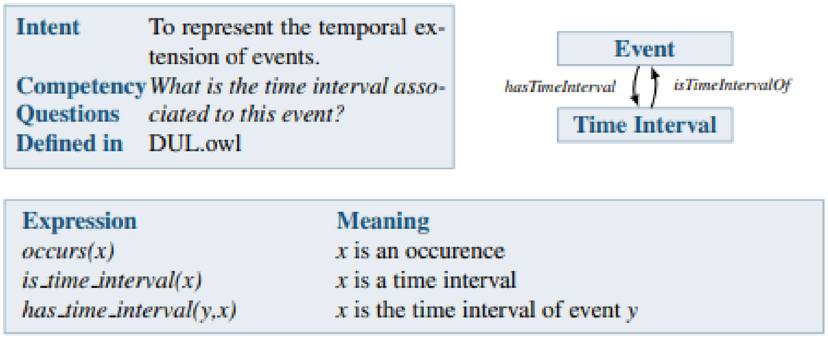


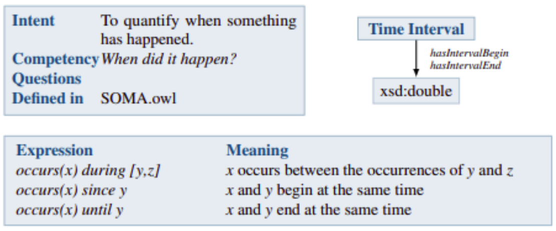
Figure 4.5: Overview of the **Action** table and some of its related tables in the **Classes as tables** schema. The links represent the relationships between the ontology classes. The links are explicitly represented as foreign keys in the database.



Intent	To represent the temporal extension of events.
Competency Questions	*What is the time interval associated to this event?*
Defined in	DUL.owl

Expression	**Meaning**
occurs(x)	*x* is an occurence
is_time_interval(x)	*x* is a time interval
has_time_interval(y,x)	*x* is the time interval of event *y*

Figure 4.6: Explanation of the *hasTimeInterval* predicate



Intent	To quantify when something has happened.
Competency Questions	*When did it happen?*
Defined in	SOMA.owl

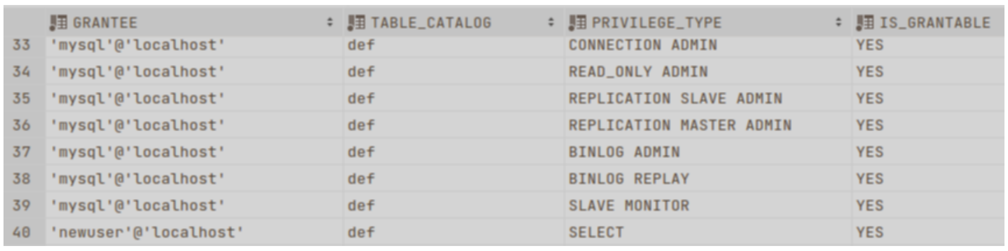
Expression	**Meaning**
occurs(x) during [y,z]	*x* occurs between the occurrences of *y* and *z*
occurs(x) since y	*x* and *y* begin at the same time
occurs(x) until y	*x* and *y* end at the same time

Figure 4.7: Explanation of the *hasIntervalBegin*, and *hasIntervalEnd* predicates.

The linking is done by finding all the TF frames that have a time stamp between the start and the end of each action, this links the robot poses and frame coordinates (which is non-symbolic) with all the symbolic data found in the triples and can be used together for adding context to the non-symbolic numeric data. In addition, the data are linked using the NEEM_ID which is a unique id that is given to each NEEM, this makes it easier to separate the NEEMS and not confuse data from different NEEMs.

4.1.6 Database as a Server

MariaDB offers the ability to add and manage users that can access the database, also the privileges of each user can be adjusted. This is easily done by having an actual table of users (shown in Figure 4.8) and their data which can be modified using SQL commands as any other table in the database, but of course, only the root user or the owner can have such a privilege to modify the table of users.



	GRANTEE	TABLE_CATALOG	PRIVILEGE_TYPE	IS_GRANTABLE
33	'mysql'@'localhost'	def	CONNECTION ADMIN	YES
34	'mysql'@'localhost'	def	READ_ONLY ADMIN	YES
35	'mysql'@'localhost'	def	REPLICATION SLAVE ADMIN	YES
36	'mysql'@'localhost'	def	REPLICATION MASTER ADMIN	YES
37	'mysql'@'localhost'	def	BINLOG ADMIN	YES
38	'mysql'@'localhost'	def	BINLOG REPLAY	YES
39	'mysql'@'localhost'	def	SLAVE MONITOR	YES
40	'newuser'@'localhost'	def	SELECT	YES

Figure 4.8: MariaDB table of users and their allowed privileges.

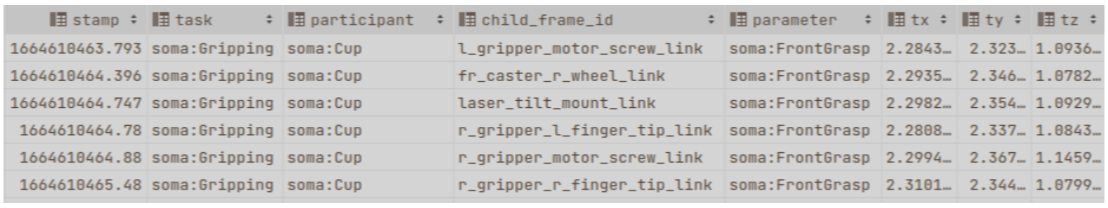
The table of users has an IP address column that specifies the address of each user, this allows for remote connections on the local network and even through ssh from non-local connections. This is especially important when it comes to sharing the database with multiple institutions across the world which is one of the goals of the NEEMs.

4.1.7 Common Table Views

Most use cases will be confined to a small number of tables from the database, joining these tables as views for the different use cases will make it much more easy to use. A table view is a table that is represented or stored as a query, this query can be used by any user directly without the need to write it, also it is possible for the user to modify this query to add other tables from the database or remove some of the joined tables in the default view. These views also serve as an example of how to use the database and how to join the tables of the predicates and the TF tables.

The currently available views

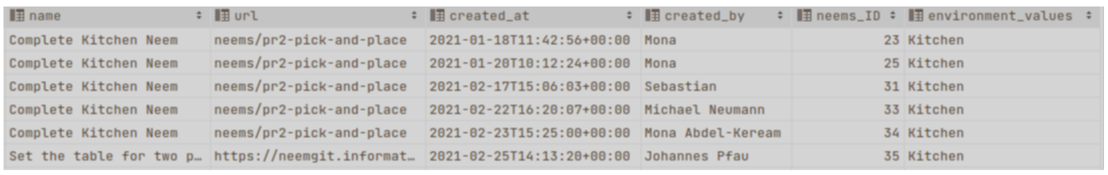
Actions and TF, this joins the symbolic robot actions with the TF data that occurred during these actions, this gives information on the poses of the robot arms and the robot location during the execution of the action. Also, this includes the time interval of each action and the objects that were acted on by the robot/agent while performing the action. The resulting table can be seen in Figure 4.9.



stamp	task	participant	child_frame_id	parameter	tx	ty	tz
1664610463.793	soma:Gripping	soma:Cup	l_gripper_motor_screw_link	soma:FrontGrasp	2.2843…	2.323…	1.0936…
1664610464.396	soma:Gripping	soma:Cup	fr_caster_r_wheel_link	soma:FrontGrasp	2.2935…	2.346…	1.0782…
1664610464.747	soma:Gripping	soma:Cup	laser_tilt_mount_link	soma:FrontGrasp	2.2982…	2.354…	1.0929…
1664610464.78	soma:Gripping	soma:Cup	r_gripper_l_finger_tip_link	soma:FrontGrasp	2.2808…	2.337…	1.0843…
1664610464.88	soma:Gripping	soma:Cup	r_gripper_motor_screw_link	soma:FrontGrasp	2.2994…	2.367…	1.1459…
1664610465.48	soma:Gripping	soma:Cup	r_gripper_r_finger_tip_link	soma:FrontGrasp	2.3101…	2.344…	1.0799…

Figure 4.9: Resulting table from the **Action and TF** common view query

NEEMs for a certain environment, this chooses the NEEMs that occurred in a certain environment, examples for environments are *Kitchen*, and *Retail Store*. This can be done by conditioning the environment column in the NEEMs metadata table to be equal to a certain environment, once the NEEMs are filtered the user can choose only the relevant data from all other tables that occurred in the chosen NEEMs. The resulting table can be seen in Figure 4.10.

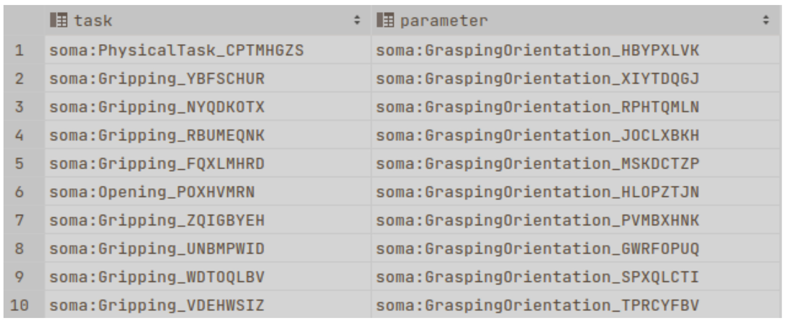


name	url	created_at	created_by	neems_ID	environment_values
Complete Kitchen Neem	neems/pr2-pick-and-place	2021-01-18T11:42:56+00:00	Mona	23	Kitchen
Complete Kitchen Neem	neems/pr2-pick-and-place	2021-01-20T10:12:24+00:00	Mona	25	Kitchen
Complete Kitchen Neem	neems/pr2-pick-and-place	2021-02-17T15:06:03+00:00	Sebastian	31	Kitchen
Complete Kitchen Neem	neems/pr2-pick-and-place	2021-02-22T16:20:07+00:00	Michael Neumann	33	Kitchen
Complete Kitchen Neem	neems/pr2-pick-and-place	2021-02-23T15:25:00+00:00	Mona Abdel-Keream	34	Kitchen
Set the table for two p…	https://neemgit.informat…	2021-02-25T14:13:20+00:00	Johannes Pfau	35	Kitchen

Figure 4.10: Resulting table from the **NEEMs for a certain environment** common view query

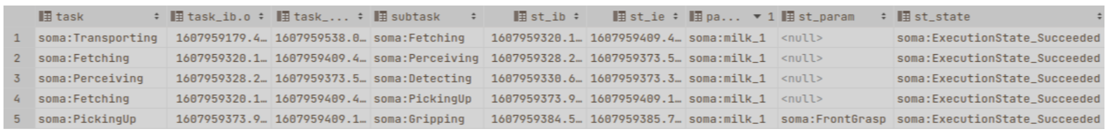
Task parameters, this shows the relevant parameters, these parameters are things like the grasping orientation and are software specific, in the NEEMs they are related to the CRAM action parameters. The resulting table can be seen in Figure 4.11.

Tasks and sub-tasks with parameters, this shows both the tasks and their sub-tasks with the parameters of each one and the objects that were acted on by the robot/agent during these tasks and sub-tasks. The resulting table can be seen in Figure 4.12.



	task	parameter
1	soma:PhysicalTask_CPTMHGZS	soma:GraspingOrientation_HBYPXLVK
2	soma:Gripping_YBFSCHUR	soma:GraspingOrientation_XIYTDQGJ
3	soma:Gripping_NYQDKOTX	soma:GraspingOrientation_RPHTQMLN
4	soma:Gripping_RBUMEQNK	soma:GraspingOrientation_JOCLXBKH
5	soma:Gripping_FQXLMHRD	soma:GraspingOrientation_MSKDCTZP
6	soma:Opening_POXHVMRN	soma:GraspingOrientation_HLOPZTJN
7	soma:Gripping_ZQIGBYEH	soma:GraspingOrientation_PVMBXHNK
8	soma:Gripping_UNBMPWID	soma:GraspingOrientation_GWRFOPUQ
9	soma:Gripping_WDTOQLBV	soma:GraspingOrientation_SPXQLCTI
10	soma:Gripping_VDEHWSIZ	soma:GraspingOrientation_TPRCYFBV

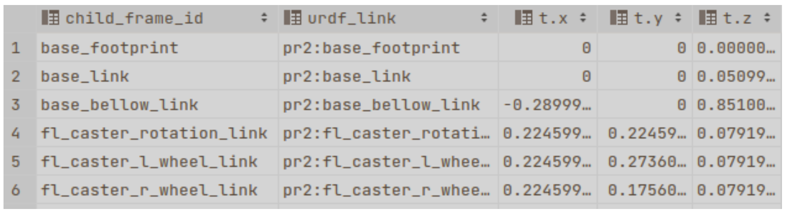
Figure 4.11: Resulting table from the **Task parameters** common view query



	task	task_ib.o	task_...	subtask	st_ib	st_ie	pa... 1	st_param	st_state
1	soma:Transporting	1607959179.4…	1607959538.0…	soma:Fetching	1607959320.1…	1607959409.4…	soma:milk_1	<null>	soma:ExecutionState_Succeeded
2	soma:Fetching	1607959320.1…	1607959409.4…	soma:Perceiving	1607959328.2…	1607959373.5…	soma:milk_1	<null>	soma:ExecutionState_Succeeded
3	soma:Perceiving	1607959328.2…	1607959373.5…	soma:Detecting	1607959330.6…	1607959373.3…	soma:milk_1	<null>	soma:ExecutionState_Succeeded
4	soma:Fetching	1607959320.1…	1607959409.4…	soma:PickingUp	1607959373.9…	1607959409.1…	soma:milk_1	<null>	soma:ExecutionState_Succeeded
5	soma:PickingUp	1607959373.9…	1607959409.1…	soma:Gripping	1607959384.5…	1607959385.7…	soma:milk_1	soma:FrontGrasp	soma:ExecutionState_Succeeded

Figure 4.12: Resulting table from the **Tasks and sub-tasks with parameters** common view query

TF data of certain robot links, this uses the symbolic knowledge of the robot and its links which is stored in the predicate tables with the TF data tables which allows filtering the TF data for certain relevant information related to the symbolic knowledge like the robot name or the link names without the need to know the frame names. The resulting table can be seen in Figure 4.13.



	child_frame_id	urdf_link	t.x	t.y	t.z
1	base_footprint	pr2:base_footprint	0	0	0.00000…
2	base_link	pr2:base_link	0	0	0.05099…
3	base_bellow_link	pr2:base_bellow_link	-0.28999…	0	0.85100…
4	fl_caster_rotation_link	pr2:fl_caster_rotati…	0.224599…	0.22459…	0.07919…
5	fl_caster_l_wheel_link	pr2:fl_caster_l_whee…	0.224599…	0.27360…	0.07919…
6	fl_caster_r_wheel_link	pr2:fl_caster_r_whee…	0.224599…	0.17560…	0.07919…

Figure 4.13: Resulting table from the **TF data of certain robot links** common view query.

