Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-Based Systems Group
Prof. G. Lakemeyer, Ph.D.

A Document-Oriented Robot Memory for Knowledge Sharing and Hybrid Reasoning on Mobile Robots

Master-Thesis

Frederik Zwilling
Matrikelnummer: 304314

Aachen, February 22, 2017

First Supervisor:	Prof. Gerhard Lakemeyer, Ph.D.
Second Supervisor:	Prof. Dr. Matthias Jarke
Advisor:	Dipl. Inform. Tim Niemueller

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Ort, Datum	Unterschrift (Frederik Zwilling)

Acknowledgments

I would like to thank Tim Niemueller and Professor Lakemeyer for the opportunity to work on the fascinating field of artificial intelligence and robotics. Especially Tim has mentored me for a long and interesting time of my studies and I am very grateful for everything he taught me.
For an exciting time with four RoboCups and great teamwork, I also want to thank the Carologistics team, especially Tim Niemueller, Dr. Sebastian Reuter, Tobias Neumann, Sebastian Schönitz, Dr. Daniel Ewert, Matthias Löbach, Victor Mataré, Johannes Rothe, Till Hofmann, Nicolas Limpert, Christoph Henke, Mostafa Gomaa, David Schmidt, Daniel Küster, Florian Nolden, Bahram Maleki-Fard, Randolph Maaßen, Professor Lakemeyer, Professor Ferrein, and Professor Jeschke.
I would also like to thank my family and friends for their support and patience.

Contents

List of Figures

Listings

List of Abbreviations

API	Application programming interface
BS	Base-station
BSON	Binary JSON
CLIPS	C Language Integrated Production System
CPS	Cyber physical system
CPU	Central processing unit
CS	Cap-station
DB	Database
DS	Delivery-station
FOL	First-order logic
GB	Gigabyte
GUI	Graphical user interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
KB	Kilobyte
KBS	Knowledge-based system
MB	Megabyte
MMM	MongoDB MultiMaster
MPS	Modular production machines
ORO	OpenRobots Ontology
OWL	Web Ontology Language
PDDL	Planning Domain Definition Language
Protobuf	Google Protocol Buffers
RAM	Random-access memory
RCLL	RoboCup Logistics League
RDF	Resource Description Framework
Refbox	Referee box
ROS	Robot Operating System
RRD	Round Robin Database
RS	Ring-station
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WiFi	Technology for wireless local area networking
XML	Extensible Markup Language

Chapter 1

Introduction

Artificial intelligence and robotics are making a fascinating progress and allow autonomous robots to advance into more complex everyday domains. Applications with high significance for our society include domestic service robots supporting elderly care, autonomous cars avoiding accidents, and intelligent factory robots allowing flexible and efficient production. To realize these intelligent and helpful robots, a couple of challenges have to be tackled. One of these is enabling robots to grasp their environment and make decisions based on their knowledge. There, robots could benefit from memorizing gained information about their environment and providing it to reasoning and planning systems. For example, domestic robots serving breakfast need a memory to know which items belong on the table and where they are stored. Intelligent factory robots need to remember where products are in the factory and what other robots are doing.

This thesis focuses on developing such a *robot memory*. Basic requirements are representing, storing, and querying information. The information can contain sensor data, recognized object positions, symbolic knowledge, and beliefs. Thus the representation has to be flexible enough to store symbolic and spatio-temporal information. The representation we chose is *document-oriented*. Pieces of information are bundled as *documents*, which are sets of *key-value pairs* with nesting and data typing. As back-end storage system, we use the document-oriented database *MongoDB*. This has multiple advantages we need for the robot memory, for example long time storage and the possibility of distribution.

Another requirement is providing the information stored in the robot memory to *knowledge-based systems (KBS)*, such as planners and reasoners. These systems need the information to reason about the environment, plan actions, and monitor plan execution. Because there are multiple KBS with different strengths and purposes, we want to combine them to create a smart robot system. For example, *PDDL* based planners can find sequences of actions required to achieve a goal. *CLIPS* reasoners can execute plans and update the world model according to observations. Motion planners, such as *OpenRAVE*, find collision free paths for grasping or driving actions. These systems need to collaborate and exchange information. By storing and sharing the information with a general robot memory, we can ensure that they work on a common and consistent world

model. The separation of storing information and working with it also allows easier exchange of components. Furthermore, a component focused on holding the memory can provide more benefits, such as scalability and sophisticated querying.

Because different components using the robot memory have special purposes and internal concepts, they require different views on the robot memory. On the one hand, they need a filter to only get relevant information. On the other hand, they require the information to be in the appropriate format. In the case of a robot position, a motion planner needs the position as geometric coordinates and a symbolic planner needs it as string describing the location (e.g. `"kitchen"`). The robot memory provides these views and supports hybrid reasoning by transforming between spatio-temporal and symbolic information. Views are provided via interfaces to multiple KBS. For example, the PDDL interface can generate files describing the state of the environment, which is represented in the robot memory. To transform between spatio-temporal and symbolic information and to enrich query results, we introduce the concept of *computables*, which allow computing information on demand. The robot memory analyzes queries to figure out if information covered by a computable is necessary for answering. In this case, it calls the component providing the information by a computation function. Computables allow interfacing perception and save computational effort. Imagine a query for the distance of two objects. This can be computed quickly, but storing all distances between each two objects and keeping them updated would be inefficient. Another feature of the robot memory is called *trigger* and notifies about changes. This feature can be used to keep a world model updated without polling or waiting for certain information. For example, a reasoner executing a production plan could register to be notified when there is a new order requiring re-planning. The robot memory is usable in a multi robot system and thus distributable.

Our contributions are the conceptual and architectural design of this generic robot memory and the implementation in the Fawkes robot software framework. In one sentence, our robot memory can be described as a document-oriented database for hybrid reasoning and knowledge sharing between multiple, potentially distributed, KBS on mobile robots. The thesis focuses on two applications, which benefit from the robot memory and are used for evaluation. Firstly the robot memory is used in the RoboCup Logistics League to share a world model between reasoners running on multiple logistics robots. Secondly it is used in a block stacking scenario inspired by the RoboCup@Home league, where it is used for hybrid reasoning and collaboration between a symbolical planner, a reasoner executive, and a motion planner.

In the following Chapter 2, we give an overview of the context and background of the thesis and the RoboCup application domains. Chapter 3 presents related work about similar approaches and methods. Chapter 4 introduces the approach of this thesis with the goals of the robot memory, its theoretical foundation, and the architecture. The implementation of our robot memory and the usage in application scenarios is covered in Chapter 5. Chapter 6 presents quantitative and qualitative evaluation results and in Chapter 7, we conclude with a summary and future work.

Chapter 2

Background

In this thesis, we will create a robot memory for knowledge sharing and hybrid reasoning. In order to do so, we first need to introduce background information. Section 2.1 gives an introduction into the thesis' context of mobile robotics and reasoning. In Section 2.2 we introduce the primary application and evaluation domains, the RoboCup Logistics League (RCLL) and the RoboCup@Home league. Afterwards, we present the software context with robot software frameworks in Section 2.3 and planners and reasoners in Section 2.4.

2.1 Mobile Robotics and Reasoning

A *robot* is a machine capable of automatically carrying out a complex series of movements [62] and belongs to the group of *cyber physical systems (CPS)*, which can be defined as computing elements combined with physical sensing and interaction [60]. In contrast to mounted assembly line robots, we focus in this thesis on *autonomous mobile robots*, thus robots that are capable of moving in their environment and are capable of operating without any form of external control for extended periods of time [9]. These robots interact with their environment to fulfill a specific task they are designed for. They are *agents*, which utilize a *sense-think-act cycle*. Sensors perceive the environment, a computation device decides what to do, and actuators interact with the environment. How autonomous mobile robots can act giving its sensing depends on its *agent architecture*. Russell and Norvig differentiate the class of simple *reflex agents* and the class of more advanced *model-based, goal-based, and learning agents* [66]. Reflex agents can only use the current sensing to compute how to react and are thus limited in their capabilities because they can't remember anything from previous cycles. All other agent architectures have an internal state, which allows to take the perception history into account and implement more complex behavior.

2.1.1 Robot Memory

We define a *robot memory* as a system that memorizes the internal state of a robot between two or more sense-think-act cycles. In the case of a model-based agent, it can include a

model of the robots environment to compensate for partial observability of the domain. A goal-based agent remembers the given goal to achieve in the environment and a learning agent remembers what was learned from previous observations [66].

Physically, a robot memory can be stored on *volatile* or *non-volatile* devices that can be read and written. A volatile device, such as *random-access memory (RAM)*, can only store information as long as it is powered. Usually, RAM acts as storage when the internal state of the robot is represented by variables in the programming languages used to implement the behavior of the robot. A non-volatile device, such as a hard disk drive, also keeps the information when it is not powered. Also a combination of volatile and non-volatile devices can be used for storage as in this thesis. A robot memory that is stored on a non-volatile device is called *persistent*.

2.1.2 Knowledge Sharing

A robot memory is not necessarily limited to a single robot. In a *multi-agent system*, a system that is composed of multiple interacting agents [76], the agents can share their internal state. In this case, a robot memory used for sharing the state is called *distributed*. Furthermore, we will call a system composed of multiple, interacting, autonomous mobile robots a *multi-robot system*. Similarly, a robot memory can be shared on a single robot between multiple agent programs, each implementing a part of the robot's behavior. The architecture of a multi-agent system can be *centralized* or *distributed* [55]. In the centralized one, a central unit stores all relevant information and/or decides which actions each agent should execute. In the distributed one, the relevant information is stored and synchronized on all agents and/or each agents makes its own decisions. This introduces the need of *consistency*, which requires that information stored in a distributed system has no contradictions.

2.1.3 Reasoning

According to Brachman and Levesque, *reasoning* is the formal manipulation of symbols representing believed propositions to produce representations of new ones [12]. Thus, new knowledge can be *logically inferred* from other knowledge, by representing the propositions, which describe what is known or believed, and then performing reasoning manipulations on it. A simple example for logical inference is the following: Based on *Caesar is a robot* and *All robots should have a memory* it can be inferred that *Caesar should have a memory*. The field of *knowledge representation* deals with the question how propositions like these can be represented in logic formalisms. Depending on the used formalism, different classes of propositions can be expressed and reasoning in these formalisms has a different computational complexity. Knowledge representation and reasoning are important parts of artificial intelligence in mobile robotics. Knowledge representation is used to model and memorize what a robot knows or beliefs about its environment and reasoning is used to infer what effects certain actions can have on the environment and which information

can be inferred by made observations. The central tasks of reasoning systems used in this thesis are updating a world model based on observations and *incremental reasoning*, which determines the next best action to do with a robot whenever the robot is idle. An important area related to reasoning is *task planning*. After specifying a domain and possible actions that can be executed in the domain, planning can yield a sequence of actions that have to be executed to reach a given goal state from an initial state. Actions are formulated with preconditions that have to be satisfied before the action can be performed and effects that describe how the state changes after executing the action. Depending on the formalism used by the planning systems, it is possible to minimize costs associated with actions, plan in temporarily constrained environments, or plan for a multi-agent system. Reasoning and task planning systems belong to the group of knowledge-based systems (KBS), which can be described as systems that intentionally ground their processes on symbolic representations, the knowledge base [12].

The robot memory developed in this thesis expands the capabilities of planning and reasoning systems by providing a back-end database for memorizing a large amount of information and by building a bridge between multiple reasoning and planning systems by transforming information between their representations. Section 2.4 presents these systems and how they can benefit from the robot memory in more detail.

2.2 RoboCup

RoboCup is an international robotics competition founded to foster research in the field of robotics and artificial intelligence [35, 79]. It provides standardized problems as a platform to compare research results. Teams compete in different leagues to benchmark their algorithms and robotic systems. A special challenge of the RoboCup is making the robotic system robust against real world complexity and challenges, such as changing light conditions and unknown obstacles.

Figure 2.1: RoboCup logo

RoboCup features a variety of leagues, each focusing on another aspect or application domain of robotics and artificial intelligence. For example there are soccer simulation leagues focusing only on software [11], and rescue robot leagues focusing on autonomy or partial tele-operation in the real world. The majority of the RoboCup leagues host soccer robots in different sizes and complexities. The leagues range from the *Small Size League* with small cylindrical robots and external perception by an overhead camera [77] to humanoid robots in teen size which need to have all sensors and computation devices on the robot [69]. The RoboCup also features more application oriented domains, for example the *Rescue League* with robots solving different challenges in disaster scenarios [36] and *RoboCup@Work* with robots operating in an industrial scenario to perform identification, handling and transporting tasks with work related objects, such as screws and nuts [39]. In the following, we describe the *RoboCup Logistics League (RCLL)*, which features logistics

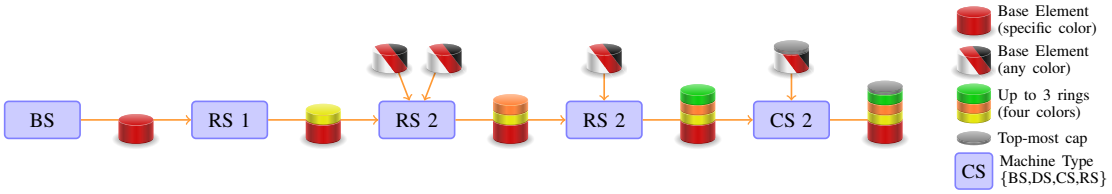


Figure 2.2: Production chain of a high complexity product in the RCLL [60]

robots in a production scenario, and the *RoboCup@Home* league featuring service robots in a domestic environment in more detail. These two leagues are used as application domains of the thesis.

2.2.1 RoboCup Logistics League

The *RoboCup Logistics League (RCLL)*[1] is an industry-oriented competition [20]. It tackles the problem of production logistics in a smart factory, where mobile robots have to plan, execute, and optimize the material and product flow between machines to produce and deliver products according to dynamic orders. Each competing team deploys a group of three robots that has to build ordered products autonomously by interacting with *Modular Production Machines (MPS)* and transporting workpieces between these machines. Figure 2.3 shows an RCLL robot filling a machine that mounts colored rings on workpieces. The robots are based on the Festo Robotino 3 platform, which uses a holonomic drive, and can be extended and programmed by the teams. The robot shown in Figure 2.3 was built by the Carologistics team and uses a laser range finder for localization, a custom made gripper for handling workpieces, and several cameras for detection of markers and light-signals mounted on the machines [57, 60]. The game is controlled by a software component called *referee box (refbox)* which acts as agent to control the environment [55]. It randomizes machine placements in the factory and production orders, communicates with participating robots, controls machines, and awards points.

Figure 2.3: Robot and MPS used in the RCLL [60]

The game consists of two phases. In the first phase, the *exploration phase*, the robots have to roam the factory to find randomly placed machines, which are to be used later. By detecting the light signal shown by the machines, the robots can determine the machine-type. For correct reports of discovered machines to the refbox the team is awarded with points, for incorrect ones points are subtracted.

In the second phase, the *production phase*, the refbox announces orders, which products have to be produced by the robots. Products are build from colored cups with optionally mounted rings and a colored cap. Figure 2.2 shows the production chain for building a

[1] RoboCup Logistics League website: `http://www.robocup-logistics.org`

high complexity product. There are four different machine types. The *base-station (BS)* provides new bases, the colored cups, as raw resource. The *ring stations (RS)* mounts colored rings on a workpiece after preparing it with a varying amount of bases. The *cap-station (CS)* mounts black or gray caps to finish a product after loading it with a cap form a shelf first. Finally, the *delivery-station (DS)* is used to deliver products.

A major challenge of the RCLL lies in the fully autonomous task planning and coordination of the multi-robot system in the dynamic environment with spatial information and time constraints. The dynamism of the environment is caused by the randomization of the factory layout, machine out-of-order times, product orders and unknown obstacles, such as the robots of other teams. Furthermore, baseline robotic challenges such as collision avoidance, perception and behavior execution need to be solved. To outperform opponent teams, the robots have to build as many ordered products as possible and deliver them in the given time window. To cope with these challenges, a decision making process needs the knowledge about the environment, which is collected by the whole robot team. Therefore the world model containing this knowledge has to be shared between the robots. This can be done by the robot memory, which also allows combining a global task planner looking in the future and a reasoner for executing plans step by step and updating the world model with current observations.

In the course of this thesis, we use a simulation of the RCLL[2] in the development and evaluation of the robot memory. The simulation is based on the open source simulator Gazebo and models the RCLL with multiple robots in a 3D environment with physics and visuals [38, 79, 80]. The simulation was developed for multi-robot coordination evaluation and has been proven as a useful tool for rapid testing and integration during the development of robot software, e.g., for agent reasoning, robot behavior, navigation graph creation, and collision avoidance. The simulation is the primary testing and development environment for implementing the robot memory in the RCLL because a full RCLL field is not available for us and maintaining a team of robots with hardware and software takes much effort and time. For the majority of robot software components, the simulation is close enough to reality to yield similar results and behaviors, because the only difference is the exchange of sensor data and actuation software by simulated data and actuation. Exceptions are visual perception by shape, and physical gripper actuation. Here, the simulation uses an approach called *multi-level abstraction*, which allows to simulate vision not only on the sensor data level (providing a simulated image), but also on the perception level (providing simulated perception results) [5].

2.2.2 RoboCup@Home League

The RoboCup@Home league is a competition about domestic service robots [75]. Robots have to assist humans in a variety of everyday tasks. These tasks include serving

[2] `https://www.fawkesrobotics.org/projects/rcll-sim/`

drinks, cleaning, setting tables, guiding or following people, helping in emergency situations, shopping, and cooking. Figure 2.4 shows an @Home robot in the competition. The goal of the RoboCup@Home league is to foster and benchmark research in the area of domestic service robots, to build a research community and to envision autonomous multi-purpose robots helping humans and especially elder people in the personal life.

Figure 2.4: RoboCup@Home robot Caesar tidying up [75]

The competition consists of multiple sub-tasks, such as finding and manipulating objects, navigation tasks, remembering persons, wait on tables, acting as nurse, and an open challenge [73]. To solve these tasks, robots need to have a wide variety of abilities, including navigation, object detection and manipulation, speech recognition, and especially human robot interaction, which requires, for example, applying everyday knowledge to incomplete task descriptions given by humans.

Important challenges in the @Home league are acting robustly in a dynamic and only partial observable domestic environment and hybrid reasoning with symbolic and spatio-temporal information. The robot memory can help here because it allows to collect information about the concrete domestic environment and, for example, log object observations to learn their distribution [49]. Furthermore, it can help with hybrid reasoning by transforming spatio-temporal knowledge into symbolic knowledge with computables. Another challenge is that detecting and manipulating objects in a personal environment can be difficult because the space the robot acts in, for example the fridge, can be very clouded. The robot memory can help here, because it is useful to keep an internal model of the environment instead of only relying on the current perception results.

2.3 Robot Software Frameworks

A *robot software framework* can be defined as design and implementation providing a possible solution for controlling a robot with basic software libraries, a communication middleware, and a run-time environment [13, 51]. Furthermore, it defines a standard for component interfaces that allows building on a variety of open source solutions for robotic problems. In the following, we introduce the widely used Robot Operating System and Fawkes [79], which is used in this thesis.

2.3.1 Robot Operating System (ROS)

The open source robot software framework *ROS*[3] is a widely used collection of useful open source libraries and tools for the development process of robot software. It can operate on many robot platforms from different domains and provides a standardized integration framework [44, 65]. The components integrated in ROS consists of processes

[3] `http://www.ros.org`

called *nodes*. These nodes can exchange information over a peer-to-peer communication network. There are communication channels called *topics*. Nodes can register a publisher to send messages on a topic or a subscriber to receive messages sent over a specified topic. There are message types defining the fixed structure of a message with nesting. A central process called `Roscore` manages the registration of publishers and subscribers. It is also a broker for nodes and configurations.

In the RCLL, ROS is used by multiple teams [10]. The Carologistics team uses ROS primarily for and localization and visualization of sensor and perception data with the `Rviz` package. In the years 2012 and 2013, Carologistics used the ROS package `move_base`, which provides local motion planning with collision avoidance.

2.3.2 Fawkes

The basis for the thesis is the robot software framework *Fawkes*[4], which is available as open source and developed at the Knowledge-based Systems Group[5] (KBSG) at RWTH Aachen University [52, 58]. It is designed to work with various robots in different domains and follows a component-based software design, which separates functional entities into individual software modules [19]. This enables reusing software solutions for robotic problems, such as localization, perception, reasoning, and behavior execution. Compiled components can be loaded as *plugins* at run-time. Plugin activity is organized in threads to make use of multiple CPU cores. The threads can be run concurrently or hooked into the main-loop of Fawkes to order the execution into a sense-think-act cycle. This ensures that higher-level components can use the latest perception data. Fawkes softly guarantees loop times by monitoring which threads are running for too long and suspending to waking them up until they recovered. These are advantages in comparison to ROS, because the components can be closer integrated. Features in Fawkes are provided as *aspects*. These are based on aspect-oriented programming [34] and give access to particular features. Threads that need to use a feature can inherit from the corresponding aspect. For example, there are aspects for logging, transforms, and specific external libraries [51].

