Symbolic Representation of Dynamic Knowledge for Robotic Teams

Dissertation by

Stephan Opfer

in partial fulfillment of the requirements
for the academic degree

Doktor der Naturwissenschaften (Dr. rer. nat.)

Submitted to the Faculty of
Electrical Engineering and Computer Science
University of Kassel, Germany
November, 2020

1st **Reviewer:**	Prof. Dr. Kurt Geihs (University of Kassel)
2nd **Reviewer:**	Prof. Dr. Daniele Nardi (Sapienza Università di Roma)
Date of Defense:	2021-03-04
Committee:	Prof. Geihs, Prof. Nardi, Prof. Stumme, Prof. Zündorf

Contents

Part I Foundations

Part II Solution

Part III Assessment

Appendices

Abstract

The research goal of this thesis is to develop the conceptional foundations for a team of autonomous robots that is capable of symbolically representing knowledge about the environment, communicating about its symbolic knowledge, and reasoning about the knowledge, while the environment changes dynamically. This research goal is motivated by several application domains. The domain of domestic service robots motivates the research goal the most. We closely analysed the application domains and the research goal, and based on this analysis, we formulated requirements for general domain independence, handling unknown and dynamic environments, and facilitating human-robot interaction.

Our proposed solution coordinates a team of autonomous robots with the ALICA Framework. In order to fulfil the requirements, we reimplemented the ALICA Framework and expanded it, for example, with a generic solver interface. The generic solver interface allows the integration of non-monotonic symbolic reasoning mechanisms and therefore, improves the domain independence of the ALICA Framework.

We further developed a symbolic knowledge base for dynamic knowledge that utilises answer set programming (ASP) as its non-monotonic reasoning core. The knowledge base is accessible through the generic solver interface from the context of the ALICA Framework, but it also offers a generic query interface that facilitates the interaction with other modules of our software architecture.

The design of our knowledge base also facilitates the interaction between humans and robots. We achieved this by integrating a commonsense knowledge database and allowing humans to teach robots at runtime. Furthermore, we developed a lean middleware that addresses the communication requirements of the domestic service robot domain and enables the knowledge-based cooperation between robots.

In addition to the experiments, dedicated to the evaluation of individual parts of our solution, we also evaluated the complete system within two different demonstrators. The Wumpus World demonstrator is a standard toy scenario for the evaluation of artificial intelligence, and the grid-based service robot simulator provides more realistic use cases. Altogether, the results of the experiments show that our system fulfils the requirements. Nevertheless, we also identified a need for further research in ASP-based reasoning about dynamic knowledge.

Zusammenfassung

Das Forschungsziel dieser Arbeit ist es, die konzeptionellen Grundlagen für ein Team autonomer Roboter zu entwickeln, welches in der Lage ist, Wissen über seine Umwelt symbolisch darzustellen, über sein symbolisches Wissen zu kommunizieren und mit dem Wissen zu schlussfolgern, während sich die Umgebung dynamisch ändert. Dieses Forschungsziel wird durch mehrere Anwendungsbereiche motiviert, wobei die Domäne der Haushaltsroboter das Forschungsziel am besten motiviert. Wir haben die Anwendungsdomänen und das Forschungsziel genau analysiert und auf Grund der Ergebnisse Anforderungen für die allgemeine Domänenunabhängigkeit, den Umgang mit unbekannten und dynamischen Umgebungen und die Erleichterung der Mensch-Roboter-Interaktion formuliert.

Unsere vorgeschlagene Lösung koordiniert ein Team autonomer Roboter mit dem ALICA Framework. Um die Anforderungen zu erfüllen, haben wir das ALICA Framework neu implementiert und es beispielsweise um eine generische Solver-Schnittstelle erweitert. Die generische Solver-Schnittstelle ermöglicht die Integration nicht monotoner symbolischer Schlussfolgerungsverfahren und verbessert daher die Domänenunabhängigkeit des ALICA Frameworks.

Weitherin haben wir eine symbolische Wissensbasis für dynamisches Wissen entwickelt, welche die Antwortmengenprogrammierung (ASP) als nicht monotonen Schlussfolgerungskern verwendet. Auf die Wissensdatenbank kann über die generische Solver-Schnittstelle aus dem Kontext des ALICA Frameworks zugegriffen werden. Sie bietet jedoch auch eine generische Schnittstelle, welche die Interaktion mit anderen Modulen unserer Softwarearchitektur erleichtert.

Das Design unserer Wissensbasis erleichtert auch die Interaktion zwischen Menschen und Robotern. Dies haben wir erreicht, indem wir eine Allgemeinwissensdatenbank integriert haben und es Menschen ermöglicht haben, Roboter zur Laufzeit zu unterrichten. Darüber hinaus haben wir eine schlanke Middleware entwickelt, die die Kommunikationsanforderungen der Domäne der Haushaltsroboter erfüllt und die wissensbasierte Zusammenarbeit zwischen Robotern ermöglicht.