4.2 Querying the Database

The common table views are very good examples that shows most use cases of the database and how to query it for all types of information that include both the symbolic (predicate tables) and the non-symbolic data (TF tables).

4.2.1 Filtering NEEMs

Let us start with the simplest one, which finds the NEEMs that have occurred in the *kitchen* environment, the first thing to do is to look at how the **neems** table is related to the **neems_environment_index** table, this can be easily done by looking at the visualization of the tables in Figure 4.14.

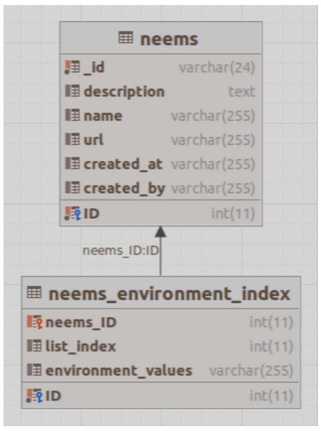


Figure 4.14: The neems and environments tables and their link in the database, they are linked using the *neems_ID* foreign key which links to the ID of the neem, the relationship is a many to many relationships where different neems can have same environment or even multiple environments

The environments are stored in a separate table because the environments are limited and are used by multiple NEEMs, also some NEEMs can have no environment specified, and since it is better to reduce the number of *NULL* values in an SQL table and reduce the redundancy of the data it is better to separate that column into another table. So the query that would select the NEEMs that occurred in the *kitchen* environment can be written as follows:

```
SELECT * FROM neems
INNER JOIN (
            SELECT * FROM neems_environment_index
            WHERE environment_values="kitchen"
            )
            AS neem_env
ON neems.ID = neem_env.neems_ID;
```

Here one first selects all columns in the **neems** table, then joins it with the **neems_ environmet_index** table conditioned on that the column *environment_values* have the value *"kitchen"*. The two tables are joined by matching the *ID* column in the **neems** table with the *neems_ID* column in the **neems_environment_index** table. This results in the table shown in Figure 4.10. This is just one example of how to choose certain NEEMs, the NEEMs can be filtered by their name, description, creator, activity name, and others.

4.2.2 Filtering TF Data

Lets say we are only interested in a specific frame on the robot, this can be easily done by selecting all rows where the value of the *child_frame_id* in the **tf** table equals to the frame name, for example if we are interested in the *"r_gripper_palm_link"* frame, the query would be as follows:

```
SELECT * FROM (
                SELECT tf.* From tf
                WHERE tf.child_frame_id = "r_gripper_palm_link"
                ) AS tf
LIMIT 300000;
```

Since the TF data is large it is better to limit the results unless you want all the data.

4.2.3 Joining the TF Data Tables

The TF data structure is similar to the one shown in Figure 4.2, but one can also see it from the tables relation diagram shown in Figure 4.15.

The TF data tables all have *ONE to ONE* relationship, this allows having the primary key of each table as the foreign key to the parent table as well. Thus the joining is done on the primary key which is always the *ID* column in the table. Thus the query that will join all the TF data tables together into one table would be as follows:

```
SELECT t.*, r.*, th.*, tf.child_frame_id FROM tf
INNER JOIN tf_header AS th
ON tf.ID = th.ID
INNER JOIN tf_transform AS tft
ON tft.ID = th.ID
INNER JOIN transform_translation AS t
ON t.ID = tft.ID
INNER JOIN transform_rotation AS r
ON r.ID = tft.ID;
```

This results in the table shown Figure 4.16.

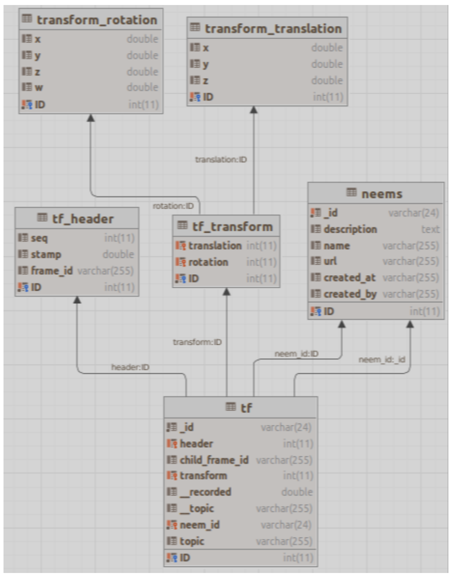
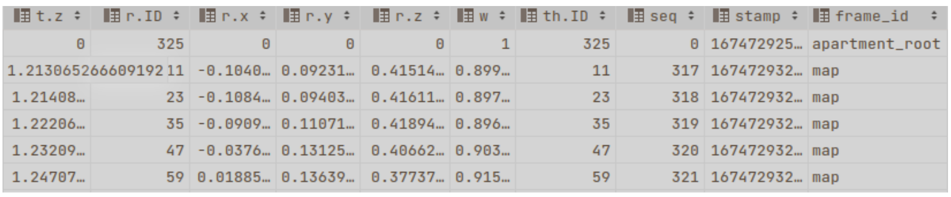


Figure 4.15: The tf tables in the database have the same structure as the tf ROS message so it is easy to convert between them and easy to understand by users familiar with the ROS message



t.z	r.ID	r.x	r.y	r.z	w	th.ID	seq	stamp	frame_id
0	325	0	0	0	1	325	0	167472925…	apartment_root
1.213065266609192	11	-0.1040…	0.09231…	0.41514…	0.899…	11	317	167472932…	map
1.21408…	23	-0.1084…	0.09403…	0.41611…	0.897…	23	318	167472932…	map
1.22206…	35	-0.0909…	0.11071…	0.41894…	0.896…	35	319	167472932…	map
1.23209…	47	-0.0376…	0.13125…	0.40662…	0.903…	47	320	167472932…	map
1.24707…	59	0.01885…	0.13639…	0.37737…	0.915…	59	321	167472932…	map

Figure 4.16: The resulting table from joining the tf tables together

4.2.4 Filtering TF Data Using Predicate Tables Information

A more interesting use case is one that combines the symbolic data in the predicate tables and the non-symbolic data in the TF data tables. Lets say we are interested in the TF data for frames that belong to a certain robot, and naturally the NEEMs have TF data from many robots or at least that is the intention, but sometimes the frame names of the robots are not uniquely identified by a certain naming pattern, here comes one of the benefits of the symbolic data.

The robot links are defined symbolically in the triples and thus in the predicate tables, these link names are prefixed with the robot name. One of the most important predicates is the *rdf:type* which is represented in the **rdf__type** table in the database, this gives the type/class of any entity in the ontology, so now we can look for entities that have type *"urdf:link"* and filter them so that we only select the links that are prefixed with *"pr2:"* which is the prefix for the *PR2* robot, and finally once we have the *PR2* link names we can filter the TF data by selecting the TF data entries that have *child_frame_id* equal to one of these link names, thus the query would be as follows:

```
SELECT tf.* FROM tf
INNER JOIN (
            SELECT DISTINCT rdft.s, rdft.ID
            FROM rdf_type AS rdft
            WHERE o = 'urdf:link' AND s REGEXP '^pr2:'
            ) AS rdft
ON tf.child_frame_id = REPLACE(rdft.s, 'pr2:','')
LIMIT 300000;
```

The results for this query can be seen in Figure 4.13.

4.2.5 Joining Related Predicate Tables

Predicate tables have three main columns, the *neem_id* column which relates the row the NEEM that it belongs to, the subject column which is the input to the predicate, and the object column which is the output of the predicate. These columns have names related to class type that is used in the column, but if no class type is defined for it in the ontology then the column names would be *'s'*, and *'o'* for the subject and object columns respectively.

An easy example is the one represented in the *Task parameters* table view, this one used two predicate tables, the first is the *dul_hasParameter* predicate table and the second is the *rdf_type* predicate table. The *dul_hasParameter* takes as input a *dul_Concept* instance and gives a *dul_Parameter* as output, but we also need to know what type is this parameter because the parameter value is just a unique name that identifies the parameter, so the name alone is useless. To find the type we use *rdf_type* predicate which tells us what is this parameter exactly when we give the parameter name as input. The query would be as follows:

```
SELECT rdft1.s AS task, rdft.s AS parameter
FROM dul_hasParameter as hpara
INNER JOIN rdf_type as rdft
ON rdft.s = hpara.dul_Parameter_o
INNER JOIN rdf_type as rdft1
ON rdft1.s = hpara.dul_Concept_s;
```

The resulting table is shown in Figure 4.11.

A more interesting example is how actions relate to tasks and how each action/task has a time interval and each time interval has a begin and an end time, these relations are represented by four predicates, the first one is *dul_executesTask* which takes an action instance and outputs the task that this action executes, the second predicate is *dul_hasTimeInterval* which takes as input a *dul_Event* and outputs a *dul_TimeInterval* for that event, the third and fourth predicates are *soma_hasIntervalBegin* and *soma_has-–IntervalEnd* which take as input a *dul_TimeInterval* and outputs a float representing the beginning and end time respectively, the query to get this information in one table is easily done as follows:

```
SELECT task.dul_Action_s AS action,
       task.dul_Task_o AS task,
       task_ib.o AS begin,
       task_ie.o AS end
FROM dul_executesTask AS task
INNER JOIN dul_hasTimeInterval AS task_ti
ON  task.dul_Action_s = task_ti.dul_Event_s
AND task.neem_id = task_ti.neem_id
INNER JOIN soma_hasIntervalBegin AS task_ib
ON  task_ti.dul_TimeInterval_o = task_ib.dul_TimeInterval_s
AND task_ti.neem_id = task_ib.neem_id
INNER JOIN soma_hasIntervalEnd AS task_ie
ON  task_ti.dul_TimeInterval_o = task_ie.dul_TimeInterval_s
AND task_ti.neem_id = task_ie.neem_id
Order by task_ib.o;
```

It is crucial to join on *neem_id* as well in each predicate to make sure data from two different neems do not mix or join together, at the end we order by the begin time so that the table shows the tasks in order of starting time, this increases the query time significantly so it should be used with care. The resulting table is shown in Figure **??**.

4.3 Machine Learning on NEEMs

4.3.1 Joint Probability Trees

The Joint Probability Trees (JPTs) model is a statistical machine-learning model that finds the most important independent variables in the data and then finds the joint probability of these variables in the form of trees. This can be considered an efficient database that summarizes the tables into trees that can be queried faster and can also generalize to unseen data that can be found through interpolating the existing data.

The way JPTs are constructed makes them a transparent and explainable machine-learning model, they can predict both symbolic and non-symbolic data, and they can also be fine-tuned to base their decisions in the tree on specific attributes in the data. Using JPTs, one can fit models on different aspects of the NEEMs, some of which are:

- Predict the task tree or the next task in the plan given the current and previous task or tasks and some other context information.
- Predict the success or failure of a task or a plan.
- Recover from preciously encountered failures.
- Predict non-symbolic data like the best position for the robot to stand in so that the probability of success of picking up or placing an object is high.

Another important question that these JPTs will answer is *"What in the context is important for each task?"*. This will give an insight into what to concentrate on and help in making more efficient models that focus on these important aspects of the context.