The communication between plugins uses a *hybrid blackboard messaging* paradigm. The blackboard lists structured entries called *interfaces*, which contain qualitative and quantitative information. Interfaces can be provided by at most one plugin at a time, the writer, and read by other plugins, the readers. This allows a flexible communication because the interface is independent from the concrete writer and readers. For example, sensor and actor plugins can be exchanged by simulation plugins [79, 80]. Furthermore, the blackboard architecture simplifies debugging because the current communication between two plugins, determined by the state of the interface, can be viewed at run-time. To send commands to an interface writer, for example a motor command to the motor controller, readers can send messages to the interface. As a communication infrastructure between different components, the blackboard only has limited possibilities to realize a

[4] `http://www.fawkesrobotics.org`
[5] `http://www.kbsg.rwth-aachen.de`

robot memory. Indeed the blackboard works well for providing some information and sending commands. However the size and structure of blackboard interfaces is fixed, and thus does not allow to dynamically represent more or other information. Furthermore the blackboard does not support long-time memory, expressive querying, or event-triggers.

2.4 Planners and Reasoners

In this thesis, there three planning and reasoning systems benefiting from the robot memory. These are *CLIPS*, a rule-based production system, the *Planning Domain Definition Language (PDDL)*, which incorporates a family of planning formalisms into a standard programming language, and OpenRAVE, a *motion planner* finding collision free actuation plans. These systems are representatives for often used types of planners and reasoners. However, the robot memory is not limited to these because it provides a general C++ interface, which can be used to integrate it into more knowledge-based systems. The three systems presented here use the robot memory to achieve new or improved functionality and are used for evaluation in this thesis.

2.4.1 CLIPS Rules Engine

The CLIPS Rules Engine[6] belongs to the class of *rule-based production systems*, which can be defined as knowledge-based system using forward chaining based on the *Rete algorithm* [66]. The Rete algorithm efficiently determines which conditions of forward chaining rules can be matched by the knowledge base, even with a large number of rules and facts by constructing a data-flow graph and keeping partial matches [28]. CLIPS is the basis of the currently used reasoner by the Carologistics team in the RCLL and implements the high-level game agent. The basic building blocks of the CLIPS Rules Engine are a *fact-base*, which is a working memory with small pieces of information, a *knowledge base* with rules and procedures, and an *inference engine* working with the knowledge base on the fact-base [31]. The pieces of information in the fact-base are called *facts* and consist of a name and a key-value structure defined in a template for *unordered facts*, e.g., `(position (name robot1) (translation 2.5 1.0 0.0)`, or an ordered list of values for *ordered facts*. An example rule is shown in Listing 2.1. Rules are composed of an *antecedent* (lines 2-4) and a *consequent* (lines 6-8). The antecedent is the first part of the rule and describes the conditions that have to be satisfied before the rule is considered for activation. It consists of a list of patterns that have to be matched by facts in the fact-base. For the antecedent in Listing 2.1, there have to be fitting `state`, `visible`, and `position` facts with matching constants and variables, which start with question marks. The consequent defines the procedure to execute when the rule is activated. It usually modifies the fact-base by retracting and asserting facts, for example to exchange the `state` fact. The inference engine checks which rule antecedents are satisfied and puts only them

[6] `http://www.clipsrules.net/`

```
1 (defrule found-machine
2   ?s <- (state SEARCHING_FOR ?machine)
3   (visible  (name ?machine))
4   (position (name robot1) (translation $?pos))
5   =>
6   (retract ?s)
7   (assert (state IDLE))
8   (printout t "Found machine " ?machine " at " ?pos crlf)
9 )
```

Listing 2.1: CLIPS rule to change a robots state when the object it searched for is visible.

on the *agenda*. Which rule of the agenda is executed, is determined by a conflict resolution strategy such as highest priority or earliest on the agenda. The consequent of the chosen rule is executed and the inference engine checks again. *Functions* encode procedural knowledge, can have side effects, and can also be implemented in C++.

The Carologistics RCLL agent implemented in CLIPS represents its knowledge about the world, the *world model*, in its fact-base. It chooses actions to take by using a technique called *Incremental Reasoning* [54]: Whenever the robot is idle, it searches for the next best task and executes it. For each task there is a rule with its preconditions in the antecedent and the actions to take in the consequent. There are also rules for modifying the world model, for example after finishing actions or getting new sensor data, task execution, and coordination with other robots. The coordination of the robot team includes synchronizing their world model and resource allocation. Because of the synchronization, each robot knows what other robots noticed and changed in the environment. The resource allocation ensures that no two robots try to use the same machine at the same time or try to achieve the same goal by choosing a redundant task. This coordination was already implemented in CLIPS rules using Google Protocol Buffers (Protobuf) messages. During this thesis, we developed an alternative implementation based on the distributed robot memory that contains the world model in a distributed database. On the one hand, this provides an easy possibility to access the world model outside of CLIPS. On the other hand, this is advantageous because the database already provides efficient synchronization and separating the synchronization from CLIPS decreases the agent's complexity.

2.4.2 Planning Domain Definition Language (PDDL)

PDDL is a standardized language for planning problems [45]. It allows modeling the nature and behavior of a domain as well as the *actions* possible to perform in a *domain description*. An additional *problem description* defines the problem to solve. Both together can be given to a PDDL-based planner that typically searches for a totally or partially ordered sequence of actions, the *plan*, to solve the problem. Listing 2.2 shows an example action contained in a domain description. Actions can have parameters and consist of preconditions and effects. The precondition is a function free FOL sentence. In the case of the picking action, it represents that the object that should be picked is on the table and

```
1 (:action pickup
2   :parameters (?ob)
3   :precondition (and (on-table ?ob) (arm-empty))
4   :effect (and (holding ?ob) (not (on-table ?ob)) (not (arm-empty)))
5 )
```

Listing 2.2: PDDL action to pick up an object from a table

that the arm is empty. The effect is a universally quantified and conditional FOL sentence, but not all FOL sentences are allowed, for example sentences containing disjunctions [45]. In the example, the effect of the action is that the arm is not empty and the object is now hold and not on the table any more. PDDL predicates, such as `(arm-empty)` and `(on-table ?ob)` in the example, describe properties and can be true or false.

There are various versions and extensions of PDDL which introduce new concepts and features. For example, PDDL 2.1 [30] adds numeric values and continuous actions. This allows finding plans in continuous environments where, for example, distances and travel times matter. Planners based on PDDL can have the advantage of being able to find complete and optimal or efficient plans depending on the model. However, this comes with the drawbacks of high computational effort for larger domains and the problem of adapting to events during plan execution. For example when the domain is not fully observable, the robot could sense changes in the environment or fail executing an action. This would require notifying the planner, incorporating the changes in PDDL and re-planning. To solve this problem, the robot memory allows combining a PDDL-based planner with an execution in CLIPS. Then the planner can generates an efficient and complete plan with coarse actions and the executive can carry out the plan step by step, while monitoring the execution and updating the world model according to perceived changes. This world model is written into the robot memory so that the world model with all changes applied can be represented in PDDL.

Because PDDL is only a standardized language for formulating planning problems, a concrete PDDL-based planner program is necessary to find plans. There is a variety of planners. Many compete in the International Planning Competition (IPC), which provides a benchmark and ranking for multiple problem domains with different complexities [43]. In this thesis, we use FF, a successful planner of the IPC. FF uses forward state space search and a heuristic that ignores the negative literals in action effects [32].

2.4.3 Motion Planners

Another kind of widely used planning in robotics is geometric motion planning. It finds collision free trajectories in an n-dimensional space to reach a goal position from the initial position. Common use cases are planning the motion of a robotic arm to grasp or place something and finding paths for driving robots. Thus motion planners use geometric knowledge in contrast to logical reasoners and planners focusing on symbolic knowledge. When searching for an trajectory, it needs to take the degrees of freedom, e.g., the joints of the robot or the maneuverability of the robot base, into account and avoiding collisions

with objects in the environment. An example motion planner framework is *MoveIt!*, which is available for ROS [16]. Another motion planner framework already integrated into Fawkes is the *Open Robotics and Animation Virtual Environment (OpenRAVE)*[7] [25]. OpenRAVE focuses on high-level commands as input and already integrates the motion planning, visualization, and simulation. Thus it is a convenient choice for a motion planner in Fawkes.

Motion planners can benefit from using a robot memory by accessing and sharing information about objects and obstacles in the environment. They can better work together with other planners and task execution components because they can access the same knowledge base. This avoids double state estimations, which can cause inconsistencies. To be usable by motion planners, the robot memory has to be able to represent hybrid information in the form of continuous positions and time related events.

[7] `http://openrave.org/`

Chapter 3

Related Work

Because there already is related work on information storage systems for robots, Section 3.1 presents systems with approaches and goals similar to this thesis. In Section 3.2, we compare database technologies and implementations that were considered as basis for the robot memory and explain our choice for the document-oriented database MongoDB in Section 3.3. Section 3.4 then presents related work about extensions of MongoDB that are similar to the extensions we add for the use of MongoDB in our robot memory.

3.1 Information Storage Systems for Robots

Because a memory can be an important part of a robot, there already are various implementations and designs for it. However, there are only some sophisticated robot memories that are separated into an own component and can be universally used by multiple other components, such as planners and reasoners. These robot memories with similar approaches and goals are presented here. We start with the knowledge processing systems KnowRob in Section 3.1.1 and OpenRobots Ontology in Section 3.1.2. Afterwards we present the Generic Robot Database based on MongoDB and Fawkes in Section 3.1.3, and Open-EASE, a combination of KnowRob and the Generic Robot Database, in Section 3.1.4.

3.1.1 KnowRob

KnowRob is an open source knowledge processing system for cognition-enabled robots [70, 71]. It is designed to store knowledge about the environment and relate it to common sense knowledge to understand vaguely described tasks, such as "set the table". For representation and inference of knowledge, KnowRob uses Description Logic and approaches of the semantic web. Each small piece of information is represented as a *Resource Description Framework (RDF) triple*, e.g., `rdf(robot, holding, cup)`, that are composed of a subject, predicate, and object. A set of these RDF triples in a network can compose common sense knowledge based on the *Web Ontology Language (OWL)* [46]. It uses *ontologies*, which can roughly be described as directed graphs setting objects or classes of

objects into relation. The graph can be represented with multiple RDF triples, where the subject and object are nodes and the predicate is a directed edge from the subject to the object. In these ontologies, it is possible to represent common sense knowledge, such as `'milk is-a perishable'`, `'refrigerator storage-place-for perishable'`, and `'refrigerator is-a box-container'`. Thus a robot asked to bring milk could infer that the milk is stored in the refrigerator, which needs to be opened first since it is a box-like container.

The implementation and interface of KnowRob is based on the logic programming language Prolog, which makes representing RDF triples and inference with ontologies intuitive. To be able to interface with perception, KnowRob uses a concept called *virtual knowledge base*. It allows computing needed information on demand instead of storing everything providently. Thus special queries can be forwarded to other components that compute the answer efficiently. Especially for perception based on sensor data or transformation of spatial relations, this is advantageous compared to computing and storing all possibly needed results in the memory. KnowRob implements it with Prolog predicates called *Computables* which call C++ functions of other components. This indeed slows down the computation of a query but is more efficient than continuously computing and storing the results or implementing and computing other components algorithms in Prolog. The concept inspired the usage of computables in this thesis.

To understand the robots environment, large and detailed ontologies are necessary. KnowRob features acquiring and connecting these from various sources, such as online common sense databases, the cloud-based robot infrastructure RoboEarth [78] and websites for deriving object information from internet shops and action recipes from how-to websites giving step by step instructions. However, it has been found that this acquired knowledge needs a lot of manual extension and verification before it can be used by the robot [72]. Encyclopedic knowledge often lacks action information needed by the robot, e.g., how to grab tools, and action recipes require human text understanding. Furthermore large ontologies limit the performance of KnowRob because of the Prolog implementation and general complexity. In contrast to the robot memory of this thesis, KnowRob focuses on common sense reasoning instead of an efficient and flexible on-line back-end. It does not support sharing knowledge between multiple robots and has a too strong focus on data only usable for reasoning components. Although it can also represent spatio-temporal knowledge, we have performance and scalability concerns with Prolog handling larger data-sets, e.g., for storing locations of found object over long time to learn the distribution where they can be found. Although KnowRob can load ontologies from a hard drive, the memory acquired during run-time is not persistently stored.

3.1.2 OpenRobots Ontology (ORO)

The OpenRobots Ontology (ORO) is an open source knowledge processing framework for robotics [40]. It is designed to enable human-robot interaction by featuring common sense

ontologies similar to KnowRob and modeling the human point of view. It also uses RDF triples and OWL ontologies. The triple storage is implemented in Jena, a Java toolkit for RDF, and the inference with Pellet, a Java OWL reasoner. Besides knowledge storing, querying and reasoning, ORO features a modular architecture between the back-end storage and front end socket server for querying. These modules add features such as events that notify external components about changes, e.g., when instances of certain classes are added or modified. Another module adds representations of alternative perspectives that should allow understanding the point of view of other agents. For example when a human asks a robot to bring a cup, there are two cups on a table and the robot should infer which cup is meant by knowing that the other cup is occluded from the human's point of view. Furthermore ORO features categorization to find differences between objects, e.g., to ask if the blue cup or the red cup was meant in a vague task description, and memory profiles for distinguishing long-term knowledge and short-term knowledge that is removed when the lifetime of a fact expires. Although these are useful features and we realized the concepts of events and memory profiles in this thesis, we do not use ORO as a basis. Due to its focus on human-robot interaction and common sense reasoning, it is not suitable for representing large amounts of data for non-reasoning components because all data would have to be stored in RDF triples and processed by the OWL reasoner. Furthermore, it does neither support synchronizing a part of the knowledge with other robots nor knowledge computation on demand.

3.1.3 Generic Robot Database with MongoDB

There already is a generic robot database developed with MongoDB as data storage [56]. It is used to log data of the robot middlewares Fawkes and ROS, and allows fault analysis and performance evaluation. To achieve this, it taps into the messaging infrastructure in ROS and the blackboard interfaces in Fawkes to store the data one to one utilizing the flexibility of MongoDB being schema free. Logged data can later be queried by the robot or a developer. This allows better evaluation and fault analysis because the data would otherwise be disposed and logging all kinds of data enables retracing errors to find the source by setting it in relation to the Data-Information-Knowledge hierarchy [1]. This work already implements a basic connection to MongoDB in Fawkes for storing documents and executing queries, which is used and extended in this thesis. It also shows that MongoDB is a good choice because it can handle querying a large database and writing a lot of data efficiently while being flexible regarding how and which kind of data is being represented. However this work lacks some important concepts needed for a robot memory as intended in this thesis. It is intended as logging facility and not as working memory holding a world model for multiple planners and reasoners. Therefore updating and querying mechanisms for planners and reasoners are missing as well as triggers, on demand computation, multi-robot synchronization, a suitable front-end, and integration into knowledge-based systems.

3.1.4 Open-EASE

Open-EASE is a knowledge processing service for robots and AI researchers [8]. It is based on a combination of KnowRob [70] and the Generic Robot Database with MongoDB in ROS [56]. It is designed as a web service accessible by robots via a socket connection and by humans via a web browser. It uses KnowRob for common sense reasoning and storing the world model of a robot. The Generic Robot Database is used to log data about robots or humans performing manipulation tasks. This data can then be accessed by using the virtual knowledge base concept to learn from human demonstration and analyze faults. The focus of Open-EASE also is on providing the recorded data and access to KnowRob to humans using the web interface. This allows using Open-EASE as eLearning tool for students, who can explore and experiment with the data and the robotic system. Furthermore it fosters reproducing experiment results, e.g., for reviews, and visualizing them. Although the system is accessible from multiple users and robots, which can query the same Open-EASE instance, it is focused on single robot data and non-collaborative processes because each user operates in his private container with an individual knowledge base. This makes sense in the context of student exercises, but is not suitable for multi-robot systems or multiple knowledge-based systems sharing knowledge. If the system is also usable or being extended for cooperating robots or planners exchanging knowledge over the web service is not mentioned in current publications or on the project website[1].

3.2 Databases

Since storing, keeping, and retrieving information with the robot memory are central tasks of this thesis, the choice of a database system is important. In the following, we present the three popular database technologies, namely relational, graph-based, and document-oriented databases. Because we later decide to use document-orientation in robot memory, we compare document-oriented databases in Section 3.2.2. MongoDB, the database implementation used as a basis in this thesis, is presented separately in Section 3.2.3.

3.2.1 Database Technologies

There are multiple database technologies, which determine how data is represented and structured. Thus they require different algorithms for storing and querying and lead to different properties, such as run-time, flexibility, scalability, and crosslinkability.

Relational

Relational databases are a classic and broadly used database technology for highly structured data. They are based on the *relational model*, introduced by Codd [18,23]. The relational model structures data into *relations*, which can be described as tables with attributes as columns and data entries as rows. Table 3.1 shows an example of a relational database

[1] `http://www.open-ease.org/`

Id	Name	Age
A	Alice	23
B	Bob	26

(a) Relation Persons

Id	Title
1	Artificial Intelligence
2	Multi Agent Systems

(b) Relation Books

Person	Book
A	1
A	2

(c) Relation Borrowed

Table 3.1: Example for a relational database model about a library

model of a library. Most relation contain a *primary key*, a unique value to identify a row in the relation, e.g., the `Id` columns in Table 3.1. These keys can be used as so-called *foreign keys* to link multiple relations, such as in Table 3.1(c). In which relations a database is structured and which attributes relations contain is defined by the *schema* of the database. To interact with a database management system following the relational model, the *Structured Query Language (SQL)* can be used. SQL is a formal language to insert, update, and delete rows, change the database schema, and search the database [21]. Queries can contain conditions for attributes and group, sort, and aggregate results. An especially useful feature of relational databases and SQL is joining multiple relations by using foreign keys. For example, this allows listing the titles of borrowed books together with the name of the borrower from Table 3.1 with the query shown in Listing 3.1

```
SELECT *
  FROM Borrowed
  JOIN Persons
    ON Borrowed.Person =
       Persons.Id
  JOIN Books
    ON Borrowed.Book =
       Books.Id
```

Listing 3.1: SQL query

Graph-Based

Graph-based databases model data structures as graphs with nodes and edges [3]. Nodes usually represent entities and edges the relations between them. Nodes and edges can have properties, such as a name. This allows, for example, to model family trees with persons as nodes, direct relation as edges, and names as properties of persons. In comparison the to relational databases, there is no database schema because the structure is intrinsically defined by the contained data. Thus graph-based databases are called *schema-free*. They are well suited for domains with transitive relations, such as `'is related to'`, and other problems that can be reduced to graph theory problems, such as connectivity, because algorithms from graph theory can directly operate on the graph. Therefore costly join operations, which would be necessary in relational databases, can be avoided [74]. A graph-based data model that is widely used in robotics is the *Resource Description Framework (RDF)* [8, 40, 71]. It represents information as triples, e.g., `rdf(robot, holding, cup)`, that are composed of a subject, predicate, and object [37]. Subjects and objects are represented by nodes and predicates by edges. The query language for RDF databases that is proposed as standard by the World Wide Web Consortium is *SPARQL* [64]. It provides operations that are similar to SQL extended by graph traversal features.

Document-oriented

Document-oriented databases store entities called *documents*, which consist of key-value pairs [15, 17, 50]. In contrast to relational databases, they are schema-free. Thus there is no predefined schema that enforces the structure of documents. Documents with similar structure are usually grouped into *collections*. Values can be accessed by their key and can have different types, such as strings, numbers, complex objects, such as dates and nested sub-documents. Therefore the documents inserted in a collection implicitly define the structure during run-time. This allows a flexible use of the database because applications are not limited to a fixed schema and can, for example, add key-value pairs during run-time when additional information needs to be stored. Furthermore this simplifies software development, in which the document structure tends to change often. Even if there are documents with different structure, the application can choose to retrieve only certain documents by giving a specialized query. Each document also contains a unique key, similar to a primary key in relational databases. In contrast to relational databases, which *normalize* information into multiple relations to minimize redundancy, document-oriented ones focus on bundling related information in documents (*denormalization*). This simplifies horizontal scaling in distributed systems because required information is available at one place. The query languages of document-oriented databases are not so standardized as of relational databases shown in Section 3.2.2. However, most document-oriented databases have the usage of the *JavaScript Object Notation (JSON)* as input format in common. An example of a JSON document is shown in Figure 3.2. JSON is a lightweight data-interchange format similar to the *Extensible Markup Language (XML)* [33]. It is human readable and assembles objects with key-value pairs. Because these objects can also be nested, they can represent documents. Compared to XML, JSON is faster and more compact [61]. In the following, we omit quotation marks around JSON keys for more compact presentation.

```
{
  "key": "value",
  "subdocument":
    {"x":3, "y":1},
  "array":
    [{"n":0.1},{"n":2}]
}
```

Listing 3.2: JSON document

3.2.2 Document-oriented Databases

There are multiple database managing systems, which are based on the document oriented model and could be used in this thesis. Here, we present three popular open source implementations that we considered. CouchDB and ArangoDB are introduced in this Subsection an MongoDB is covered in the next Subsection in more detail.