Zusätzlich zu den Experimenten, die der Bewertung einzelner Teile unserer Lösung gewidmet waren, haben wir auch das gesamte System in zwei verschiedenen Demonstratoren evaluiert. Der Wumpus World-Demonstrator ist ein Standard-Spielzeugszenario für die Evaluierung künstlicher Intelligenz, während der gridbasierte Servicerobotersimulator realistischere Anwendungsfälle bietet. Insgesamt zeigen die Ergebnisse der Experimente, dass unser System die Anforderungen erfüllt. Wir haben jedoch auch festgestellt, dass weitere Forschungen zum ASP-basierten Schlussfolgern mit dynamisches Wissen erforderlich sind.

Acknowledgements

I would like to thank all people who have accompanied me through this journey.

First of all I want to thank my supervisor Prof. Kurt Geihs, who always believed in what I did and even encouraged me to further pursue my academic career.

Over the years, there also have been several colleagues and students, some of which became friends. I would like to thank Philipp A. Baer and Roland Reichle, the founders of the Carpe Noctem Cassel RoboCup team (2006-2017). Without your dedication to this team, I probably would have never pursued a PhD. Further, I thank Hendrik Skubch and Thomas Weise who consistently inspired me and had the ability to question my beliefs in a way that allowed me to grow further and faster. I also would like to thank all the students that participated in the Carpe Noctem Cassel team over the years and therefore, suffered with and from me as member and leader. Most notably these are: Till Amma, Stefan Triller, Florian Seute, Tim Schlüter, Kai Liebscher, Janosch Henze, Eduard Belsch, Tobias Schellien, Lukas Will, Dennis Bachmann, Thore Braun, Michael Gottesleben, Nils Kubitza, Lisa Martmann, Jewgeni Beifuß, Yannick Schlamm.

I would like to thank my colleagues Stefan Niemczyk, Andreas Witsch, Harun Baraki, Stefan Jakob and Alexander Jahl for increasing the standard of my work and at the same time forming such a wonderful and thriving working environment.

I also would like to thank the following great researchers, because they inspired me and offered helpful advises: Daniele Nardi, Thorsten Schaub, Albert Zündorf, Gerd Stumme, Vladimir Lifschitz, Alessandro Saffiotti, David Vernon, Moritz Tenorth, René van de Molengraft, Bernardo Cunha, and Herman Bruyninckx.

Finally, I have to apologise to my friends and family, because I did not spend the amount of time with them that they deserve. Therefore, I would like to thank all who nevertheless stayed with me until the end.

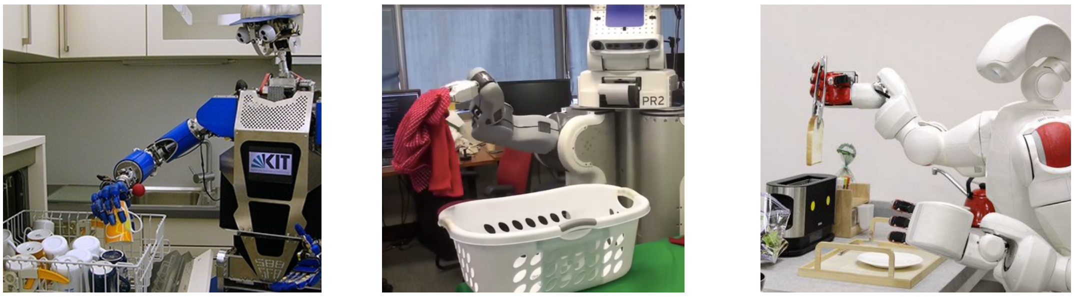
PartI

Foundations

Introduction 1

1.1 Motivation

During the last decade, autonomous robots have started to play an increasing role in our everyday life. They can vacuum-clean our living room, mow the lawn, and clean the pool. Autonomous and interactive toys become more and more intelligent. Additionally, almost all car manufacturers are developing autonomous cars. Automated guided vehicles, picking-assistants, and autonomous forklifts take care of the logistics in production plants or parcel service centres.



(a) Doing the Dishes [6]
(b) Doing Laundry [55]
(c) Serving Toast [92]

Figure 1.1: Service Robots doing Everyday Household Tasks

In the research field of domestic service robots, researchers focus on multipurpose robots, instead of single-purpose ones. The images in Figure 1.1 show some examples of state-of-the-art service robots, which can do everyday household tasks. The realisation of such multipurpose domestic service robots is the primary motivation and serves as an application scenario for this thesis. The variety of their tasks increases the number of objects the robots have to operate with tremendously[1]. Among other things, this poses the following new quality of challenges for the research community.

Generally speaking, the more a domestic service robot can do autonomously, the bigger is its value for humans. However, the developer cannot foresee the type of relevant objects that a robot will encounter in a household. Taking this constraint into account, the design of a robot that works autonomously can be challenging.