4.3.2 The Pipeline

The machine learning pipeline from NEEMs to working models is done through the following steps:

1. Figure out what data you need from the NEEMs for your task, which should include both the input and the output of your machine-learning model.
2. Check common views (section 4.1.7) and if possible just modify what you need, else write your own query, and check section 4.2 for how to do that.
3. The recommended way is to write your query in an SQL file and execute it from within your program, if you are using Python you can do that with the *SQLAlchemy*[7] or with *Pandas*[8] which will give you the resulting table from your query in a Pandas DataFrame which is very convenient to manipulate and use for tabular data.
4. Fit your machine learning model on the data.
5. Test the model with some test cases to measure its behavior and accuracy.
6. Iterate over all the steps again until the model reaches the required accuracy and behavior, sometimes the data is missing important information so iterating over the first step to figure out the required data is beneficial and then modify the query as required and fit the model again on the new data.

4.4 Test Cases

To illustrate the pipeline, it's better to describe it using examples of actual test cases.

[7]https://www.sqlalchemy.org/
[8]https://pandas.pydata.org/

4.4.1 Case 1: Predict Next Task

Problem Definition. The first step is to define the problem that the model is going to solve. To predict the next task, one has to first answer the question *"What does the robot need to know for it to be able to predict the next task in a plan correctly?"*. As easy as it may seem, it is not straightforward to get the correct answer. I started with the simplest solution possible, in which the robot only needs to know the environment and the current task as shown in Figure 4.17.

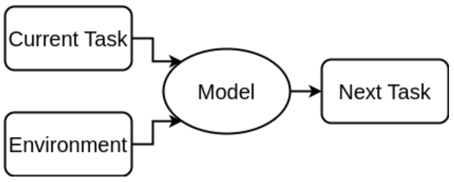


Figure 4.17: Predict Next Task from Current Task and Environment

Figure Out The Required Data. The second step is to figure out what data is needed for our model. A good start is to look at the tasks in the database and see how they are structured, Figure 4.18 shows the tasks executed for a NEEM in the kitchen.

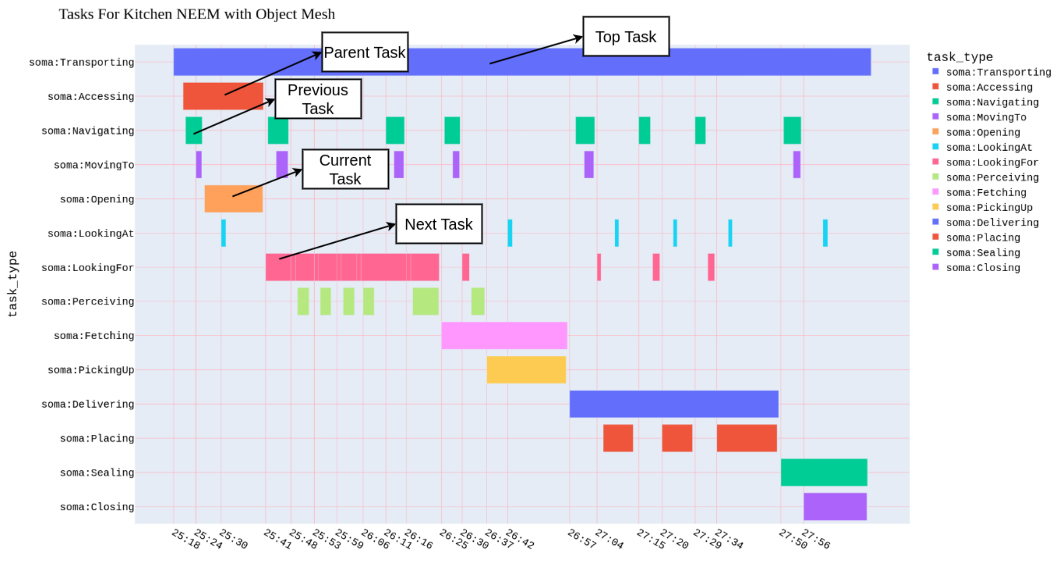


Figure 4.18: The task hierarchy of a NEEM in the kitchen. This shows the 5 task hierarchy types in which *Openning* is the *Current Task*, *Navigating* is its *Previous Task*, *LookingFor* is its *Next Task*, *Accessing* is its *Parent Task*, and *Transporting* is the *Top Task* of the whole task tree.

In this case, the first proposed model takes two inputs the current or previous task and the environment in which the robot is performing this task. Since the goal is to predict all the sequence or at least part of the sequence of tasks of a certain plan, the sequence of tasks needs to be known, in this case only every two consecutive tasks are needed because the model takes the previous/current task and outputs the next task. This means each row in the data table should contain the previous/current task, the next task, and the environment type.

Query Writing. This is not easily done using SQL queries, what can be done is similar to the **Tasks and sub-tasks with parameters** common view, where we have a column for tasks and a column for task types. But the subtasks and parameters are not needed in this case and can be removed. The task start and end times are needed for sequence order information. In addition, the execution state of each task is important since task success or failure affects the next task to be executed. Thus the table will have a total of six columns. They can be selected at the top of our query like this:

```
SELECT task.dul_Task_o AS task,
       taskt.o AS task_type,
       task_ib.o AS task_start,
       task_ie.o AS task_end,
       task_es.soma_ExecutionStateRegion_o AS task_state,
       ne.environment_values AS environment
FROM dul_executesTask AS task
```

It is important to filter the data from any unnecessary information like unknown task types which are labeled as *soma:PhysicalTask* by default, and unnecessary extra information like *owl:NamedIndividual* where every instance of a class in the ontology has this type anyway.

```
INNER JOIN rdf_type AS taskt
    ON task.dul_Task_o = taskt.s
    AND taskt.o != 'owl:NamedIndividual'
    AND task.neem_id = taskt.neem_id
    AND taskt.o not Regexp '^soma:Phy'
```

The task start and end times can be added by joining the time interval and then adding the interval begin and the interval end of each time interval.

```
INNER JOIN dul_hasTimeInterval AS task_ti
    ON task.dul_Action_s = task_ti.dul_Event_s
    AND task.neem_id = task_ti.neem_id
INNER JOIN soma_hasIntervalBegin AS task_ib
    ON task_ti.dul_TimeInterval_o = task_ib.dul_TimeInterval_s
    AND task_ti.neem_id = task_ib.neem_id
INNER JOIN soma_hasIntervalEnd AS task_ie
```

```
        ON task_ti.dul_TimeInterval_o = task_ie.dul_TimeInterval_s
        AND task_ti.neem_id = task_ie.neem_id
```

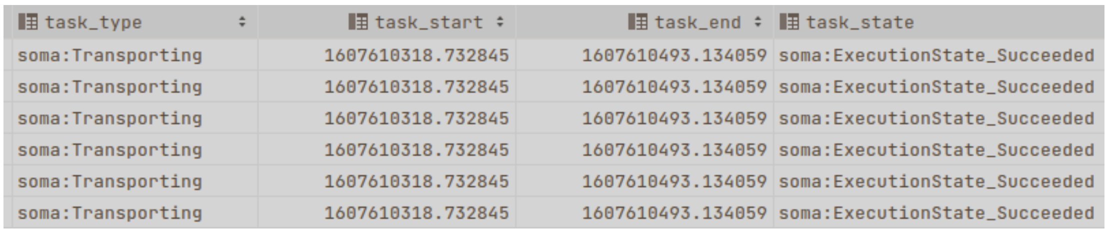
Also, the environment type and the execution state need to be added as well. To add them, a left join on the *neems_id* is used because in some cases where there is no environment or execution state specified but the data is still useful, these lines should be added to the query:

```
    LEFT JOIN soma_hasExecutionState AS task_es
        ON task_es.dul_Action_s = hc.dul_Entity_s
        AND task_es.neem_id = hc.neem_id
    LEFT JOIN neems_environment_index AS ne
        ON ne.neems_ID = task.neem_id
```

Finally, it is beneficial to order the rows according to the interval beginning time to have the correct sequence of tasks.

```
    ORDER BY task_ib.o;
```

The resulting table will be as shown in Figure 4.19. To add the next task column some processing on the data is needed using Python with the help of the Pandas package, the definition of the previous task is very important. It is important to differentiate between a parent task and a previous task. A parent task is a task that starts before or at the same time as a child task. A previous task is the last task that ends before or at the start of the current task. The final table will be as the one shown in Figure 4.20.

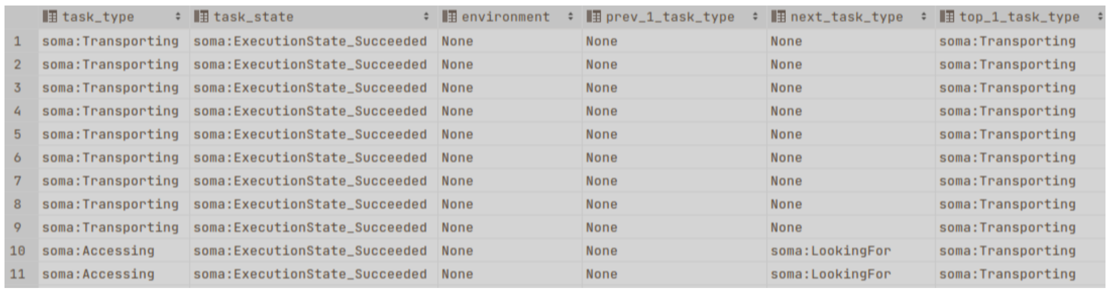


task_type	task_start	task_end	task_state
soma:Transporting	1607610318.732845	1607610493.134059	soma:ExecutionState_Succeeded
soma:Transporting	1607610318.732845	1607610493.134059	soma:ExecutionState_Succeeded
soma:Transporting	1607610318.732845	1607610493.134059	soma:ExecutionState_Succeeded
soma:Transporting	1607610318.732845	1607610493.134059	soma:ExecutionState_Succeeded
soma:Transporting	1607610318.732845	1607610493.134059	soma:ExecutionState_Succeeded
soma:Transporting	1607610318.732845	1607610493.134059	soma:ExecutionState_Succeeded

Figure 4.19: Resulting table from querying on the tasks and their related data like the task state, the task start and end times.

Model Fitting. Once the data is ready as a Python DataFrame, using it with any machine learning pipeline is easy. If using the JPTs it is as easy as the following lines of code:

```python
from jpt.variables import infer_from_dataframe
import jpt.trees
variables = infer_from_dataframe(df, scale_numeric_types=False)
model = jpt.trees.JPT(variables, min_samples_leaf=0.00005)
model.fit(df)
```



	task_type	task_state	environment	prev_1_task_type	next_task_type	top_1_task_type
1	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
2	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
3	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
4	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
5	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
6	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
7	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
8	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
9	soma:Transporting	soma:ExecutionState_Succeeded	None	None	None	soma:Transporting
10	soma:Accessing	soma:ExecutionState_Succeeded	None	None	soma:LookingFor	soma:Transporting
11	soma:Accessing	soma:ExecutionState_Succeeded	None	None	soma:LookingFor	soma:Transporting

Figure 4.20: Table for fitting a model to predict the next task with added columns like the prev task, the next task, the top task, and the parent task which were deduced from the table shown in Figure 4.19.

For more on JPTs and how to use them check the JPT documentation[9].

Test and Model Analisys. This model is used during robot operation, where the robot queries the model to know what is the next task that it should perform given the current task and the environment. A test case could be for example in a *Kitchen* environment and the robot is performing *Opening* of a drawer in the kitchen as the current task. This information is given to the model as input, then the model predicts the next task. Next, the model output is used as the new input, such that the next task is now the current task, and the environment is still the same. This is illustrated in the diagram shown in Figure 4.21.

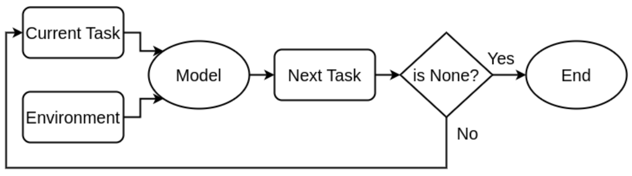


Figure 4.21: Loop over the model to predict the task sequence given an initial current task and an environment type

The problem with this approach is that the input is not enough to be able to accurately predict the output. There are many situations with the same current task and environment and different next tasks. One of the bad cases that occur with this approach is that the model will loop over two or three tasks without ever reaching an end task for the plan as shown in Figure 4.22. In this case, there are many possible scenarios, so the model marginalizes all of them and chooses the most probable scenario (i.e. the scenario that was the most frequent in its memory or database). This could work but is not ideal. This increases the model recall but lowers its precision.

[9]https://joint-probability-trees.readthedocs.io/en/stable/autoapi/jpt/index.html

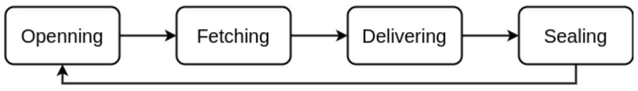


Figure 4.22: The model loops over three tasks without ever reaching an end task due to missing information

Redefine The Required Data and Iterate

The output shown in Figure 4.22 means that more context is needed for the model to accurately predict the next task. At first, I thought the model needs information on older tasks before the current task, but that resulted in a similar behavior of looping over some specific tasks although with a longer sequence this time (i.e. instead of looping over two tasks it loops over three or four tasks). The key to solving this looping problem was to understand that the robot plan is a hierarchical plan (i.e. there are parent tasks and child tasks). From this realization, if one has knowledge of what is the parent-task then, one can predict the next child-task. And this goes all the way to the top of the hierarchy, where in most cases there is an overarching top task to which all other tasks are children. This gives much more information than just the previous task. The model inputs and output is as shown in Figure 4.23.