CouchDB

Apache CouchDB[2] is an open source document-oriented database software [2, 15]. It is written in the concurrency-oriented language Erlang and uses the JSON format to

[2] `http://couchdb.apache.org/`

store data. Queries in CouchDB are realized with so called *views*, which are based on JavaScript functions, e.g., to specify constraints, aggregation, updates. The *Application programming interface (API)* of CouchDB is based on the Hypertext Transfer Protocol (HTTP) and thus focused on web applications, but interfaces for other languages, such as C++, also exist. What separates CouchDB from comparable database systems is its *multiversion concurrency control*. In a distributed system, CouchDB does not guarantee consistency. It provides a system called Multi-Master Replication, which allows writing to all replication instances. Conflicts can be solved similar to a version control system, thus multiple conflicting versions can be kept in parallel for later revision. Because of the multiversion concurrency control, CouchDB also support asynchronous replication. Thus it is possible to write on two replication instances concurrently. On a multi-robot system that would avoid idling times waiting for write access, but introduces the problem of solving conflicts later.

ArangoDB

ArangoDB[3] is an open source database managing system. What sets it apart from other database managing systems is that it combines document oriented and graph-based models in a so called *multi-model* [4, 26]. It supports the JSON format, but stores the data in a binary format called VelocityPack. The API is based on HTTP. ArangoDB uses the *Arango Query Language (AQL)*, which is inspired by SQL and extended by document and graph features that account for the dynamic schema. Queries can be enhanced with JavaScript functions, e.g., for access graph features. Also the interface focuses on JavaScript, but a C++ driver exists as well. Similar to CouchDB, ArangoDB also features multiversion concurrency control. For a robot memory, ArangoDB would have the advantage that ontological knowledge could be properly represented and queried since it has a graph structure.

3.2.3 MongoDB

MongoDB[4] is a document-oriented database. It provides many useful features, such as flexible data structures, aggregation, and distribution over multiple machines, which would take a large effort to implement manually. It has proven as powerful and scalable data storage and is widely used [17, 56]. MongoDB uses the JSON format, but internally stores and evaluates documents in the *Binary JSON (BSON)* format. Listing 3.3 shows such a document used for storing the position of a robot.

Queries in MongoDB are similar to documents because the arguments of query functions also use the document structure. An example query is shown in Listing 3.4. It searches for all documents in the collection `positions` that have matching values for `type` and `position`, and are up to date, thus with a timestamp greater than the given one.

[3] `https://www.arangodb.com/`
[4] `https://www.mongodb.com/`

```
{
  type: "position",
  name: "robot1",
  translation: {x:2.5, y:1.0, z:0.0},
  rotation: {x:0.0, y:0.0, z:0.0, w:1.0},
  timestamp : ISODate("2016-05-19T15
      :26:34.466Z")
}
```

Listing 3.3: MongoDB document representing the position of a robot

```
db.positions.find(
  {
    type: "position",
    name: "robot1",
    timestamp : {"$gt":
      ISODate("2016-05-19T15
          :26:34.000Z")}
  })
```

Listing 3.4: MongoDB query yielding the document in Listing 3.3

Multiple key-value pairs in a query are conjunctive. Values can be checked and compared as in the example with a variety of operators such as `not`, `or`, `equals`, `greater than`, etc. These queries can be evaluated very quickly, especially when *indexing* is used. Indexing is a database feature that allows to lookup documents faster than in linear time by storing a references to a portion of the data in an fastly to traverse form, for example as hash table or binary tree. For more complex queries, it is possible to add arbitrary JavaScript functions, which check if documents are matched. Furthermore, it is possible to aggregate results using the *Map-Reduce* paradigm [24], which maps resulting documents to intermediate documents which are then reduced until the end-result is found. For example this allows querying for the nearest visible object by getting all visible objects, mapping their position to a distance and reducing them to the nearest one by dropping more distant ones when comparing them. The higher expressiveness of these queries comes with the price of slower computation. However, the computational effort can be reduced by formulating queries smartly. Many usual queries, e.g. Listing 3.4, can be formulated without using `where` operators. Furthermore, complex queries can be nested to filter documents first with a fast query and execute the computationally costly query only on the remaining documents.

MongoDB also features fixed-size collections that are similar to ring buffers. These are called *capped collections* and order documents by insertion order. As Figure 3.1 visualizes, new entries replace the oldest ones when the capped collection is full. An especially useful feature of MongoDB that allows sharing knowledge between multiple robots is *replication*. Replication is a process that distributes a database onto multiple machines. One MongoDB instance called *Primary* initially copies the database to other instances called *Secondaries*. Afterwards, all modifications to the primary database, which are stored in a separate, capped collection called *operations log (Oplog)*, are forwarded to the secondaries to keep them synchronized and consistent. This is often used in web-services to distribute read-queries and their computation on multiple machines and to keep backups. Read only queries can be computed by Primary and Secondary instances. Modifying queries are forwarded to the Primaries to guarantee consistency. MongoDB can also group multiple instances into a *Replica Sets*, which automatically manages

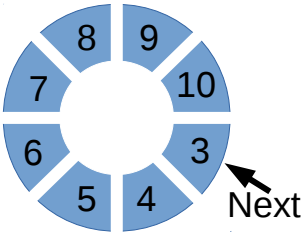


Figure 3.1: Sketch of capped collection of size 8 after inserting the numbers 1-10 in this order

the primary election when necessary. By using replication, multiple robots can robustly share a common and consistent world model. Robot specific knowledge can be kept in a separate and not replicated database to avoid unneeded communication. It is also possible to distribute write operations with a technique called *Sharding*. Sharding builds on Replica Sets and splits the database into multiple *shards*. Because each shard can have its Primary on another machine, the write operations on the database are distributed among the machines. This allows MongoDB to scale in an application with too many writes for a single machine.

3.3 Why choose MongoDB as Basis of the Robot Memory?

In this section, we compare data storage systems that can be used as a basis for the robot memory of this thesis and explain why we chose a document-oriented database and MongoDB in particular. The possible systems we compare are introduced in the background and related work and are production systems, ontologies, relational databases, and document-oriented databases.

One possibility for an underlying basis system is a production system and CLIPS in particular. The information stored in the robot memory can be represented by facts in the volatile fact base. In these facts, key-value pairs can be used according to a predefined fact template. The main advantage of a production system is its reasoning capability that allows inferring more information and knowledge from what is inserted and combining it with the existing fact base. For example, this would allow inserting new observations and deriving an updated world model, which can later be extracted. However, this would be domain dependent because special rules for updating and inferring would be required. Compared to other data storage systems, CLIPS query features for access by external components are very limited. For example, it would not scale for large data-sets due to the lack of indexing. The rule matching capabilities of CLIPS are not suitable for querying because each query would require a new rule that changes the graph underlying the RETE algorithm and thus causes a large computational overhead [28].

Ontologies and RDF triples from the Web Ontology Language are another possibility for a data storage system of the robot memory. Especially KnowRob uses these and was considered as a basis for this thesis. Information can be represented by RDF triples and embedded in an ontology graph. The main advantage is, that this allows reasoning while answering a query because the query can be reformulated using backwards chaining with the ontology as a basis. A disadvantage of this is the scalability with the sizes of the stored data and ontology. For example, the robot memory should be able to store a large data-set about object observations to learn their spatio-temporal distribution. Furthermore, the information is not stored persistently and can not be distributed in a multi robot system.

Compared to both previous approaches with ontologies and production systems, database systems focus on data storage and querying instead of reasoning. We think that they are a better basis for a robot memory because they allow separating the concerns of storing

and remembering on the one hand, and reasoning, planning, and other robotic applications on the other hand. They are designed to work as efficient data storage with rich querying and distribution features. The separation makes it possible for multiple systems to work with the robot memory and to collaborate using the robot memory for information exchange. When for example the common sense reasoning of KnowRob should be integrated into a robot system, the robot memory could store the required information in the database and provide KnowRob with the required information and ontologies. Furthermore, other systems such as a PDDL-based planner or a component learning object distributions from observations can also use the common basis in an efficient way. Before choosing a concrete database managing system, we have to decide between a relational database and a document-oriented database. Graph databases are not suited very well because the information we want to represent often does not have a graph structure, such as a world model of the RCLL or memorized object observations. Thus the main advantages of graph databases, namely searching and traversing a graph, are not so important. Relational databases provide powerful query features, such as joining over multiple relations. Document-oriented databases are schema free and thus allow a higher flexibility during development and run-time. This flexibility is important because the representations of information on the robot tend to change during development and it is possible to group documents with similar purpose and different attributes together, e.g., facts of a world model. Because of denormalization, document-oriented databases avoid joining over many relations by bundling information that belongs together in a single document, e.g., with nesting. Especially this allows document-oriented databases to scale horizontally and host a distributed robot memory. These arguments lead to the choice of a document-oriented database.

MongoDB is especially well suited for this thesis because of its performance and scalability compared to the document-oriented databases CouchDB and ArangoDB [63]. Especially compared to CouchDB, MongoDB provides faster interaction times [41]. This could be caused by the HTTP interface or by the fact that queries in CouchDB purely rely on JavaScript functions that have to be evaluated and computed during querying. The query language of MongoDB also provides an additional syntax for simple queries for faster evaluation in C++. Compared to ArangoDB, MongoDB has a larger community and more features, such as Map-Reduce and saving larger files. Furthermore it was already used successfully and efficiently in Fawkes and the RCLL as logging database [56, 59].

3.4 MongoDB Extensions

The work of this thesis uses MongoDB as a basis and extends it with some concepts that are important for the usage in robot memory. In this section, we present MongoDB extensions which are related to our extensions, such as triggers for notifications about changes in the database.

3.4.1 Trigger

In the database jargon, a trigger is a piece of code that is automatically executed after a specified event, such as a change in the database. Thus a trigger allows the user to react to events without polling. Triggers are common in relational database management systems, such as PostgreSQL [48]. MongoDB does not provide triggers, but there are multiple projects providing them as extension [27]. A simple way to realize triggers on insertion events is using capped collections of MongoDB because they order documents by the time of insertion. The lazy and iterative query evaluation strategy of MongoDB allows querying the capped collection and keeping the cursor if there are no more documents. After some time, the query can be continued from the position of the cursor to check all documents that were inserted in the meantime because they were just appended after the documents inserted before. Conveniently MongoDB uses a capped collection to synchronize changes in a Replica Set. This capped collection, called the *Oplog*, can also be queried by the user and allows listening to changes in the Replica Set just as described above. In this thesis, we also use the Oplog to implement trigger for the robot memory.

3.4.2 Multi-Master Replication

There also exist an extensions of MongoDB which implement a *Multi-Master Replication*. It is called *MongoDB MultiMaster (MMM)*[5] and enables writing to each instance of a distributed MongoDB database [22]. Changes can be immediately applied locally and are synchronized later to the other instances. This allows faster writing especially if the other instances are not available for some time and would be advantageous in the RCLL, where a crowded WiFi can lead to large latencies. However, MMM is not focused on consistency and ignores write conflicts. Other disadvantages are increased overall communication and database size. This is caused by the implementation of MMM, which uses an usual Master-Slave replication for each instance. Each instance consists of one Master of these replications and Slaves for the other. MMM then applies changes in a Slave replication to the local Master. To detect changes in a replica, MMM utilizes MongoDB's Oplog similar to this thesis. It queries the Oplog, which is capped collection, as described in the Section 3.4.1.

[5] `https://github.com/rick446/mmm`

Chapter 4

Approach

To describe the approach of this thesis, we first present the goals of the robot memory in Section 4.1. Section 4.2 then introduces the theoretical foundation how information can be represented in the robot memory and how knowledge is transferred into a knowledge based system (KBS). The concepts of the robot memory's main components are presented in Section 4.3 and Section 4.4 gives an overview of the architecture describing which components the robot memory consists of and how they interact with each other.

4.1 Goals

The design goals of the robot memory were formulated before detailing robot memory concepts and define what the robot memory should be capable of. The evaluation in Chapter 6 refers to these goals to analyze how well the implementation fulfills them.

Flexible Storage and Retrieval: The main task of the robot memory is memorizing information of the robot. Memorizing includes storing newly obtained information, updating it according to changes, removing information that is no longer valid or needed, and remembering the information by querying it again. Thus the robot memory has to provide services for these operations. It has to be usable by multiple components as information source and collaboration platform. When the robot memory is queried, it must only return the information that it is asked for. It should be flexible enough to store any kind of data, information, knowledge, and wisdom of the Data-Information-Knowledge hierarchy in any structure. For example the robot memory should be able to store world models, perception results and even large data-sets, such as remembered observations of objects.

Memory Sharing Between Knowledge-Based Systems: The robot memory has to provide a common memory to allow consistent memory sharing between multiple components. Thus multiple components have to use the robot memory at the same time and should have access to common or each other's information. This should especially allow Knowledge Based Systems (KBS) to share a common world model, even if the KBS are of a different type. For example a planner, a world model reasoner, and an executive should

work together using the robot memory as a common basis. This allows combining their specialized strengths, e.g., generating efficient plans, intuitive world model reasoning, and robust execution. Furthermore, the robot memory should help avoiding inconsistencies between KBS by providing a common knowledge base so that only one state estimation is necessary.

Special Views for Different Components: Depending on the application of different components using the robot memory, different views on the stored data are necessary. For example, a planning components needs a different part of the stored data, than a component learning distributions from object sights. Thus component-specific filters are necessary. Furthermore, different components need the stored data in different formats, e.g., symbolic or geometric, depending on the used programming language and purpose. The robot memory has to allow defining these views and then accordingly filter and transform the stored information. Also keys and identifiers, which might be named differently, have to be translatable.

Computation on Demand: To efficiently compute information only when it is necessary, the robot memory has to provide computation on demand. This means that some information is not stored in the robot memory directly, but computed when needed. It allows accessing information through the robot memory that is otherwise impracticable to compute and store continuously and for all possible queries. For example when a component rarely queries the distance of two objects, it is better to compute the result on demand than to store all distances between each two objects in the memory and keep them updated. Components using the robot memory should be able to provide knowledge computation functions (*computables*) to the robot memory. This adds a lot of versatility and expandability.

Distributed Memory for Multi-Robot Systems: The robot memory should be able to distribute parts of the memory in a multi-robot system. Multiple robots can then share a common world model for collaboration. This goal is similar to the memory sharing between knowledge-based systems on the same machine with the difference that the components on a multi-robot system work simultaneously and the additional difficulty of message loss and keeping consistency. Such a distributed robot memory is necessary in many multi-robot domains, especially in the RoboCup Logistics League (RCLL), where the whole robot team needs to know the world model and what other robots intend to achieve at the moment. Distributing the robot memory adds additional requirements and challenges, especially robustness, e.g., in the case of robots dropping out, consistency of the memory, and accessibility so each robot can use the robot memory without large latencies.

Spatio-Temporal Grounding: It is often desirable to ground stored knowledge based on location and time because this is the prerequisite for many applications, such as spatio-temporal clustering and learning distributions, e.g., at which day-times objects can be found at which positions [49]. Thus the robot memory has to be able to store knowledge tagged with spatio-temporal information and consider this information while querying.

Triggers: The robot memory should be able to notify components about relevant changes in it. These notifications about specified events are called *triggers* and should automatically be sent after a component registers an event it wants to be notified about. Events include the appearances of new information about a specific topic, modifications and deletions of it. More complex events describe conditions over whole collections, such as the amount of objects exceeding a threshold. An important use case that should be supported by the robot memory is updating reasoner predicates according to changes in the memory.

Persistent Storage: The robot memory should be persistently stored on a non-volatile device to be able to hold long-time knowledge and operate over long times, in which a robot might be restarted. This is necessary in many domains, such as for domestic service robots and RCLL robots that might be restarted. Furthermore, it should be possible to distinguish information that is persistent and information that becomes invalid after a restart and should be removed.

Human Interface and Visualization: An important factor in the development of a robotic systems is the accessibility of underlying components for easier debugging and modeling. Therefore the robot memory needs to provide an interface for developers to introspect and modify the state represented in the robot memory. A visualization would also be desirable to analyze spatio-temporal data efficiently.

4.2 Theoretical Foundation

In this section, we lay the theoretical foundation of the robot memory and its integration into KBS. First, we focus on the representation of symbolic information as it is used in KBS. Here we take the PDDL formalism as an example. The formal definition of the robot memory describes how knowledge is represented and how computables are defined. Because the robot memory is document oriented, we first define documents and their key-value pairs. This definition is derived from the specification of the Binary JavaScript Object Notation (BSON)[1] used in MongoDB. Because documents can be nested, we define them by induction. Unnested documents are defined as follows:

1. **Keys** are strings $\mathcal{K} := \Sigma^*$, where Σ contains all valid characters.

2. **Atomic values** are in a countable infinite set $\mathcal{V}_0$ of constants with integers, floating point numbers, and strings.

3. **Unnested key-value pairs:** $\mathcal{P}_0 := \mathcal{K} \times \mathcal{V}_0$

4. **Unnested documents** are sets of key-value pairs with unique keys and thus included in the power set of $\mathcal{P}_0$:
   $\mathcal{D}_0 := \{d \in \mathbb{P}(\mathcal{P}_0) | \forall (k, v), (k', v') \in d, k \neq k' \vee (k, v) = (k', v')\}$

[1] `http://bsonspec.org/spec.html`

Values and documents with a nesting depth up to n with $n >= 1$ can be defined as follows:

1. **Values:** $\mathcal{V}_n := \mathcal{V}_{n-1} \cup \mathcal{D}_{n-1}$
2. **Key-value pairs:** $\mathcal{P}_n := \mathcal{K} \times \mathcal{V}_n$
3. **Documents:** $\mathcal{D}_n := \{d \in \mathbb{P}(\mathcal{P}_n) | \forall (k,v), (k',v') \in d, k \neq k' \vee (k,v) = (k',v')\}$

This yields the set of all *finitely nested documents* $\mathcal{D} = \bigcup_{n \in \mathbb{N}} \mathcal{D}_n$. The set of all values is $\mathcal{V} = \bigcup_{n \in \mathbb{N}} \mathcal{V}_n$. Infinite nesting and documents containing themselves are not considered because they can not properly be mapped into most KBS formalisms and database systems. A *database* is a finite set of documents $\mathcal{DB} \subset \mathcal{D}$. *Queries* are represented by documents $q \in \mathcal{D}$ and yield a set of documents $r \subseteq \mathcal{DB}$ as result when executed in a database according a specification, e.g., of MongoDB [17]. The query $\{(\text{"}object\text{"}, \text{"}cup\text{"}), (\text{"}room\text{"}, \text{"}kitchen\text{"})\}$, for example, yields all documents containing these key-value pairs, which represent all cups in the kitchen.
A *computable* f is a function mapping a query document and the current state of the database to a set of computed documents:

$$f : \mathcal{D} \times \mathbb{P}(\mathcal{D}) \to \mathbb{P}(\mathcal{D})$$

The power-set of documents is in the domain of a computable because the result can also depend on the state of the database, e.g., a computable calculating the number of objects in the database. For simplicity, we assume that all sensor data, a computable can depend on, is also represented in the database. The component providing the computable has to ensure that the function is computable and terminates. With these preliminaries, we can now define a *robot memory* as tuple of a database $\mathcal{DB}$ and an ordered set of computables $\mathcal{C} = \{f_1, ..., f_n\}$:

$$\mathcal{RM} = (\mathcal{DB}, \mathcal{C})$$

The computables are ordered by a priority because a computable with higher priority can produce documents that are consumed by a computable with lower priority.

The set of documents *memorized* by a robot memory without computables equals the underlying database:

$$mem((\mathcal{DB}, \emptyset)) = \mathcal{DB}$$

For a robot memory with computables the documents memorized are:

$$mem((\mathcal{DB}, \{f_1, .., f_n\})) = mem((\mathcal{DB}, \{f_1, .., f_{n-1}\})) \cup \bigcup_{d \in \mathcal{D}} f_n(d, mem((\mathcal{DB}, \{f_1, .., f_{n-1}\})))$$

To integrate the robot memory into PDDL, we need to define *mapping functions* map_d between documents and predicates, and map_v between values and terms. As prerequisite, the connection between key-names in documents and predicates, functions, actions, and their attributes needs to be defined. Let $\mathcal{R}$ be the *set of predicate symbols*, $\mathcal{F}$ the *set of*

function symbols, and $\mathcal{A}$ the *set of actions symbols*, then

$$name_{pred} : \mathcal{R} \to \Sigma^* \text{ , } name_{func} : \mathcal{F} \to \Sigma^* \text{ , and } name_{act} : \mathcal{A} \to \Sigma^*$$

map to the strings used in document keys (e.g. $name_{pred}(\text{at}) = \text{"at"}$). To ensure unambiguity, the *name* functions must be injective and their images disjunctive. The function

$$name_{func-atr} : \mathcal{R} \times \mathbb{N} \to \mathcal{K}$$

maps the attributes of functions to a key names. $name_{pred-atr}$ and $name_{act-atr}$ are defined accordingly (e.g. $name_{pred-atr}(\text{at}, 1) = \text{"object"}$ and $name_{pred-atr}(\text{at}, 2) = \text{"room"}$). Now we can define the mapping of values in documents to terms as follows. Values that are no sub-documents can be mapped directly to the corresponding constant:

$$map_f(v) = v, v \in \mathcal{V}_0$$

Otherwise the value is a sub-document, which is mapped to a function term:

$$\begin{aligned} map_f(d) = f(map_f(v_1), ..., map_f(v_n)),& \text{ iff } f \text{ is a n-array function in } \mathcal{F}, \\ (\text{"}function\text{"}, name_{func}(f)) \in d, \forall i \in \{1..n\}&(name_{func-atr}(f, i), v_i) \in d \\ map_f(d) = nil_f, otherwise& \end{aligned}$$

where nil_f is a new constant used for not mappable documents. Similarly documents can be mapped to predicates as follows:

$$\begin{aligned} map_p(d) = p(map_f(v_1), ..., map_f(v_n)),& \text{ iff } p \text{ is a n-array predicate in } \mathcal{R}, \\ (\text{"}predicate\text{"}, name_{pred}(p)) \in d, \forall i \in \{1..n\}&(name_{pred-atr}(p, i), v_i) \in d \\ map_p(d) = nil_p, otherwise& \end{aligned}$$

For example is

$$\begin{aligned} map_p(\{(\text{"}predicate\text{"}, \text{"}at\text{"}), (\text{"}object\text{"}, \text{"}cup\text{"}), (\text{"}room\text{"}, \text{"}kitchen\text{"})\}) \\ = at(map_f(\text{"}cup\text{"}), map_f(\text{"}kitchen\text{"})) = at(cup, kitchen). \end{aligned}$$

This theoretical foundation of the robot memory and the mapping of queried documents into predicates is the basis for the generation of PDDL problem descriptions. The direction from PDDL back to the robot memory is limited to storing the resulting plan found by a PDDL-based planner. The plan $\mathcal{L}$ is an ordered sequence of parameterized actions

$$\mathcal{L} = (a_1(v_{1,1}, v_{1,2}, ..., v_{1,k_1}), a_2(v_{2,1}, ..., v_{2,k_2}), ..., a_j(v_{j,1}, ..., v_{j,k_j})).$$

where a_i is an action with k_i parameters and j is the length of the plan. The resulting plan is then mapped into a document containing the actions as sub-documents:

$$map_L(L) = \{(\text{"}plan\text{"}, \text{"}success\text{"})\} \cup_{i \in \{1..j\}} \{(n_i, map_a(a_i))\},$$

where $\mathcal{L}$ is defined as above, n_i is the number i as string, and map_a maps a parameterized action to a sub-document, similar to the inverse of map_f:

$$map_a(a_i(v_{i,1}, ..., v_{i,k_i})) = \{(\text{"}action\text{"}, name_{act}(a_i))\} \cup_{m \in \{1..k_i\}} \{(name_{act-atr}(a_i, m), v_{i,m})\}$$

If the planner can not find any plan, the result is mapped to the document $\{(\text{"}plan\text{"}, \text{"}fail\text{"})\}$.

