Faculty 03: Mathematics/Computer Science

Bachelor's Thesis

PyCRORM: A Relational Approach to Episodic Memory in PyCRAM

David Prüser

Matriculation No. 606 236 6

August 13 2024

First Examiner: Prof. Michael Beetz, PhD
Second Examiner: Dr. Thomas Röfer
Advisor: Tom Schierenbeck, M.Sc.

Declaration of Authorship

I hereby confirm that the thesis I am submitting is my own original work. Any use of other materials and works of other authors is properly acknowledged at their point of use.

Bremen, August 13 2024

David Prüser

Contents

Contents i

1 Introduction **1**
1.1 Motivation 1
1.2 Roadmap 2

2 Related Work **5**
2.1 Cognitive Robot Abstract Machine 5
2.2 Adaptive Control of Thought-Rational 6

3 Preliminaries **7**
3.1 Cognitive Architectures 7
3.1.1 Perception 8
3.1.2 Action selection and execution 8
3.1.3 Reasoning and Decision-making 8
3.1.4 Attention 9
3.1.5 Memory 9
3.1.6 Learning 10
3.2 PyCRAM 10
3.2.1 Designator 11
3.2.2 TaskTree 13
3.3 Databases 13

4 Contribution **17**
4.1 *PyCRAM* classification 17
4.2 Functional requirements 19
4.3 Approach 20
4.4 ORM-Class Structure 21
4.5 Mappings 24
4.6 Querying in the ORM 26
4.6.1 Views 26

5 Evaluation **29**
5.1 *PyCRORM* Usage Demo 29
5.2 Learning Demo 37
5.3 Evaluating data-intensive applications 40

5.3.1 Reliability . . . 40
5.3.2 Scalability . . . 41
5.3.3 Maintainability . . . 42
5.3.3.1 Operability . . . 42
5.3.3.2 Simplicity . . . 43
5.3.3.3 Evolvability . . . 44

6 Conclusion **47**
6.1 Future Work . . . 47

Bibliography **49**

Chapter 1

Introduction

1.1 Motivation

The field of artificial intelligence has been around for many decades, providing many subfields of research. According to the definition of Russell & Norvig in their widely known book *Artificial Intelligence: A modern Approach*, the question about what artificial intelligence is can be divided into four different possible definitions:

> „*AI* is the field of study that tries to build ..., 1. Systems that think like humans, 2. Systems that act like humans, 3. Systems that think rationally and 4. Systems that act rationally. " (Stuart Russell & Peter Norvig, [RN21])

Rational in this case means that the system achieves the best possible - or at least the best expected - outcome, when acting under uncertainty.

Significant advancements have been made in developing such systems [KT20]. One subfield of artificial intelligence, cognitive robotics, introduced the approach of cognitive architectures (3.1) to model human cognition and behavior in an attempt to create cognitive robotic systems. One such framework, PyCRAM (3.2, 4.1), aims to model human cognition to develop software enabling robotic agents to achieve autonomous behavior. This framework is actively developed and utilized in research at the Institute of Artificial Intelligence (IAI)[1] at the University of Bremen [Dec].

Modeling the human mind is not an easy task due to humans complex decision-making, learning, memory capabilities, and emotions. However, in recent research, several characteristics have been addressed as critical components of cognitive systems (3.1). Among these characteristics, the ability to learn from past experiences and events, improve decision-making, and choose a certain (optimal) action sequence out of multiple possibilities is one of the most critical abilities of a cognitive agent. Yet, the way components like learning and decision-making are modeled differ in different architectures.

Although *PyCRAM* already implements many characteristics (3.1) of human cognition like decision-making and action selection as described in Chapter 4.1, it currently has no inherent learning capabilities. As the short- and long-term memories are the most important characteristic in a human to learn a new skill and improve knowledge like language understanding,

[1] https://ai.uni-bremen.de/

memory is also one of the most critical components in cognitive architectures. One important aspect of the memory necessary for learning purposes is episodic memory (3.1.5), which enables an agent to store past experiences and recap them whenever needed [Ver22]. To properly implement learning, the architecture needs a reliable, correct and fast episodic memory component that stores the robot's past experiences and can properly track the robot's environment with its objects and their relationships. The memory needs to be easily usable and user-friendly, and it also needs to achieve high insert- and querying-speeds, since machine learning scenarios often deal with lots of data.

The current type of episodic memory used in *PyCRAM* is a *NoSQL* approach called *Narrative Enabled Episodic Memory (NEEM* (2.1)). This tool has proven itself in various tasks and is used by multiple frameworks, especially within the *Everyday Activity Science & Engineering Collaborative Research Center (EASE CRC)*[2] ecosystem [Bee+20]. However, the implementation was not specifically designed to meet the needs and requirements of *PyCRAM*. As a result, it is not readily accessible or easily integrated within the architecture, and it may not effectively capture all relevant details and relationships. The system interfaces with an external library and utilizes a *NoSQL* database, which is often suboptimal for representing complex relationships between different structures, given the availability of more suitable database options for this purpose. Also, *NoSQL* has no standard querying language, making it harder to use for users which have no experiences with this type of database [NPP13]. Querying multiple *NEEMs* at once, as of right now, is not possible, which might lead to complications during training and testing in learning in the cognitive architecture [Bas24b].

NEEMs not being easily queryable and usable in general, plus their potentially not ideal relationship capturing make it not a perfect fit for *PyCRAM's* needs and requirements, especially in terms of learning. The episodic memory component should be optimized for its architectures needs. *PyCRAM* needs a new memory component that fulfills all the aforementioned requirements, is optimized for its architecture and can be used without further understanding of databases, querying languages and details of the structure and internal functioning of the memory.

This leads to multiple questions that will be answered thoroughly in this thesis. The first and most important question: **Can a suited episodic memory component be introduced to *PyCRAM*, which excels in terms of usability, performance and relationship capturing?** The second question: **Does this memory component enable learning capabilities within *PyCRAM*?**

1.2 Roadmap

To create a proper scientific elaboration, this thesis is divided into multiple chapters.

The first chapter, **Chapter 2** introduces related work in terms of cognitive modeling and memory systems of cognitive agents. It describes two popular cognitive architectures and their

[2] https://ease-crc.org/

implementations of memory, which have been used in research for many years.

The next chapter, **Chapter 3** gives an overview of foundations of subjects necessary to understand this thesis and its implementation, such as cognitive architectures, *PyCRAM* and databases. It is important to understand the fundamentals to further understand the contribution and the added value of the work done.

Chapter 4 contains the core of this thesis. It begins by discussing *PyCRAM* as a cognitive architecture. It explains some of the functionality of *PyCRAM* and compares them to typical characteristics of cognitive architectures. This is important to understand *PyCRAM's* goals and ideas and also create awareness of the state of the framework.

Afterwards, another section recaps and further elaborates on the technical and functional requirements of the memory defined in the introduction and the approach that can be chosen based on these requirements and needs. Once the requirements have been worked out, the chosen approach and type of storage are communicated, and the contribution, namely *PyCRAM ORM (PyCRORM)*, is explained in detail.

Once *PyCRORM* is defined and explained, a working example of this new memory component is shown in **Chapter 5**, followed by an example of learning done with the help of *PyCRORM*. This gives a good impression of the capabilities and communicates design choices connected to the requirements defined earlier. Providing a learning scenario is great to illustrate not only the functionality of *PyCRORM*, but also its influences on learning as a typical characteristic of cognitive architectures. At the end of the chapter, an evaluation of the capabilities of *PyCRORM* in comparison to *NEEMs* is done. This is crucial to prove, that the new memory component actually adds value to *PyCRAM* over the *NEEMs* and second, in scientific research in general.

The last chapter, **Chapter 6** wraps up this thesis with an overview of the work done and an idea of possible future work. It revisits the initial goals and requirements.

Chapter 2

Related Work

Research on cognitive architectures (3.1) and episodic memory systems (3.1.5) has been around for more than 50 years and has improved a lot ever since [Tha12]. Nowadays, the total number of cognitive architectures is expected to be in the 300s [KT20]. This chapter gives an overview of previous research on such architectures and especially episodic memories within these architectures. There are many approaches and lots of literature to these architectures with their implementation of memory in recent research. This chapter gives an overview of some popular and historically relevant cognitive architectures and their approach to memory as part of their structure.

2.1 Cognitive Robot Abstract Machine

***C**ognitive **R**obot **A**bstract **M**achine (CRAM)*[1] is a cognitive architecture written in common Lisp created for designing and deploying software on autonomous, cognitive robotic agents. It was first mentioned in a scientific article in 2010 and was developed in research to analyze and improve task performance and execution on everyday household tasks in a typical living environment [BMT10]. It aims to create a control system that enables a robotic agent to perform complex everyday activity tasks in a household environment in an autonomous, independent way. Task-performing in *CRAM* can be done with a plan language [BMT10].

PyCRAM, which the contribution of this thesis is implemented for is a Python reimplementation of *CRAM*, meaning it started of with similar goals.

Since *CRAM* models cognition and achieves autonomous behavior, *CRAM* also provides an implementation of an episodic memory called Narrative Enabled Episodic Memories *(NEEMs)*. They create episodes for every task and every performed action and store it in their memory. *NEEMs* consist of two parts, the *NEEM experience* and the *NEEM narrative*.

The *experience* stores low-level robot data like sensory information but also holds details about robot or object poses while keeping track of the time of each experience. This is done with the help of ontologies, which are used to describe the environment, the robot acts in.

The *experience* is connected to the *narrative* which stores information about the general episode like goals, the task at hand and descriptions [Bee+20].

Data in *NEEMs* is generally stored in a *NoSQL* database, namely a *Document-oriented*

[1] https://cram-system.org/

database with *JSON* documents using MongoDB[2] as their *Database Management System (DBMS)* (3.3). This approach leads to having high flexibility in schema definition and high scalability, providing *NEEMs* with the ability to store huge amounts of data and change their schema.

There have been attempts to migrate *NEEMs* into a relational database schema, see [Bas24b].

2.2 Adaptive Control of Thought-Rational

***A**daptive **C**ontrol of **T**hought-**R**ational (ACT-R)*[3] is one of the biggest currently maintained cognitive architectures in the world. Its origins can be backtracked to 1973, when Gordon Bower and John R. Anderson designed the *Human Associative Memory (HAM)* [AB14]. It has been extended and improved ever since.

ACT-R ultimately models human cognition, like any other cognitive architecture. It is divided into different core modules for different aspects of cognition-modelling, with its long-term memory *(LTM)* being one of these modules.

It does not define an explicit episodic memory within its *LTM*, but rather stores its knowledge in a declarative and a procedural memory component (3.1.5), with the declarative memory including information about episodes. In general, knowledge in *ACT-R* is represented by chunks, with chunks being structured pieces of information. These chunks consist of a chunk type and different slots filled with associated values, which can be seen as columns in a table in a relational database, containing different details about the entity. These chunks can be referenced by other knowledge in the memory via pointers.

The declarative module is the collection of these chunks, which can be added and updated. In order to update them they are first retrieved from the memory. Once retrieved, the slots can be updated with new values [LGL15].

The procedural system does not consist of chunks but rather stores information about skills and procedures. It consists of different pairs of condition-action rules, where a condition leads to a certain action. The condition defines the chunk and its slots, that must be present in a certain buffer to trigger the action. This definition is a reference to a chunk in declarative memory [Gal13].

To conclude, different architectures follow different approaches to modeling human cognition and implementing (episodic) memory. Therefore, there is no clear „right way“ to implement memory, but one of the biggest cognitive architectures in recent research in the form of *ACT-R* uses an approach similar to and related to a relational database.

[2]https://www.mongodb.com/
[3]http://act-r.psy.cmu.edu/

Chapter 3

Preliminaries

This chapter introduces some basic concepts and definitions needed to understand the contribution and grasp its functionality and its effects on the current architecture and research.

It is divided into different parts, starting with a section about the definition of cognitive architectures and typical characteristics usually modeled and implemented in these architectures. Understanding this topic is important to get an idea of the goals of typical robotic cognitive modeling and to get familiar with some general key components of this thesis.

Another section introduces *PyCRAM* and some of its modules that will be used or extended within the contribution of this thesis. A basic understanding of *PyCRAM's* goals and functionality is crucial to comprehending *PyCRAM's* classification in Chapter 4.1 and getting a grasp of the entrance point of the upcoming implementation of the memory component.

The last section addresses database and *SQL* concepts, giving a basic overview of keywords like *NoSQL* databases, relational databases, *Object Relational Mappings/Mapper (ORM)* or *Views*. These basics are also important in order to get an understanding of the functionality of the new memory component and the possibilities and limitations of different database types that could be chosen as a backend of the memory. Understanding these topics is also important for comparing and evaluating the new memory component (*ORM*) against the old one (*NEEMs*), which is done in Chapter 5.3.

3.1 Cognitive Architectures

Based on the definition of artificial intelligence (*AI*) given in Chapter 1, *AI* can be defined as building systems that think and act like humans (Subpoints 1. and 2.). Lots of research has been done on building such systems, as Chapter 2 suggests. In cognitive robotics, cognitive architectures are one approach to address this challenge.