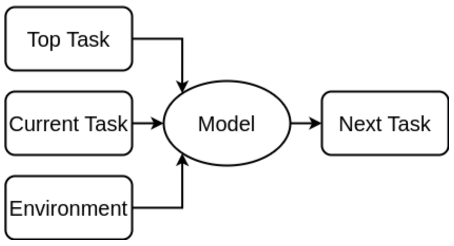


Figure 4.23: Predict Next Task from Top Task, Parent Task, Current Task, and Environment

The output of this model behaves as required and produces the expected task sequence for the plan correctly as shown in Figure 4.24.

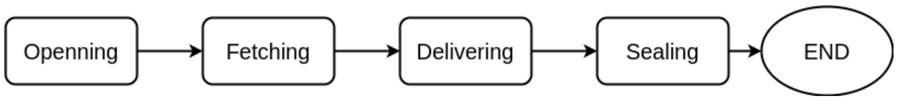


Figure 4.24: Correct task tree that does not loop, this when the model has knowledge of the Top Task, Current Task, and Environment as the model shown in Figure 4.23

But this still does not work for all cases, there are still different cases with different next tasks that have the same *Top Task*, *Current Task*, and *Environment*, for example, the success or failure of the current task affects the next task. If the task failed then a common strategy is to retry the current task again or even do a failure recovery task. In fact, some tasks are more common to fail than to succeed on the first try, thus for these tasks, in that case, the model would still go into the looping behavior.

A solution to that is to let the model know what was the previous tasks to this task, and what is the parent task of each task. The new model would be like the one illustrated in Figure 4.25.

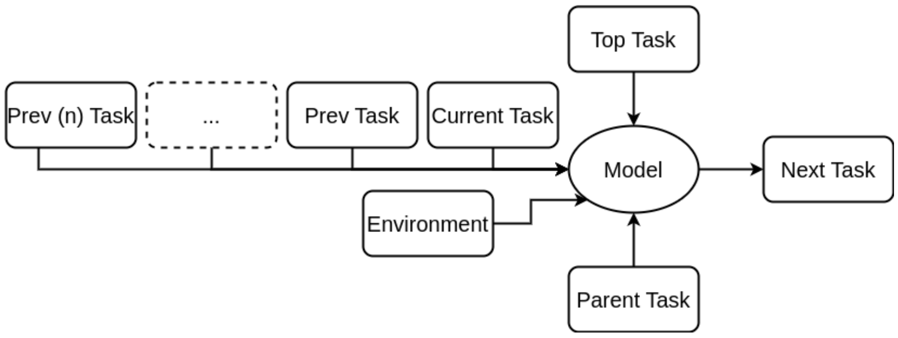


Figure 4.25: Predict Next Task from Top Task, Parent Task, Current Task, Prev Task(s), Task State, and Environment.

4.4.2 Case 2: Failure Recovery

Problem Definition. *Failure Recovery* is very similar to predicting the next task but with the condition, the next task is a success and the current task is a failure. Thus we can use a similar model to the one we used to predict the next task but with some modifications. The most apparent attribute that needs to be added to our model is the state of each task. The state is the resulting state which could be either *succeeded* or *failed* or *none*. The state was used in the **Predict Next Task Model** but only for the current task not for all tasks. Another addition is the cause of failure in case the task state is *Failed*. The new model is the one illustrated in Figure 4.26.

Figure Out The Required Data. The required data is the same as the **Predict Next Task** but with the addition of the information of the *Task State* for all tasks and the *Cause of Failure* in the case of a *Failed* task state.

Write The Query. To add the *Cause of Failure*, the following line needs to be added to the *SELECT* part of the query:

```
t_sat.dul_Description_o AS task_failure_type
```

The *dul:Satisfies* predicate table takes as input a *dul:Situation*. The tasks are a type of *dul:Situation* and this predicate outputs a *dul:Description* for that input situation. The description of the task is *NONE* if the task state is *Succeeded*, but if the task state

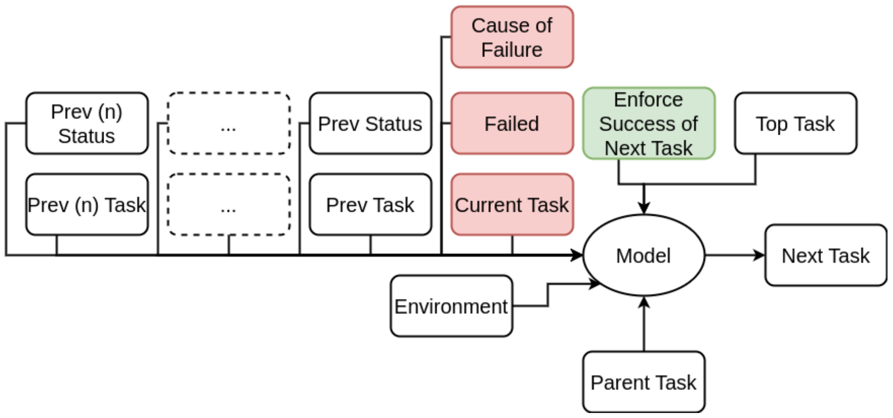
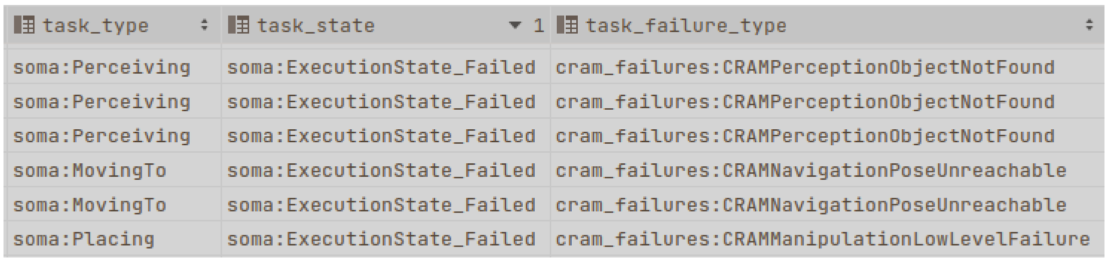


Figure 4.26: **Failure Recovery Model**, the difference between this and the **Predict Next Task Model** shown in Figure 4.25 is the addition of the state of each task, the cause of failure in case the state is *Failed* and the enforcing of the success condition of the next task.

is *Failed* the description is the cause of failure. Then the following lines need to be added at the end of the query before the ordering, this line joins the required data from the *dul_satisfies* table, the join is done using the task instance name which is stored in the *dul_Action_s* column of the *task* table and referencing it to the *dul_Situation_s* column in the *dul_satisfies* table:

```
Left join dul_satisfies AS t_sat
ON task.dul_Action_s = t_sat.dul_Situation_s
```

The resulting table from the new modified query with the failure type is shown in Figure 4.27.



task_type	task_state	task_failure_type
soma:Perceiving	soma:ExecutionState_Failed	cram_failures:CRAMPerceptionObjectNotFound
soma:Perceiving	soma:ExecutionState_Failed	cram_failures:CRAMPerceptionObjectNotFound
soma:Perceiving	soma:ExecutionState_Failed	cram_failures:CRAMPerceptionObjectNotFound
soma:MovingTo	soma:ExecutionState_Failed	cram_failures:CRAMNavigationPoseUnreachable
soma:MovingTo	soma:ExecutionState_Failed	cram_failures:CRAMNavigationPoseUnreachable
soma:Placing	soma:ExecutionState_Failed	cram_failures:CRAMManipulationLowLevelFailure

Figure 4.27: The resulting table from modifying the query of **Predict Next Task** and adding the failure type information to be used for the **Failure Recovery Model**

After retrieving the query result, some modifications are made to it to add the columns for the previous, parent, and next tasks and their attributes. The attributes are the task state and the cause of failure.

After fitting the model and testing with the following inputs:

```
task_state = soma:ExecutionState_Failed
next_task_state = soma:ExecutionState_Succeded
task_type = soma:Placing
top_1_task_type = soma:Transporting
```

The resulting task tree adds a recovery strategy that involves navigating and looking for a new location to perform the placing task as shown in Figure 4.28. Actually, this shows a very important aspect of JPTs as a machine learning model and of the *NEEMs* as a database, because the system has implicitly learned how the automated program that was designed to recover from failure works. This failure recovery mechanism has been designed and programmed by the engineers and researchers working on the *CRAM* cognitive architecture. Through the *NEEMs* the experiences of the robot and the reasoning process have been successfully captured and stored, and through the JPTs the exact failure recovery mechanism and logic have been learned.

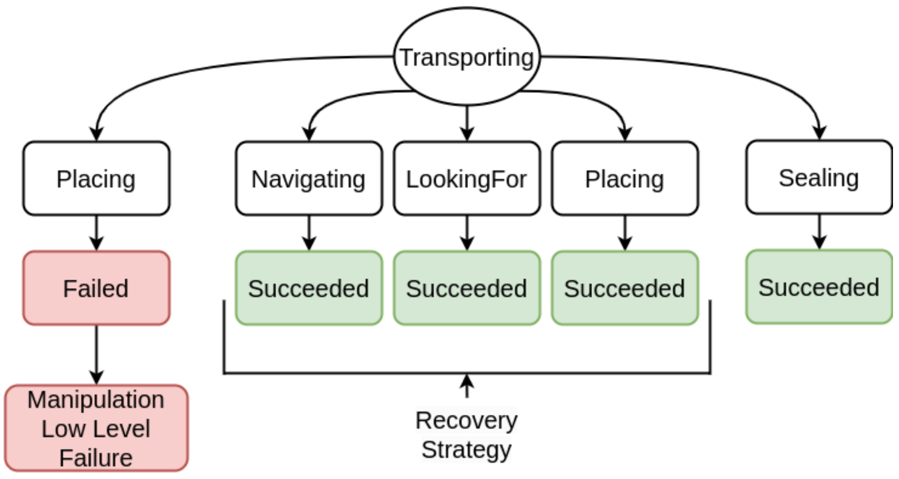


Figure 4.28: The resulting task tree where the initial task is *Placing* and the initial task state is failed, the resulting tree performs a recovery strategy that involves navigating to a new location and looking for a new placing location

4.5 Analysis & Discussion of Results

4.5.1 Database Tables Diagrams

The NEEMs metadata are stored in the *neems table*, and any table that is prefixed with *neem_*, Figure 4.29 shows the metadata tables, some attributes of the metadata are separated into their own tables because they exhibit a many to many or one-to-many or many-to-one relationships with the NEEMs. If they have a one-to-one relationship then they will be in the *neems table*.

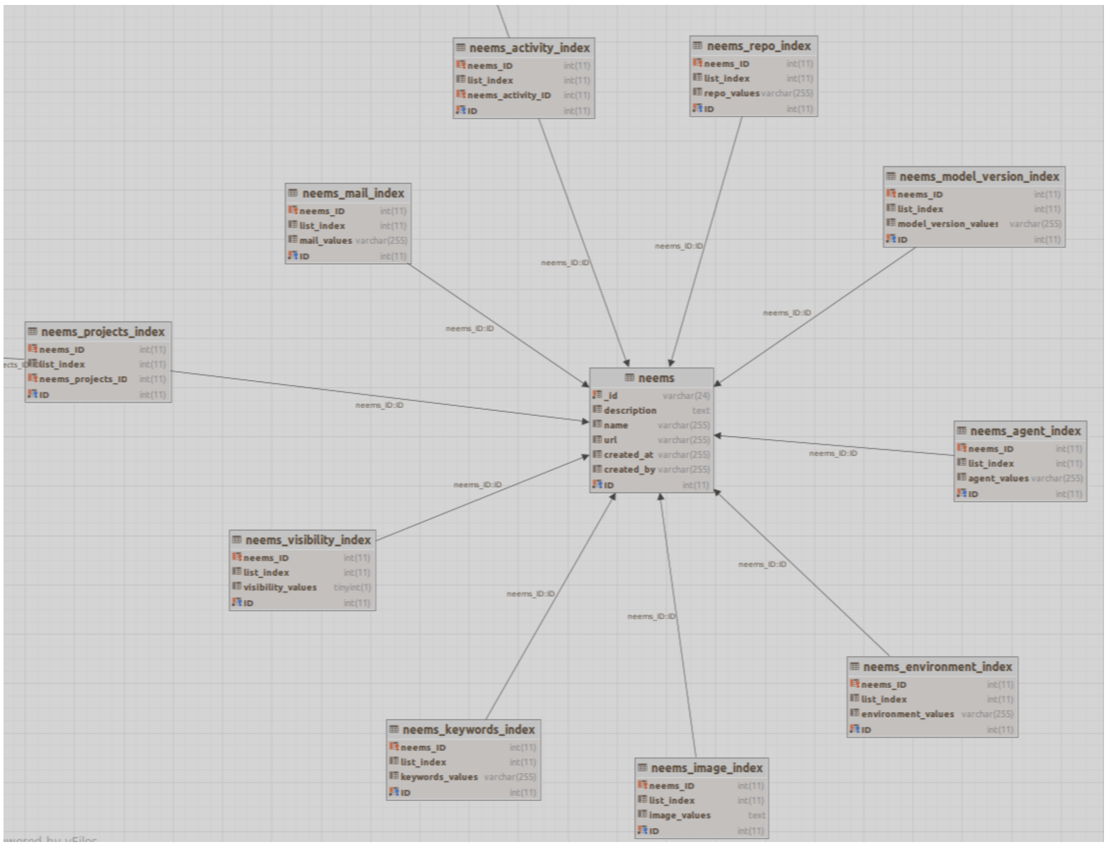


Figure 4.29: The tables for the metadata of the NEEMs.