For the knowledge representation in the robot memory and the integration into PDDL, the closed world assumption, stating that all true predicates are known to be true, is kept. Thus if there is no document $d \in mem(\mathcal{RM})$ so that $map_p(d) = p(v_1, v_2)$, $p(v_1, v_2)$ is known to be false.

Not included in the theoretical foundation focused on PDDL is that the robot memory can also map spatio-temporal information into other systems such as CLIPS and C++. Here, the documents in $mem(\mathcal{RM})$ are not directly mapped into the used representation of CLIPS, e.g., as facts, or C++, e.g., as variables. Instead, there are functions for querying, which yield a set of documents, and functions to receive the value of a key-value pair in different types. Furthermore, there are functions to build documents and queries iteratively pair-by-pair and for inserting, updating, and removing.

4.3 Robot Memory Concepts

This section presents the concepts behind important aspects of the robot memory in more detail. These concepts are computables for computation on demand, trigger for notifications about events in the database, interfaces of the robot memory to knowledge-based systems, and the distribution of the robot memory in a multi-robot system.

4.3.1 Computables

Computables are functions that provide information or knowledge on demand. Thus the information provided by the computable is not stored directly in the database, but will be computed when some application asks for it. The robot memory manages the list of available computables and checks which computables need to be called to answer a query. The concrete computables are provided by other components. For example, a vision component can register a computable that computes which objects are currently visible at which position. We use computables in our robot memory for three main reasons:

- Computables ease dealing with complex data in knowledge-based systems. By using the robot memory interface, KBS can access functions that are provided by other

components and compute information that is difficult or impossible to compute inside the KBS. For example, a KBS using a computable can ask for the closest object even if the KBS does not support arithmetic operations. This way, KBS can bypass some expressiveness and complexity limits.

- Computables decrease the computational effort that is necessary to collect the information contained in the robot memory. For some problems, it is not appropriate to continuously compute and store the results in the robot memory. Rather it is better to compute the results on demand only when it is necessary and only with the parameters needed at the moment. One common example of these problems in the robotics context is perception, which can be computationally costly and might be needed only rarely. For example, the detection of objects on a table is only necessary when the robot is searching for something on a table. Another set of problems that are well suited to be solved by computables deal with values changing rapidly over time and a variety of possible parameters. For example, the transformation result of object coordinates change rapidly while the robot is moving and there might be many objects that have to be transformed to the different frames used by the robot. Thus it is impractical to compute and store the positions of all objects in all frames and keep them updated. It is a better idea to provide the transforms in the robot memory as computable so the transformation only needs to be done when it is needed and only for the subset of objects that are currently needed.

- Computables simplify hybrid reasoning by transforming spatio-temporal information into symbolical information and vice versa. Because of the KBS interface and the decreased computational effort mentioned in the previous two reasons, computables are well suited to simplify dealing with hybrid information in KBS. For example computables can provide a KBS with the information in which room the robot is based on the current geometric position stored in the robot memory. An example for the transformation of symbolic to geometric information is finding a free position on a table for a planner action specifying to place an object somewhere on the table.

According to the formal definition of computables in Section 4.2, computables are functions taking a query as input and returning a set of computed documents. Furthermore, computables can also use the documents currently provided by the robot memory as basis for the computation, e.g., to transform the coordinates written in the robot memory to another frame. The robot memory has to decide when the function of a computable needs to be executed given an asked query. To decide this, each computable also contains a definition which queries can be answered by it, called the *computable spec*. This definition is represented by a query. Whenever a query on the robot memory matches this computable spec, the computable is called. Furthermore, each computable has a priority defining in which order computables are executed if a query can be answered by multiple computables. This allows computables to depend on the results of other computables. The documents resulting from a computable are temporarily added to the robot memory

so that a query can be evaluated on computed information and persistent information in the database of the robot memory.

4.3.2 Triggers

Triggers are mechanisms that invoke a callback function if a specified event occurs in the robot memory. The main application of triggers in this thesis is to notify components using the robot memory about changes of the stored information. Furthermore they can be used to check for more complex conditions each time a relevant document in the robot memory changes. Triggers consist of a function and definition of the event, that should cause the function to be invoked. The robot memory manages triggers, which were registered by other components, checks when the defined events occur, and then call the according function. The events are defined by queries about insertions, modifications, or deletions of documents with specific properties. For applications using the robot memory, triggers can be useful because they allow notifications in a convenient and efficient way. Applications do not have to implement polling and the queries defining events only need to be executed on changes in the robot memory. An examples for a component utilizing triggers is an executives that wants to be notified about changes to keep an internal world model up to date. Furthermore, triggers allow sending messages in the flexible document format.

We note here that triggers are not intended to automatically work together with computables. To solve the complex question if some element of the image area of an arbitrary computable function matches a trigger query, the computable function would have to be evaluated for all possible input queries. Because this contradicts the intention behind computables, there is no automatic evaluation of triggers on computables. However since the results of computables are temporarily inserted in the database, triggers do work on computables when the computable was invoked by an external query. Thus it is possible to combine triggers and computables by periodically querying the robot memory to activate the computable. Furthermore it is possible to bypass this limitation by replacing the computable by a system that periodically inserts the required computation results into the robot memory.

4.3.3 Knowledge-Based System Interface

The interface between knowledge-based systems and the robot memory defines how features of the robot memory can be accessed and how the robot memory can be embedded in the language concepts of the KBS. The features of the robot memory include the basic operations, query, insert, delete, and update, the registration and callback for triggers, and the possibility to provide computables. Depending on the KBS, the robot memory features have to be provided in the appropriate programming language and representation. For example to provide the information contained in the key-value structure of documents in CLIPS, unordered facts can be used. However facts do not allow nesting, therefore the nesting has to be resolved somehow. Another example is the representation as pred-

icates in PDDL, which requires the mapping from documents to predicates as shown in Section 4.2. Thus the interface has to transform documents into the representation and programming language used by the KBS. This transformation can also require deriving symbolic information from spatio-temporal one. An example is the room a robot is in if the robot memory only contains the geometric position of the robot. Furthermore KBS often are specialized on some area, such as planning or world model reasoning. Therefore only a part of the information contained in the robot memory might be relevant to the KBS. The interface has to filter out non relevant parts.

Due to the difference of KBS, each KBS language requires a separate interface to the robot memory. This separate interface can be provided by a separate component using the robot memory and the environment of the KBS. It can also bridge to specific concepts used by the KBS, such as the problem definition in PDDL instead of the procedural interaction with the robot memory. The filtering of the relevant part of the robot memory can use additional query constraints or be limited to specific collections. It is also possible to rename keys of key-value pairs in documents to obtain the correct identifiers used in the KBS.

4.3.4 Multi-Robot Synchronization

To distribute the robot memory in a multi-robot system, each robot has to host one instance of the robot memory. These instances communicate to share and synchronize their memorized information. There are a shared and a private part of the robot memory, which are both accessible by the robot. The private part is only visible and usable by the robot hosting it and the shared part is accessible by all robots in the system. This enables sharing all necessary information while saving bandwidth by not sending information that is not relevant for other robots. The shared and private part of the robot memory are separated as different databases. All write operations on the shared part are mutually locked and executed on a single primary instance. Read operations can be evaluated on all instances for faster querying. Thus the shared robot memory is eventually consistent because all updates on the primary reach the secondary with a slight delay. Each document is consistent in itself since additions, updates, and deletions of single documents are atomic. Inconsistencies of document sets should not be a problem for the robot memory because documents should be denormalized and thus bundle all relevant information. The distributed robot memory has to be robust against the drop out of robots. Thus, it has to stay operational, keep all shared information by having a copy on each robot, and reintegrate the dropped out robot when it is started again.

Triggers and computables work in a distributed robot memory. However, the robot that wants to be notified by the trigger or that wants to use the computable has to register it on its own robot memory instance.

4.4 Architecture

The architecture contains standard concepts of software engineering, most importantly a layered architecture, functional abstraction, and data abstraction [14] to allow re-usability and adaptability to changing requirements and use cases. Figure 4.1 shows the architecture of the robot memory and is explained in the following. The architecture organizes all

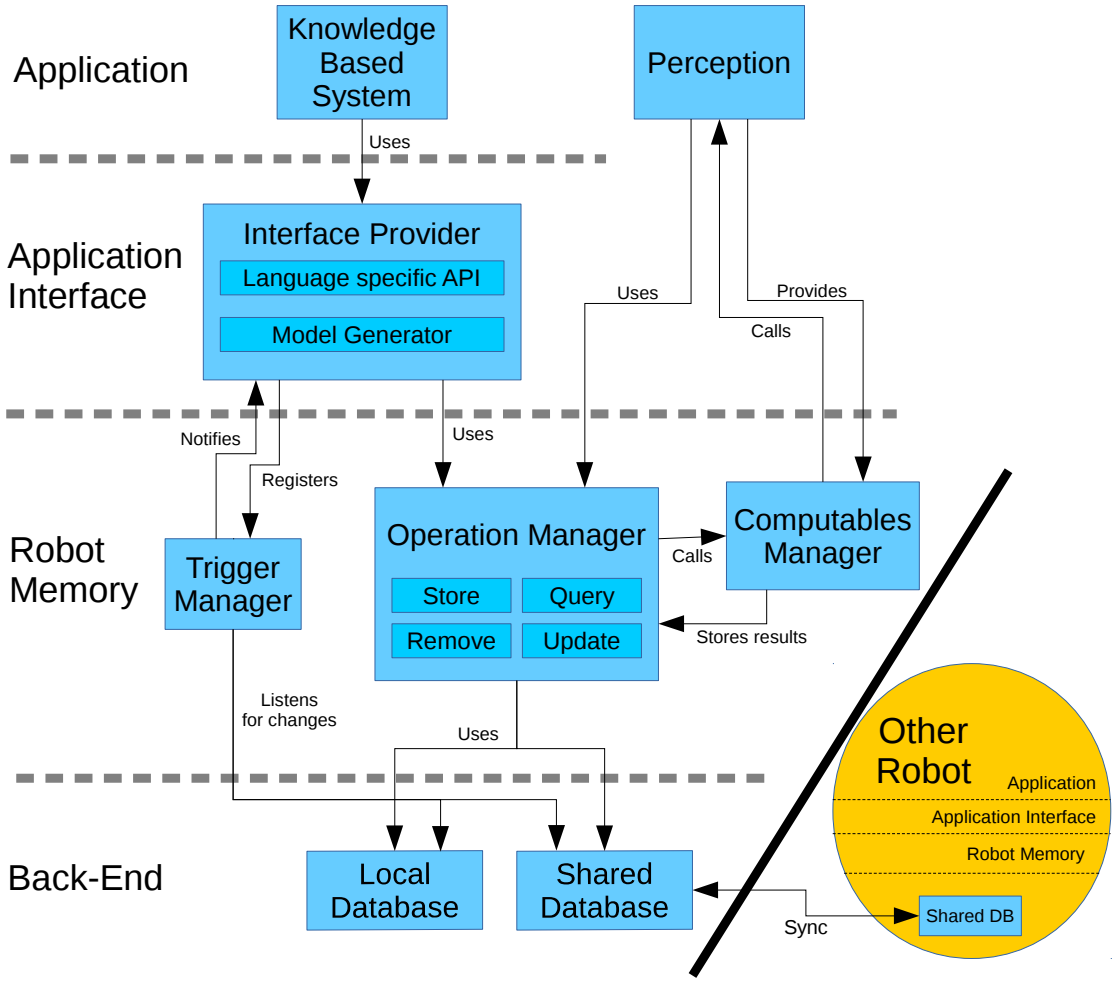


Figure 4.1: Architecture of the Robot Memory

components into four layers: Back-End, Robot Memory, Application Interface, and Application. The Back-End contains databases for storing knowledge and executing queries. There is a local private database, which is private to the robot, and a shared database distributed between multiple robots using the same architecture. This distribution can utilize the replication features of the database. The Robot Memory-layer contains the central functions that distinguish the robot memory from a pure database. It contains the operation manager, which is used to execute store, update, query, and delete operations. The operation manager is an intermediate component that realizes most of the robot memory's functionality by modifying and enhancing queries, storage, and update requests before applying them to the database. The operation manager also handles the database interaction

and distinguishes if operations should be executed on the local or the shared database. If a query asks for knowledge provided by a computable, the operation manager forwards it to the computable manager. The computable manager keeps a record of which computable is provided by which application and requests the demanded computation. The component responsible for triggers receives and manages registrations of applications that want to be notified in the case of an event. It listens to logged changes in the databases to check if events happen and notifies applications accordingly. Additionally the operation manager can setup the robot memory by loading a set of initial documents from a file, e.g., for a new RCLL game. Between the robot memory and a planner or reasoner components, there is the Application Interface layer. For each KBS, it includes an Interface Provider component, which makes the function of the robot memory available to the KBS and the special programming language used. It is not necessary for components that can directly use the interface of the robot memory (available in C++). The responsibilities of the interface provider also include transforming query results into the appropriate format used by the KBS and eventually renaming key-names if they have be mapped to match the names used in the KBS model. Because languages and concepts of knowledge-based systems can be very different, each KBS needs a separate Interface Provider. The Interface Provider can also contain a model generator, which creates models for the KBS, e.g., a problem definition or initial fact base, from a template and additional information resulting from queries. Finally the Application layer contains all components using the robot memory either through the Interface provider or directly through the robot memory interface. KBS can also register triggers through their interface provider and perception components can provide computation functions for computables. The sources and consumers of knowledge stored in the robot memory are the applications using it. The ordinary data flow starts in the application requesting storage, modification, or querying of knowledge (over the language specific interface). The operation manager forwards the request enhanced with management information to the database. For queries, the result is transferred back from the database over the operation manager (and the interface provider) to the KBS or perception component. For computables, the only difference in the data flow is the knowledge computation before executing the query in the database. The operation manager additionally initiates the knowledge computation via the Computables Manager, which forwards the query to the application providing the specific computable. The documents resulting from the computable are then stored by the computables manager in the robot memory, where it is queried afterwards.

Chapter 5

Implementation

The implementation of the robot memory is separated according to the layered architecture. Section 5.1 presents the implementation of the back-end containing the MongoDB databases and Section 5.2 presents the implementation of the robot memory with the operations-, trigger-, and computable manager. Both application layers are covered in Section 5.3. We conclude with the description of the evaluation scenarios and how they were implemented in Section 5.4.

5.1 MongoDB Back-End

The back-end of the robot memory is based on MongoDB databases. It acts as storage of the robot memory and executes insert, query, update, and delete operations. The representation of stored information utilizes the document structure of MongoDB with key-value pairs and nested sub-documents. This allows a flexible storage of information pieces in the structure that is induced by the application and can be expanded by meta information from the robot memory.

An example document stored in the back-end is shown in Listing 5.1. It contains knowledge about a bottle of milk, as needed by a planner told to bring it, with additional information where it should be stored later. Additionally the document contains meta information needed by the robot memory, e.g., if the document should be stored persistently or removed at restart and when it should be dropped because its use-time has expired. How these documents are parsed, serialized, stored, and queried again is handled by MongoDB. To connect to MongoDB from the robot memory, we use the `mongodb` plugin in Fawkes. It was developed for the implementation of the Generic Robot Database [56]

```
{
  _robmem_info:
  {
    persistent: true,
    decay_time: ISODate(
      "2016-05-19T 23:50:00.000Z")
  },
  type: "object info",
  name: "milk_1",
  position: {x:2.5, y:1.0, z:0.0},
  storage_place: "refrigerator"
}
```

Listing 5.1: Representation of a knowledge item in the back-end

and utilizes the C++ driver of MongoDB. We also extended this plugin, e.g. to properly connect to a distributed replica set.

As the architecture in Figure 4.1 shows, the MongoDB backend contains a local and a shared part. Each part is operated by a separate `mongod` instance, which is the daemon process of the MongoDB system. It manages data access, handles data requests, and performs background management operations. Each `mongod` instance is accessible over its unique TCP port and can contain multiple databases. The databases of the robot memory that are only locally relevant are managed by the local `mongod` instance. The shared `mongod` instance manages the distribution of shared databases in a multi-robot system. Thus we configured it to be part of a replica set as already described in Section 3.2.3. This ensures the efficient distribution of the databases and handles important distribution issues, such as Primary election, consistency, and synchronization. The robot memory needs multiple connections to both `mongod` instances because single connections, as provided by the C++ driver, are not thread safe. Thus we use the connection manager of the `mongodb` Fawkes plugin to create two connections each for operations, computables, and triggers. After some experiments with sharding, we decided against using it to ensure that at least one robot has write access to the full robot memory. Based on our experience of WiFi issues in the RCLL, where the network was unusable for several seconds, it can be a problem that every robot could have to wait until a Primary for a shard of the world model is reachable. Furthermore we expect that the classical motivation for sharding, namely a bottleneck in processing power for updates, is not given.

The MongoDB back-end also contains the operations log (Oplog), a separate capped collection containing the list of changes to the database with a timestamp. The Oplog is accessed by the robot memory to analyze database changes for triggers. However, the Oplog is only created for replicated database, because its original purpose is logging changes to propagate them to other instances in the Replica Set. Thus we also have to configure the local `mongod` instance as Replica Set, but this set only contains itself because its databases are only locally relevant.

Because the robot memory requires a special configuration of the MongoDB daemons, we also implemented an automated setup of the back-end. This setup starts the `mongod` processes as sub-processes of Fawkes if they are not already running. Each process needs to be configured, most importantly, with the Replica Set name, the database path, where the databases are persistently stored on the hard-drive, and the Oplog size. After the start-up, the `mongod` instances need to be initialized, when they are started for the first time. Thus the automated setup sends commands for initialization of the Replica Set with information about the members of the set. Furthermore, commands for the query evaluation are sent, to allow query evaluation on Secondaries. When Fawkes is terminated, also the MongoDB sub-processes are finalized again.

5.2 Robot Memory

The robot memory middle-layer implements most functions of the robot memory that exceed a typical database. Because this is a central and large part of this thesis, we split the implementation into the query language in Section 5.2.1, the operation manager in Section 5.2.2, computables in Section 5.2.3, triggers in Section 5.2.4, and unit-tests in Section 5.2.5. The robot memory layer can be accessed in Fawkes through the `RobotMemoryAspect` and through sending messages to the blackboard interface of the robot memory. Both the aspect and the interface are provided by the `robot-memory` plugin.

5.2.1 Query Language

A central question is which query language should be used between applications and the robot memory because this determines the expressiveness and has a large impact on the performance. We choose to use the query language of MongoDB as it is already used between the robot memory and the back-end. This has the following advantages compared to using other query languages, such as SQL, SPARQL, XQuery [47], and JSONiq [29]: For the applications using the robot memory, the query language of MongoDB is a good choice because it is an intuitive query language and has been proven as efficient for usual and well designed queries [56]. It is also highly expressive when using additional JavaScript functions or the MapReduce paradigm [17]. For the robot memory, it requires no translation before applying queries to the database. Furthermore it allows extending and modifying queries because queries are structured as documents with key-value fields and can be nested or executed in sequence. Another advantage of the MongoDB query language is that it can easily be parsed, e.g., from a string by using the MongoDB C++ driver. The resulting object can be analyzed and modified for example to add key-value pairs or to check if computables are queried.