The capability to operate in a dynamic environment is a typical requirement for robots that work in environments populated with human beings. The reason for this is that humans insert, remove, and displace objects in the environment and are themselves moving

[1] The average German person owns 10.000 objects, and the average American household contains 300.000 objects.

obstacles from the robot's point of view. Autonomously operating with unknown objects in a dynamically changing environment leads to even more significant challenges. The robot cannot be programmed in advance to function correctly in all situations, since the programmer cannot know the relevant details of every situation beforehand. However, equipping the robot with human-like cognitive capabilities like learning, reasoning, and planning, as done in this thesis, helps to solve this problem.

The user of the robot can assist in achieving the first cognitive capability. Instead of learning by itself, the robot could be *taught* by its user, since the user knows best the desired behaviour of the robot. Nevertheless, the typical user of a domestic service robot will not have the programming skills of a computer scientist, to adapt the robot to her needs. It should be possible for the user to interact with the robot naturally and *tell* the robot about new knowledge. A promising way to achieve this is the usage of symbolic knowledge representation. One of the inherent advantages of symbolic representations is that adding new knowledge to a running system is straight forward [115, pp. 8-9].

The symbolic knowledge representation is not an end in itself. It always serves the purpose of the second cognitive capability, i.e. to reason efficiently and to deduce new knowledge. Therefore, it would be possible for a service robot to use a priori known facts about typical application domains (background knowledge) in combination with perceived or deduced knowledge to answer questions or to choose the most efficient way to fulfil its current task.

The third cognitive capability to deal autonomously with dynamic environments is the capability of planning. Classical planning algorithms find sequences of actions, whose execution will achieve a specific goal. The actions are often generic and predefined, which means they have to be instantiated with specific entities of the environment of the robot, to be executable. The action to *grasp an object*, for example, needs to be instantiated with a grounded symbol that is representing a particular physical object to make the execution of the action change the environment. Symbolic knowledge representations inherently consist of symbols, which also represent physical objects of the environment of the robot. Therefore, planning is a special case of reasoning, i.e. reasoning about sequences of actions and their effect on the environment.

From our point of view, it is unlikely for several reasons that we will have only one service robot in our future homes. A robot equipped with actuators for all possible domestic service tasks will probably be more expensive and less effective than several robots that have actuators for a single purpose. Further, a single service robot is a single point of failure. If the vacuum cleaner component of a multi-service robot, for example, becomes entangled with the rug fringe, not a single further task will be done. With several service robots, only the floor remains dirty in some places. Another reason is that several robots can be more efficient, as they can parallelise their work. Further, multiple robots can share their knowledge about the environment to obtain a more complete representation of their environment. A more comprehensive environment model, for example, enables more efficient path planning or object localisation. Consequently, topics such as multi-robot planning, multi-robot learning, cooperation, multi-robot task allocation, and coordination are relevant fields of research for future service robotic applications.

In summary, the domain of service robots motivates teams of autonomous robots that utilise a symbolic knowledge representation to support several required cognitive capabilities. Additionally, besides service robots, motivations for our research arise from two other areas: disaster scenarios and space missions.

In disaster scenarios, the environment is often hazardous for human beings and therefore suggests to use robots wherever possible. In 2011, for example, the disaster of the nuclear power plant Fukushima Daiichi was primarily caused by a tsunami. The environment changed due to the flood and the closer area around the plant was contaminated by radiation, making it impossible for people to access the power plant. Even for most robots, the environment was too extreme. The radiation destroyed the wiring and therefore needed to be especially shielded against radiation. It took several years to develop a robot that is capable of getting closer to the melting core. Sensors that can create images of the environment under the influence of the radiation are still under development [15]. Another effect of the radiation is that the robots can only be remotely controlled via specially shielded cables since the radiation superimposes any radio control signal. Therefore, the underwater navigation of a robot in the debris of the collapsed reactor building is problematic without entangling the connected wires, which is necessary for remote control. As a result, disaster scenarios motivate teams of autonomous robots that, e. g., can operate without connected cables.

Compared to domestic and human-populated environments, disaster scenarios are not necessarily dynamic in the sense that they are continuously changing. However, due to changing water levels after a flood, aftershocks and consecutive explosions, the environment can, although not continually, change. Furthermore, the environment is often unknown, because disasters like tsunamis change the situation in a way that makes existing maps of landscapes and buildings useless. An autonomous robot, operating in such an unknown environment, needs to adapt its knowledge about it continuously. Finally, the demand for a symbolic knowledge representation can be justified in disaster scenarios as well. As mentioned for the domain of service robots in the household, symbolically represented knowledge is suitable for planning algorithms, eases human-robot interaction, and is exchanged easily. All three properties of symbolic knowledge representations are also advantages in the context of disaster scenarios. In this context, autonomous robots need to plan their actions in unknown environments, rescue forces need to interact with the robots, and the knowledge about the environment needs to be adapted to different situations.