For the predicate tables, each predicate is a table with subject, object, and neem_id columns, and the table name is the predicate name, prefixed with the ontology prefix, the subject and object columns are post-fixed with '_s' and '_o' respectively. Figure 4.30 shows the predicate tables.

The TF data is linked with the triples through a table called *tf_header_soma_ hasIntervalBegin*, it was constructed by comparing *tf_header* timestamp with *soma_hasIntervalBegin*, and *soma_hasIntervalEnd* predicates. Figure 4.31 shows the linked TFs and predicate tables.

4.5.2 Querying the Database

Table 4.2 shows the analysis of the common queries, focusing on the execution time, rows count, and number of joins in each query. The table shows a clear effect of the number of rows on the execution time compared to the slight effect of the number of joins. The query with the least time has the least number of rows to search which is the *NEEMs Filter* query, while the query with the longest execution time is *Filter TF Data* query which has the highest number of rows to search in.

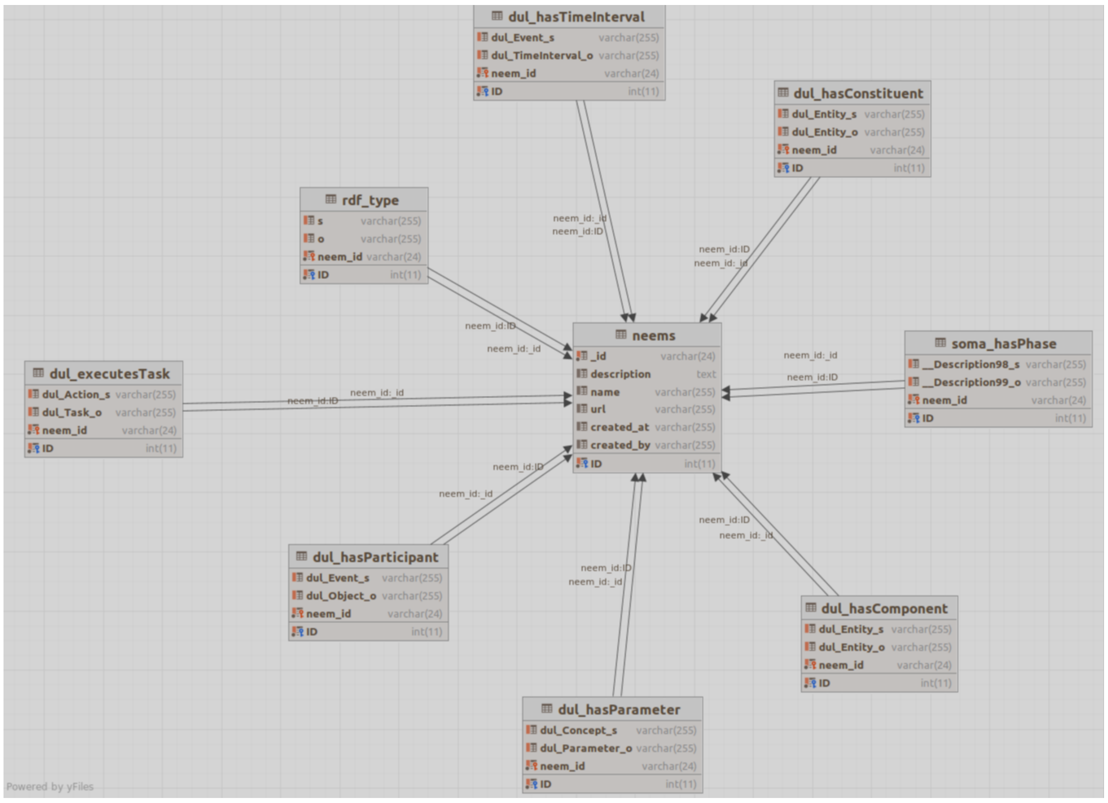


Figure 4.30: The tables for the predicates of in the NEEMs and their relationship with the metadata table of the NEEMs.

4.5.3 Analyses of Joint Probability Trees (JPTs)

The Model Decision Tree

An important aspect of JPTs is that they are transparent models since they are a form of decision trees. This gives us the ability to look at the decision tree and have an understanding of how the model behaves and be able to follow its logic. Figure 4.32 shows part of a large generated decision tree with 133 leaves and 9608 parameters. It is easy to follow the decisions made by the model, from this figure, one can see that if the *Top Task* is *soma:MovingTo* and the *Task Type* is *soma:Placing* then the most probable *Next Task* is *soma:PickingUp* and the second most probable is *soma:Closing*.

The number of leaves is a tuneable parameter for the model, the more the number of leaves the more closely the model follows the database but the more costly it is to perform a prediction in terms of computation and time. The best is to manually go from a low number of leaves and slowly increase it until the model behaves correctly. For example, a tree with 5 leaves is shown in Figure 4.33, this clearly cannot generate a correct task tree, which means the number of leaves needs to be increased. Then one can decrease it again while observing the likelihood of its predictions, using this likelihood

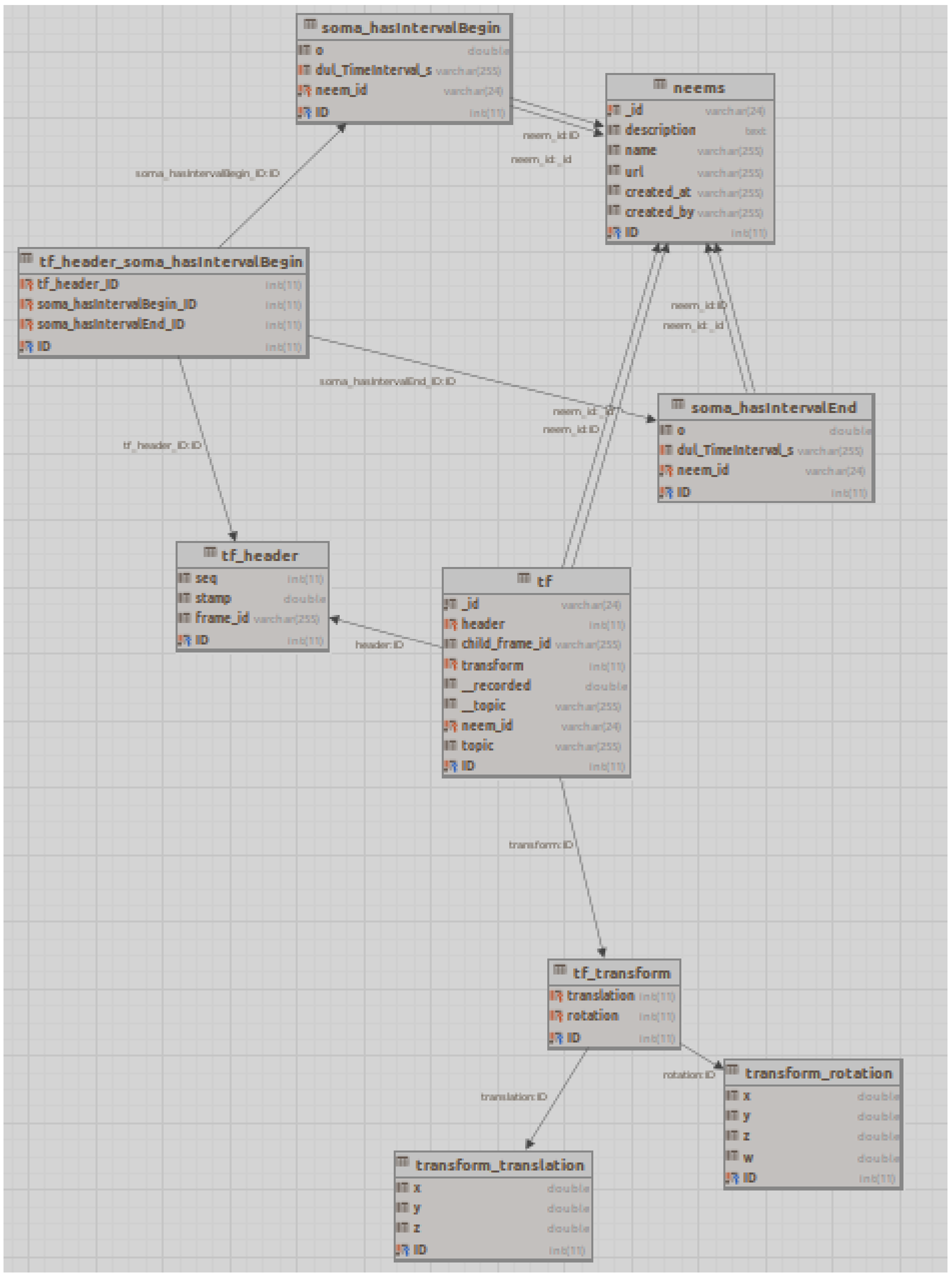


Figure 4.31: Linked TFs and predicate tables using the time stamp of the TF data and the time interval of the events defined by the *soma:hasTimeInterval* predicate.

one can design some limits on the likelihood such that if it is lower than this limit (for example, a likelihood of 0.3) then reject this prediction. This way one can save more resources without having to use a large number of leaves.

Table 4.2: Analysis of the common queries execution time. The time is for retrieving only the first 500 rows.

Query Name	Execution Time (s)	Max Rows Count	No. of Joins
NEEMS Filter	0.02	63	2
Tasks and Parameters	0.03	269	3
Actions and TFs	0.05	269	16
Tasks, Subtasks, and Parameters	0.07	16074	19
Filter TF Data	2.2	18089147	7

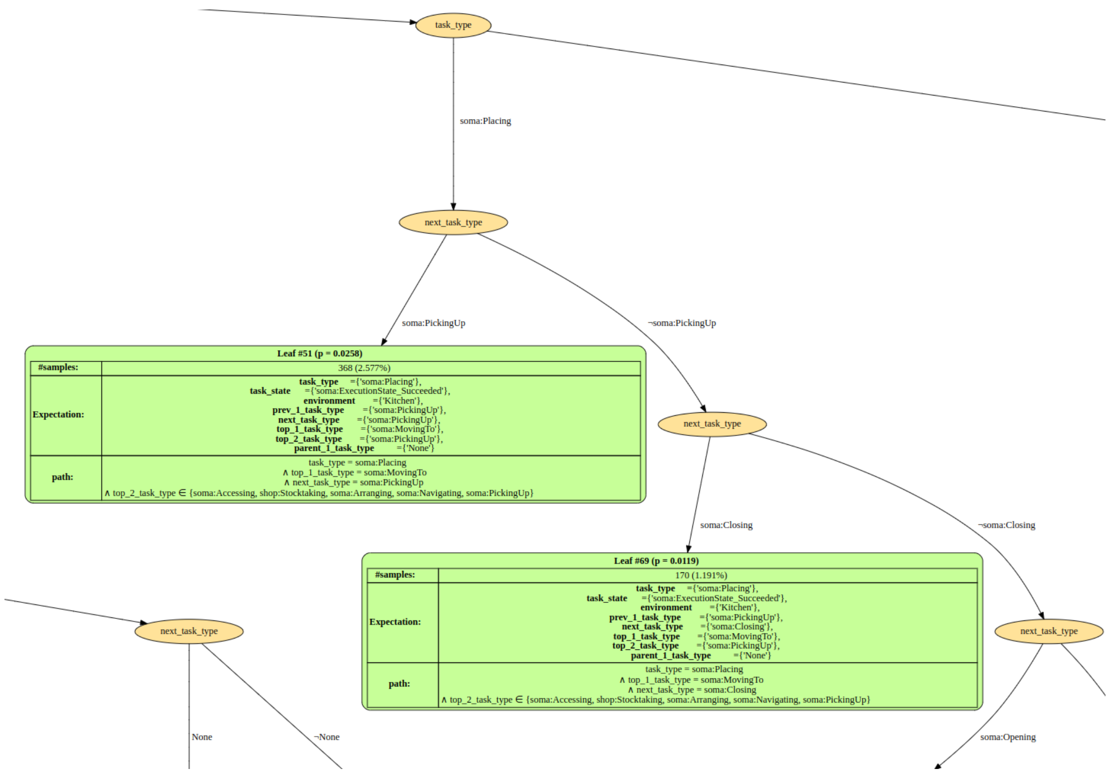

task_type
soma:Placing
next_task_type
soma:PickingUp
¬soma:PickingUp
Leaf #51 (p = 0.0258)
#samples: 368 (2.577%)
Expectation:
task_type ={'soma:Placing'},
task_state ={'soma:ExecutionState_Succeeded'},
environment ={'Kitchen'},
prev_1_task_type ={'soma:PickingUp'},
next_task_type ={'soma:PickingUp'},
top_1_task_type ={'soma:MovingTo'},
top_2_task_type ={'soma:PickingUp'},
parent_1_task_type ={'None'}
path:
task_type = soma:Placing
∧ top_1_task_type = soma:MovingTo
∧ next_task_type = soma:PickingUp
∧ top_2_task_type ∈ {soma:Accessing, shop:Stocktaking, soma:Arranging, soma:Navigating, soma:PickingUp}
next_task_type
soma:Closing
¬soma:Closing
Leaf #69 (p = 0.0119)
#samples: 170 (1.191%)
Expectation:
task_type ={'soma:Placing'},
task_state ={'soma:ExecutionState_Succeeded'},
environment ={'Kitchen'},
prev_1_task_type ={'soma:PickingUp'},
next_task_type ={'soma:Closing'},
top_1_task_type ={'soma:MovingTo'},
top_2_task_type ={'soma:PickingUp'},
parent_1_task_type ={'None'}
path:
task_type = soma:Placing
∧ top_1_task_type = soma:MovingTo
∧ next_task_type = soma:Closing
∧ top_2_task_type ∈ {soma:Accessing, shop:Stocktaking, soma:Arranging, soma:Navigating, soma:PickingUp}
next_task_type
None
¬None
next_task_type
soma:Opening