Over the years, numerous cognitive architectures with diverse capabilities and objectives have been designed and implemented, leading to extensive research in this area. Kotseruba and Tsotsos, in their comprehensive review of 40 years of research in cognitive architectures, define them as:

> „a part of research in general AI, ... with the goal of creating programs that could reason about problems across different domains, develop insights, and adapt to new situations and reflect on themselves. Similarly, the ultimate goal of research in

cognitive architectures is to model the human mind, eventually enabling us to build human-level artificial intelligence.“ (Iuliia Kotseruba & John K. Tsotsos, [KT20])

These programs are often utilized to endow robotic agents with cognitive capabilities. In the context of artificial intelligence, an agent is defined as an entity that acts in a specific manner [RN21].

Some popular examples of cognitive architectures include *CRAM* (2.1), *ACT-R* (2.2), *CLARION* and *Soar*[1].

The term „cognitive architecture“ is somewhat loosely defined, lacking specific criteria for classification. Nevertheless, given that the ultimate objective involves modeling the human cognition and behavior, it is possible to identify common characteristics of human cognition that many architectures incorporate to create cognitive agents [Ver22]. As previously mentioned, research goals and focuses in research vary, leading to different requirements for cognitive architectures and their design. Consequently, some architectures possess features that others do not, and vice versa [KT20][LLR09].

Some of these characteristics include:

3.1.1 Perception

Perception is a critical property when developing a fully cognitive agent. It is essential for enabling the agent to communicate and interact with its environment. Similar to human senses, the agent requires mechanisms to receive external inputs and appropriately store this information to respond to changes in a dynamic world. Common perceptual modalities include vision, sound, and smell [KT20]. Data acquired by sensory modules of an agent are typically saved to the agent's sensory memory and, depending on the current task, moved to the working memory.

3.1.2 Action selection and execution

Another capability of an agent is action selection and execution. Often, the agent is not only designed to model human cognition but also to utilize this cognition to interact with its environment. Typically, acting in an environment involves executing a sequence of actions. However, the agent must be aware of its own skill set, the motor capabilities of its actuators, and the information obtained through its sensors to adapt based on the outcomes of these actions. Thus, to execute action sequences effectively, the agent requires close interaction with its memory [RSS12] to obtain and use this capability. Sensory information is stored within the working memory by perceptors, while the agent's capabilities are recorded in procedural memory.

3.1.3 Reasoning and Decision-making

Reasoning involves the process of drawing conclusions and making inferences based on current knowledge. It encompasses the establishment and application of logic, rules, and heuristics within the environment and for the agent. To effectively model human cognition, there must be a structured representation of the world and the relationships between objects, enabling the

[1] https://soar.eecs.umich.edu/

agent to comprehend its environment and engage in reasoned activity. In cognitive robotics, this capability is pivotal for establishing a belief state and enabling the agent to react and adapt to a dynamic environment [Oli+19]. Unlike humans, robotic agents lack inherent understanding of their environment. One method to formally describe domains of interest, such as the environments in which agents exist and act, is through the use of ontologies [Küm24]. Reasoning can be categorized into various types, including analogical reasoning, deductive reasoning, and moral reasoning [LSB14].

Reasoning serves as the foundational framework for decision-making. The beliefs and experiences of a cognitive agent guide its decisions by evaluating potential outcomes and identifying appropriate actions. Strong decision-making abilities are essential for effectively modeling cognitive processes.

3.1.4 Attention

Attention refers to the agent's ability to assess its environment and prioritize specific sensory data over others. When combined with reasoning, attention can influence changes in action selection and execution. Ideally, the agent autonomously determines which inputs to ignore and which require its focus [BK11]. Depending on the task and goals, evaluating the same environment may yield different outcomes, as certain external factors may need to influence the agent's next actions in one sequence but may be irrelevant in another.

3.1.5 Memory

A fundamental characteristic of a cognitive agent is its knowledge, particularly the representation of that knowledge. To effectively act and react in a partially known and dynamic environment, an agent must be capable of accessing and storing information about the environment itself, which it typically receives through its sensors, as well as information about its own set of actions and the limitations of its actuators. Additionally, the agent must store information about past experiences and learned behaviors. Cognitive architectures, which aim to replicate human-like processes and structures, incorporate memory components based on principles similar to those of human memory. Typically, this involves the design of both short-term memory and long-term memory systems. However, because memory systems can be tailored to specific requirements, the design of memory components within cognitive architecture can vary. The complete structure of a typical memory system is illustrated in Figure 3.1.5.

Short-term memory is often divided into **sensory memory** and **working memory**. Sensory memory stores recent information acquired by low-level sensors. Working memory is analogous to a personal computer's random access memory (RAM), storing information pertinent to the current task, such as goals, knowledge about objects, possible actions, and environmental details [Ver22]. It also receives data relevant to the current task from the sensory memory.

Long-term memory is generally used to store an agent's previous experiences in solving various tasks in the form of episodes, as well as to hold knowledge about objects in the environment, goals, and other relevant information. Long-term memory often distinguishes between procedural and declarative memory.

Procedural memory stores information about motor tasks and action sequences, as well as

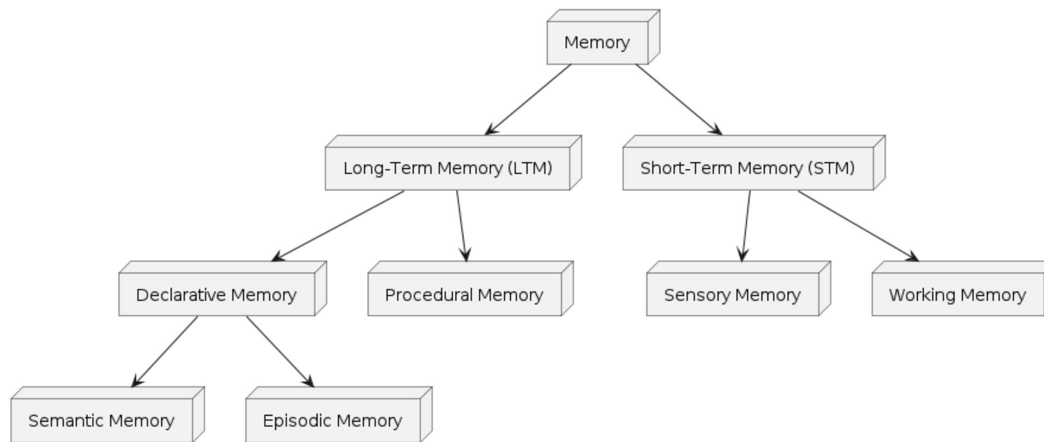


Figure 3.1 Typical Partitioning of Memory

learned behavior. It serves as the repository for the "how" in an agent's functioning [Su+16].

Declarative memory, on the other hand, can be divided into two subcategories and pertains to the "what" [Su+16]. **Semantic memory** functions similarly to procedural memory but focuses on information about the environment and the task at hand, rather than the capabilities of the agent's actuators. These two types of memory share a significant relationship. [Leó16]. **Episodic memory** stores past experiences and events, such as poses before and after executing actions, often recorded with a timestamp indicating when the change or event occurred. This provides comprehensive access to earlier episodes.

3.1.6 Learning

Learning entails leveraging prior experiences to enhance performance over time or adapt to ambiguous environments [TH12]. As discussed in Chapter 3.1, a prevalent objective in cognitive architectures is the development of intelligent systems. The capacity to assimilate experiences and external influences constitutes a fundamental attribute of human cognition. Therefore, any attempt to emulate human cognitive structures must include modeling the phenomenon of learning.

Typically, knowledge is not inherently embedded within the architecture itself but is acquired and retained through the execution of various experiments. Thus, memory plays a pivotal role in facilitating the learning process. Similar to the categorization of memory types, learning in cognitive robotics can be delineated into several categories: perceptual learning, procedural learning, declarative learning, associative learning, non-associative learning, and priming [KT20]

3.2 PyCRAM

As mentioned in the Introduction, *PyCRAM*[2] is a framework for designing and deploying software on robotic architectures to achieve high levels of robot autonomy and control [Dec]. It is a Python re-implementation of *CRAM* (Cognitive Robot Abstract Machine) written in common

[2] https://github.com/cram2/pycram

Lisp. It uses *ROS*[3] (Robot Operating System), which comes along with certain functionality like sensor data processing.

One of the framework's primary features is the so-called "plan language", which gives the end-user a way to implement and execute robot plans on different robots. These plans can be designed without much knowledge about *PyCRAM's* architectural structure and the implementation of different components. They can be executed on different robots like the PR2 or the *IAIs* Boxy.

PyCRAM also presents the possibility of simulating these plans within the project. With the *BulletWorld* being a physics-based simulation environment, the user can do realistic and fast robotic simulations [Dec19], meaning the user does not need to explicitly test a plan on the robot itself. The *BulletWorld* is designed around *PyBullet*[4], a library for real-time physics simulation.

PyCRAM defines multiple modules typical for cognitive architectures and can therefore be seen and used like such architectures, as described in Chapter 4.1.

3.2.1 Designator

As described above, *PyCRAM* can be used to create plans that run on different kinds of robotic architectures. These plans consist of one or more designators, a construct already used in *CRAM*, and then further extended in *PyCRAM* [BMT10]. An example of a typical plan within *PyCRAM's* kitchen environment where the robot moves to a counter, picks up a bottle of milk, and places it back on the counter can be seen in Figure 3.2.

Designators are a straightforward approach to represent different aspects of the experiment's environment and to execute certain tasks the architecture is supposed to perform. Using Pythons object-oriented programming approach, every designator is represented by an associated class. Plans usually consist of several instances of these classes.

Designators are a key aspect of this thesis since they represent the base of the contribution as described in Chapter 4.

These designator can be divided into four different types: action designator, motion designator, object designator and location designator.

- **Action Designator**

  Action designators describe different actions a robot is able to perform.

  These actions come with a perform() function which executes one or more motions connected to the action.

  They take a list of arguments, which specify details about this instance of an action, depending on the action itself. Table 3.1 shows all the actions currently implemented in *PyCRAM*.

- **Motion Designator**

  Motion designators are the motions that are performed by the robot, triggered by the

[3] https://ros.org/

[4] https://pybullet.org/wordpress/

Action Designator	Motion Designator
CloseAction	ClosingMotion
DetectAction	DetectingMotion
GraspingAction	LookingMotion
GripAction	MoveArmJointsMotion
LookAtAction	MoveGripperMotion
MoveTorsoAction	MoveJointsMotion
NavigateAction	MoveMotion
OpenAction	MoveTCPMotion
ParkArmsAction	OpeningMotion
PickUpAction	WorldStateDetectingMotion
PlaceAction	
ReleaseAction	
SetGripperAction	
TransportAction	

Table 3.1 All action and motion designator in *PyCRAM*

action.
In previous versions of *PyCRAM* motions were intended to be called by themselves within plans, but nowadays motions are rather used internally. One reason is failure handling. *PyCRAM* provides failure handling for actions by default but does not provide failure handling for motions.
See Table 3.1 to see all motions implemented in *PyCRAM*.

- **Object Designator**
  Object designators are used to describe objects that exist in the environment. They are often needed by action designators in order to execute certain actions like picking up or transporting an object. An example is the milk bottle, which is used in the Pick and Place plan above.

- **Location Designator**
  In contrast to all other designators, location designators can be seen as helper classes. They give information about locations in different contexts. Examples are the *CostmapLocation*, which uses cost maps in order to create locations with certain constraints, and the *AccessingLocation*, which states different poses from which a drawer could be opened.
  While this type of designator is very important for the functionality of *PyCRAM*, it is not as important for the goal of this bachelor thesis. It is used to get information about the poses of objects or the robot in order to be able to use certain actions that need to specify poses, but these types of designators are nothing supposed to be stored in the episodic memory.

```python
from pycram.process_module import simulated_robot
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.datastructures.enums import ObjectType, WorldMode, Arms
from pycram.object_descriptors.urdf import ObjectDescription
from pycram.designators.action_designator import *

extension = ObjectDescription.get_file_extension()
world = BulletWorld(mode=WorldMode.GUI)
milk = Object("milk", ObjectType.MILK, "milk.stl", pose=Pose([1.3,
    1, 0.9], [0, 0, 0, 1]))
robot = Object(robot_description.name, ObjectType.ROBOT,
                       robot_description.name + extension)
kitchen = Object("kitchen", ObjectType.ENVIRONMENT, "kitchen" +
    extension)
object_description = ObjectDesignatorDescription(["milk"])

with simulated_robot:
    ParkArmsActionPerformable(Arms.BOTH).perform()
    NavigateActionPerformable(Pose([0.6, 0.4, 0], [0, 0, 0, 1])).
    perform()
    MoveTorsoActionPerformable(0.3).perform()
    PickUpActionPerformable(object_description.resolve(), "left",
    "front").perform()
    PlaceActionPerformable(object_description.resolve(), "left",
    Pose([1.3, 1, 0.9], [0, 0, 0, 1])).perform()
```

Figure 3.2 Pick and Place plan in *PyCRAM*

3.2.2 TaskTree

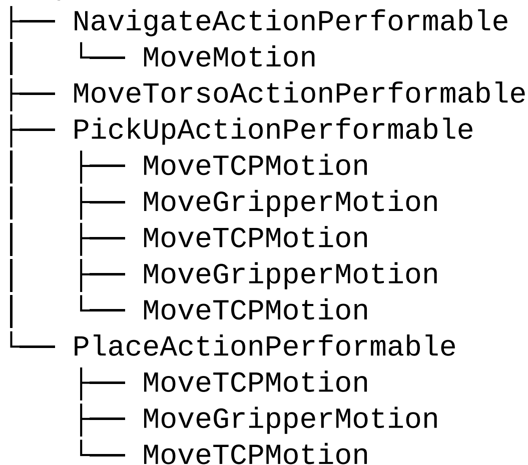
Internally, *PyCRAM* uses a tree-like structure to represent and manage tasks triggered by plans. This *TaskTree* consists of nodes (*TaskTreeNodes*) which are instantiated whenever an action or motion designator's perform() function is called, meaning that every node represents one action or motion. Regarding cognitive architectures, implementing some kind of action-execution-sequence-tracking is quite common. In practice, the *TaskTree* belonging to the pick and place plan example shown in Figure 3.2 can be seen in Figure 3.3.