Space missions, such as exploring the surface of other planets, provide similar requirements. Spacewalks are hazardous for humans due to radiation and lack of oxygen and therefore, robots step in wherever possible. Unfortunately, the robots can only be partially remote-controlled, as the communication delay between Earth and for example, Mars is between 3 and 21 minutes and is interrupted for hours, depending on planetary constellations. The considerable communication delay motivates, e. g., partially autonomous robot designs to allow safe long-term navigation in unobservable terrain. Furthermore, as the term *exploration* indicates, the environment is unknown and requires the planning capabilities of the robots, similar to disaster scenarios.

All three discussed scenarios contribute to the motivation for this thesis. However, the domain that motivates the research goal formulated in Section 1.2 the most is the application of service robots in human-populated environments.

1.2 Problem Statement

The following research goal summarises the general objective of the work presented in this thesis:

> *The development of conceptional foundations for a team of autonomous robots that is capable of*
>
> - *symbolically representing knowledge about the environment,*
> - *communicating about its symbolic knowledge,*
> - *and reasoning about the knowledge,*
>
> *while the environment changes dynamically.*

Although very concise and straight to the point, the research goal provides many different aspects that have to be taken into account, while pursuing it. The different aspects are elaborated in the following paragraphs and narrowed down to clarify the scope of this work.

1.2.1 Knowledge

The research goal of this thesis focuses on symbolic knowledge representation and reasoning formalisms. Therefore, the following sections about communication and environment partially relate their problem description and requirements to knowledge representation and reasoning formalisms, too. The autonomous team of robots should be able to communicate and understand not only the knowledge of the other but also the semantics of the performed speech acts. Furthermore, the environment of the robots should be representable by the knowledge representation formalism, as well as humans, should be able to understand and manipulate the knowledge of the robots through communication.

Due to the partially unknown environment, the knowledge must be expandable at runtime. Therefore, chosen knowledge representation and reasoning mechanisms must be able to integrate new knowledge at runtime and to handle or avoid inconsistencies. There are three different sources of new knowledge: humans communicating with robots, robotic sensor values describing the environment, or messages from other robotic team members. As humans communicate new knowledge to the robots, the knowledge representation mechanism typically limits the language that humans can use. This constrained language needs to be comfortable to "speak" and expressive enough to teach robots. Finally, an efficient reasoning mechanism is required to reason about the knowledge such that conclusions are useful, although the environment changes fast.

1.2.2 Communication

Communication among team members is an important aspect of teams. A team of autonomous robots that, for example, operates in a household needs to allocate requested tasks to team members and sometimes even synchronise the execution of the tasks, through communication. Therefore, the robots need to speak the same language and need to have a similar understanding of the communicated concepts. Considering the possibility that the robots are heterogeneous systems and not necessarily originating from the same developer or company, gives rise to complex technical aspects of communication, such as different data formats, protocols, routing, and communication technologies. Nevertheless, this work spares most of the technical aspects of communication. It expects the team members to be able to send and receive any data between each other correctly. Instead, this work focuses on non-functional characteristics of the communication which influence the performance of the team, how the structure in the team affects possible communication patterns, and the semantics of communicated knowledge. However, to conciliate arbitrary peculiarities of non-functional communication properties like reliability, latency, and bandwidth with a robust architecture for a team of autonomous robots is one of the goals of this thesis. It, therefore, influences its design decisions like a common thread through this work.

Hence, this thesis considers the following communication specific requirements. Each team member needs to be able to run independently from available communication to its team members since the communication itself or the team members can fail. Furthermore, single-point-of-failures and any bottle-necks should be avoided in the design wherever possible. Finally, the team members might not be known in advance and can leave and join the team at runtime.

1.2.3 Environment

A human-populated domestic environment can be very challenging for autonomous robots. This work focuses on the complexity and variety of objects that are relevant to service robots within such an environment. The robots, for example, cannot know all the objects they will encounter, since the developers of the robots cannot know all the objects either. Furthermore, humans in a household will change the environment continuously, e. g., by displacing objects, walking around, and opening and closing doors.

As humans itself belong to the environment of the robots and the robots need to interact with their environment, they also need to interact with humans, for example, through communication. Finding a universal language that robots and humans understand is challenging. Humans cannot be expected to know the utilised knowledge representation and reasoning formalism of the robots. Furthermore, it is difficult for robots to understand the natural language of humans.

1.2.4 Requirements

The following requirements summarise the problem statement as elaborated in the previous subsections:

R1: Domain Independence The solution must be designed in a way that allows its application independent of the domain.

R2: Handling of Unknown Environments The solution must reliably work in an unknown and unpredictable environment in order to maintain the autonomy of the agents.

R3: Handling of Dynamic Environments The solution must be able to handle dynamic environments.