Figure 4.32: This shows part of the generated decision tree for a model that was fit on the **Predict Next Task** table, where the number of leaves is 133, and the model size (number of parameters) is 9608. Here the figures illustrate an important part of the decision tree that focuses on the *Task Type* and *Next Task Type*. The green rectangles show the most probable explanation (MPE) by showing the most probable values for the model variables for the given path in the decision tree.

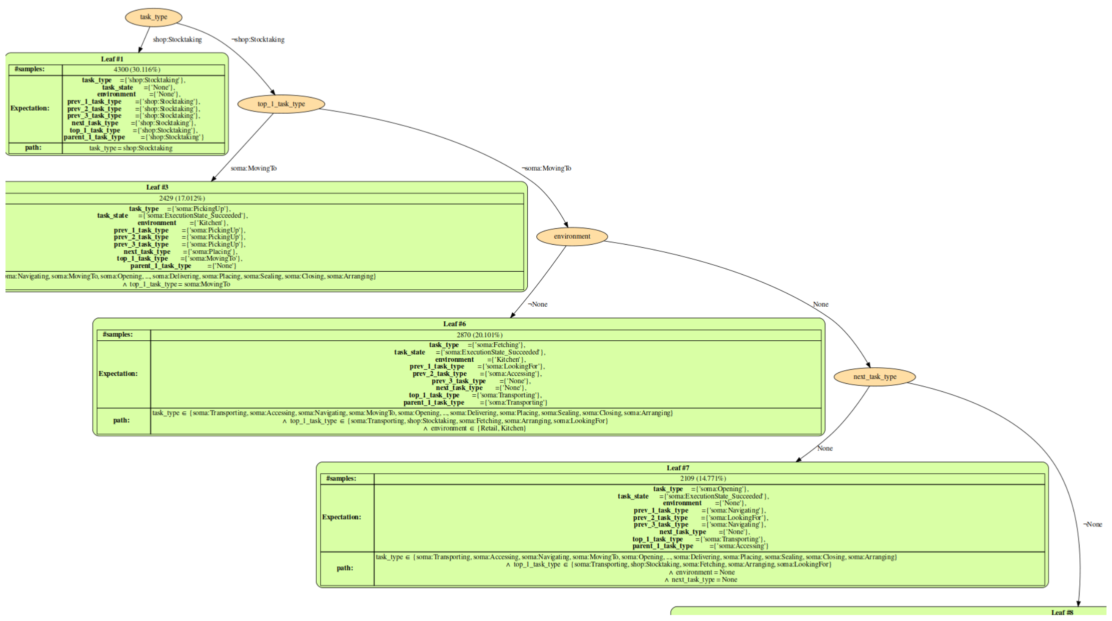

task_type
shop:Stocktaking
¬shop:Stocktaking
Leaf #1
#samples:
4300 (30.116%)
Expectation:
task_type ={'shop:Stocktaking'},
task_state ={'None'},
environment ={'None'},
prev_1_task_type ={'shop:Stocktaking'},
prev_2_task_type ={'shop:Stocktaking'},
prev_3_task_type ={'shop:Stocktaking'},
next_task_type ={'shop:Stocktaking'},
top_1_task_type ={'shop:Stocktaking'},
parent_1_task_type ={'shop:Stocktaking'}
path:
task_type = shop:Stocktaking
top_1_task_type
soma:MovingTo
¬soma:MovingTo
Leaf #3
2429 (17.012%)
task_type ={'soma:PickingUp'},
task_state ={'soma:ExecutionState_Succeeded'},
environment ={'Kitchen'},
prev_1_task_type ={'soma:PickingUp'},
prev_2_task_type ={'soma:PickingUp'},
prev_3_task_type ={'soma:PickingUp'},
next_task_type ={'soma:Placing'},
top_1_task_type ={'soma:MovingTo'},
parent_1_task_type ={'None'}
ma:Navigating, soma:MovingTo, soma:Opening, ..., soma:Delivering, soma:Placing, soma:Sealing, soma:Closing, soma:Arranging}
∧ top_1_task_type = soma:MovingTo
environment
¬None
None
Leaf #6
#samples:
2870 (20.101%)
Expectation:
task_type ={'soma:Fetching'},
task_state ={'soma:ExecutionState_Succeeded'},
environment ={'Kitchen'},
prev_1_task_type ={'soma:LookingFor'},
prev_2_task_type ={'soma:Accessing'},
prev_3_task_type ={'None'},
next_task_type ={'None'},
top_1_task_type ={'soma:Transporting'},
parent_1_task_type ={'soma:Transporting'}
path:
task_type ∈ {soma:Transporting, soma:Accessing, soma:Navigating, soma:MovingTo, soma:Opening, ..., soma:Delivering, soma:Placing, soma:Sealing, soma:Closing, soma:Arranging}
∧ top_1_task_type ∈ {soma:Transporting, shop:Stocktaking, soma:Fetching, soma:Arranging, soma:LookingFor}
∧ environment ∈ {Retail, Kitchen}
next_task_type
None
¬None
Leaf #7
#samples:
2109 (14.771%)
Expectation:
task_type ={'soma:Opening'},
task_state ={'soma:ExecutionState_Succeeded'},
environment ={'None'},
prev_1_task_type ={'soma:Navigating'},
prev_2_task_type ={'soma:LookingFor'},
prev_3_task_type ={'soma:Navigating'},
next_task_type ={'None'},
top_1_task_type ={'soma:Transporting'},
parent_1_task_type ={'soma:Accessing'}
path:
task_type ∈ {soma:Transporting, soma:Accessing, soma:Navigating, soma:MovingTo, soma:Opening, ..., soma:Delivering, soma:Placing, soma:Sealing, soma:Closing, soma:Arranging}
∧ top_1_task_type ∈ {soma:Transporting, shop:Stocktaking, soma:Fetching, soma:Arranging, soma:LookingFor}
∧ environment = None
∧ next_task_type = None
Leaf #8

Figure 4.33: This shows a complete JPT-generated tree with a total of 5 leaves. This shows that 5 leaves in this task are not enough for the model to behave correctly and generate correct task trees similar to the ones in the *NEEMs* database.

Querying The Database vs Querying The Model

If one has access to the database during operation, then it is possible using SQL to directly query the database. The model provides an efficient way to query for information from experience because it summarizes the database and finds the most relevant independent variables and the joint probability of these independent variables. The first test is to check if the model predictions agree with the database. For that the evidence that is given to the model as input is also used to query the database for the table rows that coincide with this evidence or input. One can see from Figure 4.34 that the model output agrees with the most probable output found from the data statistics.

To measure the benefit of using the model over the data, one could compare the query time difference between querying the model and querying directly the database. Table 4.3 shows the time taken to query the database and the model in different use cases.

It is important to mention that the JPTs models have the ability to generalize to new instances that are not in the database, especially in predicting non-symbolic data like robot arm pose, robot base location, etc.

Table 4.3: Comparing the different JPT models with the database in terms of the query time and correctness of the output. The number in the name of the model is the number of leaves, and the mean likelihood is the mean log-likelihood of the model with respect to the database (higher is better), the best one is underlined. The database consists of 14k rows and 8 columns. The correctness of the output is decided by comparing the predicted task tree to the one from the database. These tests are for the **Predict Next Task** test case.

Method	**Query Time (s)**	**Mean Likelihood**	**Correct Output**
Database (14k rows)	3.2	-	-
JPT-12	0.037	-6.29	False
JPT-69	0.15	-4.4	True
JPT-217	0.61	-4.1	True

4.5.4 Predict Next Task Simulation Results

Using PyBullet[10] a physics engine and a simulator with a PR2 robot[11] in an apartment environment with a small kitchen.

The learned model to **Predict Next Task** has been tested in this simulation using *PyCram*[12] as the robot cognitive architecture which does the planning and execution of the robot tasks and reasoning process. The task trees that are in the *NEEMs* database are based on the original *CRAM* (Cognitive Robot Abstract Machine) which is in *Lisp* programming language, while *PyCram* is a migration of the *Lisp* based *CRAM* to *Python*.

Figure 4.35 shows the screenshots of the simulation results (Also available on Youtube[13]) of executing the generated task tree predicted by the learned JPT model to predict the next task iteratively. The robot and environment start with an initial state where the robot just finished opening a drawer in the kitchen table and there is a milk bottle inside that drawer. This information given to the model of the initial state is just the task that was just completed which is *soma:Opening* and the top task which is *soma:Transporting* this information was enough for the model to correctly figure out the remaining tasks of the plan.

To find a good initial state where the picking up of the milk is certainly possible given the robot's kinematics and the environment. The task tree shown in Figure 4.36 is executed and the good initial state is defined by the found successful positions. This is actually how it is done without the machine learning model in the real implementation

[10]https://pybullet.org/wordpress/
[11]https://robotsguide.com/robots/pr2?interactive=1
[12]https://github.com/cram2/pycram
[13]https://youtu.be/LjDn91rC7RY

of the *CRAM* plans. One can see that having a model that can directly get these good poses is very beneficial.

In *PyCRAM*, every motion requires the generation of cost maps, these cost maps are based on constraints of the kinematics of the robot, the obstacles in the environment, and any user-specified constraints. There is cost maps for robot locations as seen in Figure 4.37, and there are arm cost maps as the one seen in Figure 4.38. Using these costmaps the poses are randomly picked, and that is how it works in *CRAM* as well. In the next steps, the machine learning model will be used to decide on the next tasks, while these coast map heuristics will still be used to figure out the poses of the arms and the locations of the robot. Although these could also be learned from the *NEEMs*, this is not the focus of this experiment.

After each step the model is called again with the new information fromt the previous step to predict the next task until the model predicts *None* as the next task indicating that this is most likely the last task of the plan. The generated plan is the one shown in Figure 4.24.

4.5.5 Resulting Software

The resulting software is two repositories and a Jupyter Notebook tutorial[14] on the actual usage of the whole pipeline on a robot in a simulation environment. The first repository called *neems_to_sql*[15] provides the software for converting *NEEMs* from MongoDB to MariaDB. The second repository provides software for executing the machine learning pipeline on the *NEEMs* starting from querying to testing of learned models, the repository is called *neems_research*[16].

4.5.6 Analysing deviations from the Initial Planification

The first task group **Migrate NEEMs from NoSQL to SQL** took much longer than expected due to my lack of experience in the domain of databases. A lot of time has been spent on studying databases and understanding how to use SQL and how to make good, efficient, and maintainable schemas that work well with time and can be regurarly updated.

The consequence of this was that I had to spend an extra month in the development of the *NEEMs* Database and less amount of time in the Machine Learning and Robotics applications on the data in the Database. Nevertheless, the experiments and the simulation provided provide concrete evidence to the efficacy of the resulting Database and the benefit of *NEEMs*.