5.2.2 Operation Manager

The MongoDB query language provides the basis for the operation manager, which receives insert, query, update, and deletion requests. Received requests are analyzed using the MongoDB C++ driver and modified before being applied to the back-end. This way, the operation manager can add meta information of the robot memory to documents and queries, and the computable manager can base its decision if computables need to be executed on the contents of a query.

The basic operations on the robot memory are insertions, queries, updates, and deletions. For all four, the operation specifies on which collection in which database the operation should be performed. The collection string contains both separated by a dot, e.g., `'robmem.worldmodel'` for the worldmodel collection in the robmem database. Using the configurable list of shared databases, the collection string also specifies whether the

operation should be performed on the local or shared `mongod` instance. Because connections to MongoDB are not thread safe, the operation manager has to guarantee the mutual exclusion of connection usage. Furthermore, the operation manager handles exceptions caused by operations, e.g. when a query can not be parsed or an object contains invalid fields.

An **insertion** operation also specifies the document that should be inserted. An example insertion of a document in C++ is shown in Listing 5.2. The object is built by adding key-value pairs. The `position` field holds a sub-document constructed from a JSON string. Before performing the insertion on the database, the operation manager can add meta information of the robot memory, for example the insertion time if the collection is configured to be a short time memory. The meta information is contained in a sub-document under the `_robmem_info` key, as shown in Listing 5.1.

```
mongo::BSONObjBuilder b;
b.append("object", "cup");
b.append("color", "red");
b.append("clean", false);
b.append("pose",
  fromjson("{x:0,y:1}"));
robot_memory->insert(
  "memory.dishes", b.obj());
```

Listing 5.2: Inserting a document about a red cup in C++

For **queries**, the operation also specifies the query itself. Listing 5.3 shows how a query can be executed, how to iterate over returned documents, and how to access their key-value pairs. The operation manager analyzes the query and modifies it similar to how it modifies documents. It adds a filter and a read preference. The filter hides meta information of the robot memory from the application. The read preference specifies that the query should be on the executed closest available member of a replica set. This should always be the local one because it has the lowest latency and thus leads to a faster query times than performing queries on remote replica set members. Furthermore the query is passed to the computation manager to check if a computable has to be executed.

```
mongo::BSONObjBuilder q;
q.append("object", "cup");
QResCursor res = robot_memory->query(
  "memory.dishes", q.obj());
while(res->more())
{
  BSONObj cup = res->next();
  std::string color =
    cup.getField("color").String();
  printf("There is a %s cup.",
    color.c_str());
}
```

Listing 5.3: Query for printing all cup colors

Update operations contain a query specifying which documents should be updated and a document containing the changes that should be applied. An example update operation is shown in Listing 5.4. It uses shift operators to fill documents with key-value pairs and the greater than operator `$gt`. By default the first matching document in the specified collec-

```
mongo::BSONObjBuilder query;
query << "object" << "cup"
  << "pose.x" << fromjson("{$gt:0}");
mongo::BSONObjBuilder update;
update << "$set" << fromjson(
                    "{clean:true}"));
robot_memory->update("memory.dishes",
  query.obj(), update.obj());
```

Listing 5.4: Update operation to mark one cup in an area as clean

tion is replaced by the update document. It is possible to modify this behavior with additional options. There are options for performing the update on all matching documents and for inserting the document even if no document in the collection matches the query (called *upsert*). To keep key-value pairs in the original document, the update document needs to contain updated values in a `$set` sub-document. In contrast to query operations, updates do not lead to the execution of computables because the updated information would be lost shortly after when the computable exceeds its caching time. However, it is still possible to update already computed documents because they are handled as regular documents during the caching time.

Deletion operations have an attached query specifying which documents should be removed. Listing 5.5 shows a delete operation. In contrast to updates, deletions are performed on all documents matching a query by default. The operation manager does not need to perform special modifications of this query.

```
mongo::BSONObjBuilder query;
query << "object" << "cup"
  << "color" << "black";
robot_memory->remove(
  "memory.dishes",query.obj());
```

Listing 5.5: Deletion of all black cups

Additionally to these basic operations, the operation manager can also export the content of a collection to a file (called *dump*). It can also import a collection from a file in an operation called *restore*. These operations can be used to save and load a world model that is contained in a collection. For example in the RCLL, this allows setting the world model in a prepared state to test the behavior of the agent in that state. The dump and restore operations are implemented by calling the `mongodump` and `mongorestore` command line tools of MongoDB with appropriate parameters.

5.2.3 Computables

Computables allow the computation of documents on demand, that is when a query operation asks for a document that can be provided by the computable. The computable manager is the entity used for registering and unregistering computables as well as for checking which computables should be executed for a query.

Listing 5.6 and Listing 5.7 show how queries, documents, and functions are involved in

```
{
  compute: "sum",
  x: {$exists:true},
  y: {$exists:true}
}
```

a: Query defining the computable specification

```
{
  compute: "sum",
  x: 15,
  y: 4
}
```

b: Query requiring the computable

```
{
  compute: "sum",
  x: 15,
  y: 4,
  sum: 19
}
```

c: Resulting document

Listing 5.6: Queries and documents involved in a computable for addition

a computable by an addition example. When some component registers a computable

on a collection, it needs to give a reference to the function performing the actual computation and a definition which documents can be provided by the computable. This definition is specified by the *computable spec*, as shown in Listing 5.6a. Whenever the robot memory has to answer a query, for example Listing 5.6b, that is matched by the computable spec, the computable is executed. The referenced function for the addition computable is shown in Listing 5.7 and yields a list of computed documents that, in this example, only contains the result shown in Listing 5.6c. The result contains the `compute`, `x`, and `y`, fields because it needs to be matched later by the original query in Listing 5.6b.

The computable manager checks whether queries passed to it can be answered by registered computables. Because the MongoDB C++ driver supports no unification or matching method outside the database, we perform the matching inside the database. Firstly the robot memory query is inserted as document in a separate and empty collection. Secondly the computable spec is executed as query on that collection to check if the robot memory query is matched by the computable spec. Thus when the computable spec returns the robot memory query as result, the function of the computable is called. This function (Listing 5.7) takes the query causing the computation (Listing 5.6b) and the collection as parameters. Thus it can use values in the query (`x` and `y`) to determine what should be computed. The function then computes a list of resulting documents (list containing Listing 5.6c) and returns it to the computable manager. The example in Listing 5.9 shows how this computable can be registered, invoked, and removed. By defining the computable spec appropriately, i.e., with `$exists`, it is ensured that all values required by the function are provided by the query. It is also possible have optional arguments. When optional arguments are not given, a computable could create documents for all possible values. For example a computable for blackboard interfaces in Fawkes, could returns documents for all interface ids of a type, when no interface id is given in the query.

```
std::list<mongo::BSONObj>
compute_sum(mongo::BSONObj query,
            std::string collection)
{
  int x = query.getField("x").Int();
  int y = query.getField("y").Int();
  mongo::BSONObjBuilder b;
  b << "compute" << "sum"
    << "x" << x << "y" << y
    << "sum" << x + y;
  return {b.obj()};
}
```

Listing 5.7: Function of a computable

After a computable returned the list of computed documents, the computable manager inserts these documents in the specified collection of the robot memory. Finally, the operation manager evaluates the original query as usual on the collection. The query can return computed documents as well as persistently stored documents. Thus the computed information is embedded in the already existing information. Another advantage of evaluating the original query on the collection, is that the query can have more key-value fields than required by the computable, for example to filter computed documents or add aggregation functions.

To lower the computational effort of computables, we implemented caching of computed documents. Whenever a query is identical to a query that caused computation in the caching time before, the no computables are executed for it. Computed documents remain in the robot memory during this caching time. This is implemented by adding the expiration time of computed documents to their meta information and periodically removing documents with passed expiration time. A typical caching time could be the loop time of Fawkes.

Computables can depend on the current content of the robot memory, by using the robot memory themselves in their function. This can be used for hybrid reasoning by transform documents already contained in the robot memory into other representations. To allow computables to depend on the result of other computables, there is a prioritization. The computable manager uses this prioritization to determine the order in which computables are checked.

The `robot-memory` plugin in Fawkes provides two basic computables. One provides access to the blackboard by computing documents containing the fields of blackboard interfaces with matching types and ids. The other transforms documents describing 3D positions into other frames by using the transform aspect of Fawkes. All other computables have to be provided by other plugins.

5.2.4 Triggers

Triggers notify components about changes in the database, for example to keep a reasoner internal world model up to date. To register a trigger, the component has to provide the name of the collection that should be observed, a reference to a callback function, and a query defining the event in which the callback function should be called. An example query used for being notified about new or modified orders in the RCLL is {`relation: "order"`}. This query is used to search the Oplog for changes. The Oplog is a capped collection in MongoDB, which logs all changes to propagate them in a Replica Set.

Listing 5.8 shows an example document contained in the Oplog. Besides the collection name and timestamp of the change, it shows the type the causing operation (`op:"i"` stands for insertion) and the inserted, updated, or removed document as sub-document under the `o` key. For each new trigger, the trigger manager creates a cursor on Oplog in MongoDB with the trigger query. Initially this cursor points to the end of the Oplog (position 3 in Figure 5.1). Because the Oplog is a capped collection, new changes, e.g., 4 and 5 in Figure 5.1, are appended behind the position the

```
{
  ns: "syncedrobmem.clipswm",
  ts: Timestamp(1485369072, 5),
  op: "i",
  o: {
    _id : ObjectId("58c14a14"),
    relation: "order",
    id: 2,
    complexity: "C0",
    delivery-gate: 3,
    quantity-requested: 1,
    quantity-delivered: 0,
    begin: 209,
    end: 282 }
}
```

Listing 5.8: Document in the Oplog

cursor points to. During the main loop of Fawkes, the trigger manager checks for all tailable cursors if there are new documents in the Oplog containing `o` object that match the according query. As Oplog changes are checked, the tailable cursor is forwarded. Thus if Figure 5.8 shows the document labeled with 4 or 5 it would be returned by the query. The trigger manager then calls the callback function and passes the document from Figure 5.8. This way, the application is notified and can react to the change with the information contained in the returned document from the Oplog.

5.2.5 Unit-tests

To ensure the quality and correctness of the robot memory, we implemented unit tests for the important features of this layer. Unit testing is part of the software development processes test driven development and extreme programming [6, 7]. Unit tests are automated software tests for small units of a program. They use the functions or components that should be tested and check if the computed result matches the expected one. Thus they help to locate bugs, ensure the quality of the unit, and fasten the development because they can verify if the tests still pass after changes in the unit. We use unit tests for the robot memory by using the C++ interface of the robot memory and checking if the results are correct for examples of common use cases. For example there is one test that checks insert, update, and query operations by inserting a document into an empty collection, updating it, and querying it by an updated value. The test shown in Listing 5.9 checks if computables work by registering the computable showed in Listing 5.7 and Listing 5.6, executing the query in Listing 5.6b and checking if the result contains the correct sum from Listing 5.6c. Unit-testing in Fawkes is based on Google's C++ testing framework *gtest*[1]. To test the functionality of the robot memory in a fully running Fawkes process with dependencies of the robot memory, such as the `mongodb` plugin, we integrated our unit tests into a separate plugin that is started with the robot memory when the tests are executed.

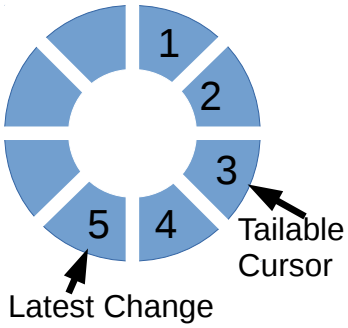


Figure 5.1: Tailable cursor on the Oplog

5.3 Application Interface

The application interface layer provides the features of the robot memory for applications such as planners and reasoners. Components written in C++ can directly interface the robot memory layer. This interface is identical to the interface between the robot memory and and the interface providers. For example perception plugins in Fawkes are usually written in C++ and thus can provide computables or store documents directly in the robot memory. Planner and reasoner components often utilize specialized languages. For

[1] `https://github.com/google/googletest`

```
TEST_F(RobotMemoryTest, ComputableCallAddition)
{
  TestComputable* tc = new TestComputable();
  Computable* comp = robot_memory->register_computable(fromjson(
      "{compute:'sum',x:{$exists:true},y:{$exists:true}}"),
      "robmem.test", &TestComputable::compute_sum, tc);
  QResCursor qres = robot_memory->query(
      fromjson("{compute:'sum',x:15,y:4}"), "robmem.test");
  ASSERT_TRUE(qres->more());
  ASSERT_TRUE(19, qres->next().getField("sum").Int());
  robot_memory->remove_computable(comp);
}
```

Listing 5.9: Unit test for registering and invoking a computable

these components and languages, there are multiple interface providers we implemented. In the following, we present the interface providers for PDDL, CLIPS, and OpenRAVE.

5.3.1 PDDL Interface Provider

To find a plan with a PDDL-based planner, a domain description and a problem description need to be provided for the planner. The domain description is usually fixed because the predicates defining the world state and the actions defining the capabilities of the robot often are fixed for a domain. The problem description, depends on the world model of the robot and varies in the concrete predicates and the amount of objects. Thus the problem description depends on the content of the robot memory and has to be constructed before planning. This process belongs to the model generation of the interface provider and is implemented in the `pddl-robot-memory` plugin. In Section 4.2, we defined the mapping from documents in the robot memory to PDDL theoretically. Nevertheless, we need to specify which documents in the robot memory should be mapped to PDDL and which strings are mapped to which predicates, functions, and their attributes. These are the *name* functions. To specify both, we make use of the template engine *ctemplate*[2]. The template engine takes a *template file* as input and uses a *dictionary* defined during run-time to exchange *template markers* in the file by desired string values. An example template file for the blocks world domain is shown in Listing 5.10. The result after generating the problem description based on information in the robot memory is shown in Listing 5.11. A simple template marker is `<<GOAL>>`. During the model generation, the dictionary is filled with the information that `<<GOAL>>` should be replaced by `(on A B)`. There also are special markers such as `<<ON>> <</ON>>`. They enclose an environment and are called *list markers* because they insert the enclosed environment for each list entry of the dictionary on the `ON` marker. We use list markers to insert a predicate for each document that is returned by a query. The query is located in the front marker behind the `|` symbol. The enclosed environment can contain markers that are substituted by key-value pairs of documents. In the example given in Listing 5.10, the query {`relation:'on-table'`} yields

[2] `https://code.google.com/p/google-ctemplate`

```
(define (problem blocks_world_generated)
  (:domain blocksworld)
  (:objects A B C D)
  (:goal <<GOAL>>)
  (:init
    <<#ONTABLE|{relation:'on-table'}>>
      (on-table <<object>>) <</ONTABLE>>
    <<#ON|{relation:'on'}>>
      (on <<object_top>> <<object_bottom>>)
    <</ON>>
    <<#HOLDING|{relation:'holding'}>>
      (holding <<object>>) <</HOLDING>>
  ))
```

Listing 5.10: PDDL problem description template

```
(define (problem
    blocks_world_generated)
  (:domain blocksworld)
  (:objects A B C D)
  (:goal (on A B))
  (:init
    (on-table A)
    (on-table C)
    (on D A)
    (holding B)
  ))
```

Listing 5.11: Generated PDDL problem description

the documents {`relation:'on-table', object:'A'`}, what is mapped to `(on-table A)`, and {`relation:'on-table', object:'B'`}, what is mapped to `(on-table B)`. To distinguish markers inside these environments, they are written in lower case.

The process of the model generation starts with an interface message containing the goal to the `pddl-robot-memory` plugin. Then the plugin parses the problem description template to extract the queries of the list markers. This is a separate step, because passing parameters to a list marker is not supported by ctemplate. The plugin fills the dictionary with the goal and entries for each document returned by the robot memory for a list marker query. For each such list entry, all key-value pairs of the documents are inserted into the dictionary, where the key name in the document equals the marker name in the template. For sub-documents, the key names are concatenated so that the marker name is the path to each value. Markers that are not in the dictionary are later substituted by the empty string. Finally, the dictionary is passed with the template file to ctemplate, which generates the resulting problem description.

The direction from PDDL to the robot memory is implemented in the `pddl-planner` plugin. When the plugin receives an interface message, it starts the FF planner with the domain and generated problem description files and waits until the planner finished. Then the resulting plan is passed, transformed to a document as specified in the theoretical foundation in Section 4.2, and inserted into the robot memory. Afterwards an executive can fetch the plan, for example after being notified by a trigger that there is a new plan.

5.3.2 CLIPS Interface Provider

The Interface Provider for CLIPS is implemented in the `clips-robot-memory` plugin as a CLIPS-feature. *CLIPS-features* are extensions of an environment, can be loaded during run-time, and can provide access to C++ functions. In contrast to PDDL, CLIPS can use procedures. Thus the CLIPS *robot-memory feature* can wrap the interface functions of the robot memory and provide them as functions inside the CLIPS environment. Listing 5.12 shows an example function in CLIPS for querying the robot memory. The function takes the collection name and the query as a string concatenated by clips and yields a cursor to documents about not ended orders. This cursor can then be used by

```
(rm-query "syncedrobmem.clipswm"
          (str-cat "{relation:'order', end:{$gt:" ?gametime "}}"))
```

Listing 5.12: CLIPS function to execute a query

```
{ relation: "cap-station", name: "M-CS1", cap-loaded: "NONE",
  caps-on-shelf: NumberLong(3), "sync-id": NumberLong(13) }
```

```
(cap-station (name M-CS1) (cap-loaded NONE) (caps-on-shelf 3) (sync-id 13))
```

Listing 5.13: Mapping between robot memory document and an unordered CLIPS fact

further functions to check if there are more documents, fetching the document, and reading values from it. Furthermore, the robot-memory feature provides higher abstracted functions, which can convert a fact reference to a document and asserts facts constructed from documents. These functions use a similar mapping as presented in the theoretical foundation. Listing 5.13 shows how a document is mapped to an unordered fact by the function `(rm-assert-from-bson ?doc)`, where `?doc` is a pointer to a document. The `relation` key-value pair is matched to the fact template name and the other key-value pairs are matched to slot names of this template. There is a transformation between lists in CLIPS and sub-documents with numbered key-value pairs. Other sub-documents are passed as reference to the sub-document.

Similar to the model generation in PDDL, it is also possible to create an initial fact base in CLIPS. This is done by querying the initial state from the robot memory and asserting the corresponding facts to the fact base. The robot-memory feature also provides access to triggers, which are a good way to keep the fact base in CLIPS updated according to changes in the robot memory. However, executing callback functions to notify about triggers does not fit to the programming methodology of the production system CLIPS. Rather CLIPS reacts to events by having rules with facts representing the event in the antecedent. Thus, when the CLIPS interface provider is notified about a trigger activation, it asserts a fact representing this to the fact base. This fact contains an identifier defined during the registration of the trigger by a CLIPS agent, and a pointer to the Oplog document causing the trigger event.

5.3.3 OpenRAVE Interface Provider

OpenRAVE is a motion planner that finds collision free paths and movements of a robot, e.g., for an arm. It searches for the path in an internal representation of the robot's environment, the *scene*. During setup or run-time, this scene is filled with objects the robot knows about in its environment, either for interaction or collision avoidance. To make sure the motion planner scene is be consistent with the world model stored in the robot memory, we implement the OpenRAVE interface provider in the `openrave-robot-memory` plugin. The plugin constructs and updates the motion planner scene from documents in the robot memory. Based on configuration values, it queries the robot memory for objects

in the environment and inserts, updates, or removes them from the OpenRAVE scene. Listing 5.14 shows an example document, which can be used to construct the OpenRAVE scene. The key-value pair `block:"B"` specifies the type of the object and the name used to move or remove it later. The type also defines the model used in the scene according to the configuration. Furthermore the translation and rotation in a given frame are used to place the object at the right position. Here, the transform computable can be used to transform the position into the frame used by OpenRAVE. For example the frame used for the Jaco arm has its origin in the base of the arm.

```
{
  block: "B",
  frame: "map",
  translation:
   [0.43, -0.04, 0.01],
  rotation:
   [0, 0, 0.99, 0.08]
}
```

Listing 5.14: Document used to construct the OpenRAVE scene

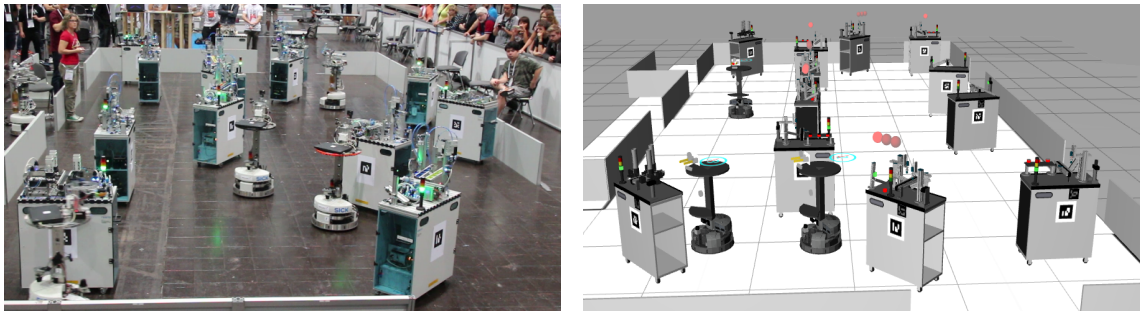
5.4 Application Scenarios

The two scenarios are chosen in a way that they cover all major features of the robot memory and that they are prototypes for applications the robot memory is made for. In Section 5.4.1, we present how the robot memory is used in the RCLL to synchronize the world model between multiple robots, and in Section 5.4.2, we present the use of the robot memory in a block stacking scenario with a robot arm.