3.3 Databases

An essential topic across various domains, including artificial intelligence, cognitive modeling, and beyond, is the storage, retrieval, and management of data. From large corporations requiring efficient systems to store customer and employee information to cognitive architectures aiming to model human memory, and web applications handling vast amounts of data, needs are diverse. Related data is typically stored in a unified format at a centralized location, with the format determined by the specific requirements of the task or the nature of the data. This organized collection of related data, maintained in a consistent format, is referred to as a database

```python
from pycram.tasktree import task_tree

pycram.orm.base.ProcessMetaData().description = "Pick and Place plan in PyCRAM"
task_tree = task_tree
print(anytree.RenderTree(task_tree))
```



```
NoOperation
├── NavigateActionPerformable
│   └── MoveMotion
├── MoveTorsoActionPerformable
├── PickUpActionPerformable
│   ├── MoveTCPMotion
│   ├── MoveGripperMotion
│   ├── MoveTCPMotion
│   ├── MoveGripperMotion
│   └── MoveTCPMotion
└── PlaceActionPerformable
    ├── MoveTCPMotion
    ├── MoveGripperMotion
    └── MoveTCPMotion
```

Figure 3.3 *TaskTree* of a Pick and Place plan in *PyCRAM*

[CB05]. Over the years, numerous design approaches have emerged, each tailored to meet particular needs and requirements. Among these, two of the most widely recognized approaches are relational databases and *NoSQL* databases.

***NoSQL (Not Only SQL)* databases** are a schema-less approach allowing flexible data modeling. NoSQL databases are typically used when working with high amounts of data in which relationship modeling is not the focus and high flexibility and scalability are desired [SK11]. An example of a NoSQL database is MongoDB[5] which uses json-like files for data storage.

In contrast, a **relational database** is a rather fixed approach to storing and managing data. It consists of a structured set of tables with varying amounts of columns in each table. Once defined, this structure can not be modified easily. Connections and relationships between tables can be defined by keys. A primary key describes a minimal amount of columns that uniquely identify every row in the table. A foreign key holds a reference to the primary key of a different table, defining a relationship between them. These relationships are one of the strongest selling points of relational databases.

Relational Database Management Systems (RDBMS) are used to work and interact with the database, e.g. to define its structure, store, update and retrieve data. *RDBMS* come with transaction management, meaning that they implement some way to deal with multiple parallel accesses to the database to ensure data safety and consistency [PPJ17]. Analogous, ***Database Management Systems (DBMS)*** sometimes exist for non-relational databases.

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

Reliability, data safety, and correctness of a relational database are ensured by the ***ACID*** properties. *ACID* stands for *A*tomicity (if one part of the transaction fails, everything fails), *C*onsistency (after every transaction, the database is in a valid state), *I*solation (different concurrent transactions are isolated and do not affect one another) and *D*urability (once a transaction made permanent changes to the database, the changes are kept even in case of errors) [Jat+12]. All major RDBMS follow the *ACID* properties. Data retrieval is done with a query language, in the case of relational databases via the *S*tructured *Q*uery *L*anguage *SQL*. Examples of popular *RDBMS* include MySQL[6], PostgreSQL[7], SQLite[8] and MariaDB[9].

One of the biggest strengths in relational databases is relationship modeling. When working with cognitive architectures, many objects or designators in the robot's environment need to define relationships to each other. These environments are often described by programming languages with the usage of object-oriented programming (*OOP*). Therefore, interactions between the programming language's objects in memory and data in the database can be done via so-called ***Object Relational Mappings (ORM)***. An *ORM* is an interface sitting between an application and a (relational) database. It can be used whenever the application follows an object-oriented programming style, meaning that it contains classes that are instanced. It usually works in a way, where it maps classes (objects) within the application to a corresponding table in the database. This can be achieved by defining a mapper-class structure in the existing code, where a mapper class containing the table structure is assigned to a class in the *OOP* structure. This is done for every class that is supposed to be connected to the database. Afterward, whenever an instance of a class is created or updated, the mapping class synchronously changes data in the database. The mapping classes resort to an *Object Relational Mapper's* internal structure for database interaction handling. One framework that enables developers to create such mappers is *SQLAlchemy*[10].

SQLAlchemy is a Python toolkit for SQL functionality within the language. It helps the user with database creation and management and enables developers to focus their work on Python. When keeping database interaction separate from the application, multiple languages and syntaxes need to be known and used, which can be limiting and exhausting. *SQLAlchemy* consists of two components, the *Core* and *ORM*. *SQLAlchemys* documentation describes the *Core* as

> „... a fully featured SQL abstraction toolkit, providing a smooth layer of abstraction over a wide variety of DBAPI implementations and behaviors, as well as a SQL Expression Language which allows expression of the SQL language via generative Python expressions." (SQLAlchemy authors and contributors, [ACb])

The *ORM* is an optional extension that is based on the core and provides mapping functionality as described above [ACb].

A fundamental functionality of relational databases is the ability to query the data stored

[6]https://www.mysql.com/
[7]https://www.postgresql.org/
[8]https://www.sqlite.org/index.html
[9]https://mariadb.org/
[10]https://www.sqlalchemy.org/

within them. However, queries can become quite large and repetitive, especially when they are used for similar purposes or involve advanced operations. To address this issue and improve efficiency, the concept of **views** was introduced. Views are essentially virtual tables that are not part of the database's permanent schema but function as independent entities that can be queried and updated [CP84].

A view often consists of a join of multiple columns from different tables, as required by the query itself. Unlike temporary query results, views persist beyond the initial querying environment. This allows them to be queried like a standard table, with the added benefit of being automatically updated whenever the original columns in their respective tables change. As a result, using views can significantly enhance performance when executing complex queries multiple times. The ”virtual table” is defined once within the session, allowing it to be queried directly without needing to recreate the same joins for repeated queries.

Chapter 4

Contribution

This chapter starts by discussing *PyCRAM's* goals and functionality in the context of the goals and characteristics of typical cognitive architectures, classifying it as such. Since the previous chapter introduced both *PyCRAM* and cognitive architectures, classifying *PyCRAM* is a logical step. The classification also provides further details about *PyCRAM's* modules which might help to understand later parts of the chapter, introducing the new memory component. It also justifies the necessity of a functioning memory component in this architecture.

After classifying *PyCRAM* as a cognitive architecture, the chapter then provides a section about the technical and functional requirements of the contribution. It makes sense to keep in mind the ultimate goals and requirements when creating a new episodic memory. It is followed by the approach chosen, based on the functional requirements.

Afterward, *PyCRORM*, the implemented memory is introduced and explained. The implementation is divided into different subchapters. The first one explains the structure of *PyCRORM* and the inheritance pattern of all mapper classes. Once done, another section introduces how mappings are done in practice. Once the structure and the process of mappings are defined, another section addresses the usage of the implementation (*PyCRORM*), including a practical example. Querying in *PyCRORM* is crucial and thus explained in detail in another subsection.

The code written for this thesis can be found either in the main repository of *PyCRAM*[1] or in a fork[2] of *PyCRAM* on Github[3]. Within *PyCRAM*, most of the code can be found in the *orm* module and the *designator* module.

4.1 *PyCRAM* classification

Chapter 3.1 introduced different common characteristics of cognitive architectures used to classify a system as such. These characteristics include perception, action selection and execution, attention, reasoning and decision-making, memory and learning. But depending on the goals of the architecture, other properties can be defined and some may be ignored. Characteristics are adapted to the requirements of the architecture, leading to different architectures implementing different properties.

[1] https://github.com/cram2/pycram

[2] https://github.com/davidprueser/pycram/tree/dev

[3] https://github.com/

PyCRAM, as defined in Chapter 3.2, is a framework for creating and deploying software on robots. Some of its goals include achieving high levels of robot autonomy and research cognitive agents.

It uses *ROS* internally. *ROS* serves as an interface for communication between hardware and software, in the case of *PyCRAM*, it enables the framework to communicate with the sensors and other components and processes within its structure, giving it the ability to send commands to its actuators and perceive information through its sensors among other capabilities. Actual perception capabilities are achieved by an external module called *RoboKudo*[4], which is a perception framework for robot manipulation tasks. It is also developed and maintained by the *IAI*. This framework integrates *ROS* and can therefore work with its data.

Furthermore, *PyCRAM* defines a *TFBroadcaster*(*T*rans*F*orm *B*roadcaster), which is keeps track of the world state and the objects inside with details like the poses of each object with timestamps. The *Transform Broadcaster* generally extends a *ROS* package *TF* and creates so called coordinate frames for every aspect of the environment like the base frame, gripper frame or head frame [FMM]. These frames change (transform) whenever the corresponding component within the environment changes. The broadcaster transmits these changes to other components of the architecture [Foo13]. The *PyCRAM TFBroadcaster* exchanges messages with the *ROS TF* module using a publisher. This kind of environmental management is crucial in multiple regards. First of all, it enables **action selection and execution**, since it provides a way to keep track of changes that can then be accessed by future actions. This is important because in practical action planning and selection, the robot needs to know all the details about the world to execute an action, e.g. the current pose of the agent or a pose of an object. This information enables the agent to decide whether or not an action can be executed. Second, it also provides some form of **semantic memory** as defined in 3.1.5, since this way it can store details about the general environment like a global structure.
Another characteristic that *PyCRAM* comes along with is **reasoning**. Reasoning is a critical characteristic when creating autonomous behavior is the goal. It was one of the earliest features in *PyCRAM*. For robotic agents, it is crucial to have a form of reasoning implemented to decide if some sequence of actions or motions is possible to execute. Robots do not have reasoning capabilities by default like humans do and do not have any emotions or preferences when confronted with a task. Every decision made relies solely on its reasoning capabilities and the developer's instructions. Among others, some of the most important reasoning capabilities of agents in *PyCRAM* are the reachability of an object by the agent's grippers, visibility of an object, or an object being blocked by another object when the agent tries to pick it up. *PyCRAM* implements some further reasoning capabilities [Dec19].

Like it implements a semantic memory, it also provides a module for **episodic memory**, the *NEEMs*. As mentioned in Chapter 1 they were not created for *PyCRAM* itself and are therefore imported as an external module, with an interface to create communication between *PyCRAM* and *NEEMs*.

[4]https://robokudo.ai.uni-bremen.de/index.html

Chapter 3.1 defined goals that systems called cognitive architectures often follow and also described common characteristics of such architectures. Based on these goals and characteristics, *PyCRAM* can be classified as a cognitive architecture. It tries to achieve high levels of autonomy on robotic agents, meaning to equip them with the best cognitive abilities possible and it also implements many typical components of these architectures.

However, learning is something that most cognitive architectures do implement to equip the robotic agent with even better cognition and it is certainly something that is currently missing as a feature in *PyCRAM*. As described in Chapter 1, to implement learning, an architecture needs some implementation of episodic memory that is best suited for this use case. *NEEMs* are not an ideal fit for *PyCRAM's* use case which is learning, as Chapter 4.3 describes. Therefore, the following section defines functional requirements for an episodic memory component in *PyCRAM*, especially in the context of learning.

4.2 Functional requirements

Multiple goals and requirements for the implementation can be defined.

1. Designator Mapping and Tracking:
   - The episodic memory system shall maintain comprehensive mapping and tracking of all utilized designators within the environment, including action designators, motion designators, and object designators. This mapping must reflect the dynamic state of the environment and support real-time updates.
2. Incremental Memory Updates:
   - Any modification or execution of designators must trigger internal, incremental updates to the episodic memory, preserving the history of previous states and changes. This ensures a chronological and historical perspective of the system's operations, enabling analysis of past episodes and analysis of decision-making.
3. User-Friendliness and Accessibility:
   - The system must provide an intuitive and accessible interface for end-users, eliminating the necessity for specialized knowledge of database structures or query languages such as SQL. It should prioritize ease of access and user interaction.
4. Object Relationship Mapping:
   - The episodic memory system shall capture and represent relationships between objects within the environment, enabling a relational understanding of interactions and dependencies. This relational mapping is crucial for the cognitive architecture's reasoning and decision-making processes.
5. Scalable Data Storage:
   - The system must offer scalable data storage solutions capable of accommodating an indefinite number of plans and designators. The architecture should provide efficient handling of extensive datasets without compromising performance or responsiveness.
6. Efficient Data Querying for Machine Learning Purposes:
   - The episodic memory must support efficient querying mechanisms to provide fast access to stored data, particularly for machine learning applications requiring sub-

stantial amounts of data for training and testing. While the robotic agent's access to its long-term memory does not need to be instantaneous, the architecture must support timely retrieval.

7. Consistency and Integrity Assurance:
   - Mechanisms shall be implemented to ensure data consistency and integrity, preventing corruption or inconsistency during concurrent accesses or updates. This is essential for maintaining the reliability of the episodic memory system.