R4: Facilitate Human Interaction The solution must facilitate the interaction between agents and humans.

1.3 Solution Approach

The approach to reach the given research goal is based on the multi-robot framework ALICA [77]. The framework includes three parts: a language for describing the behaviour of a team of autonomous agents, a modelling tool to design programs that represent such behaviour, and a runtime engine that allows controlling an agent according to the given program. In this thesis, the ALICA Framework forms the foundation for organising and coordinating a team of autonomous robots and is reimplemented and expanded to provide more domain independence.

In order to find an appropriate knowledge representation and reasoning formalism, several different logical formalisms are examined and compared. Further, the development of a general solver interface allows the utilisation of different formalisms, as needed by different domains. Nevertheless, our particular focus is on answer set programming [79] as one of the most intriguing state-of-the-art reasoning formalisms. answer set programming offers unique features for handling inconsistencies and allows non-monotonic reasoning. As a result, it is possible to expand the knowledge represented with answer set programming such that even humans can teach robots knowledge about their environment. The integration of the commonsense knowledge database ConceptNet5 [39] provides background knowledge for the domestic service robot domain, and its knowledge is automatically joined to the robots knowledge base, depending on the content of the human-robot interaction.

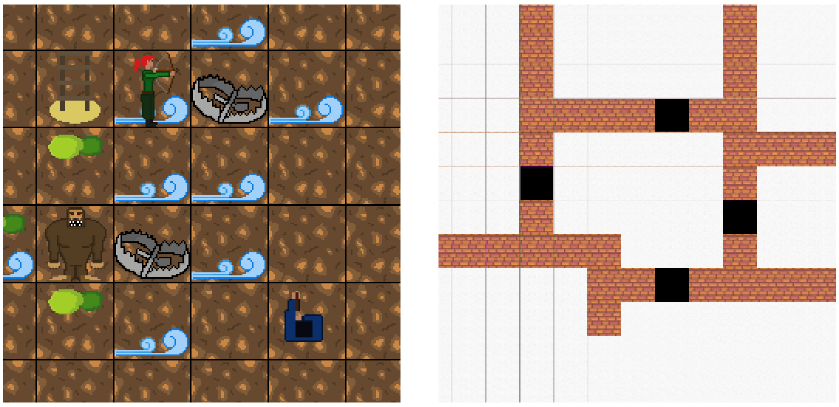

(a) Wumpus World Simulation (b) Service Robot Simulation

Figure 1.2: Evaluation Scenarios

The presented solution is evaluated within two sophisticated test scenarios. The well-known Wumpus World provides an easy to understand test environment, and it facilitates the comparison of the presented approach with other research results (see Figure 1.2a). In particular, the Wumpus World scenario shows the benefit of cooperating by exchanging symbolic knowledge. The second evaluation scenario is a grid-based 2.5D service robot simulation environment (see Figure 1.2b). It is similar to the Wumpus World but completely asynchronous and dynamic. The grid-based approach abstracts from continuous motion planning and sensor value processing, but still offers most features of a real service robotic environment, including human-robot interaction.

1.4 Contributions

The presented work substantially contributes to the effort of two research communities, namely *artificial intelligence* and *robotics*, to fuse their individual achievements into systems that stand out by mastering scenarios of unmatched complexity, scale, heterogeneity, and unpredictability. In the case of this thesis, it is the fusion of state-of-the-art symbolic **knowledge representation and reasoning** formalisms from the field of artificial intelligence with a **team coordination framework** from the field of robotics and thereby addressing requirements from highly dynamic and human-populated environments.

The reimplementation and extension of the team coordination framework ALICA achieve a **new quality of domain-independence** that allows the framework to be applied in various highly dynamic scenarios.

Furthermore, we identified **requirements for symbolic knowledge representation and reasoning formalisms** to cope with dynamic and human-populated domains. Additionally, we provide a **thorough analysis of several different formalisms** regarding these requirements. According to this analysis, **answer set programming (ASP)** is one of the most promising formalisms and therefore integrated as an additional solver into the ALICA Framework. A **sophisticated interface** to the solver makes it a **knowledge base** capable of **handling dynamic knowledge**, allows humans to **teach agents**, and allows **agents to share their symbolic knowledge** during cooperation.

Another contribution of this work is to avoid that agents have to learn domain-specific knowledge via machine learning algorithms from scratch since these algorithms are not scalable in most realistic scenarios. Instead, it allows humans to teach agents as well as agents to **extract knowledge from a commonsense knowledge database automatically**. Finally, the presented system can **understand implicit human requests** by utilising the commonsense knowledge database.

1.5 Structure of the Thesis

The structure of this thesis divides it into three parts. Apart from this introductory chapter, Part 1 comprises the foundations of this thesis. Part 2 presents the solution for the research problem (see Section 1.2) with all necessary details to understand and to proceed with this line of research. Further, it includes work that either has similar research goals or is using the same tools and formalisms. In order to objectively assess the solution presented in this thesis, Part 3 describes the utilised evaluation scenarios and critically discusses the corresponding results.