5.4.1 RoboCup Logistics League

A central component of the Carologistics robot software in the RCLL is the CLIPS agent. It maintains a world model, decides what the robot should do with incremental reasoning [54], and monitors the execution of robot actions. The decision approach and world model maintenance is distributed. Robots have to coordinate with each other and notify each other about changes in the environment. Furthermore, they can drop out and might be restarted. Thus, the CLIPS agent needs to synchronize its world model with the other robots. The existing implementation used by Carologistics in 2016 uses Protobuf messages which are being sent between the agents. In this application scenario, we exchange the synchronization via Protobuf by using the robot memory. Thus the world model of CLIPS has to be represented in the robot memory and distributed over all robots. On the one hand, this lays the foundation for using the robot memory in the RCLL in future work. On the other hand, this simplifies the CLIPS agent by separating the synchronization. The synchronization by MongoDB is advantageous because it only sends necessary changes over the network instead of the whole world model, as it is done by the CLIPS agent. Another advantage is the persistency, because robots keep their state if one or even all of them are restarted.

During the development and evaluation of the RCLL scenario, we mainly depend on the simulation of the RCLL shown in Figure 5.2b. This simplifies and fastens the

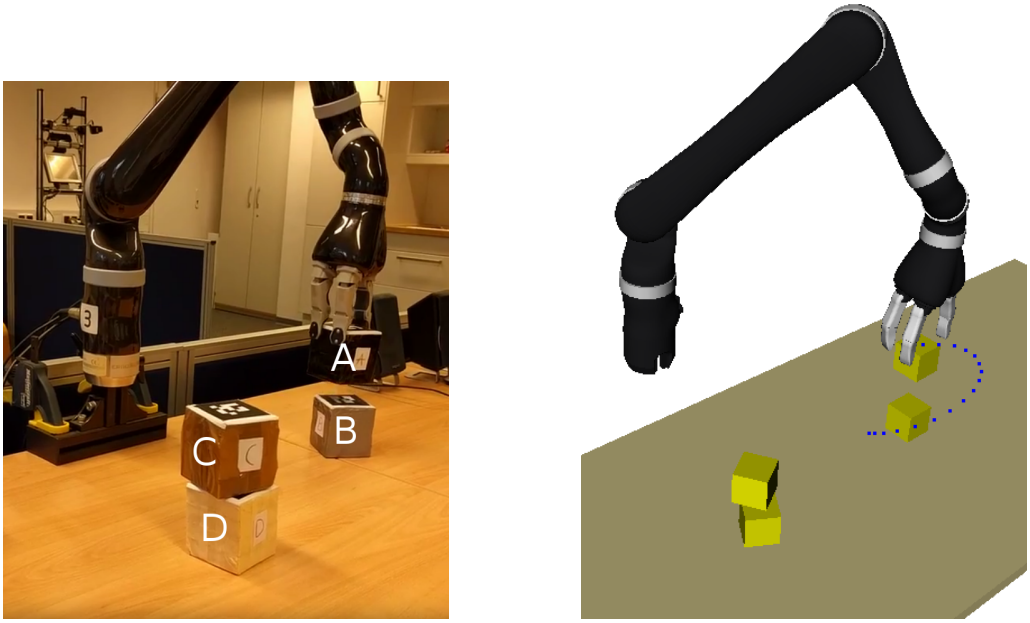


(a) At the RoboCup 2016 in Leipzig (b) In the Gazebo simulation

Figure 5.2: Field of the RoboCup Logistics League

development because we had no full RCLL field available and properly setting up all robots takes a lot effort. However, this does not make a significant difference for the robot memory because the simulation can be distributed over multiple machines and because the interface and usage is identical in simulation and real world. Also the robot behavior, perception results, and factory layout (Figure 5.2) are similar. The environment agency by the Refbox is identical. Different are WiFi issues occurring at RoboCup competitions because the simulation only introduces a message loss for UDP packages.

The distribution of the robot memory in the RCLL is achieved by configuring the robot memory to use the replica set with all robots and loading the `robot-memory` plugin on all robots of the team. The world model existing in CLIPS is already properly initialized at the start of the game and has to be transferred to the robot memory. During the setup phase, one of the agents removes the old world model in the robot memory, which persisted from a previous game. Then all facts defining the world model are inserted into the robot memory by using the fact-to-document function of the CLIPS robot-memory feature. All facts that are represented in the robot memory have the unique field `sync-id`. This field is used to clearly identify which fact is represented by which document and vice versa. During the game, facts inside CLIPS are modified, for example because a robot commits to a task, observes a change, or applies the effect of actions it performed. For this modification, the agent uses the special modify function `(synced-modify)`, which uses a syntax similar to the original `(modify)` function of CLIPS. `(synced-modify)` applies the modification to the fact that should be modified by calling `(modify)` and furthermore updated the corresponding document in the robot memory by using the `sync-id` in the query. Similarly there are the functions `(synced-assert)` and `(synced-retract)`. Other robots are notified about such a change in the robot memory by using triggers. Each agent registers a trigger at its own `robot-memory` plugin. Whenever there is a change, a trigger fact is asserted according to Section 5.3.2 and an update rules consumes this fact and updates the fact with the corresponding `sync-id`.



(a) Arm picking up block `A` (annotated) (b) Same situation in OpenRAVE scene

Figure 5.3: Blocks world scenario with Jaco arm

5.4.2 Blocks World with a Robotic Arm

The second application scenario is the blocks world with a robotic arm. Blocks world is a classical planning problem about planning the actions of a robotic arm to stack blocks until a given goal state is reached [68]. An example goal is stacking the blocks `A` on `B` and `B` on `C`. If the initial state is shown in Figure 5.3a, the planner has to figure out that a solving sequence of actions is to put `A` on the table first, stack `B` on `C` second, and stack `A` on `B` third because the arm can only hold one block at a time. Implementing this scenario with a robot arm, is a good evaluation scenario for the robot memory, because in contrast to the theoretical problem, multiple components (perception, planners, executive) have to work together and there are additional problems such as occlusion and geometric representation and manipulation. Figure 5.3a shows this scenario and the Kinova Jaco arm we are using. The scenario is inspired by the RoboCup@Home league, which also deals with hybrid reasoning, perception, manipulation, planning, and memorizing observations in the real world. Furthermore we choose this scenario because it requires many features of the robot memory. In the following, we present the different components operating in this scenario and how they collaborate by using the robot memory:

Perception: Blocks are located and identified by a marker attached to the top surface. We use the results of the detection by the `tag-vision` plugin with a Kinect version 1 camera mounted on the ceiling. We implemented the `blocks-world-perception` plugin, which fetches the position of detected markers and updates the last observed position in the robot memory. Because the `tag-vision` plugin publishes its results on the blackboard, we use the blackboard computable to obtain the interface data as documents and

the transform computable to transform block coordinates from the camera frame into the map frame. Furthermore this plugin provides computables for transforming the 3D positions into symbolic information needed later by a PDDL component. It provides the symbolic information if a block is on the table by comparing the height of the block to the height of the table and if a block is clear to be grabbed by checking if the marker is currently visible. There is also a computable describing unknown blocks. If Figure 5.3a shows the current situation, the plugin can detect that block `C` is not on the table. By using the information in the robot memory, the plugin also detects that there is an unknown block below `C` because the robot memory does not contain any information about `C` standing on another block yet.

PDDL-based planner: The classical part of the blocks world scenario is solved by a PDDL planner. There is a fixed domain description describing the available predicates `on, holding, on-table, clear, arm-empty` and the possible arm actions `pickup, putdown, stack, unstack`. We use the PDDL robot memory interface to generate the problem description based on the current state represented in the robot memory. The template is an extended version of Listing 5.10 and fills in the information, which is memorized in the robot memory or provided by computables of the `blocks-world-perception` plugin. All this information is symbolic and spatial coordinates are filtered out so that the PDDL planner can work with it. The resulting plan provided by the FF planner is inserted into the robot memory.

CLIPS executive: We use CLIPS executive as top level executive for user interaction, invocation of the planner, and monitoring plan execution. When a new goal state is specified by the user, the executive invokes the PDDL model generation with the goal first and then starts the planner. It uses a trigger to be notified when the resulting plan is added to the robot memory. When planning failed, it starts again, otherwise the plan is converted into a CLIPS task with several steps as specified by the plan. The task-step construction of the executive is also used in the RCLL CLIPS agent. The task is executed step by step. For each step, the executive calls the behavior engine [53] to execute actions defined in the step. Because the behavior engine requires the concrete position where the arm should pick or put a block, the executive queries the robot memory to obtain the position of the block with the name given in the action. For the putdown action, the executive cycles through free positions on the table. After successfully executing a step, the executive updates the world model in the robot memory. For example when block `B` was stacked on `C`, it adds a document describing that `B` is now placed on `C` and removes the document describing that `B` is hold. These updates allow the system to remember the current state because the vision can no longer detect `C` due to occlusion by `B`. When a step fails, the execution of the plan is aborted and the executive starts again with model generation and planning. In most cases, the system can continue because the current state is remembered in the robot memory.

OpenRAVE motion planner: The collision free motion of the arm is planned with OpenRAVE. Figure 5.3b shows the motion planner scene of OpenRAVE and the path it

planned in blue. The scene is constructed with by the OpenRAVE robot memory interface, which queries the position, name, and type of all objects in the environment from the robot memory. An example document, which can be inserted from the robot memory into the OpenRAVE scene is Listing 5.14. Positions are transformed by the transform computable into the frame of the robot arm, which is used by OpenRAVE. The scene construction is invoked by the executive between steps. The motion commands OpenRAVE plans and controls are send by the behaviors executing each step.

Chapter 6

Evaluation

Qualitatively, Section 6.1 evaluates how well the robot memory fulfills the goals defined in the approach. After some remarks about the context of the quantitative evaluation in Section 6.2, we analyze the durations of different robot memory operations in Section 6.3. This includes comparing the durations for different expressiveness levels and database sizes to find a compromise between expressiveness and performance. Furthermore, we analyze the overhead of database operations introduced by robot memory features. In Section 6.4 we present the performance of the system in both introduced application scenarios. This includes CPU and memory usage, network throughput, and statistics about robot memory operations.

6.1 Qualitative Evaluation

As a guideline for the quantitative evaluation, we use the goals defined in Section 4.1. We discuss how well the features of the robot memory work based on the quantitative evaluation and the experience using the robot memory in the RCLL and blocks world application scenarios. Both scenarios were chosen in a way that all important robot memory features are covered.

Flexible Storage and Retrieval: As demonstrated in the RCLL and blocks world, the robot memory properly works as storage for robot data, information, and knowledge. Insert, query, modify, and remove functions work as supposed and are quality controlled by unit-tests. The robot memory is flexible enough to maintain various kinds of information, such as symbolic knowledge about a world model in the RCLL, spatial information about block positions in the blocks world, and temporal information about how long cached information has to be kept. From our experience in the application scenarios, complex queries are rarely required and can often be made faster by reformulation. Because of JavaScript usage, aggregation, and Map-Reduce, queries are very expressive. However due to the document-oriented nature, it is inconvenient to join documents from different collections as it would be done in relational databases. This problem rarely occurs because the document structure allows storing additional information, e.g., by utilizing

sub-documents. The storage and retrieval functions of the robot memory are available to a variety of components via the C++ interface or the interface providers for PDDL, CLIPS, and OpenRAVE.

Memory Sharing Between Knowledge Based Systems: The robot memory allows memory sharing between multiple knowledge based systems. In the RCLL, we share the world model between multiple robots using the same CLIPS reasoning engine. In the blocks world scenario, the memory is shared between different kinds of components, namely a symbolic PDDL planner, the motion planner OpenRAVE, a CLIPS executive, and a perception plugin. The robot memory ensures consistency because it is a central storage and KBS can keep their internal world model updated with triggers. Multiple state estimations by different systems are avoided. As shown in both application scenarios, the robot memory is well suited as basis for the collaboration and combination of multiple KBS. Accessing a common storage is more convenient than accessing each others working memory because of a well defined interface and higher exchangability. Another advantage of memory sharing with the robot memory is that different planners can easily be exchanged and thus compared. In the RCLL, we provide the world model in the robot memory and show how a planner and CLIPS as executive can work together. This is also the foundation for future work with different planning systems in the RCLL.

Special Views for Different Components: Special views on the robot memory are primarily realized by the used queries. A query can filter which part of the robot memory is shown to a component by selecting a collection and filtering documents with the query language of MongoDB. Queries can also be used to rename keys of key-value pairs in returned documents. The robot memory does not directly rename keys or values. Thus, if some planner uses another identifier for an object than a reasoner, one of them has to implement the mapping, for example by providing a computable. The robot memory also simplifies hybrid reasoning by providing a symbolic or spatio-temporal view if appropriate computables are registered. For example, a computable in the blocks world provides `on-table` predicates, which are derived from block positions. Special view on the robot memory are also provided by model generators such as the PDDL model generator, which creates a PDDL problem description from information represented in the robot memory. In a distributed system, the robot memory focuses the view on the shared and local memory of a robot and hides the local memory of other robots.

Computation on Demand: The robot memory provides computation on demand with computables. In the application scenarios, computables are a useful tool to provide functions of perception components to KBS, which only need to execute a usual robot memory query. They are also useful for hybrid reasoning because they can transform information in the robot memory to be usable by different components, as described in the previous paragraph. Compared to computing everything in advance, computables are more efficient because they are only evaluated when required and for required parameters. A disadvantage we noticed when heavily using computables in the blocks world scenario is that the data flow over computables is difficult to track and thus to debug during the

development. This is caused by the volatility of computed documents in the robot memory and the flexibility of document-orientation. Similar to programming languages with dynamic typing, comparisons can unintendedly fail, for example because of spelling mistakes in a key name or comparison of the string `"1"` with the number 1. Because queries only return an empty result instead of throwing an error in such cases, the developer has to search for the problem. We simplify this search by providing a configurable logging mechanism of executed queries, computables, and results. A major advantage of computables is the added extensibility by allowing any computation function to be interfaceable over the robot memory. For example it is possible to compensate for not choosing a common sense reasoner as basis for the robot memory by integrating it with computables. KnowRob, for example, could be connected to the robot memory by querying information from the robot memory, adding it to its ontologies, and providing KnowRob queries via computables.

Distributed Memory for Multi-Robot Systems: In a multi-robot system, the robot memory can be distributed and thus allows information sharing. It is robust against failures of single robots. The content of the robot memory is eventually consistent over the whole system because changes are distributed with a slight delay. As showed in Figure 6.5, the network usage is small and only depends on changes in the robot memory. In the RCLL application scenario, the synchronization of the world model with the robot memory is convenient. It frees the CLIPS agent from implementing its own synchronization and provides more features. Especially useful is the focus on sending only changes and detecting when a full re-synchronization is necessary. Previously the CLIPS agent sent the full world model once a second because it did not know which changes other agents already received. The distributed robot memory works well together with triggers to keep the world model updated for all CLIPS agents.

Spatio-Temporal Grounding: The robot memory allows spatio-temporal grounding of the information stored in it. Due to the flexible structure and representation in documents with various types, such as floats and dates, adding spatio-temporal information is simple. For example, we use spatial positions of objects in the blocks world and times for caching of computables. The robot memory can also consider spatio-temporal information in queries. Simple operations, such as comparison, can be expressed in the query language and complex operations can be performed by computables as, for example, shown by the transform computable.

Triggers: By using triggers, components can be notified about changes in the robot memory. This is especially useful for keeping a world model in a KBS updated according to the robot memory. In the RCLL application scenario, this works well and frees the agent from implementing synchronization or pulling mechanisms to search for changes. Another beneficial use case for triggers is message passing, e.g., for passing a plan to an executive. This way of message passing is simple. KBS can use the existing interface to the robot memory and messages can be changed in the development with small effort because documents have no fixed structure in contrast to messages over the blackboard or Protobuf. To ensure a reasonable computation time, triggers only analyze the changes

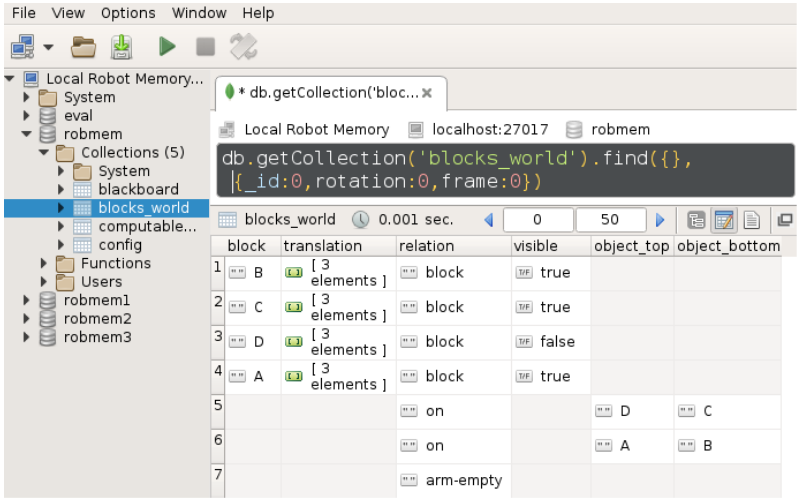


Figure 6.1: Robomongo screenshot

caused by single operations on the robot memory. We considered allowing triggers to describe events of whole collections, such as the amount of objects exceeding a threshold. We decided against it because this would require periodically evaluating queries over whole collections and could slow down the robot memory for large collections or complex queries. Analyzing single changes in the robot memory suffices for use cases, such as keeping a world model updated. More complex problems, such as monitoring the amount, can still be implemented efficiently by creating a trigger for the insertion and deletion of such objects and then checking the object count in the callback function. Furthermore, triggers can not notify about documents provided by computables without invoking the computable from the application. This is due to the theoretical complexity of verifying if an arbitrary function yields a specified result for any input values. However triggers do work on the result of computables. Thus the application has to specify when and with which query the computable is invoked. Because of these two restrictions of triggers, they are computationally fast as shown in Section 6.3.5. They work well in both application scenarios and are convenient for components using the robot memory.

Persistent Storage: The robot memory provides a persistent storage. The content of the robot memory is stored on the hard-drive and remains after restarting Fawkes or the whole system. Thus the robot memory enables a robot to have a long-time memory. It is possible to mark documents that should not be stored persistently, by adding a expiration time or a key-value pair describing that this document should be removed during a restart. In a multi-robot system, the distributed robot memory is even robust against a restart of the whole system because of the persistency.

Human Interface and Visualization: There are multiple possibilities to manually interface the robot memory during the development. On the one hand, it is possible to monitor and modify the content of the robot memory by using MongoDB tools. There are the `mongo` command line program provided by MongoDB and GUI based programs, such as *Robomongo*[1], which provides a table like view on collections as shown in Figure 6.1

[1] `https://robomongo.org/`

```
{ name: "99898", position: "299", tidied: "299" }
{ name: "42305", position: "942", tidied: "231" }
```

Listing 6.1: Documents of the tidy up scenario. One is misplaced.

and allows to perform modifications without being familiar to the MongoDB syntax. This leads to simple and efficient monitoring during the development. On the other hand, the robot memory can be accessed over blackboard interface messages, which can be sent over a browser using the `webview` plugin. Thus it is possible to use robot memory features, such as computables. To support debugging, the robot memory logs operations, computable calls, and trigger activations when specified in the configuration. It is possible to visualize spatio-temporal data in the robot memory. For example, the computation times in Figure 6.2 were inserted as documents in the robot memory and plotted using the `matplotlib` and `pymongo` libraries for Python.

6.2 Evaluation context

The benchmarks presented in this chapter were performed on Fedora 23 systems with 16 GB RAM, a spinning hard-drive, and an Intel Core i7-3770 with four hyper-threading enabled cores at 3.4 GHz. For local benchmarks, everything, including Fawkes, MongoDB, the Gazebo simulation and ROS, ran on a single system. For distributed benchmarks, three identical systems were used. Each system hosted its own Fawkes, ROS, and MongoDB processes, and one system additionally, hosts the Gazebo simulation.

Since the robot memory runs inside the Fawkes framework and thus inside the Fawkes process, the performance of the robot memory is influenced by Fawkes and can not directly be measured separately. The influences are caused by other components, such as localization, reasoning, and collision avoidance, and by the thread management of Fawkes. To softly guarantee loop times for other threads, Fawkes suspends threads with a too long loop iteration. When measuring durations of robot memory operations, this can increase large measurement values because if an operation takes longer than the allowed loop iteration time, the time being suspended is added to the measurement.

6.3 Robot Memory Operations

In this section, we analyze the run-time durations of robot memory operations, such as insert, query, and update. We also analyze how these change with computables and triggers. The goal is to find out how fast the robot memory works. Besides the influence of the implementation, the durations heavily depend on the underlying complexity, e.g., of a query, and *domain size*, which is the amount of information stored in the robot memory. Here, we want to find out how the durations scale with increasing domain size and increasing complexity.