4.3 Approach

Chapter 4.1 explained that *PyCRAM* currently uses the external module *NEEMs* as its episodic memory. This tool, however, is not perfectly suited for the task of learning and *PyCRAM's* needs in general based on the requirements defined above. It uses a *NoSQL* database, which is not ideal for modeling relationships between objects in an environment. Since it is an external tool, it is not specifically designed to model all different designators used in *PyCRAM* and be used for learning purposes. It does not map any of *PyCRAM's* designator but rather has its own internal types, which are based on the designators defined in *CRAM*. *PyCRAM's* designators are based on the ones in *CRAM*, so due to their similarity, *NEEMS* did not have any problems with *PyCRAM's* structure yet. This might change and lead to complications once *PyCRAM's* internal functioning differs too much from *CRAM's*. Querying *NEEMs* can also be quite challenging since *NoSQL* has no standard querying language. They use *MongoDB*, which has its own querying language, but it is not easy to use, as can be seen in Chapter 5.3. A user would need further information and details about querying syntax and *NEEMs* internal database structure. New *NEEMs* can also only be created by developers of the *NEEM* project.

Thus, *PyCRAM* needs a newly developed episodic memory that is directly designed for the task that is learning.
Based on the requirements defined above, a storage system that can store huge amounts of data with fast insert and query speeds that, while doing so, also maps relationships between different structures and objects and also ensures data integrity and safety, is needed.

One possible way to achieve that is by using a relational database. As described in 3.3, relational databases excel in relationship design. They can be used to define relationships between different tables in the form of foreign keys. They are easily accessible and queryable through the widely popular language *SQL*. Relational databases are often managed by *RDBMS*, which ensure data integrity and safety and transaction handling by following the *ACID* properties. However, even when using a relational database, the memory still needs a structure in which it can capture the world and monitor changes of any designator.

PyCRAM uses an object-oriented approach in its plan language and environment, meaning that every designator in the environment, i.e. actions and objects, is just an instance of a corresponding Python class. Therefore, to monitor all the designators in the current world state, an attempt to map every Python class representing a certain designator to a table in the database, seems fitting. This can be done by ***O**bject-**R**elational **M**appings (ORM)*. *SQLAlchemy* can be used for help with mappings since it provides lots of database and mapping support. As described in Chapter 3.3 about *ORMs*, a common approach of these mappings is to define

a mapper class for each class that is to be mapped and then, whenever an instance is created, invoke the mapper to create a corresponding database table entry of that type. *SQLAlchemy* comes with two different mapping styles for object-relational mappings the user can choose between, imperative mappings and declarative mappings. While an imperative mapping keeps the class definition and the table definition separate, a declarative mapping combines these features, meaning that with a declarative mapping, the user would define a class that serves as the table and within the class, class attributes, that serve as the columns of the table. This approach is very fitting for *PyCRAM* since there doesn't need to be any differentiation between classes and tables when trying to map classes one-to-one.

4.4 ORM-Class Structure

The memory module is structured into different parts that work closely with the concept of inheritance within Python. An overview of the structure and the relationships between tables can be seen in Figure 4.1.

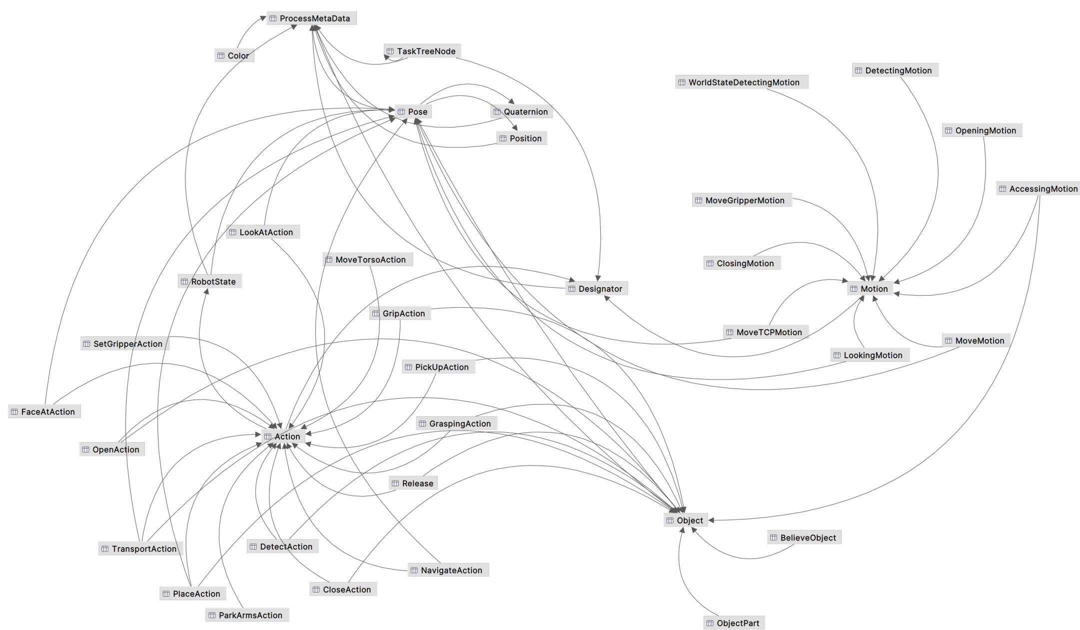


Figure 4.1 ORM class structure including relationships

A ***Base*** class lays the foundation for every mapper class. It inherits from *SQLAlchemy's DeclarativeMapping* class which defines the mapping style. It is an abstract class, meaning it does not represent a table itself, but rather sets attributes or other properties, that other classes inherit from and pick up. The *Base* class creates an *id* column and other settings, e.g. having the name of the class be the name of the corresponding table, which all other classes in *PyCRORM* inherit from. The id is a unique identifier for every entry in a table, enabling it to serve as the primary key in every table. It increments automatically for every entry in a table.

A class ***ProcessMetaData*** creates descriptive metadata about the *PyCRAM* plan at hand, storing at what time the table was created, and by whom it was created, and stores a description of the plan and the current *PyCRAM* version. The *Base* defines a relationship to the metadata.

A relationship in *PyCRORM* is a function that indicates that a table belongs to or depends on another one. It is usually defined by two attributes, one attribute holding the relationship itself and one attribute holding the id of the related object.

The *Base* having a relationship to the *ProcessMetaData* indicates that every class that inherits from the *Base*, inherits the relationship to the *ProcessMetaData*. Since every class inherits from the *Base* as explained above, every table is connected to a corresponding metadata. This makes sense and is desired behavior since the metadata holds descriptive information about the plan, and therefore about all the designators and entries in the plan. Metadata is unique for an experiment, possibly leading to thousands of rows in different tables pointing to the same metadata object. It is important for episodic memory since queries might be related to a certain plan or the time of execution matters, both of which are defined in the metadata. In the end, episodic memory consists of episodes, with episodes often being each plan.

With the *Base* and *ProcessMetaData* defined, actual one-on-one mapper classes can be defined. Mapped should be all designators, but also a couple of other classes.

First of all, a class ***RobotState*** tracks the current state of the robot at all times, including real-time changes. The class is a necessity for all action designators. The (robotic) agent is expected to execute actions, but these actions need information about the current state of the agent, including the robot's pose, its torso height, and the type of robot. These traits are defined as attributes, which define the columns to be created.

Another class ***Pose***, which consists of a ***Position*** and ***Orientation*** also needs to be mapped, due to many properties in the world needing information about their own pose or the pose of other elements. To name a few, like mentioned in the previous paragraph, the *RobotState*, which needs the robot's pose, objects, that need their own pose, or some actions, e.g. an action that moves an object to a different position. It needs the *RobotState*, but also the pose the object is supposed to be moved to.

A class ***TaskTreeNode*** maps every *TaskTree* node in the plan. It stores the action executed, the start time and end time of the node, a status, a reason, and the parent. It is critical since it gives more context to the executive plan and its actions and motions, and also keeps track of the status of each node, meaning that it holds details about the success of an action or motion. The *TaskTreeNode* is one of the tables that influence the functionality the most within *PyCRORM*. The reason for that is discussed in Chapter 4.5.

A key aspect of the implementation is the mapping of object, motion, and action designator. ***Designator*** represents a base class for every designator, storing the designator type.
A class ***Object*** maps objects within the world, e.g. a milk bottle. ***ObjectPart*** and ***BelieveObject*** are two abstractions of the *Object*.
Motion serves as a base class for all motions. It inherits from *Designator* and is inherited by each motion. It holds every executed motion's type. Every derived motion has its own class/table and creates different attributes/columns, based on the motions requirement. Table 4.1 presents a list of every mapped motion designator as of July 2024.
Action serves as a base class for all actions. Like the *Motion* class, it inherits from *Designator*, stores information about every executed action's type and lots of derived actions are defined. However, as mentioned above, it also stores a relationship object to the *RobotState* to keep

track of the robot's current state. All the actions have their own tables, holding information dependent on the action. For example, the mapper class for the *MoveTorsoAction*, which is used to change the height of the robot's torso can be seen in Figure 4.2.

```
class MoveTorsoAction(Action):
    """ORM Class of pycram.designators.action_designator.
↪MoveTorsoAction."""

    id: Mapped[int] = mapped_column(ForeignKey(f'{Action.
↪__tablename__}.id'), primary_key=True, init=False)
    position: Mapped[Optional[float]] = mapped_column(default=None)
```

Figure 4.2 Mapper-Class for the MoveTorsoAction

Table 4.1 presents a list of defined mappings for the action designators.

To conclude this section it can be said, that inheritance and the general structure of classes in *PyCRORM* is crucial for the mapping to avoid explicitly defining every setting and column, that all the tables share, like the id. It also leads to the possibility of defining a new mapper class without much difficulty, providing advanced flexibility. Using the designator classes directly for the mappings without defining explicit mapper classes for every designator is not an option since then, a user would need to think about the structure of the designator's table structure in the databases during creation. This is not ideal because the user should be able to use *PyCRAM* without having to think about the memory in the background.

mapped actions	mapped motions
CloseAction	AccessingMotion
DetectAction	ClosingMotion
GraspingAction	DetectingMotion
GripAction	LookingMotion
LookAtAction	MoveGripperMotion
MoveTorsoAction	MoveMotion
NavigateAction	MoveTCPMotion
OpenAction	OpeningMotion
ParkArmsAction	WorldStateDetectingMotion
PickUpAction	
PlaceAction	
ReleaseAction	
SetGripperAction	
TransportAction	

Table 4.1 List of all mapped action and motion designator

4.5 Mappings

Chapter 4.4 describes the general structure of the relational database and the mapper classes. However, just creating classes with attributes that are then translated into tables with columns by SQLAlchemy does not lead to actual data being automatically retrieved and inserted into the database. Actual mappings of the designator's properties with the corresponding inserts must still be defined.

To create the mappings, it is crucial to understand how SQLAlchemy works. Whenever SQLAlchemy and its modules are suppossed to be used in regards to a database, an engine object needs to be created which receives the URL of the database as its parameters [ACa]. A *Session* is used to communicate between the code and the database. It binds itself to the engine and needs to be opened and closed manually whenever communication with the database is desired, in the case of *PyCRAM* whenever a plan is executed that should use the memory [ACc]. This makes it possible to create and execute plans that do not use the memory in the background. That can bring performance improvements and may be desired for tests and simulations. The session can part an instance of an object into five different states. In the transient state, the objects instance is not currently in the session, meaning it is not saved to the database. With an *add()* function, the instance can be moved to the pending state. This means, that it was added to the session, but is not yet flushed or committed to the database. With a *flush()* or *commit()*, the in-memory pending state of the objects can be flushed or committed to the actual database, moving them to the persistent state. Flushing is a subset of committing. When committing, instances get flushed to the database, making them persistent, and afterward these changes get finalized and made permanent. Just flushing does not make the changes to the database permanent. The last two states are the deleted and detached states, which are not relevant to this thesis [ACd].

This means, that to create the actual mappings, every instance needs to be added to the session and committed to the database. Two additional functions are added to each class that is supposed to be mapped by a corresponding mapper class to manage that functionality, one named *to_sql()* and the other *insert()*. *to_sql()* returns the mapper class that belongs to the specified class. For instance, the *CloseAction*'s *to_sql* would return the *CloseAction* of the *ORM*. The *insert()* first calls the *to_sql* and creates an object of the *ORM* class. This class defines certain attributes, that represent the columns of the table. These attributes get mapped one by one. As described in Chapter 4.4, mapper classes define relationships to other tables whenever a column also references another object. The relationship is defined by two attributes, an attribute that holds the relationship and the object a relationship is supposed to be defined on. This attribute, however, does not create an actual column in the table. It is rather used internally to work on the relationship object, e.g. in joins while querying. The other attribute actually exists in the table and holds the id of the object's entry in its own table, defining a foreign key. So when the attribute of the *ORM* class represents a foreign key to another mapped class, an instance of that object is created and its own *insert()* is called on that instance, a pose for example. So once the instance of the other mapped class is created and inserted, it can be assigned as the value of the attribute holding the actual object. This leads to the second

attribute, which is the foreign key holding the id of the referenced entry, automatically being filled with the object's id in the relationship attribute.

Once the attributes of the object are mapped, the object can be added to the session to move it to the pending state.

To insert the whole plan into the database, *PyCRAM* uses its *TaskTree* (3.2.2). The *ORM* uses this *TaskTree* structure to do the final mappings which lead to database tables being created or extended by a designator. The *TaskTree* iterates recursively through all actions. That makes it possible to define an *insert()* function for the *TaskTreeNodes* class, which for each node invokes the node's *insert()* as well. This means, that in *PyCRAM*, designators are not automatically mapped whenever an instance is created or a plan is executed, but rather whenever the root node's *insert()* is triggered.