Part I - Foundations

Chapter 2: In Chapter 2, it is explained how basic terms like an intelligent agent or multi-agent systems are understood according to current research and within this thesis. Furthermore, an introduction to the field of cognitive robotics with a focus on corresponding robotic software architectures is given.

Chapter 3: The ALICA Framework is described in Chapter 3 as presented in [77], to distinguish the contributions made by this thesis from the former state of the ALICA Framework. Further, this chapter identifies the disadvantages and properties that have to be adapted for the research goal of this thesis.

Chapter 4: The definitions of *knowledge* and *reasoning*, as used in this thesis are given in Chapter 4. Furthermore, problems common to symbolic reasoning systems are discussed, and it is shown how different formalisms attempt to solve them. Finally, answer set programming is explained, and arguments are given for its suitability as a dynamic knowledge base.

Part II - Solution

Chapter 5: The software architecture of this work is presented in Chapter 5. At first, general design principles are explained and afterwards, the actual implementation of these principles is described.

Chapter 6: The extensions and necessary adaptations to enhance the domain independence of the ALICA Framework are described in Chapter 6. Among others, these include a solver interface, abstraction from communication middleware, and reduction of dependencies.

Chapter 7: The knowledge representation and reasoning formalism chosen for this work is answer set programming (ASP). How this formalism can be used to create a dynamic knowledge base, is described in Chapter 7.

Chapter 8: In Chapter 8, based on the extended version of the ALICA Framework and the ASP-based knowledge base, it is explained how humans can teach robots and how the commonsense knowledge database ConceptNet 5 supports this process.

Chapter 9: For sharing knowledge in a robotic team, a corresponding middleware and speech acts are necessary. Chapter 9 introduces Cap'n Zero, a versatile and lean middleware and explains how this work allows robots to cooperate by sharing knowledge.

Chapter 10: In Chapter 10, the works of other researchers that pursue similar research goals are discussed, and differences to the work presented in this thesis are given. Hereby, the focus is on frameworks for multi-agent coordination and knowledge-based robotic systems.

Part III - Assessment

Chapter 11: The two different demonstrators, used for the evaluation, are explained in Chapter 11. On the one hand, the Wumpus World Simulator offers results comparable with all formalisms and frameworks, able to solve the Wumpus World. On the other hand, the Service Robot Simulator demonstrates the applicability of the presented solution in dynamic environments.

Chapter 12: In Chapter 12, the executed experiments are documented. The presented results are analysed and critically discussed from different viewpoints.

Chapter 13: Chapter 13 concludes this thesis by summarising the presented work and providing pointers for continuing the direction of research endorsed by this thesis.

Autonomous Robots 2

The work presented in this thesis connects research from the adjacent areas of artificial intelligence and robotics. On closer examination, there are even more research areas and sub-areas related to the presented work. Among others, there are knowledge representation and reasoning, cognitive science, intelligent agents, cognitive robotics, action theory (philosophy), and multi-agent systems. This chapter dives into the terminologies and critical concepts of these research areas and explains the semantics of the terms as used within this thesis.

2.1 Intelligent Agents

The definition of the term *agent* in computer science is very inclusive. Sometimes it denotes specific processes running on a computer (e.g. ssh-agent), and sometimes the term does not distinguish between humans, robots, and computers altogether. Nevertheless, all definitions of the term agent have in common that the agent is separated from its environment and is capable of interacting with it. Therefore, the depiction of an agent and its environment in Figure 2.1 is very general.

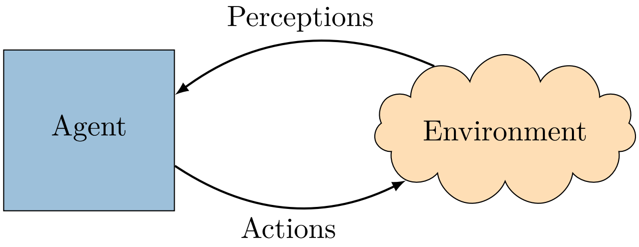


Figure 2.1: Agent Environment Cycle

The agent perceives its environment and is, depending on the context, able to manipulate it through its actions. Afterwards, the agent recognises the effects of its actions on the environment, and thus the cycle between agent and environment starts over. In the presented work, humans are not understood as agents. However, it is assumed that agents have a physical embodiment and therefore, the term *robot* is used interchangeably.