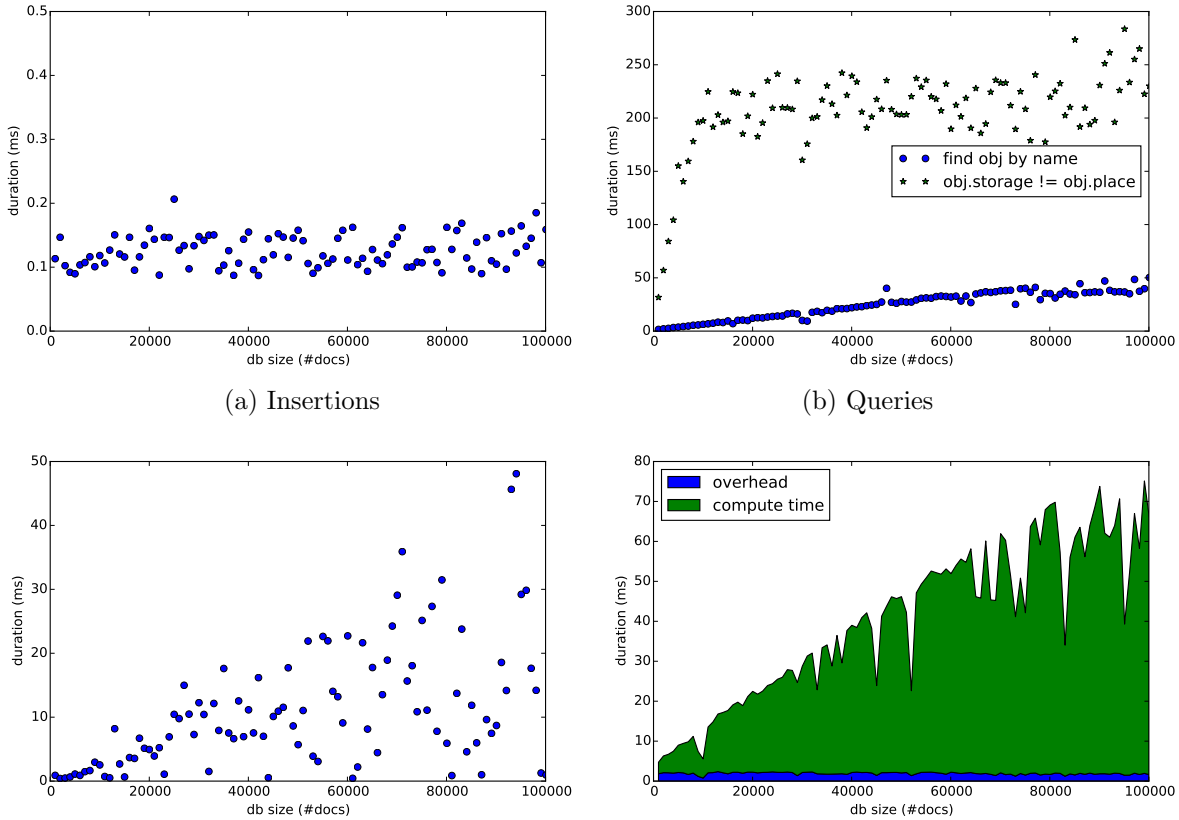
The structure of the data used for this evaluation is chosen in a way to be similar to information that is usually stored in a robot memory. Furthermore, the structure should simplify the analysis. The used data mimics a robot's knowledge about objects and their positions to solve a tidy up scenario. For each object, the robot memory contains a document with the object's name, the current position, and the tidied place it belongs to. Names are unique and tidied places are randomly chosen out of 1000 possibilities. Objects are at a random misplaced position with probability 0.01 and at their proper tidied place otherwise. The object's name is a string to enforce string comparisons, which are expected to be often needed in a real scenario. Two example documents are shown in Listing 6.1. At first, we do not use indexing for the analysis of operation durations because in the worst case queries do not benefit from indexing. In Section 6.3.6 we evaluate the impact of indexing separately. An evaluation plugin in Fawkes generates the data and calls random operations. To analyze how the durations scale with the domain size, the plugin calls the operations repeatedly after increasing the domain size from 0 to 100000 documents by steps of 1000 documents.

The analysis is separated into the main classes of operations, namely insertions in Section 6.3.1, queries in Section 6.3.2, and updates in Section 6.3.3. Because deletions behave very similar to updates, both search for documents and work with them, they are also covered in the section about updates. Section 6.3.4 covers and explains the impact of computables on query durations and Section 6.3.5 deals with the impact of event triggers. Operations that are performed rarely are not analyzed. For example dropping or restoring a collection and registration of triggers and computables are only performed once during initialization. In the course of implementing and evaluating the application scenarios, their durations were not noticeable compared to the initialization time of the whole robot software.

6.3.1 Insertions

The insertion of a new document into the robot memory is overall the fastest operation. Figure 6.2a shows the distribution of insert durations with increasing domain size. It is noticeable that the duration does not depend on the domain size. The insertion time is small with an average of $0.125ms$. However, there are single outliers not present in Figure 6.2a. These outliers occur periodically, when MongoDB flushes database changes from the RAM to the hard drive. We could reduce the occurrence frequency of these outliers to one in $60s$, which is the default flushing frequency of MongoDB. To reach this, we had to disable the forward logging of MongoDB, called Journaling. The outliers are not shown in Figure 6.2a because it is unlikely to hit one outlier in $60s$ by measuring the duration of each 1000th insertion and the insertion of all 100000 documents lasts less than $15s$. When measuring all insertion durations, outliers took about $0.3s$ because of the large amount of written documents and the threading in Fawkes. The overhead of the robot memory compared to the duration of the raw MongoDB operation has an average

of 0.006ms, which is about 5.5%. The overhead is caused by verifying the collection name, deciding if the local or distributed database is used, adding meta-data, and mutual exclusion of database usage. The robot memory overhead measured here does not include checking if triggers were activated by inserting the document. This is caused by the asynchronous and separate checking by the trigger manager. The size and the complexity of inserted documents did not make a noticeable difference.



(a) Insertions

(b) Queries

(c) Updates

(d) Computables

Figure 6.2: Duration of robot memory operations with increasing domain size (Different duration scale for each operation used)

6.3.2 Queries

Queries are more complex operations than insertions and also take more time. Figure 6.2b shows that query durations heavily depend on many factors, such as domain-size and query complexity. The query durations, when searching for an object by its unique name, linearly depend on the domain size. The other plot shows the query durations for finding objects that are not at their tidied place and thus would have to be tidied up. The query is more complex because it contains a comparison between two values that is formulated in JavaScript. From 0 to about 10000 documents, also a linear increase can be seen. With more documents, the average duration does not increase because of a query evaluation trick

of MongoDB. MongoDB lazily evaluates queries and returns a cursor to the first batch of n documents that match the query. While the user iteratively fetches documents from the cursor, MongoDB continues the query evaluation if the batch was emptied. For the `'find object name'` query, the whole database was searched because the unique object could not fill the batch. However it is possible to request a smaller batch, e.g., only one document, for faster computation. The overhead of the robot memory query operation does not depend on the domain size and has an average duration of $0.6ms$. The largest part of that overhead is caused by the computable manager, even if in this scenario no computables are registered. The computable manager performs a single insert and remove on a separate collection to prepare checking if the query matches a computable and to clean up afterwards. Because of this insertion, the duration spikes observed for insertions can also occur for queries. As Figure 6.2b shows, the query duration heavily depends on the complexity of the query. Especially the expressive JavaScript functions are expensive compared to the MongoDB internal query language without JavaScript. By formulating queries smartly, complex questions can often be efficient in practice. For example, checking whether the dishwasher is full is a complex question because it considers a set of document that have to fulfill the condition of being in the dishwasher, and their aggregated volume has to exceed a threshold. Thus it could be a computationally costly JavaScript function. But the query can also be formulated efficiently by first filtering only the objects in the dishwasher with a fast JavaScript-free query and then summing up the objects volume to check if it exceeds a given threshold. This way the costly sum computation, which can be done by aggregation or JavaScript, only needs to be computed for the few objects in the dishwasher.

6.3.3 Updates and Deletions

Updates consist of a query defining what should be updated and some key-value pairs, which should be changed. Thus the duration of an update should be similar to a query plus an insertion. Figure 6.2c shows the duration of position updates of objects with a random name. Because the used update operation only updates a single document, the durations are roughly uniformly distributed with an upper bound matching the duration of `'find object'` queries in Figure 6.2b. Thus the duration of update operations also scale linearly with the domain size. Durations of queries searching for only one document also behave as in Figure 6.2c. Because delete operations also consist of a query and a fast change operation, they behave similar to updates. The difference is that by default deletions are executed on all documents and updates only on a single one. For both deletions and updates, the overhead does not depend on the database size. Both updates and deletions have an average overhead duration below $0.02ms$. In contrast to query operations, the computable manager is not involved, because computed documents should not be updated and they do not need to be removed manually.

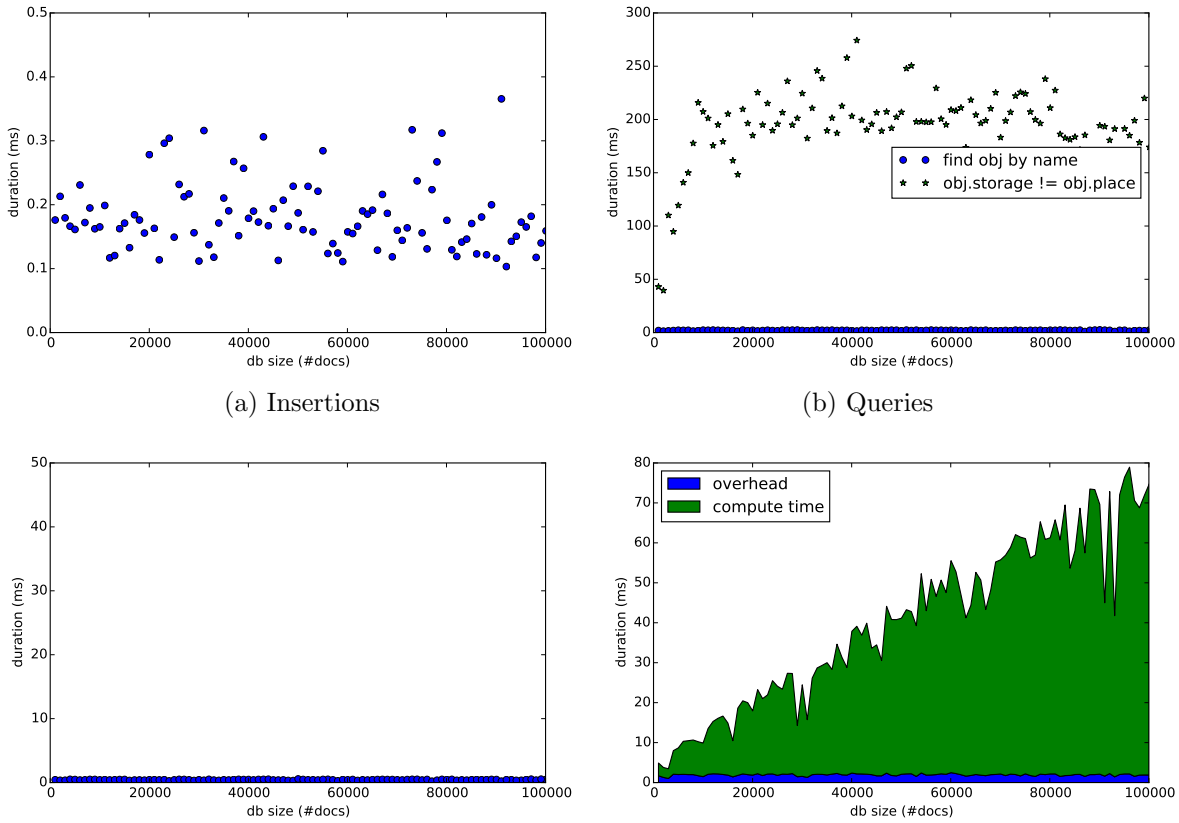
6.3.4 Computables

Computables generate new information when it is demanded by a query. Thus each new query is checked by the computable manager as explained in Section 6.3.2. Figure 6.2d shows the time needed for checking if a query demands a computable and for computing the results. The computable used in Figure 6.2d calculates the distance of the robot to the closest object in the database and yields the object's name, position, and distance. The measurement does not include the query duration of finding the computed document. The computation time of the computable itself scales linear with the domain size because each object's distance needs to be checked. The time of the overhead does not depend on the domain size and has an average duration of $1.8ms$. The overhead includes inserting the original query in MongoDB, evaluating the demand query for each registered computable to check if the original query demands the computable, and a clean up operation. Furthermore, the results of the computable need to be inserted into the robot memory. Asynchronously in the main loop of the robot memory, computed documents are removed again. Thus the time needed for computables depends on the amount of registered computables, the amount of computed documents, and mainly the computation time needed for each matching computable itself. This computation time can get large if the computable solves problems with higher complexity. For example a computable that finds the pair of objects with the smallest distance to each other would scale quadratically with the domain size. To avoid repeated computation of the same data, we implemented caching for computables. As long as specified by the caching time, the computation step is skipped if the new query is identical to the query that demanded the computation before. The remaining overhead of the computable manager is the same.

6.3.5 Trigger

Triggers notify an application when something in the database changes as specified in the trigger query. For this benchmark, we registered a trigger on new objects in a certain location, e.g., in the laundry box. Whether a trigger event happened, is checked each loop iteration of the robot memory by the trigger manager. Because trigger queries, which check for these events, are only executed on the Oplog and not on the whole database, the time needed to check trigger depends on the amount of registered trigger and the amount of database changes (insertions, updates, and deletions). The current domain size has no influence. During the insertion of the 100000 objects, the trigger manager checked for trigger events 2249 times with an average duration of $0.27ms$. That makes $0.006ms$ per insertion. This duration also depends heavily on the formulation of the trigger query, e.g., in the case of JavaScript usage. These measurements do not include the computation time needed by the callback function of the trigger. Because the callback function is executed directly, its computation time influences the loop time of the robot memory and can in the worst case block it.

6.3.6 Indexing



(a) Insertions

(b) Queries

(c) Updates

(d) Computables

Figure 6.3: Duration of robot memory operations with indexing. For each operation the duration scale is identical to Figure 6.2

As Figure 6.3 shows in comparison to Figure 6.2, indexing by the object name has a significant impact on some query and update durations. The scales used both figures are identical for the same operation. Both can be faster if the query is restricting the value of the key used as primary index because then the built index can be used to search for documents with this key-value pair. 'Find object by name' queries in Figure 6.3b have an average duration of $1.96ms$ and updates in Figure 6.3c an average duration of $0.4ms$. If the query only restricts non indexed key-value fields or a comparison of values, such as the query about places in Figure 6.3b, the index can not be used. In this case, query durations are not noticeably affected because it is still necessary to search through all documents until the batch is full. The duration of insertion operations is larger with indexing because MongoDB has to update the index. The durations for computables do not change in this scenario because the function searching for the closest object still has to iterate over all documents.

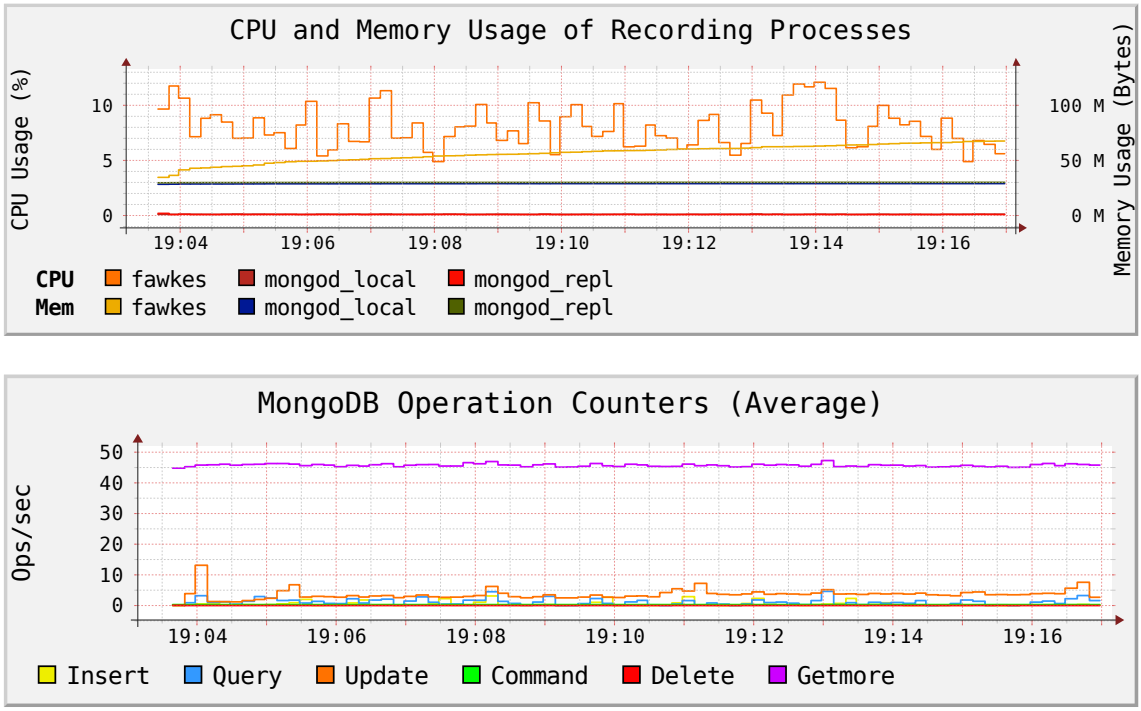


Figure 6.4: Benchmark during a locally simulated RCLL game

6.4 Performance in Evaluation Scenarios

To evaluate the performance of the robot memory, we performed benchmarks in multiple application scenarios. The measurements during the experiments were stored and plotted with the Round Robin Database Tool (RRD)[2] as in [56].

The first scenario is a **simulated RoboCup Logistics League** game, in which the synchronization of the world model between the robots was implemented with the robot memory. Figure 6.4 shows the CPU and memory usage, as well as the operations performed by MongoDB. For clarity only the CPU and memory usage of one Fawkes instance of the three robots is plotted. Both MongoDB processes, one for the robot local robot memory and one for the distributed worldmodel, have a CPU usage below 1% and a memory usage below 40 MB. This is not surprising because the robot's world model that needs to be synchronized in the RCLL only has a size of 60 to 100 documents with an average object size of 390 bytes. It only grows slowly because most of the documents are updated and there are few new documents for incoming orders and and work-pieces in production. This is also shown by the MongoDB operation count shown in Figure 6.4, which matches the operation count of the robot memory. The values include the database operations of all three robots. Figure 6.4 shows that the mostly used operation is the *Getmore* operation, which checks for an existing query cursor if there are more matching documents than in the previous batch. This is caused by the trigger manager because it periodically checks

[2] `http://www.rrdtool.org`

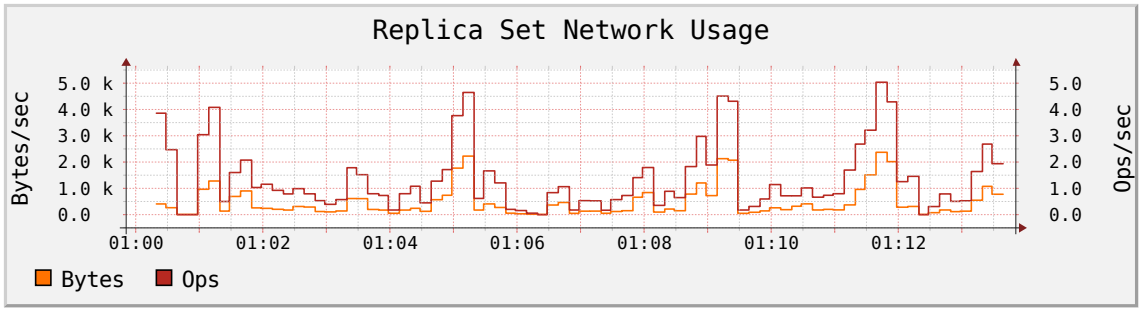


Figure 6.5: Network usage of distributed robot memory during an RCLL game

the Oplog for new changes that have to be reported to the CLIPS agent. Since the robot memory runs inside the Fawkes process, its CPU and memory usage is included in the values of Fawkes in Figure 6.4. The growing memory usage of Fawkes is also present in benchmarks of the RCLL simulation without the robot memory and the CPU curve looks similar. Thus the robot memory plugin has no large impact on the performance in this scenario. The size of the database on the hard-drive is about 1097 MB for the distributed part and 1060 MB for each local one. The size does not change noticeably over time. 1 GB of each database is occupied by the Oplog, which has a constant size because it is a capped collection.

Figure 6.5 shows the network usage of a Replica Set to distribute the robot memory over the multi-robot system in the RCLL. The measurements are provided by MongoDB's server statistics and only consider the packages sent from the Primary to a Secondary. MongoDB uses TCP for Replica Sets. The network usage mostly stays below 1 KB/s and rarely reaches about 2 KB/s. The usage is roughly proportional to the number of operations. For comparison, the previous CLIPS agent, which implements the world model synchronization with Protobuf messages, sends between 7 KB/s and 9 KB/s in about 12 UDP packages over broadcast. The previous CLIPS agent acting as Master sends the whole world model to other robots once a second because it does not know if previous UDP packages reached other robots or if a robot was restarted and requires the whole world model again. What is interesting to notice in Figure 6.5 is that the amount of registered operations is smaller than the amount of insert, update, and delete operations in Figure 6.4. This is due to updates by the CLIPS agent, that do not apply to any document or change a value in document to its current value. This is detected by MongoDB and not distributed over the Oplog, because the operation did not change anything.