Once all actions with their poses, necessary objects, motions, and other attributes are inserted and added to the session, hence being in the pending state, they can be committed and therefore made permanently available in the database, meaning they are moved to the persistent state of the session, which makes them queryable.

To conclude this section, in order to create a designator or other object, there needs to be a mapper class in the *PyCRORM* module in *PyCRAM* and a *to_sql()* and *insert()* that maps that designator or object.

Still, usability is one of the highest valued goals towards the implementation of the memory component. For structures, that have similar definitions, like the action designators, which all save the robot state, and often store poses or objects, it is exhausting and time and space-consuming to map every designator by hand. Thus, *PyCRORM* defines an *ActionAbstract*, which when inherited from, automatically maps all the attributes of a newly defined action, given that they are not mapped manually and a corresponding mapped class is defined. This means, that it is only necessary for the action to inherit from the *ActionAbstract* to do the mappings. The mapping of the *MoveTorsoAction* can be seen in Figure 4.3.

The only thing that changed from the original designator was the addition of the inheritance of the *ActionAbstract* and the addition of the *orm_class* attribute. The rest is done by the abstract mapping. This is great for usability and maintainability since a user creating a new designator does not need to know anything about *PyCRORM* itself, but just inherits from the abstract mapper and defines one attribute that references the corresponding mapper class.

With the overall mapper class structure and the actual mappings being defined, *PyCRORM* can be used. The mapper-classes and mappings for corresponding classes being explained above lead to designators of plans written in *PyCRAM's* plan language being iteratively inserted as episodes into the memory, once the root of the corresponding *TaskTree* is inserted. The memory tracks all the objects and designator according to the functionality described above, also capturing changes throughout the plan, e.g. an object that gets moved by the agent multiple times. For an example of the use of the mappings, see Chapter 5.

```python
@dataclass
class MoveTorsoActionPerformable(ActionAbstract):
    """
    Move the torso of the robot up and down.
    """

    position: float
    """
    Target position of the torso joint
    """
    orm_class: Type[ActionAbstract] = field(init=False,
    default=ORMMoveTorsoAction)

    @with_tree
    def perform(self) -> None:
        MoveJointsMotion([robot_description.torso_joint], [self.
    position]).perform()
```

Figure 4.3 Mapping for the MoveTorsoAction

4.6 Querying in the ORM

As explained in Chapter 1 and 4, one of the biggest goals of *PyCRAM* and therefore one of the biggest requirements for *PyCRORM* is to be able to do learning within the cognitive architecture that *PyCRAM* is. One capability, especially for machine learning purposes, is querying. Querying in relational databases is usually done via *SQL* statements, that define the subject of the query, which are then executed. However, that requires knowledge about the syntax of the querying language and the overall structure of *PyCRORM*, making querying not very user-friendly and usable, which was one of the requirements defined in 4.2. Therefore, through the functionality provided by *SQLAlchemy*, *PyCRORM* can be queried right in Python without the use of *SQL* directly. The query written with Python functions is internally translated to a query the database can understand and execute.

Querying can be split into two parts, the first being a select statement, and the second being the execution of that statement. The select statement defines the desired output from the database. It is a function that takes the columns to be queried as its paramaters, with the ability to output the whole table. Other details like joins, group by or the amount of rows to be taken from the database can be added as an extra function at the end of the statement. Executing the statement can be done by calling *execute()* or *scalars()* on the session, and adding the statement as a parameter.

This way of defining and executing queries is simple and straightforward.

4.6.1 Views

Some, often complex and long parts of queries are repeatedly used when working with *PyCRORM*, especially in the context of learning. These queries can be put into *Views* in *PyCRORM*, which are virtual tables created at the beginning of the session along with the meta-

data, which can be queried like any normal table, leading to simplification of the use of certain statements with complex constraints and often resulting in faster querying times, since the constraints like joins do not have to be computed at querying time but are already computed when the metadata is created. *PyCRORM* offers the creation of an arbitrary number of *Views* to provide great accessibility and usability.

Chapter 5

Evaluation

With the overall structure of the mapper classes and mappings defined in Chapter 4, this chapter evaluates the new memory component and compares it to the old one. It starts with some simple examples of the usage of *PyCRORM*. It follows up with a more complex demo that shows learning mechanisms in practice, using *PyCRORM*. It then evaluates *PyCRORM* in the context of the functional requirements defined in Chapter 4.2 and in contrast to the *NEEMs*.

5.1 *PyCRORM* Usage Demo

This demo mediates a broad idea about the possibilities and capabilities of *PyCRORM* including the setup of a session and connection to the database, insertions of mapped classes, and retrieval of the inserted data. At the end of the demo, the creation of a new designator and the requirements to make it eligible for *PyCRORM* are shown.

The first thing to be done is the creation of an engine and a session, to be able to communicate with the database. In this demo, an *SQLite* in-memory database was chosen, but the choice is up to the user entirely. Afterward establishing a connection, the database structure and metadata are created.

[1]:
```
import sqlalchemy.orm
from pycram.orm.base import Base
import pycram.orm.action_designator

engine = sqlalchemy.create_engine("sqlite+pysqlite:///:memory:",
  ↪echo=False)
session = sqlalchemy.orm.Session(bind=engine)
Base.metadata.create_all(engine)
session.commit()

session
```

[1]:
```
<sqlalchemy.orm.session.Session at 0x7f2319f1a8b0>
```

With the session being up and running, and the structure created, a sample plan using PyCRAMs plan language can be created. First, the world is defined, then, a TaskTree is constructed, holding the sequence of actions executed.

```
[2]: from pycram.designators.action_designator import *
from pycram.designators.location_designator import *
from pycram.process_module import simulated_robot
from pycram.datastructures.enums import Arms, ObjectType
from pycram.tasktree import with_tree
from pycram.worlds.bullet_world import Object, BulletWorld
from pycram.designators.object_designator import *
from pycram.datastructures.pose import Pose
import anytree

world = BulletWorld()
pr2 = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
kitchen = Object("kitchen", ObjectType.ENVIRONMENT, "kitchen.urdf")
milk = Object("milk", ObjectType.MILK, "milk.stl", pose=Pose([1.3,
  ↪1, 0.9]))
cereal = Object("cereal", ObjectType.BREAKFAST_CEREAL,
  ↪"breakfast_cereal.stl", pose=Pose([1.3, 0.7, 0.95]))
milk_desig = ObjectDesignatorDescription(names=["milk"])
cereal_desig = ObjectDesignatorDescription(names=["cereal"])
robot_desig = ObjectDesignatorDescription(names=["pr2"]).resolve()
kitchen_desig = ObjectDesignatorDescription(names=["kitchen"])

@with_tree
def plan():
    with simulated_robot:
        ParkArmsActionPerformable(Arms.BOTH).perform()
        MoveTorsoAction([0.3]).resolve().perform()
        pickup_pose = CostmapLocation(target=cereal_desig.
  ↪resolve(), reachable_for=robot_desig).resolve()
        pickup_arm = pickup_pose.reachable_arms[0]
        NavigateAction(target_locations=[pickup_pose.pose]).
  ↪resolve().perform()
        PickUpAction(object_designator_description=cereal_desig,
  ↪arms=[pickup_arm], grasps=["front"]).resolve().perform()
        ParkArmsAction([Arms.BOTH]).resolve().perform()
```

```
        place_island =
    ↪SemanticCostmapLocation("kitchen_island_surface", kitchen_desig.
    ↪resolve(),
                                                    cereal_desig.
    ↪resolve()).resolve()

        place_stand = CostmapLocation(place_island.pose,
    ↪reachable_for=robot_desig, reachable_arm=pickup_arm).resolve()

        NavigateAction(target_locations=[place_stand.pose]).
    ↪resolve().perform()

        PlaceAction(cereal_desig, target_locations=[place_island.
    ↪pose], arms=[pickup_arm]).resolve().perform()

        ParkArmsActionPerformable(Arms.BOTH).perform()

plan()

# set description of what we are doing
pycram.orm.base.ProcessMetaData().description = "Tutorial for
    ↪getting familiar with the ORM."
task_tree = pycram.tasktree.task_tree
print(anytree.RenderTree(task_tree))
```

```
[INFO] [1719315347.448089]: Ontology [http://www.ease-crc.org/ont/
  ↪SOMA-
HOME.owl#]'s name: SOMA-HOME has been loaded
[INFO] [1719315347.449039]: - main namespace: SOMA-HOME
[INFO] [1719315347.449760]: - loaded ontologies:
[INFO] [1719315347.450509]: http://www.ease-crc.org/ont/SOMA-HOME.
  ↪owl#
[INFO] [1719315347.451221]: http://www.ease-crc.org/ont/DUL.owl#
[INFO] [1719315347.451760]: http://www.ease-crc.org/ont/SOMA.owl#
[INFO] [1719315348.457787]: Waiting for IK service:
/pr2_left_arm_kinematics/get_ik
NoOperation
└── NoOperation
    ├── ParkArmsActionPerformable
    ├── MoveTorsoActionPerformable
    ├── NavigateActionPerformable
    │   └── MoveMotion
```

```
├── PickUpActionPerformable
│   ├── MoveTCPMotion
│   ├── MoveGripperMotion
│   ├── MoveTCPMotion
│   ├── MoveGripperMotion
│   └── MoveTCPMotion
├── ParkArmsActionPerformable
├── NavigateActionPerformable
│   └── MoveMotion
├── PlaceActionPerformable
│   ├── MoveTCPMotion
│   ├── MoveGripperMotion
│   └── MoveTCPMotion
└── ParkArmsActionPerformable
```

To insert the data into the database, the *TaskTree* needs to be iteratively inserted, starting at its root.

[3]:
```
task_tree.root.insert(session)
```

```
Inserting TaskTree into database: 100%|██████████| 20/20 [00:00<00:
↪00,
246.74it/s]
```

[3]:
```
TaskTreeNode(id=1, action_id=None, action=None,
start_time=datetime.datetime(2024, 6, 25, 13, 35, 44, 64330),
↪end_time=None,
status=<TaskStatus.RUNNING: 1>, reason=None, parent_id=None,
↪parent=None,
process_metadata_id=1)
```

It can be observed, that the insertion into the database happened instantaneous. 20 nodes were inserted with a speed of 246.74 iteration per second.

With the data inserted into the database, queries can be computed, requesting any kind of data from the database. Querying in *PyCRORM* is quite easy and especially, provides a good user-experience, due to the ability to write queries with Python functions, which are easy to understand and work like a user would expect them to work. A query that graps the aboves plan's metadata looks like this:

[4]:
```
from sqlalchemy import select
```

```
print(*session.scalars(select(pycram.orm.base.ProcessMetaData)).
  ↪all())
```

```
ProcessMetaData(id=1, created_at=datetime.datetime(2024, 6, 25, 11,
  ↪35, 55),
created_by='dprueser', description='Tutorial for getting familiar
  ↪with the
ORM.', pycram_version='fab4c2d0b234032f4a94c5e8c6a1a33ee775c2dc')
```

It is to be remembered, that querying can be done by either calling *.scalars()* or *.execute()* based on the desired return type of the data. Another query that graps all *NavigateActions* executed would look like this:

```
[5]: navigations = session.scalars(select(pycram.orm.action_designator.
       ↪NavigateAction)).all()
     print(*navigations, sep="\n")
```

```
NavigateAction(id=3, process_metadata_id=1, dtype='NavigateAction',
robot_state_id=3, robot_state=RobotState(id=3, pose_to_init=False,
torso_height=0.3, type=<ObjectType.ROBOT: 8>, pose_id=3,
  ↪process_metadata_id=1),
pose_to_init=False, pose_id=4)
NavigateAction(id=12, process_metadata_id=1, dtype='NavigateAction',
robot_state_id=6, robot_state=RobotState(id=6, pose_to_init=False,
torso_height=0.3, type=<ObjectType.ROBOT: 8>, pose_id=12,
process_metadata_id=1), pose_to_init=False, pose_id=13)
```

This example demonstrates relationship modeling in *PyCRORM*. The output shows two *NavigateAction*s. Both of them have a parameter *robot_state*. This robot state, however, is no actual column in the database, as described in Chapter 4.5, but an internal attribute that holds the referenced *RobotState* object. The foreign key referencing this object is stored in the *robot_state_id* parameter, which is an actual column in the database. Having the ability to see the actually referenced object within the query result leads to great possibilities in multiple regards, like checking data integrity, easy use of the database, and better accessibility.