This work addresses domestic service robots that can do everyday household tasks. These tasks can be challenging for an agent to handle and therefore, generally speaking, the agents should be intelligent in order to assist humans. The question "What makes an agent intelligent?" is not easy to answer. In the literature [127, 94, 120], agents are typically characterised by their properties. Accordingly, an agent can be autonomous, proactive, rational, social, goal-based, model-based, reactive, and much more, but no general definition of an intelligent agent is given. Instead of defining intelligence itself, 1950 the computer scientist Alan Turing proposed an intelligence test, not for agents, but machines in general. He said that every machine passing the *Turing test* [164], could be considered to be intelligent. Informally speaking, the Turing test is passed by a machine, if a human interrogator cannot distinguish between another human and the machine by means of written interaction with both of them. Unfortunately, at the time of writing this thesis, no machine clearly and undisputedly passed the Turing test, and therefore, could be an example of an intelligent machine or agent. Thus, the test has been widely criticised, albeit the critical analysis helped to understand what it takes to create intelligent behaviour.

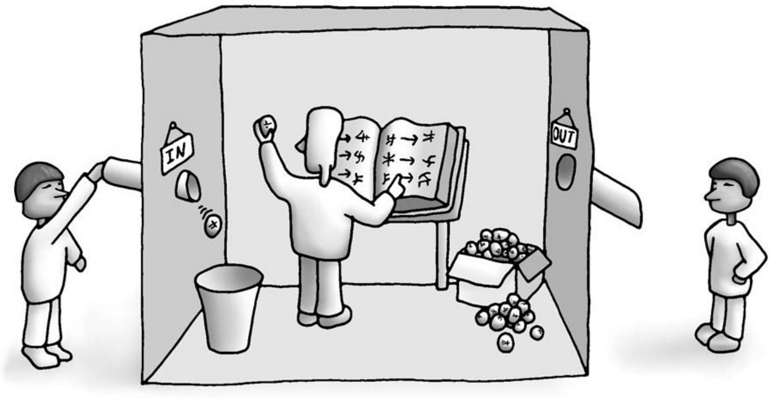


Figure 2.2: The Chinese Room Thought Experiment[1]

The most famous critique on the Turing test represents John Searle's thought experiment, known as the *Chinese Room argument* [155]. Figure 2.2 shows the setup of the experiment. In the middle, there is a sealed room, and only Chinese symbols can enter and leave the room. Inside the room, there is an infinite amount of Chinese symbols and a rulebook that describes which Chinese symbol should leave the room according to the Chinese symbols that have entered the room. Apart from that, there is one person inside the room that does not understand Chinese at all. It merely follows the rules written in that rulebook. From outside, native-speaking Chinese persons send questions in the form of Chinese symbols into the room, and the Chinese symbols that leave the room represent answers to the questions.

This setup can be seen as a Chinese variant of the Turing test and the agent environment cycle in Figure 2.1. Following that idea, the analogies to a computer running in a domestic service robot environment would be as follows: The room would be the computer, the

[1] Source: https://commons.wikimedia.org/wiki/File:2-chinese-room.jpg - Available under the Creative Commons License

rulebook would be a program, and the person inside would be the central processing unit (CPU) that can execute the program. Imagine that such a computer would pass the Turing test due to its perfectly designed program and CPU. Still, the argument of John Searle would be: No matter how well written the rulebook inside the room is, the person still does not understand Chinese. According to Searle, the CPU is just executing syntactic operations, and the Chinese symbols have no meaning for it. As a result, Searle refutes the conclusion of Turing that any computer capable of passing the Turing test would be equivalent to the human mind. The key in his argument is that the CPU is only doing syntactic operations without understanding its semantics or meaning.

Following the Chinese room argument, the term intelligent agent, as used in this work, is referring to robots executing actions and communication that appears to be intelligent relative to the observer and apart from that have nothing in common with an actual intelligent and conscious mind. Nevertheless, one of the many replies to Searle's thought experiment matches the work presented in this thesis. The reply states that the meaning and semantics of the Chinese symbols could be encoded into a comprehensive library in the room that extends the original rulebook. ConceptNet 5 [71] is currently one of the largest and best curated commonsense knowledge databases that exist and can be seen as an extension of the rulebook for domestic service robots. The integration of ConceptNet 5 into the software architecture of the robots presented in this work is described in Chapter 4. However, following the argument of Stevan Harnad [68], the extended rulebook reply to Searle's Chinese room argument is just trying to learn Chinese with the help of a Chinese-Chinese dictionary: In the end, the library still contains only symbols that explain other symbols, without adding any semantics. Stevan Harnad, founder and former editor of the Behavioural and Brain Sciences journal [156] in which Searle's article [155] was published, argues that the semantics of symbols need to be grounded in sensorimotor transduction that happens when the human brain perceives the environment through its corresponding body. Developing a service robot, capable of solving Harnad's interpretation of the symbol grounding problem (see Section 4.1.2) through collecting sensorimotor experiences is out of the scope of this work. Instead, extensive commonsense knowledge is integrated into the software architecture of the robots, in the form of symbols that have precise semantics in the mind of human observers. As a result, the behaviour of the robots is perceived to be intelligent.