The second scenario is the **Blocks World** with the Kinova Jaco arm. The robot memory is included in the following processes: the overhead vision detects markers on blocks and publishes their position on the blackboard. The PDDL model generation constructs a problem description from symbolic information in the robot memory. A part of this information is provided by computables for `on-table` and `clear` predicates. There are also computables for the transformation of block positions and accessing the blackboard. The computables for predicates are provided by a perception plugin. This

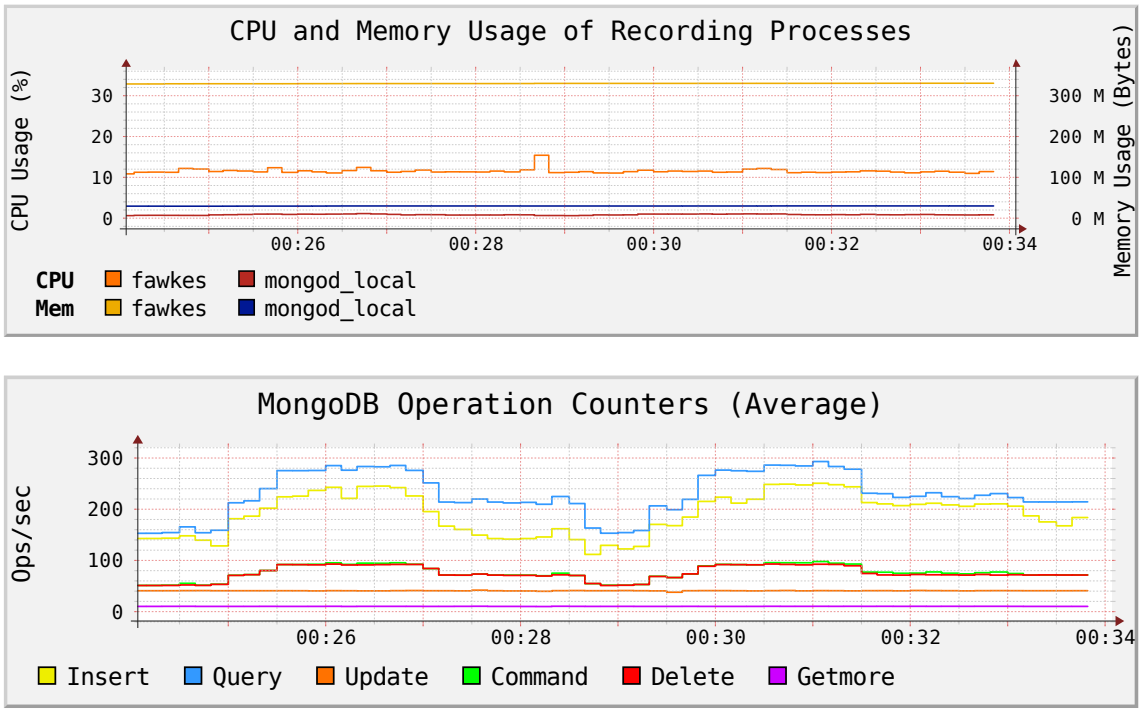


Figure 6.6: Benchmark during Blocks World demo

plugin continuously writes the block positions into the robot memory for demonstration purposes. This is not very efficient because continuously updating block positions requires executing the blackboard computable and transform computable very often. A computable that only inserts block positions when necessary would be more efficient.

Figure 6.6 shows the CPU and memory usage as well as the operation count in MongoDB during the execution of two plans with 8 and 12 actions.

The amount of update operations is constant because the block positions and their visibility state are constantly updated. The large amount of queries, insertions, and deletions is mainly caused by the blackboard and transform computables, which are executed to update the block positions. The blackboard computable inserts the interface values into the robot memory and the transform computable queries these values and inserts the transformed values. For each query that needs to be checked by the computable manager, one insert and deletion is performed as well as a query for each registered computable. The CPU usage of Fawkes and the CPU and memory usage of MongoDB are similar to the results of the RCLL benchmark.

Chapter 7

Conclusion

In this chapter, we summarize the thesis in Section 7.1 and give an outlook to possible future work in Section 7.2.

7.1 Summary

In this thesis, we have designed and implemented a document-oriented robot memory for knowledge sharing and hybrid reasoning on mobile robots. The robot memory allows robots to advance into more complex everyday domains by providing a generic and capable information storage, which is the basis for memorizing a complex environment and thus for reasoning about it. It can represent complex information and expressive queries. As collaboration platform for KBS, it provides a common and consistent information basis. Furthermore, it simplifies hybrid reasoning by storing spatio-temporal information and allowing transformation into symbolic information on demand.

In the theoretical foundation, we specified how information can be represented in documents and how these can be mapped into KBS. The theoretical foundation also describes the concept of computables, which allows components to provide computation on demand. By using query matching, the robot memory checks if information covered by the computable is queried and then invokes the function of the component providing the computable. As discussed in the evaluation, computables are convenient for transforming hybrid information and for interfacing perception. Another important concept is notification about changes in the robot memory with triggers. The robot memory searches the Oplog of MongoDB with the query defining the trigger event and invokes the callback function if there is a matching change. Triggers are useful for keeping internal world models of KBS updated. They also have turned out to be convenient for message passing.

In contrast to other approaches from related work, our robot memory supports sharing information in a multi-robot system. We built a robust, consistent, and efficiently distributed memory by utilizing Replication Sets of MongoDB and distinguishing between a local and a shared database. In this context, triggers are useful for notifying about updates by other robots. Based on the layered architecture, we implemented the back-end

database with MongoDB and the core robot memory containing the operation-, trigger-, and computable-manager. To provide the robot memory to reasoners and planners, we implemented interface providers for PDDL, CLIPS, and OpenRAVE. The PDDL interface can generate problem descriptions from a template and the information contained in the robot memory. The CLIPS interface provides CLIPS functions for robot memory operations and can map between documents and facts by comparing key-value pairs to fact template definitions. The OpenRAVE interface builds the motion planner scene from object positions and types represented in the robot memory.

We used the robot memory in two application scenarios for knowledge sharing and hybrid reasoning. Both scenarios showed that the concepts of the robot memory are beneficial. In the RCLL, the robot memory distributes the world model between multiple robots, which use triggers. Important benefits are the robust, simple, and efficient world model synchronization and the persistent storage. In the blocks world scenario, the robot memory is the basis for the collaboration of perception, a PDDL planner, a CLIPS executive, and the motion planner OpenRAVE. Here, the main benefits are the world model sharing between planners and reasoners of different types and the hybrid reasoning support with computables. As presented in the quantitative evaluation, the robot memory works efficiently and only adds a small overhead compared to raw MongoDB operations. The increase in computation time for complex queries and computables is reasonable and caused by the computational complexity of the queries themselves.

An important challenge we faced in the thesis is the conceptual tuning of robot memory concepts to achieve a balance between expressiveness and computation time. For example, we decided that triggers can only be evaluated on single changes of the robot memory instead of the whole set of documents. For queries and computables, we decided to allow all features of the MongoDB query language for high flexibility and expressiveness and make the application responsible for efficient query formulation. Further challenges are the design and implementation of on demand computation in a document-oriented system and making the representations between CLIPS and MongoDB compatible. This is due to CLIPS using strings and symbols, which are both mapped to string values in documents, but still have to be distinguishable.

Concluding, this thesis has designed and implemented a useful, scalable, and persistent robot memory for hybrid reasoning and knowledge sharing. Major benefits are representation of complex information and expressive queries with document-orientation, hybrid reasoning with computables, and world model sharing between knowledge-based systems.

7.2 Future Work

Because the robot memory provides the back-end information storage of a robot system, it is meant to be the foundation for other components using it and thus for future work. Especially planners and reasoners benefit from collaboration possibilities and the hybrid reasoning capabilities of the robot memory. There are two ongoing master theses, which

already use the robot memory and implement new global planning approaches in the RCLL [42, 67]. Both utilize the robot memory as common storage of the world model, which is updated by the CLIPS executive and read by a central planner. Furthermore, they use the robot memory to provide plans to the executive with triggers. Both also base their implementation on prototypes developed in the application scenarios of this thesis. On the one hand, the RCLL application scenario provides the distributed memory, which can be accessed by their planners and is kept updated by the CLIPS executive. On the other hand, the blocks world scenario provides a prototype showing how a CLIPS executive can fetch plans from the robot memory with triggers, represent it as task, and execute it step by step. The first thesis by Björn Schäpers uses reactive answer set programming with real-time constraints [67] and the second one by Matthias Löbach uses a PDDL planner and converts the resulting plan into a simple temporal network [42]. The second one also makes use of our PDDL model generation and considers using computables for travel distance calculations.

Future work on the robot memory itself can further improve it in various ways. The robot memory can be extended by more interfaces for KBS, such as Golog, which can also benefit from the theoretical foundation. Furthermore the robot memory can be improved internally by incorporating confidence values or more knowledge representation concepts, such as beliefs. Potentially beneficial MongoDB features that we did not cover are geospatial indexing and queries. These allow to query documents with location information by proximity or intersection. A possible improvement of computables is the addition of computed key-value pairs in already existing documents.

As part of Fawkes, the robot memory is open source and thus available for many future projects.

Bibliography

[1] Russell L Ackoff. From Data to Wisdom. *Journal of applied systems analysis*, 16(1):3–9, 1989.

[2] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide.* " O'Reilly Media, Inc.", 2010.

[3] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[4] ArangoDB GmbH. ArangoDB v3.1.7 Documentation. `https://docs.arangodb.com/3.1/Manual/index.html`, retrieved Jan 18th 2017.

[5] Daniel Beck, Alexander Ferrein, and Gerhard Lakemeyer. A Simulation Environment for Middle-Size Robots with Multi-level Abstraction. In *RoboCup Symposium*, 2008.

[6] Kent Beck. *Extreme Programming Explained: Embrace Change.* addison-wesley professional, 2000.

[7] Kent Beck. *Test-Driven Development: By Example.* Addison-Wesley Professional, 2003.

[8] Michael Beetz, Moritz Tenorth, and Jan Winkler. Open-EASE. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 1983–1990. IEEE, 2015.

[9] George A Bekey. *Autonomous Robots: From Biological Inspiration to Implementation and Control.* MIT press, 2005.

[10] Reinaldo A. C. Bianchi, H. Levent Akin, Subramanian Ramamoorthy, and Komei Sugiura. *RoboCup 2014: Robot World Cup XVIII.* Springer, 2014.

[11] Joschka Boedecker and Minoru Asada. Simspark–concepts and application in the robocup 3d soccer simulation league. *Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, 2008.

[12] R.J. Brachman and H.J. Levesque. *Knowledge Representation and Reasoning.* The Morgan Kaufmann Series in Artificial Intelligence Series. Morgan Kaufmann, 2004.

[13] Herman Bruyninckx. OROCOS: design and implementation of a robot control software framework. In *Proceedings of IEEE International Conference on Robotics and Automation*. Citeseer, 2002.

[14] Frank Buschmann, Kevin Henney, and Douglas C Schmidt. *Pattern-oriented software architecture, on patterns and pattern languages*, volume 5. John wiley & sons, 2007.

[15] Rick Cattell. Scalable SQL and NoSQL Data Stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[16] Sachin Chitta, Ioan Sucan, and Steve Cousins. Moveit![ROS topics]. *Robotics & Automation Magazine, IEEE*, 19(1):18–19, 2012.

[17] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly, 2013.

[18] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[19] Mark Collins-Cope. Component based development and advanced OO design. Technical report, Technical report, Ratio Group Ltd, 2001.

[20] RCLL Technical Committee. RoboCup Logistics League: Rules and Regulations 2016. `http://www.robocup-logistics.org/rules`, 2016.

[21] Thomas M Connolly and Carolyn E Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Pearson Education, 2005.

[22] Rick Copeland. Working with MongoDB MultiMaster. `https://dzone.com/articles/working-mongodb-multimaster`, retrieved Jan 20th 2017.

[23] Carlos Coronel and Steven Morris. *Database Systems: Design, Implementation, & Management*. Cengage Learning, 2016.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[25] Rosen Diankov and James Kuffner. OpenRAVE: A Planning Architecture for Autonomous Robotics. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, 79, 2008.

[26] Lucas Dohmen. A Declarative Web Framework for the Server-side Extension of the Multi Model Database ArangoDB. Master thesis, RWTH Aachen University, Chair of Information Systems Databases, 2014.

[27] Kalpana Dwivedi and Sanjay Kumar Dubey. Implementation of Data Analytics for MongoDB Using Trigger Utility. In *Computational Intelligence in Data Mining—Volume 1*, pages 39–47. Springer, 2016.

[28] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1), September 1982.

[29] Ghislain Fourny et al. *Jsoniq the SQL of NoSQL*. Citeseer, 2013.

[30] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20, 2003.

[31] Joseph C. Giarratano. *CLIPS Reference Manuals*, 2007. `http://clipsrules.sf.net/OnlineDocs.html`.

[32] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *arXiv preprint 1106.0675*, 2011.

[33] ECMA International. *The JSON Data Interchange Format*, 2013.

[34] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented Programming*. Springer, 1997.

[35] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The Robot World Cup Initiative. In *1st Int. Conference on Autonomous Agents*, 1997.

[36] Hiroaki Kitano, Satoshi Tadokoro, Itsuki Noda, Hitoshi Matsubara, Tomoichi Takahashi, Atsuhi Shinjou, and Susumu Shimada. Robocup Rescue: Search and Rescue in Large-Scale Disasters as a Domain for Autonomous Agents Research. In *Systems, Man, and Cybernetics, 1999. IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on*, volume 6, pages 739–743. IEEE, 1999.

[37] Graham Klyne and Jeremy J Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. 2006.

[38] Nathan Koenig and Andrew Howard. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In *Int. Conf. on Intelligent Robots and Systems*, 2004.

[39] Gerhard K Kraetzschmar, Nico Hochgeschwender, Walter Nowak, Frederik Hegger, Sven Schneider, Rhama Dwiputra, Jakob Berghofer, and Rainer Bischoff. RoboCup@Work: Competing for the Factory of the Future. In *Robot Soccer World Cup*, pages 171–182. Springer, 2014.

[40] Seéverin Lemaignan, Raquel Ros, Lorenz Mösenlechner, Rachid Alami, and Michael Beetz. ORO, a knowledge management platform for cognitive architectures in robotics. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3548–3553. IEEE, 2010.

[41] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 15–19. IEEE, 2013.

[42] Matthias Loebach. Centralized Global Task Planning with Temporal Aspects on a Group of Mobile Robots in the RoboCup Logistics League. Master thesis proposal, Maastricht University, February 2017.

[43] Derek Long and Maria Fox. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research (JAIR)*, 20:1–59, 2003.

[44] Aaron Martinez and Enrique Fernández. *Learning ROS for Robotics Programming*. Packt Publishing Ltd, 2015.

[45] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language. Technical report, AIPS-98 Planning Competition Committee, 1998.

[46] Deborah L McGuinness, Frank Van Harmelen, et al. OWL Web Ontology Language Overview. *W3C recommendation*, 10(10):2004, 2004.

[47] Jim Melton. SQL, XQuery, and SPARQL: what's wrong with this picture. In *Proc. XTech*, 2006.

[48] Bruce Momjian. *PostgreSQL: Introduction and Concepts*, volume 192. Addison-Wesley New York, 2001.

[49] Deebul Nair. Predicting Object Locations using Spatio-Temporal Information by a Domestic Service Robot: A Bayesian Learning Approach. Master thesis, Hochschule Bonn-Rhein-Sieg University of Applied Sciences, 2016.

[50] Ameya Nayak, Anil Poriya, and Dikshay Poojary. Type of NOSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013.

[51] Tim Niemueller. Developing A Behavior Engine for the Fawkes Robot-Control Software and its Adaptation to the Humanoid Platform Nao. Master thesis, RWTH Aachen University, Knowledge-Based Systems Group, April 2009.

[52] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. Design Principles of the Component-Based Robot Software Framework Fawkes. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 300–311. Springer, 2010.

[53] Tim Niemueller, Alexander Ferrein, and Gerhard Lakemeyer. A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao. In *RoboCup Symposium*, 2009.

[54] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. Incremental Task-level Reasoning in a Competitive Factory Automation Scenario. In *AAAI Spring Symposium 2013 - Designing Intelligent AI*, 2013.

[55] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. The RoboCup Logistics League as a Benchmark for Planning in Robotics. In *WS on Planning and Robotics (PlanRob) at Int. Conf. on Aut. Planning and Scheduling (ICAPS)*, 2015.

[56] Tim Niemueller, Gerhard Lakemeyer, and Siddhartha S. Srinivasa. A Generic Robot Database and its Application in Fault Analysis and Performance Evaluation. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2012.

[57] Tim Niemueller, Sebastian Reuter, Daniel Ewert, Alexander Ferrein, Sabina Jeschke, and Gerhard Lakemeyer. The Carologistics Approach to Cope with the Increased Complexity and New Challenges of the RoboCup Logistics League 2015. In *RoboCup Symposium – Champion Teams Track*, 2015.

[58] Tim Niemueller, Sebastian Reuter, and Alexander Ferrein. Fawkes for the RoboCup Logistics League. In *RoboCup Symposium 2015 – Development Track*, 2015.

[59] Tim Niemueller, Sebastian Reuter, Alexander Ferrein, Sabina Jeschke, and Gerhard Lakemeyer. Evaluation of the RoboCup Logistics League and Derived Criteria for Future Competitions. In *RoboCup Symposium*, 2015.

[60] Tim Niemueller, Frederik Zwilling, Gerhard Lakemeyer, Matthias Löbach, Sebastian Reuter, Sabina Jeschke, and Alexander Ferrein. *Industrial Internet of Things: Cybermanufacturing Systems*, chapter Cyber-Physical System Intelligence – Knowledge-Based Mobile Robot Autonomy in an Industrial Scenario. Springer, 2016.

[61] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of JSON and XML Data Interchange Formats: A Case Study. *Caine*, 2009:157–162, 2009.

[62] Oxford Dictionary of English. Definition of 'robot'. Oxford University Press, 2010.

[63] Fábio Roberto Oliveira and Luis del Val Cura. Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages 230–235. ACM, 2016.

[64] Eric Prud, Andy Seaborne, et al. SPARQL Query Language for RDF. *W3C recommendation*, 2006.

[65] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[66] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: a modern approach*, volume 3. Upper Saddle River: Prentice hall, 2010.

[67] Björn Schäpers. Centralized Global Planning using Reactive Answer Set Programming with Real-time Constraints for Autonomous Mobile Robots. Master thesis proposal, RWTH Aachen University, Knowledge-Based Systems Group, November 2016.

[68] John Slaney and Sylvie Thiébaux. Blocks World revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

[69] Peter Stone, Tucker Balch, and Gerhard Kraetzschmar. *RoboCup 2000: Robot Soccer World Cup IV*. Springer Science & Business Media, 2001.

[70] Moritz Tenorth and Michael Beetz. KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots. Part 1: The KnowRob System. *Int. Journal of Robotics Research (IJRR)*, 2013.

[71] Moritz Tenorth and Michael Beetz. Representations for robot knowledge in the KnowRob framework. *Artificial Intelligence*, 2015.

[72] Moritz Tenorth, Ulrich Klank, Dejan Pangercic, and Michael Beetz. Web-Enabled Robots. *Robotics & Automation Magazine, IEEE*, 18(2):58–68, 2011.

[73] Loy van Beek, Kai Chen, Dirk Holz, Mauricio Matamoros, Caleb Rascon, Maja Rudinac, Javier Ruiz des Solar, and Sven Wachsmuth. RoboCup@Home 2015: Rule and regulations. `http://www.robocupathome.org/rules/2015_rulebook.pdf`, 2015.

[74] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the 48th annual Southeast regional conference*, page 42. ACM, 2010.

[75] Thomas Wisspeintner, Tijn Van Der Zant, Luca Iocchi, and Stefan Schiffer. Robocup@home: Scientific competition and benchmarking for domestic service robots. *Interaction Studies*, 10(3), 2009.

[76] Michael Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.

[77] Stefan Zickler, Tim Laue, Oliver Birbach, Mahisorn Wongphati, and Manuela Veloso. SSL-Vision: The Shared Vision System for the RoboCup Small Size League. In *Robot Soccer World Cup*, pages 425–436. Springer, 2009.

[78] Oliver Zweigle, René van de Molengraft, Raffaello d'Andrea, and Kai Häussermann. RoboEarth: connecting Robots worldwide. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 184–191. ACM, 2009.

[79] Frederik Zwilling. Simulation of the RoboCup Logistic League with Fawkes and Gazebo for Multi-Robot Coordination Evaluation. Bachelor thesis, RWTH Aachen University, Knowledge-Based Systems Group, December 2013.

[80] Frederik Zwilling, Tim Niemueller, and Gerhard Lakemeyer. Simulation for the RoboCup Logistics League with Real-World Environment Agency and Multi-level Abstraction. In *RoboCup Symposium*, 2014.