Due to the inheritance mapped in the ORM package, all executed actions can also be obtained with just one query.

```
[6]: actions = session.scalars(select(pycram.orm.action_designator.
       ↪Action)).all()
     print(*actions, sep="\n")
```

```
ParkArmsAction(id=1, process_metadata_id=1, dtype='ParkArmsAction',
robot_state_id=1, robot_state=RobotState(id=1, pose_to_init=False,
torso_height=0.0, type=<ObjectType.ROBOT: 8>, pose_id=1, process_metadata_id=1),
arm=<Arms.BOTH: 3>)
MoveTorsoAction(id=2, process_metadata_id=1, dtype='MoveTorsoAction',
robot_state_id=2, robot_state=RobotState(id=2, pose_to_init=False,
torso_height=0.0, type=<ObjectType.ROBOT: 8>, pose_id=2, process_metadata_id=1),
position=0.3)
NavigateAction(id=3, process_metadata_id=1, dtype='NavigateAction',
robot_state_id=3, robot_state=RobotState(id=3, pose_to_init=False,
torso_height=0.3, type=<ObjectType.ROBOT: 8>, pose_id=3, process_metadata_id=1),
pose_to_init=False, pose_id=4)
PickUpAction(id=5, process_metadata_id=1, dtype='PickUpAction',
robot_state_id=4, robot_state=RobotState(id=4, pose_to_init=False,
torso_height=0.3, type=<ObjectType.ROBOT: 8>, pose_id=6, process_metadata_id=1),
object_to_init=False, arm='left', grasp='front', object_id=1)
ParkArmsAction(id=11, process_metadata_id=1, dtype='ParkArmsAction',
robot_state_id=5, robot_state=RobotState(id=5, pose_to_init=False,
torso_height=0.3, type=<ObjectType.ROBOT: 8>, pose_id=11,
process_metadata_id=1), arm=<Arms.BOTH: 3>)
NavigateAction(id=12, process_metadata_id=1, dtype='NavigateAction',
robot_state_id=6, robot_state=RobotState(id=6, pose_to_init=False,
torso_height=0.3, type=<ObjectType.ROBOT: 8>, pose_id=12,
process_metadata_id=1), pose_to_init=False, pose_id=13)
PlaceAction(id=14, process_metadata_id=1, dtype='PlaceAction', robot_state_id=7,
robot_state=RobotState(id=7, pose_to_init=False, torso_height=0.3,
type=<ObjectType.ROBOT: 8>, pose_id=15, process_metadata_id=1),
object_to_init=False, pose_to_init=False, arm='left', pose_id=17, object_id=2)
ParkArmsAction(id=18, process_metadata_id=1, dtype='ParkArmsAction',
robot_state_id=8, robot_state=RobotState(id=8, pose_to_init=False,
torso_height=0.3, type=<ObjectType.ROBOT: 8>, pose_id=20,
process_metadata_id=1), arm=<Arms.BOTH: 3>)
```

Of course all relational algebra operators, such as filtering and joining also work in ORM queries. An example would be to query all the poses of objects, that were picked up by a

robot. Since a relationship between the *PickUpAction* table and the *Object* table and between the *Object* table and the *Pose* table in *PyCRORM* class schema is defined, a join can be used between these tables connected by relationship objects.

[7]:
```
object_actions = (session.scalars(select(pycram.orm.base.Pose)
                      .join(pycram.orm.action_designator.PickUpAction.
  ↪object)
                      .join(pycram.orm.object_designator.Object.pose))
                      .all())
print(*object_actions, sep="\n")
```

```
Pose(id=7, orientation_to_init=False, position_to_init=False,
time=datetime.datetime(2024, 6, 25, 11, 35, 54, 249805),
  ↪frame='map',
position_id=7, orientation_id=7, process_metadata_id=1)
```

It can be observed, that the joins were not done in a typical sql kind of way. The relationship objects defined in *PyCRORM*'s classes were used to join directly on the objects hold in these attributes, written like *PickUpAction.object* or *Object.pose*. This works because *SQLAlchemy* enables *PyCRORM* to automatically create the joins, so it is only necessary to join on the attributes that hold the relationship. This is a huge advantage over writing sql queries by hand, since join conditions do not have to be configured manually.

Now that inserting and querying data is shown to work as expected and desired, a last section shows the creation of a new designator in *PyCRORM* package. This is also done with ease. To create a new action designator, a mapping for the class and an ORM mapper-class are needed. A new action, that logs what a robot is saying could look like this:

[8]:
```
from sqlalchemy.orm import Mapped, mapped_column, Session
from pycram.orm.action_designator import Action
from dataclasses import dataclass


# define ORM class from pattern in every pycram.orm class
class ORMSaying(Action):

    id: Mapped[int] = mapped_column(sqlalchemy.
  ↪ForeignKey(f'{Action.__tablename__}.id'), primary_key=True,
  ↪init=False)
    # since we do not want to add any custom specifications to our
  ↪column, we don't even need to define mapped_column, sqlalchemy
  ↪does this internally.
```

```
    text: Mapped[str]

# define brand new action designator

@dataclass
class SayingActionPerformable(ActionAbstract):

    text: str
    orm_class = ORMSaying

    @with_tree
    def perform(self) -> None:
        print(self.text)
```

The *ORM* mapper-class contains the columns needed for the table. The actual designator inherits from the *ActionAbstract* and thus receives automatic mappings, leading to not having to create mappings manually.

Since this class got created after all the other classes got inserted into the database it has to be inserted manually.

[9]:
```
ORMSaying.metadata.create_all(bind=engine)
```

Now, a *SayingAction* can be created and inserted. Since the *BulletWorld* is no longer needed, it can be closed.

[10]:
```
# create a saying action and insert it
SayingActionPerformable("Patchie, Patchie; Where is my Patchie?").
  ↪perform()
pycram.tasktree.task_tree.root.insert(session)
session.commit()

world.exit()
```

```
Patchie, Patchie; Where is my Patchie?

Inserting TaskTree into database: 100%|██████████| 21/21 [00:00<00:
  ↪00,
1102.80it/s]
```

One thing to note is that committing the object to the session fills its primary key. Hence, there is no worries about assigning unique IDs manually. Finally, the data quality and correctness can be checked in the database.

```
[11]: session.scalars(select(ORMSaying)).all()
```

```
[11]: [ORMSaying(id=37, process_metadata_id=1, dtype='ORMSaying',
        ↪robot_state_id=17,
      robot_state=RobotState(id=17, pose_to_init=False, torso_height=0.3,
      type=<ObjectType.ROBOT: 8>, pose_id=41, process_metadata_id=1),
        ↪text='Patchie,
      Patchie; Where is my Patchie?')]
```

This example showed some capabilities of *PyCRORM*, including the setup, the execution of a plan, the subsequent querying and the creation of a new designator. All these things were done with ease without the need to have much knowledge about databases in general. When looking at the functional requirements defined in Chapter 4.2 it can be observed that this demo alone shows how well goals 1. through 4. are fullfilled. It provided a great insight into the usage and acceled at relationship capturing, accessibility and user-experience. This

5.2 Learning Demo

However, one of the research questions defined in Chapter 1 of this thesis was „Does this memory component enable learning capabilities within *PyCRAM*?". This question has not been discussed at all. Now that the memory component has been implemented and functions according to the specified requirements, such as relationship capturing, we can explore its potential for developing learning capabilities. Consequently, a demonstration is presented below, where an agent tries to move and pick up an object in its environment. This demonstration is a segment of a more comprehensive example available in the *PyCRAM* documentation[1]. The demonstration, along with the internal learning mechanisms in *PyCRAM*, was developed by Tom Schierenbeck of the *IAI* using *PyCRORM* for data storage and retrieval.

PC specifications:

- **CPU:** 11th Gen Intel Core i7-11700 @ 2.50GHz x 16
- **GPU:** AMD Radeon RX 6700XT
- **Memory:** 32GB (2x16) DDR 4 @ 3200MT/s

The initial step involves constructing the world, the robot, the object to be picked up (in this case, a bottle of milk), and initiating an *ORM* session. It is important to note that the constructed world consists solely of the robot and the object, and does not represent a complex

[1] https://pycram.readthedocs.io/en/latest/notebooks/improving_actions.html

environment such as the kitchen environment used in the previous demonstration.

Following this, a probabilistic model can be established to describe the processes involved in moving toward and picking up objects. The default policy employed in this model, which attempts to pick up an object while maintaining an optimal distance, is illustrated in Figure 5.1. Testing this default policy on a dataset of 50,000 samples resulted in a modest success rate of 15.8%.

```
[1]: pycram.orm.base.ProcessMetaData().description = "Experimenting␣
       ↪with Pick Up Actions"
     model.sample_amount = 50000

     with simulated_robot:
         model.batch_rollout()

     task_tree.insert(session)
     session.commit()
```

```
[INFO] [1723034017.823800]: Waiting for IK service:
/pr2_right_arm_kinematics/get_ik
100%|██████████| 50000/50000 [1:20:41<00:00, 10.33it/s, Success
Probability=0.158]
Inserting TaskTree into database: 100%|██████████| 488043/488043␣
 ↪[06:19<00:00,
1285.39it/s]
```

Figure 5.1 Marginal View of Relative x and y Position of the Robot with respect to the Object

This also resembles a great example of data-intensive inserting into *PyCRORM*. The *TaskTree*, executing 50,000 pick-up tasks using the default policy took 80 minutes and 41 seconds to finish up all 50,000 nodes, achieving 10.33 iterations per second. Inserting the *TaskTree* into the database took 06 minutes and 19 seconds for 488,043 nodes, with 1,285.39 iterations per second. So even though inserting all the data took around 6.5 minutes, it is nothing compared to the general speeds of the plans. It can be observed, that performance-wise in terms of inserting speeds, *PyCRORM* uses its potential and does not take up much additional time.

The outcome of the default policy is stored as episodes in *PyCRORM*. Under the hood, it

uses a *View* to store all essential columns combined into one virtual table, since data from multiple different tables was needed. All successful samples can be queried by ensuring that *view.status == TaskStatus.SUCCEEDED* holds, with *view.status* being the *TaskTree* table's *status* column. If the node succeeds, the robot in this sample is in a position to pick up the object.

```
[2]: samples = pd.read_sql(model.query_for_database(), engine)
     samples
```

```
[2]:         arm  grasp  relative_x  relative_y
     0     right   left   -0.428668    0.434859
     1      left   left   -0.522978    0.300486
     2      left   left   -0.215690    0.664172
     3      left  right   -0.260081   -0.522400
     4     right  right   -0.373218   -0.606202
     …       …     …          …           …
     7881  right   left   -0.171496    0.767967
     7882   left  front    0.075736    0.355137
     7883  right   left    0.037081    0.736527
     7884  right  front    0.111385    0.615652
     7885  right  front    0.190658    0.596994

     [7886 rows x 4 columns]
```

To improve the success rate of the model, a better policy is needed. Therefore, a new policy needs to be learned. Using existing data in *PyCRORM*, a new model that can be used as a new policy can be learned with the *JPT* (Joint Probability Trees) learning algorithm. Since the focus of this thesis is not on how learning algorithms work, this will not be explained further.

Assigning the new policy to the model and then trying this policy leads to a new success rate of 92.8%, a huge increase over the default policy, providing great results.

```
[3]: model.policy = learned_model
     model.sample_amount = 50000

     with simulated_robot:
         model.batch_rollout()
```

```
100%|██████████| 50000/50000 [2:31:44<00:00,  5.49it/s, Success
Probability=0.928]
```

This demo has shown a practical example of a simple learning mechanism utilizing *PyCRORM* to achieve the agent's goals. The agent improved its success rate by a lot, without explicitly getting told where the object or the robot stands. It used the *ORM*, to learn from previous

episodes to improve success rate. This provides a great answer to the second research question defined in Chapter 1. *PyCRORM* can consequently be used to achieve learning tasks in *PyCRAM*, providing a great new possibility in research on cognitive architectures.

5.3 Evaluating data-intensive applications

Throughout this thesis, *NEEMs* have been identified to be the predecessors of *PyCRORM* in terms of episodic memory within *PyCRAM*. As the new memory component is intended to replace *NEEMs* in PyCRAM, it is essential to compare the functionality and characteristics of *PyCRORM* with *NEEMs* to ensure that transitioning to this new episodic memory is justified. Both *PyCRORM* and *NEEMs* are examples of data-intensive applications, which inherently possess specific requirements and characteristics. Based on the widely accepted definitions provided in Martin Kleppmann's text, „Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems“[Kle17] data-intensive applications can be analyzed and evaluated based on three primary features: *Reliability*, *Scalability*, and *Maintainability*.

5.3.1 Reliability

Reliability describes data integrity and safety, particularly in the event of human errors, software glitches, or hardware malfunctions. Reliability does not imply that every operation consistently yields valid, usable results. Instead, it emphasizes that when faults occur or the system is incorrectly used, robust error handling intervenes to bring the system to a secure state [Kle17]. The greater goal is to provide a system capable of storing and processing valid data.

When evaluating the reliablity of *PyCRORM* and *NEEMs* within *PyCRAM*, it must be differentiated between the reliability of the database, and the reliability of the mapper or, in the case of *NEEMs*, the interface within *PyCRAM*, and its associated backend.

Both the mapper and the interface are designed to utilize *PyCRAM*'s plan language and store its outputs in their respective databases. Consequently, in the event of execution faults, the plan language error handling commences, making this a *PyCRAM* issue. However, data in the *NEEMs* database is not exclusive to *PyCRAM*. Many *NEEMs* are created within the *CRAM* framework and do not hold any data from *PyCRAM*, making it hard to ensure data integrity when querying the database. Currently, there is no mechanism to query or filter *PyCRAM*-specific *NEEMs* exclusively.

Additionally, differences between *PyCRORM* and *NEEMs* are evident when examining the data received from plans. *PyCRORM* follows a straightforward structure based on inheritance, allowing developers familiar with *PyCRAM*'s plan language to grasp its functionality easily. This design simplifies the verification of correct behavior, as the functionality must be defined only once. *PyCRORM* relies on the *SQLAlchemy* library, which integrates its own security and

safety measures. These measures include verifying that the data inserted into the database are equal to the data types specified in the mapper and ensuring the validity of joins. Currently, *PyCRORM* utilizes *MariaDB* as its relational database management system (*RDBMS*), leading to ACID compliance (3.3) and therefore maintaining data integrity, safety, and reliability.