[14]https://github.com/AbdelrhmanBassiouny/pycram/blob/complex_plans/examples/neems_to_cram_plan_tutorial.ipynb
[15]https://github.com/AbdelrhmanBassiouny/neem_to_sql
[16]https://github.com/AbdelrhmanBassiouny/neems_research

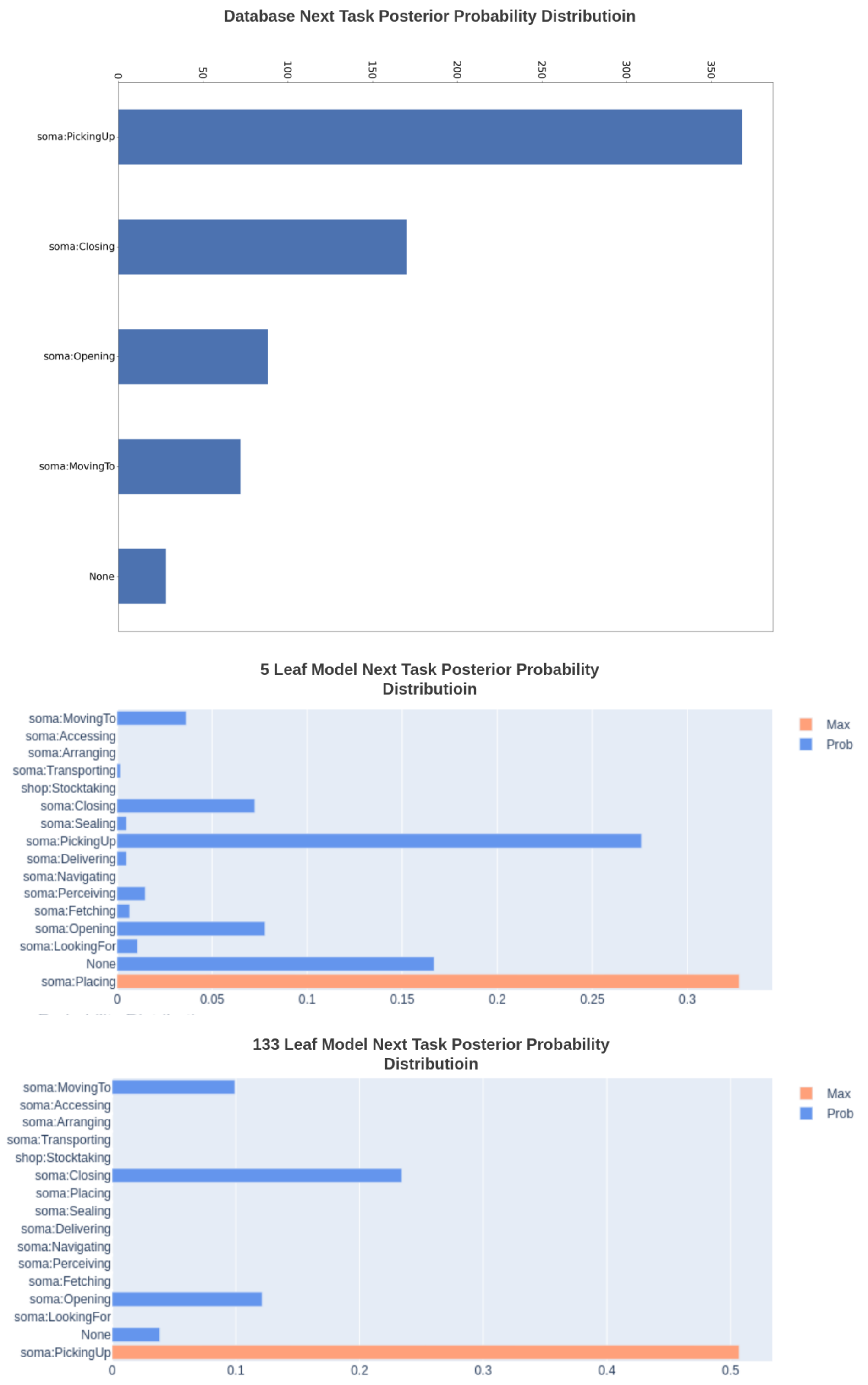


Figure 4.34: The posterior probability distribution of the *Next Task Type* variable from the database (Top), the 5-leaf model (middle), and the 133-leaf model (Bottom). The given evidence for this posterior is that the *Task Type* is *soma:Placing* and the *Top Task Type* is *soma:MovingTo*.

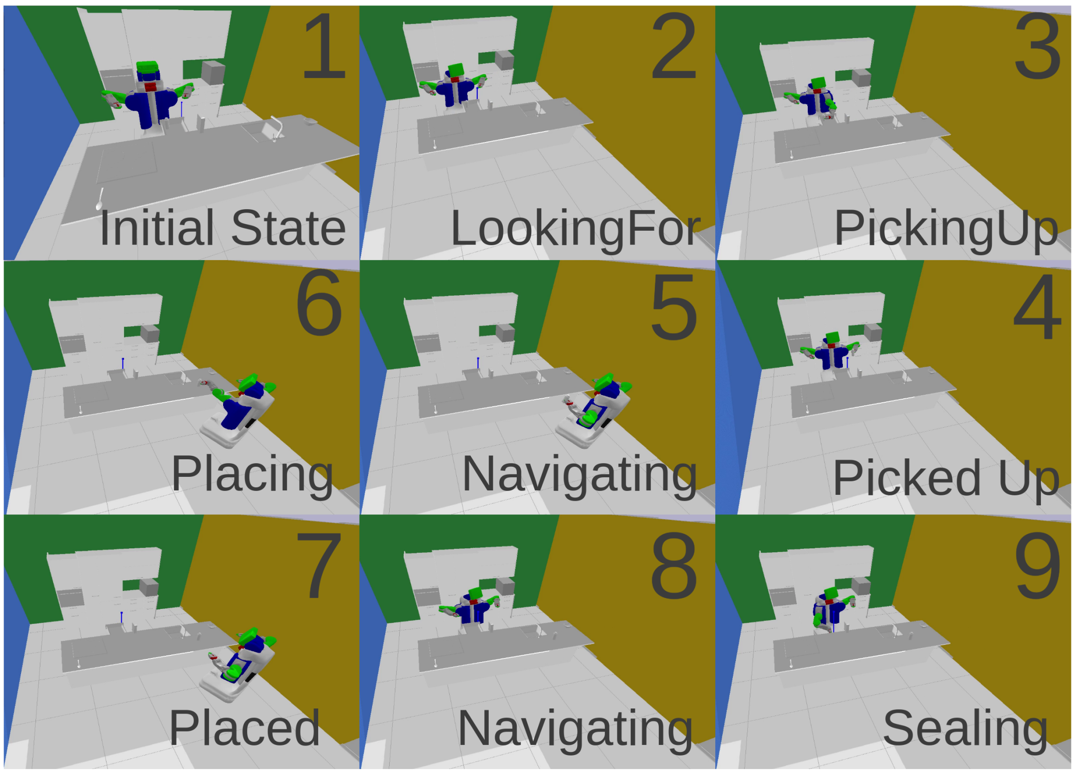


Figure 4.35: This shows the simulation results of the task tree predicted from a JPT model executed in PyBullet using PyCRAM cognitive system to execute the generated plan. The robot and environment start with an initial state where the robot just finished opening a drawer in the kitchen table and there is a milk bottle inside that drawer in step (1).

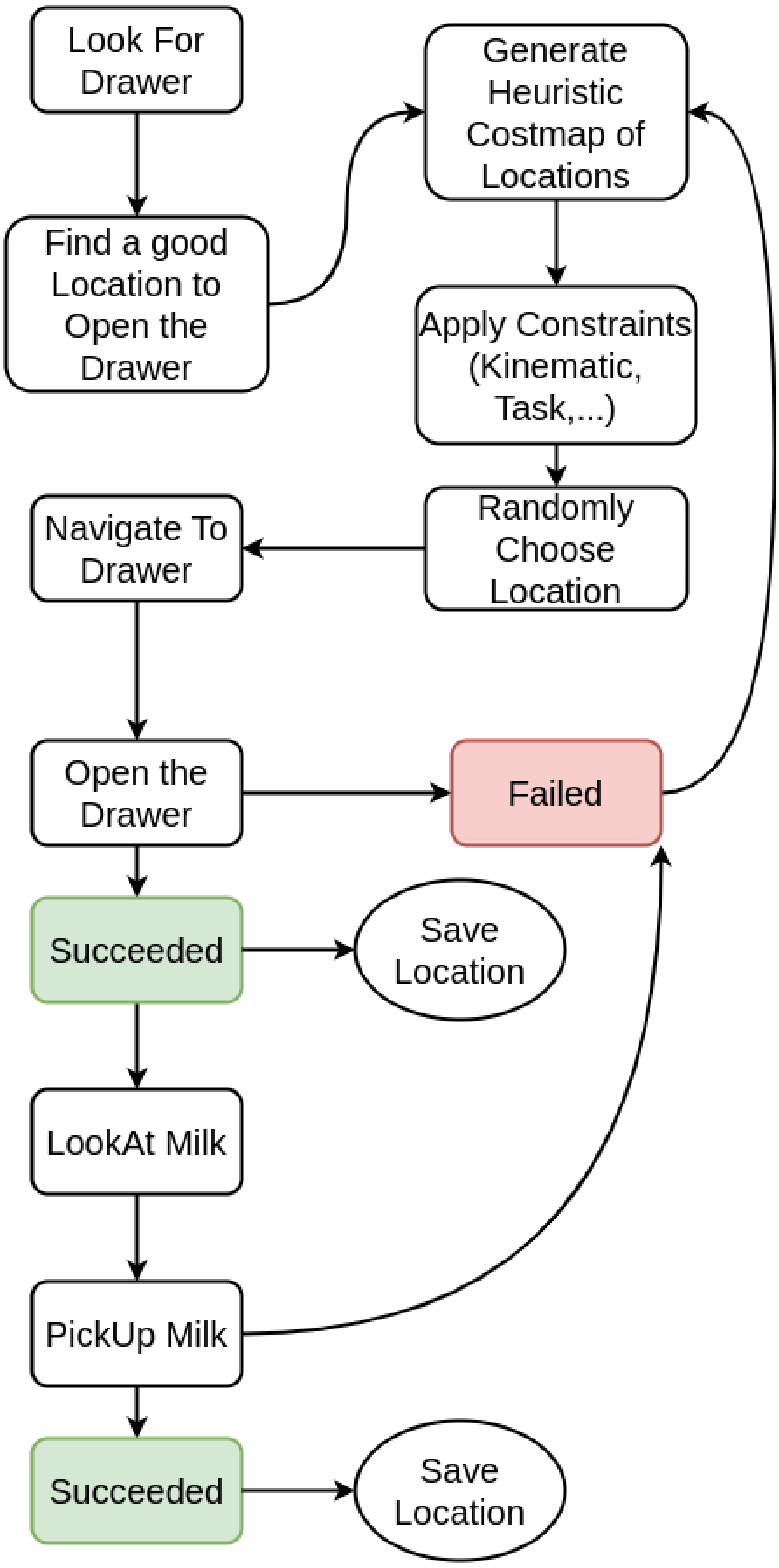


Figure 4.36: This shows the task tree that gets executed before using the machine learning model to initialize the environment and robot state in a good initial state where there is certainly a possible way to pick up the milk from the drawer. This also shows how without the machine learning model executing such tasks would be much slower and more prone to failure.

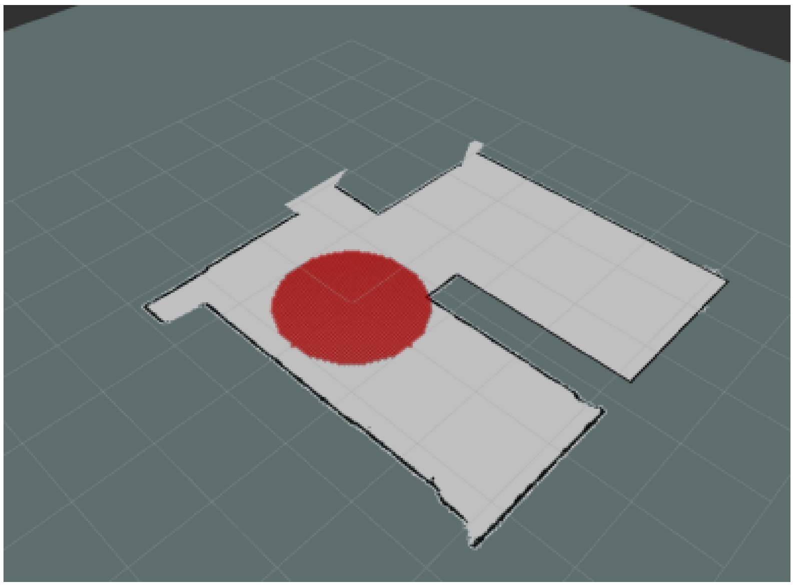


Figure 4.37: Visualized costmap of pr2 standing locations

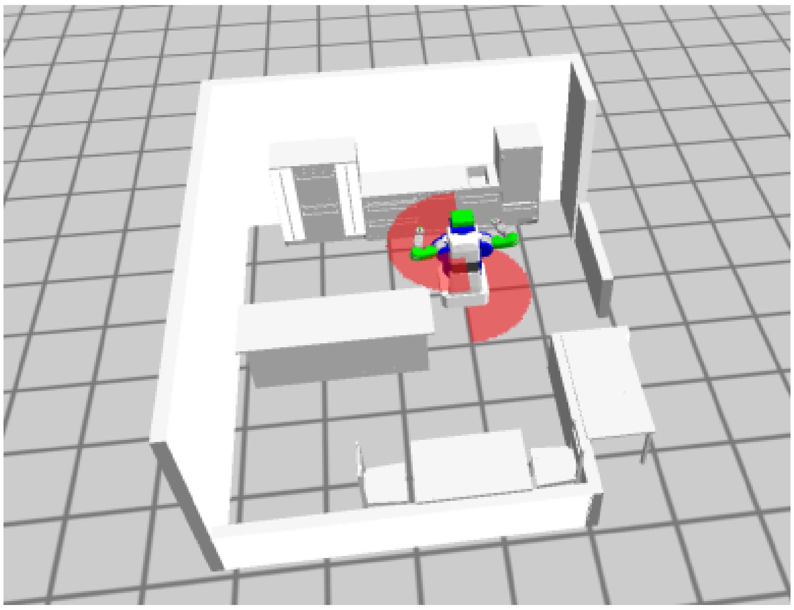


Figure 4.38: Visualized costmap of PR2 arms

Chapter 5

Conclusions and Future Work

Contents

5.1 Conclusions

This master thesis contributed to moving robotics a step forward towards *big data* by taking the *Narrative-Enabled Episodic Memories* (NEEMs) and putting them in an SQL database and making them accessible through the cloud.