Furthermore, a crucial component of both memory systems is relationship capturing. Relational databases, with their primary and foreign key features, are ideally suited to ensure accurate relationship patterns. *NEEMs* lack this capability due to their chosen approach. Studies indicate that, despite excelling in other areas, *NoSQL* databases struggle with proper relationship capturing, making *PyCRORM* the superior choice in this critical aspect [KP17].

In conclusion, since *PyCRORM* was designed directly for *PyCRAM*, ensuring reliability is a feasible task. Since it is a mapper, it can use *PyCRAM*'s, *SQLAlchemy*'s, and the *RDBMS*'s safety and reliability features, making it highly reliable overall. The database is only filled with *PyCRAM* plans.

NEEMs on the other hand, were not designed for *PyCRAM* originally and the codebase within the backend is much larger, containing lots of data of other applications, which are not easily separable. While *DBMS* bring the same database standard to both memories, The *ORM*'s relationship-capturing mechanisms outweigh the *NEEM*'s. Its larger internal size and it's usage in different applications make it also harder to ensure the *NEEM*'s reliability.

5.3.2 Scalability

When developing a data-intensive application, it is crucial to provide optimal performance in terms of insertion speeds, query speeds, and scalability in general [Kle17].

Therefore, comparing *NEEMs* and *PyCRORM* in terms of performance and scalability is important. Evaluating and comparing both memories is best done by using the same data in each respective memory. However, *NEEMs* can only be created by developers of the *NEEM* project. Thus, executing a *PyCRAM* plan that gets stored both as a *NEEM* and as an entry in *PyCRORM* can not be done without much more effort.

However, it is possible to look at the general types of databases behind both memories and do a qualitative comparison of *NEEMs* and *PyCRORM* based on these types. *NEEMs* use a *NoSQL* approach featuring a document-oriented database with *MongoDB*. *PyCRORM* uses a relational database.

As outlined in Chapter 3.3, *NoSQL* databases are often favored over relational databases when flexibility and scalability in structure are primary requirements, and relationship modeling is less critical. The reason for that is the internal structure of the database and the ability to modify it. With exponential amounts of data being processed by the database, *NoSQL* databases shine in being able to scale out (horizontal scaling), which means they can scale by adding more machines or nodes to a system to handle the increased load. Relational databases traditionally were limited to only scale up (vertical scaling), which means that in order to handle increased loads of data, relational databases could only improve performance

by adding more resources to the system, e.g. more *RAM*, or better *CPU*. This was an expensive and often impractical approach [MAI14]. Nowadays, modern *RDBMS* provide features to also achieve horizontal scaling in relational databases [Kle17].

When looking at performance, none of the types can be characterized as the faster approach. Depending on the goals, one might be chosen over the other. Studies like [LM13] have found, that between different *NoSQL* databases, some were more performant than a relational database in terms of reading, writing, and deleting data, while others were less performant. In the case of *NEEMs*, a comparison between *NEEMs* and an *SQL* equivalent of the *NEEMs* concluded, that when querying all gripping actions executed in a *NEEM*, the *ORM* query was faster than the *Mongo* query and a general *SQL* query faster than the *ORM* query [Bas24a].
Queries that have proven more performant for *NEEMs* over *PyCRORM* were queries on specific episodes. *NEEMs* with their structured documents store all information related to an episode in the same place, i.e. having a great data locality, as described in Chapter 5.3.3.2. Relational databases with their table structure, split data connected to one episode into different tables, and therefore contain lots of data from different episodes in the same tables. So to query data from a certain episode, *PyCRORM* might have to use multiple joins and iterate through lots of tables that also contain data not needed for the query. A query on *NEEMs*, on the other hand, only needs to access the documents connected to the episode, ignoring the rest of the database. This might lead to faster retrieval times for this kind of query. However, machine learning is usually done on lots of data, making queries on single episodes rather rare.

To conclude this section, *NEEMs* generally do not offer superior scalability, even though they use a *NoSQL* approach. In terms of performance a general advantage of one system over the other can not be concluded. Looking at data retrieval performance, some types of queries might favor *NEEMs*, some might favor *PyCRORM*.

5.3.3 Maintainability

Though scalability and reliability are crucial in data-intensive systems, maintainability is also critical. Of what use is a system, if it can not be understood by new engineers and it can't be changed without further complications, and adapt to future tasks and requirements? Especially when working in a team-based environment, it is important to keep maintenance cost as low as possible by still providing a system that fullfills all the requirements and is easily adaptable to changes in other departments.

Therefore, a well-maintained system should contain the following characteristics: 1. *operable*, 2. *simple* and 3. *evolvable* [Kle17].

5.3.3.1 Operability

Operability describes the capability of keeping a system smooth and running without having to put too many people and too much effort and therefore, time into the task. Keeping the system healthy and up to date is critical.

As for *PyCRORM*, operability is provided rather easily. Since its only purpose is to map *PyCRAM*'s environment into a database, keeping the system up to date can be easily done.

As long as *PyCRAM*'s internal structure doesn't change, and it remains object-oriented, *PyCRORM* is flexible and reliable enough to handle changes and keep up operability without any support. It is currently maintained by a single person, and does not need advanced attention. Error handling, as described in 5.3.1 is informative and straightforward and the code base is well documented, also providing good visibility into the runtime behavior. It is well documented and achieves a test coverage of more than 90%, and therefore aligns with modern software practices [MD14].

Since *NEEMs* are used, and have interfaces in multiple different systems primarily throughout the EASE software ecosystem, it is much harder to operate, also due to its comparably much larger codebase. Though the *PyCRAM* interface is not large and thus not hard to maintain and to keep up to date, operating the internal structure with the *NEEM Background*, the *NEEM Narrative* and the *NEEM Experience* and ensuring that changes of internal functionality within *NEEMs* also work in the case of *PyCRAM* is much harder. The *NEEMs* project follows a much larger goal and is not fitted primarily for *PyCRAMs* purposes, leading to complicated operability.

5.3.3.2 Simplicity

A system is considered simple if the code is expressive, there are no tangled dependencies, and terminology is consistent. Good documentation can also help with simplicity. Of course, smaller projects are usually more simple by default [Kle17].

This means that *PyCRORM* is much simpler to use by default, due to its size. *NEEMs* come with much-advanced mechanisms and details, making them preferable in some applications with certain requirements like storage of sensor information, but it makes them much more complex than *PyCRORM*.

Also, *PyCRORM* was designed to be user-friendly. To be able to use it, the user does not have to know much about the internal structure and does not even need knowledge about the querying language, as is shown in Chapter 5.1 and Chapter 5.2. To use *NEEMs*, the user needs advanced knowledge about the system and also needs to fulfill a whole checklist, as described in Chapter 7 of [Bee+20].

5fc8ff968f880006aa208e19_triples > s

_id	graph	s	p	o	o*	scope	p*
5fdcdaa7b3e25a241f04a1f8	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#actsThrough	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#ObjectProperty	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a1f9	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#actsThrough	http://www.w3.org/2000/01/rdf-schema#subPropertyOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	[4 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a1fa	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#actsThrough	http://www.w3.org/2000/01/rdf-schema#domain	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#SocialAgent		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a1fd	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#actsThrough	http://www.w3.org/2000/01/rdf-schema#isDefinedBy	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a1fe	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#actsThrough	http://www.w3.org/2000/01/rdf-schema#label	lang(en,'acts through')		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a1ff	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#actsThrough	http://www.w3.org/2000/01/rdf-schema#label	lang(it,'agisce mediante')		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a200	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#ObjectProperty	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a201	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	http://www.w3.org/2002/07/owl#inverseOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a202	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#SymmetricProperty	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a203	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#TransitiveProperty	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a204	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	http://www.w3.org/2000/01/rdf-schema#domain	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Entity		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a205	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	http://www.w3.org/2000/01/rdf-schema#range	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Entity		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a208	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	http://www.w3.org/2000/01/rdf-schema#label	associatedWith		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a209	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#characterizes	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#ObjectProperty	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a20b	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#characterizes	http://www.w3.org/2002/07/owl#inverseOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#isCharacterize		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a20c	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#characterizes	http://www.w3.org/2000/01/rdf-schema#domain	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Concept		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a20d	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#characterizes	http://www.w3.org/2000/01/rdf-schema#range	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Collection		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a20f	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#characterizes	http://www.w3.org/2000/01/rdf-schema#isDefinedBy	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a210	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#characterizes	http://www.w3.org/2000/01/rdf-schema#label	lang(it,caratterizza)		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a211	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#characterizes	http://www.w3.org/2000/01/rdf-schema#label	lang(en,characterizes)		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a222	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#classifies	http://www.w3.org/2000/01/rdf-schema#domain	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Concept		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a23e	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#concretelyExp	http://www.w3.org/2000/01/rdf-schema#subPropertyOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a23f	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#concretelyExp	http://www.w3.org/2002/07/owl#inverseOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#isConcretelyE		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a240	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#concretelyExp	http://www.w3.org/2000/01/rdf-schema#domain	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#InformationRe		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a241	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#concretelyExp	http://www.w3.org/2000/01/rdf-schema#range	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#SocialObject		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a243	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#concretelyExp	http://www.w3.org/2000/01/rdf-schema#isDefinedBy	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a244	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#concretelyExp	http://www.w3.org/2000/01/rdf-schema#label	lang(en,'concretely expresses')		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a245	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#concretelyExp	http://www.w3.org/2000/01/rdf-schema#label	lang(it,'esprime concretamente')		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a246	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipate	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#ObjectProperty	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a247	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipate	http://www.w3.org/2000/01/rdf-schema#subPropertyOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#associatedWi	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a248	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipate	http://www.w3.org/2002/07/owl#inverseOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipate		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a249	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipate	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#SymmetricProperty	[1 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a24a	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipate	http://www.w3.org/2000/01/rdf-schema#domain	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Object		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a24b	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipate	http://www.w3.org/2000/01/rdf-schema#range	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Object		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a258	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#covers	http://www.w3.org/2000/01/rdf-schema#label	lang(it,ricopre)		{ 1 fields }	[1 elements]
5fdcdaa7b3e25a241f04a263	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#definesRole	http://www.w3.org/2000/01/rdf-schema#subPropertyOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#defines	[4 elements]	{ 1 fields }	
5fdcdaa7b3e25a241f04a276	DUL	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#describes	http://www.w3.org/2002/07/owl#inverseOf	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#isDescribedBy		{ 1 fields }	[1 elements]

Figure 5.2 Example of data stored in a NEEM's _triples document

When looking at the database and the simplicity of the data itself, *PyCRORM* is much easier to understand and use. Like all relational databases, *PyCRORM* provides a clear table structure with expressive names in both the table itself and its columns, which provide an easily understandable data structure. On the other hand, the *NEEMs* being a document-oriented database, are not as simple to understand. Every *NEEM* consists of four *JSON*-like documents stored in the database, with its name being its idea. It is not expressive, and data in these documents is not easily understandable. For instance, the *NEEM* with the id 5fc8ff968f880006aa208e19 provides four documents, their names being 5fc8ff968f880006aa208e19_annotations, 5fc8ff968f880006aa208e19_inferred, 5fc8ff968f880006aa208e19_tf, 5fc8ff968f880006aa208e19_triples. These documents contain data using *JSON* syntax. Looking at the 5fc8ff968f880006aa208e19_triples document, when transforming the raw data into a table view, which best compares to *PyCRORM* structure, the data is still hard to understand without knowledge about the *NEEM's* triples system and ontological structures. An excerpt of the *NEEM's* data stored in the _triples can be seen in Figure 5.2.

```python
from sqlalchemy import select
from pycram.orm.action_designator import GripAction
from pycram.orm.base import ProcessMetaData


def get_gripping_action_orm(session, plan_id):
    return session.scalars(select(GripAction).
    ↪where(ProcessMetaData.id==plan_id)).all()
```

Figure 5.3 *ORM* Query which gets all Gripping Actions from a plan

However, since simplicity in the usability of the episodic memory component is a key requirement, the internal structure of the data in the database might not be as important, as long as the access and therefore querying shines in ease of use.

Looking at a query that collects all the gripping actions executed in an episode (in a plan), it can be observed, that querying *PyCRORM* is much more intuitive and also takes less code to compute this query. The *ORM* query can be seen in Figure 5.3, *NEEM* query in Figure 5.4.

Simplicity primarily leads to improved usability and accessibility, which represent key requirements for the episodic memory. With *PyCRORM* being much simpler in multiple regards as described above, based only on user experience and ease of use, *PyCRORM* outweighs the *NEEMs* by a lot.

5.3.3.3 Evolvability

System requirements and purposes may change over time. Therefore, a system needs to be able to evolve to its changing requirements.

In the case of *PyCRORM*, it can easily adapt to any future requirements as long *PyCRAM*'s general object-oriented approach persists. Due to the inheritance pattern and the automatical mapping via setting an attribute of the class leads to the capability of easily adding new designator or other objects without the need to change the overall structure. As for the *NEEMs*,

```
def get_gripping_action_neem(neem_id) -> List[Dict]:
    return [{"$match": {"p": "http://www.ontologydesignpatterns.
  ↪org/ont/dul/DUL.owl#executesTask"}},
            {
                "$lookup":
                    {
                        "from": f"{neem_id}_triples",
                        "localField": "o",
                        "foreignField": "s",
                        "as": f"{neem_id}"
                    }
            },
            {"$match": {f'{neem_id}.p': 'http://www.w3.org/1999/02/
  ↪22-rdf-syntax-ns#type',
                        f'{neem_id}.o': 'http://www.ease-crc.org/
  ↪ont/SOMA.owl#Gripping'}},
            {"$unwind": f"${neem_id}"},
            {
                "$project": {
                    f"{neem_id}.s": 1,
                    "_id": 0
                }
            }]
```

Figure 5.4 *NEEM* Query which gets all Gripping Actions from a *NEEM* [Bas24a]

changes in requirements both in themselves and in *PyCRAM* may lead to further complications. If the structure and requirements of the *NEEMs* change over time, they may not be suitable for *PyCRAM* anymore. The fact, that the *NEEMs* in their database are primarily created by *CRAM* which might have other future requirements and goals than *PyCRAM*, leading to differences in the *NEEMs* itself, does not help this case. The other way around, adapting to new requirements in *PyCRAM* might not be as big of a problem for the *NEEMs* but may lead to necessary changes to the interface, depending on the form of adaption.

Chapter 6

Conclusion

This thesis introduced *PyCRORM*, an object-relational approach to episodic memory within the cognitive architecture *PyCRAM*.

The thesis followed a clear structure, beginning by discussing *PyCRAM* as a cognitive architecture, emphasizing the critical role of episodic memory. It identified limitations within the existing memory component (*NEEMs*) and outlined *PyCRAM*'s requirements for knowledge acquisition and representation. These insights paved the way for introducing the new relational approach, *PyCRORM*.

Subsequently, the implementation of *PyCRORM* was explained, accompanied by practical examples to illustrate its functionality. The examples demonstrated its user-friendly nature, emphasizing the ease of querying in Python without requiring extensive knowledge of *SQL*. This aspect addressed a key limitation of *NEEMs*, which demanded familiarity with *NoSQL* syntax and the complex structure of *NEEMs*.

A comparative analysis between the old memory component and the new approach highlighted the potential of *PyCRORM*, offering a clear overview of its advantages over *NEEMs* within *PyCRAM*. This comparison justified *PyCRORM*'s adoption as *PyCRAM*'s episodic memory system.

The conclusion of this thesis provided comprehensive answers to the research questions outlined in Chapter 1 *PyCRORM* successfully meets the requirements defined in Chapter 4.2, presenting a user-friendly and accessible memory system suitable for diverse applications. As demonstrated in Chapter 5.2, *PyCRORM*'s performance extends to learning purposes within the cognitive architecture, fulfilling the primary motivation for developing a new memory component.

As of writing this conclusion, *PyCRORM* has replaced *NEEMs* in *PyCRAM* and is actively employed in research, serving as its episodic memory system.

6.1 Future Work

Although *PyCRORM* functions effectively, there are opportunities for further improvements. *PyCRORM* captures all relevant episodic information, yet *PyCRAM* stores additional data types, such as sensory information and ontologies, in separate systems. Integrating these data types into *PyCRORM*'s relational database is a current task under development, aiming to centralize information storage.

Additionally, optimizing the relational database's table structures presents an opportunity for future work. Currently, due to *PyCRAM*'s learning requirements, the tables do not obey to "normal-form" standards, resulting in the tables and data not being minimal. Transforming these structures to conform with normal-form standards could improve performance, making this an intriguing area for future research.

In summary, *PyCRORM* stands as a significant advancement in episodic memory systems within *PyCRAM*. While it fulfills current needs, continued development and exploration of future work will further expand its capabilities, reinforcing its role in advancing cognitive robotics research.

Bibliography

[AB14] John R. Anderson and Gordon H. Bower. *Human associative memory*. Psychology press, 2014 (cit. on p. 6).

[ACa] SQLAlchemy Authors and Contributors. *Engine Configuration — SQLAlchemy 2.0 Documentation*. URL: *https://docs.sqlalchemy.org/en/20/core/engines.html* (visited on 07/10/2024) (cit. on p. 24).

[ACb] SQLAlchemy Authors and Contributors. *Features - SQLAlchemy*. URL: *https://www.sqlalchemy.org/features.html* (visited on 07/17/2024) (cit. on p. 15).

[ACc] SQLAlchemy Authors and Contributors. *Session Basics — SQLAlchemy 2.0 Documentation*. URL: *https://docs.sqlalchemy.org/en/20/orm/session_basics.html* (visited on 07/10/2024) (cit. on p. 24).

[ACd] SQLAlchemy Authors and Contributors. *State Management — SQLAlchemy 2.0 Documentation*. URL: *https://docs.sqlalchemy.org/en/20/orm/session_state_management.html* (visited on 07/10/2024) (cit. on p. 24).

[Bas24a] Abdelrhman Bassiouny. *neem_to_sql Github repository*. 2024. URL: *https://github.com/AbdelrhmanBassiouny/neem_to_sql/blob/master/src/mongo_vs_sql_query.py* (cit. on pp. 42, 45).

[Bas24b] Abdelrhman Bassiouny. "Towards Bigdata in Robotics: Machine Learning Pipeline for Robot NEEMs (Narrative-Enabled Episodic Memories) in an SQL Database". 2024 (cit. on pp. 2, 6).

[Bee+20] Michael Beetz, Daniel Beßler, Sebastian Koralewski, Mihai Pomarlan, Abhijit Vyas, Alina Hawkin, Kaviya Dhanabalachandran, and Sascha Jongebloed. "Neem handbook". In: (2020) (cit. on pp. 2, 5, 43).

[BMT10] Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. "CRAM — A Cognitive Robot Abstract Machine for everyday manipulation in human environments". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 1012–1017. DOI: *10.1109/IROS.2010.5650146* (cit. on pp. 5, 11).

[BK11] Momotaz Begum and Fakhri Karray. "Visual Attention for Robotic Cognition: A Survey". In: *IEEE Transactions on Autonomous Mental Development* 3.1 (2011), pp. 92–105. DOI: *10.1109/TAMD.2010.2096505* (cit. on p. 9).

[CB05] Thomas M. Connolly and Carolyn E. Begg. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005 (cit. on p. 14).

[CP84] Stavros S. Cosmadakis and Christos H. Papadimitriou. "Updates of Relational Views". In: *J. ACM* 31.4 (Sept. 1984), pp. 742–760. ISSN: 0004-5411. DOI: *10.1145/1634.1887*. URL: *https://doi.org/10.1145/1634.1887* (cit. on p. 16).

[Dec19] Jonas Dech. "PyCRAM - Accurate Physics-based Environment for Executing Mobile Pick and Place Plans". 2019. URL: *https://cram-system.org/_media/jonas_dech_bsc.pdf* (cit. on pp. 11, 18).

[Dec] Jonas Dech. *Welcome to pycram's documentation! — pycram documentation*. URL: *https://pycram.readthedocs.io/en/latest/* (visited on 07/20/2024) (cit. on pp. 1, 10).

[Foo13] Tully Foote. "tf: The transform library". In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. Apr. 2013, pp. 1–6. DOI: *10.1109/TePRA.2013.6556373* (cit. on p. 18).

[FMM] Tully Foote, Eitan Marder-Eppstein, and Wim Meeussen. *ROS TF package homepage*. URL: *https://wiki.ros.org/tf* (visited on 07/04/2024) (cit. on p. 18).

[Gal13] Daniel Gall. "A rule-based implementation of ACT-R using Constraint Handling Rules". PhD thesis. Master Thesis, Ulm University, 2013 (cit. on p. 6).

[Jat+12] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. "A survey and comparison of relational and non-relational database". In: *International Journal of Engineering Research & Technology* 1.6 (2012), pp. 1–5 (cit. on p. 15).

[Kle17] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, Inc., 2017. ISBN: 9781491903100 (cit. on pp. 40–43).

[KT20] Iuliia Kotserub and John K. Tsotsos. "40 years of cognitive architectures: core cognitive abilities and practical applications". In: *Artificial Intelligence Review* 53 (2020), pp. 17–94. DOI: *https://doi.org/10.1007/s10462-018-9646-y* (cit. on pp. 1, 5, 8, 10).

[Küm24] Michaela Kümpel. "Actionable Knowledge Graphs-how daily activity applications can benefit from embodied web knowledge". PhD thesis. Universität Bremen, 2024 (cit. on p. 9).

[KP17] Douglas Kunda and Hazael Phiri. "A Comparative Study of NoSQL and Relational Database". In: *Zambia ICT Journal* 1.1 (Dec. 2017), pp. 1–4. DOI: *10.33260/zictjournal.v1i1.8*. URL: *https://ictjournal.icict.org.zm/index.php/zictjournal/article/view/8* (cit. on p. 41).

[LLR09] Pat Langley, John E. Laird, and Seth Rogers. "Cognitive architectures: Research issues and challenges". In: *Cognitive Systems Research* 10.2 (2009), pp. 141–160. ISSN: 1389-0417. DOI: *https://doi.org/10.1016/j.cogsys.2006.07.004*. URL: *https://www.sciencedirect.com/science/article/pii/S1389041708000557* (cit. on p. 8).

[Leó16] Carlos León. "An architecture of narrative memory". In: *Biologically Inspired Cognitive Architectures* 16 (2016), pp. 19–33. ISSN: 2212-683X. DOI: *https://doi.org/10.1016/j.bica.2016.04.002*. URL: *https://www.sciencedirect.com/science/article/pii/S2212683X16300184* (cit. on p. 10).

[LM13] Yishan Li and Sathiamoorthy Manoharan. "A performance comparison of SQL and NoSQL databases". In: *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. 2013, pp. 15–19. DOI: *10.1109/PACRIM.2013.6625441* (cit. on p. 42).

[LSB14] John Licato, Ron Sun, and Selmer Bringsjord. "Structural representation and reasoning in a hybrid cognitive architecture". In: *2014 International Joint Conference on*

Neural Networks (IJCNN). 2014, pp. 891–898. DOI: *10.1109/IJCNN.2014.6889895* (cit. on p. 9).

[LGL15] Martyn Lloyd-Kelly, Fernand Gobet, and Peter CR Lane. "Piece of Mind: Long-Term Memory Structure in ACT-R and CHREST." In: *CogSci*. 2015 (cit. on p. 6).

[MD14] Robert C. Martin and Jürgen Dubau. *Clean Coder: Verhaltensregeln für professionelle Programmierer*. Vol. 1. mitp Verlags GmbH & Co. KG, 2014 (cit. on p. 43).

[MAI14] Mohamed A. Mohamed, Obay G. Altrafi, and Mohammed O. Ismail. "Relational vs. nosql databases: A survey". In: *International Journal of Computer and Information Technology* 3.03 (2014), pp. 598–601 (cit. on p. 42).

[NPP13] Ameya Nayak, Anil Poriya, and Dikshay Poojary. "Type of NOSQL databases and its comparison with relational databases". In: *International Journal of Applied Information Systems* 5.4 (2013), pp. 16–19 (cit. on p. 2).

[Oli+19] Alberto Olivares-Alarcos, Daniel Beßler, Alaa Khamis, Paulo Goncalves, Maki K. Habib, Julita Bermejo-Alonso, Marcos Barreto, Mohammed Diab, Jan Rosell, João Quintas, and et al. "A review and comparison of ontology-based approaches to robot autonomy". In: *The Knowledge Engineering Review* 34 (2019), p. 29. DOI: *10.1017/S0269888919000237* (cit. on p. 9).

[PPJ17] Robert Poljak, Patrizia Poščić, and Danijela Jakšić. "Comparative analysis of the selected relational database management systems". In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2017, pp. 1496–1500. DOI: *10.23919/MIPRO.2017.7973658* (cit. on p. 14).

[RSS12] Mathis Richter, Yulia Sandamirskaya, and Gregor Schöner. "A robotic architecture for action selection and behavioral organization inspired by human cognition". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 2457–2464. DOI: *10.1109/IROS.2012.6386153* (cit. on p. 8).

[RN21] Stuart Russell and Peter Norvig. *Artificial Intelligence, Global Edition A Modern Approach*. Pearson Deutschland, 2021, p. 1168. ISBN: 9781292401133. URL: *https://elibrary.pearson.de/book/99.150005/9781292401171* (cit. on pp. 1, 8).

[SK11] Christof Strauch and Walter Kriha. "NoSQL databases". In: *Lecture Notes, Stuttgart Media University* 20.24 (2011), p. 79 (cit. on p. 14).

[Su+16] Yun Su, Xiaowei Zhang, Philip Moore, Jing Chen, Xu Ma, and Bin Hu. "Declarative and procedural knowledge modeling methodology for brain cognitive function analysis". In: *in Science Robotics* (2016), p. 50 (cit. on p. 10).

[Tha12] Paul Thagard. "Cognitive architectures". In: *The Cambridge handbook of cognitive science* 3 (2012), pp. 50–70 (cit. on p. 5).

[TH12] Kristinn Thórisson and Helgi Helgasson. "Cognitive Architectures and Autonomy: A Comparative Review". In: *Journal of Artificial General Intelligence* 3.2 (2012), pp. 1–30. DOI: *doi:10.2478/v10229-011-0015-3*. URL: *https://doi.org/10.2478/v10229-011-0015-3* (cit. on p. 10).

[Ver22] David Vernon. "Cognitive Architectures". In: *Cognitive Robotics*. The MIT Press, May 2022. ISBN: 9780262369329. DOI: *10.7551/mitpress/13780.003.0015*. URL: *https://doi.org/10.7551/mitpress/13780.003.0015* (cit. on pp. 2, 8, 9).