This thesis work showed that NEEMs (including its two types of data, that is, symbolic and non-symbolic) containing the robot sensor and motor data, and the semantic information of the task, the plan, the environment state, the robot state, the robot reasoning process, etc. is a key to enable big data in robotics and more specifically in robotic manipulation. Linking all these types of data together using the database key system and giving them the correct structure and data types makes it much more convenient for machine learning pipelines.

The *Joint Probability Trees* (JPTS), a transparent model based on decision trees has been used on two machine learning tasks using the NEEMs SQL database. The purpose was three-fold, the first is to show the importance of the context and the semantic data that is a unique part of the NEEMs database compared to any other database. The second purpose is to show a complete robotics machine learning pipeline starting from problem description, and going through all of the steps like figuring out required information, querying the database for that information and generating tables, then fitting the machine learning model and evaluating it in simulation. The third and last purpose was to show the benefit of JPTs as a transparent and efficient machine learning model and show what parameters affect its performance and how being transparent has successfully guided the tuning and model improvement and debugging process.

The first test case was to try to fit a model to *Predict Next Task*, that is, it tries to predict the next task that the robot should perform given the context which includes information on current and previous tasks, the environment, and the task states whether succeeded or failed. Applying this model iteratively generates a complete task tree that can be executed on an actual or simulated robot. The simulation has been chosen for convenience and ease of control given the time and resources available.

The second test case was on the task of *Failure Recovery. Failure Recovery* is very similar to *Predict Next Task* problem with the difference that the last performed task has failed so more information related to the cause of failure is required in addition to enforce the success of the next task.

Finally, it has been shown that each problem requires some context for a model to learn to solve the problem successfully. The NEEMs is an approach which provides this context in an efficient and scalable manner. Each developed software is publicly available on my Github[1] and has been tested. A command line interface is available for the conversion of the NEEMs database from MongoDB to MariaDB and has been used by multiple people at the Institute For Artificial Intelligence (IAI), University of Bremen, Germany.

5.2 Future work

Although the main purpose of this master thesis work has been to make the NEEMs available in SQL and test them in a complete machine learning pipeline, an important side goal has been to show the importance and benefit of the NEEMs and this still has much more potential to be shown.

The NEEMs are information-rich, and the two test cases that have been done in this work are not nearly enough to cover the variety of tasks that could be learned from them, some of which are: learning the task parameters, fitting a model to learn to predict the best location for the robot to stand on to pick or place an object, learning how to perform pouring, learning how to interact properly in the presence of human collaborators, learning motions and plans from humans performing a certain task from NEEMs that involve humans in virtual reality, and much more.

[1]https://github.com/AbdelrhmanBassiouny

A very interesting application is to learn manipulation plans and motions from video demonstrations (see Figure 5.1). The NEEMs already have a huge advantage in annotating parts of videos and simulations using time stamps to relate them to a specific task or motion in the plan. In addition, scene graphs will play a fundamental role in video understanding and in bridging the gap between the sub-symbolic data in the video to the symbolic data in the ontologies and enable the application of logic and reasoning.

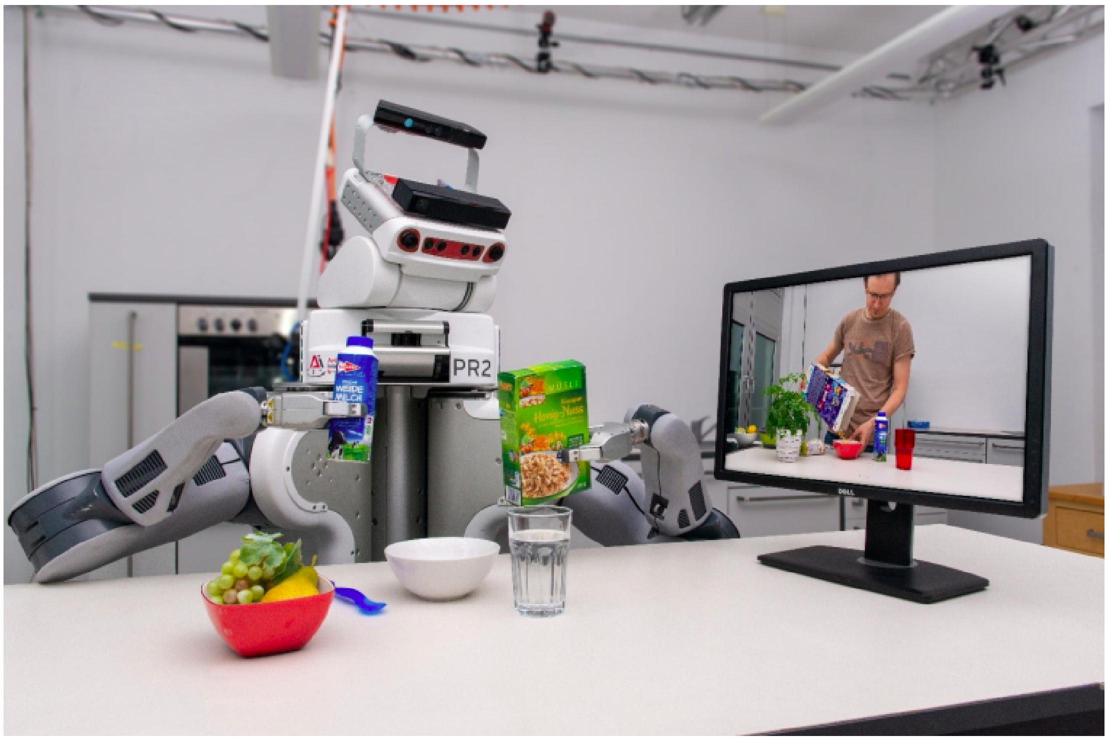


Figure 5.1: PR2 robot learning plans and motions to perform a certain task from a video demonstration.

Using Scene Graphs to annotate the videos not just of real videos but also of virtual reality simulations where much more data and annotation can be easily done and retrieved can a very important step toward learning from demonstrations. Figure 5.2 shows how the *NEEMs* can be used to store all this information due to it having the capability to store symbolic data from the scene graphs that can be modeled as predicates and the motion trajectories from the humans in the virtual reality and the real world in its non-symbolic database. Using the very easy SQL language to query this information for all kinds of machine-learning tasks will push the research forward in the area of learning from demonstrations.

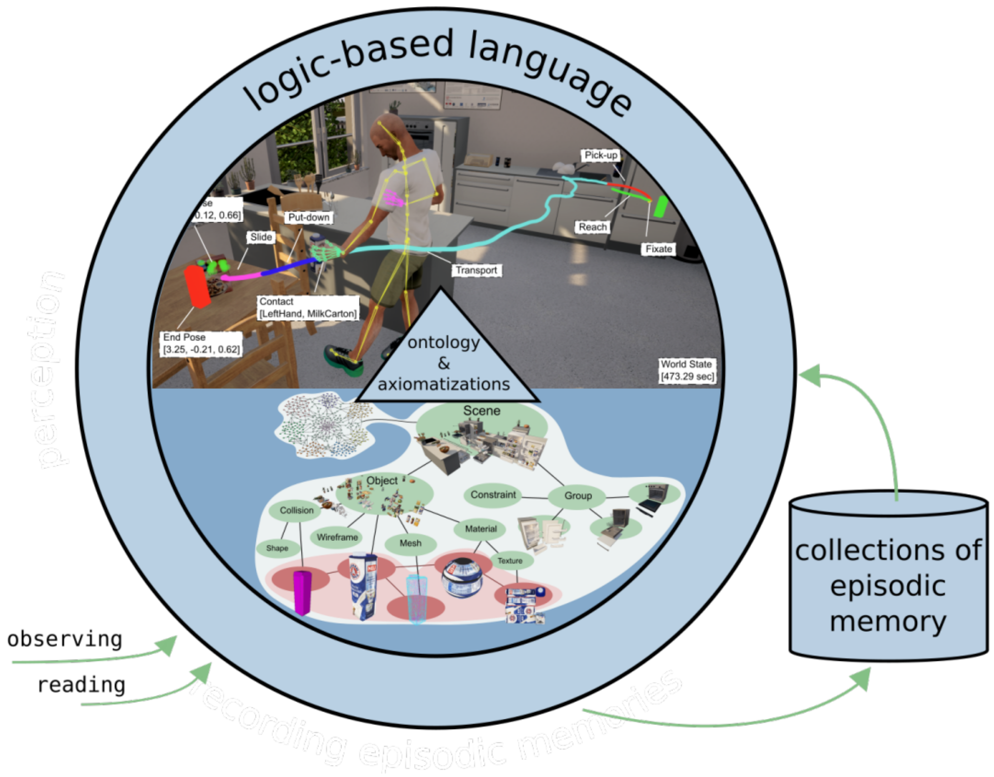


Figure 5.2: *NEEMs* used with virtual reality and scene graphs to enable learning from demonstrations. Storing and learning both the semantics from the scene graphs and also the non-symbolic motions trajectories and motion primitives.

Bibliography

[1] JR Anderson, D Bothell, MD Byrne, S Douglass, and C Lebiere. &amp; qin, y.(2004). *An integrated theory of mind. Psychological Review111*, pages 1036–1060.

[2] Michael Beetz, Daniel Beßler, Andrei Haidu, Mihai Pomarlan, Asil Kaan Bozcuoğlu, and Georg Bartels. Know rob 2.0—a 2nd generation knowledge processing framework for cognition-enabled robotic agents. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519. IEEE, 2018.

[3] Michael Beetz, Lorenz Mosenlechner, and Moritz Tenorth. Cram — a cognitive robot abstract machine for everyday manipulation in human environments. pages 1012 – 1017, 11 2010.

[4] Daniel Beßler, 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 Agents1.* 12 2021.

[5] Manfred Jaeger. Probabilistic decision graphs - combining verification and ai techniques for probabilistic inference. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.*, 12(Supplement-1):19–42, 2004.

[6] Kazuhiko Kawamura, Stephen M Gordon, Palis Ratanaswasd, Erdem Erdemir, and Joseph F Hall. Implementation of cognitive control for a humanoid robot. *International Journal of Humanoid Robotics*, 5(04):547–586, 2008.

[7] Davis E Kieras and Davis E Meyer. An overview of the epic architecture for cognition and performance with application to human-computer interaction. *Human–Computer Interaction*, 12(4):391–438, 1997.

[8] Iuliia Kotseruba, Oscar J Avella Gonzalez, and John K Tsotsos. A review of 40 years of cognitive architecture research: Focus on perception, attention, learning and applications. *arXiv preprint arXiv:1610.08602*, pages 1–74, 2016.

[9] Pat Langley and Dongkyu Choi. A unified cognitive architecture for physical agents. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1469. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[10] Daniel Nyga, Mareike Picklum, Tom Schierenbeck, and Michael Beetz. Joint probability trees, 2023.

[11] Fabian Peller-Konrad, Rainer Kartmann, Christian R. G. Dreher, Andre Meixner, Fabian Reister, Markus Grotz, and Tamim Asfour. A memory system of a robot cognitive architecture and its implementation in armarx. *Robotics Auton. Syst.*, 164:104415, 2022.

[12] David Vernon. Cognitive architectures. *Cognitive Robotics. MIT Press, Cambridge*, 2022.

[13] Jan Winkler, Moritz Tenorth, Asil Bozcuoğlu, and Michael Beetz. Cramm — memories for robots performing everyday manipulation activities. 12 2013.

Appendix A

Source code

NEEMs To SQL:
https://github.com/AbdelrhmanBassiouny/neem_to_sql
This is a repository for converting *NEEMs* from MongoDB to MariaDB.

NEEMs Research:
https://github.com/AbdelrhmanBassiouny/neems_research
This is a repository for fitting Joint Probability Trees (JPTs) on tasks that could be learned from the *NEEMs* database. More specifically the two tasks of **Predict Next Task** and **Failure Recovery**.

NEEMs with PyCRAM in Jupyter Notebook:
https://github.com/AbdelrhmanBassiouny/pycram/blob/complex_plans/examples/neems_to_cram_plan_t
This is a Jupyter Notebook that shows how to use the learned JPTs with PyCRAM in simulation.