In order to let the agents behave intelligently, the actions of the agents are based on commonsense and domain-specific knowledge as well as on situational knowledge that the agents gain through the perception of their environment (see Chapter 8). In addition to the direct influence of knowledge on the behaviour of the agent, general properties in the design of an agent can make it appear to be intelligent, too. Before these properties are explained, we describe the properties of the environment that is expected in this work.

2.1.1 Properties of the Environment

The environment of the agents in this work is that of a typical household. The terms that describe the environment are taken from the standard reference of Russell and Norvig [47]. For the agents, the environment can only be *partially observed* and thus is a *stochastic* environment. A human in another room, for example, cannot be observed through walls and the moment when she spills some coffee that demands the robots to clean up the mess cannot be anticipated. Actions performed by agents can influence future decisions and actions. This property is denoted as being *sequential*. Finally, the environment is *dynamic* and can, therefore, change on its own, and the state of the environment is *continuous*. As a result, the environment changes while the agent is deliberating, and the state space of the environment is infinite. Although not part of the environment, it is essential to note that there are other agents *cooperatively* operating in the environment, too.

2.1.2 Design Properties of the Agents

The properties of the environment, as described in Section 2.1.1, have a direct impact on the design properties of the agents. These properties are divided into three groups: decision making, communication, and general design principles. The terms again are taken from Russell and Norvig [47], but also from standard references for rational agents and multi-agent systems [127, 94, 120].

Decision Making

Specific properties in agent literature describe how agents choose between different possible actions, and it is worth noting that an agent can follow several of these properties at once. The most simple decision-making option is denoted as being *reactive*. A reactive agent performs its actions in reaction to an exterior event. The actions and events are often directly coupled, which allows for short reaction time. Agents that only follow this principle are called reflex agents and are hardly perceived as being intelligent. Nevertheless, due to the short reaction time, such a design is often preferred when actions are very time-critical, like in avoiding unexpected obstacles. The problem with reflex agents in the environments described above is that they cannot fully observe the environment and therefore, do not perceive all relevant exterior events.

Another complementary approach is a *model-based* agent. A model-based agent maintains a model of its environment and chooses actions according to this model. Simply put, the model describes how the environment works and is therefore suitable for predicting future events. As a result, the agent can react to events that happen or will happen in the model of the environment, instead of directly observing the environment. If the model, for example, comprises how and where humans walk under certain circumstances, collisions can be avoided by considering this model during path planning.

Reactive and model-based agents make their decisions according to the environment, but without being aware of a goal that needs to be achieved, the behaviour of an agent is somewhat limited. *Goal-based* agents have, apart from a model of the environment, also an explicit representation of a goal. Often this goal describes a state of the environment that can be achieved step-wise and therefore allows to measure progress in fulfilling the goal. Finally, a more general property of an agent is to be rational.

A *rational* agent always makes decisions that are optimal to achieve a particular goal while considering its current beliefs. The agents in this work are, to some extent, reactive, model-based, goal-based and rational all together. However, reactive behaviour is not addressed within this work and is only used in situations in which the direct mapping of events to actions reduces the complexity of the decision-making process. Some goals the agents try to achieve are extracted from tasks that humans assign to the agents. Other goals are more inherent to the agents, and in order to achieve them, the agents *proactively* perform actions, i. e. without exterior events causing these actions to be performed.

Communication

Section 2.3 presents communication patterns and protocols in detail. Here standardised terms for the general attitude of agents towards interaction are given. Generally speaking, agents are *interactive* if they communicate with their environment, humans or other agents. More specific are *social* agents that choose their interaction according to individual preferences. In this work, the agents are expected to be *cooperative*, and therefore they answer each other's requests benevolently.

General Design Principles

Apart from communication and decision making properties, there are more general design principles of the software architecture of the agents. Most important for this work is that the agents are *adaptive*. Adaptive agents change their behaviour due to experiences. In particular, they can learn and therefore adapt their behaviour according to the learned knowledge. Section 8.1 explains how humans can teach robots via the software developed for this thesis. In the case of the Chinese room thought experiment, allowing humans to teach, would mean that the Chinese speaking persons outside the room have a way to rewrite the rulebook according to their needs. This rewrite possibility is not only beneficial for the users of the service robots, but it is also necessary, because the developer, for example, cannot foresee the objects that are present, how they work, or need to be handled in a specific household.

The agents in this work are designed to be *persistent*. The lifetime of software agents, like the ssh-agent on a personal computer, is very short. In contrast to this, the agents in this work ideally operate 24 hours 7 days a week. This lieftime requires self-reflective skills, like monitoring battery levels and searching for charge stations, in order to stay operational. Although relevant for the design of the general architecture (see Chapter 5), self-* properties [119] are not explicitly addressed within this work. Instead, the agents are designed to be *robust* in general, meaning it is expected that faults happen and they are handled appropriately.

Finally, the agents are *autonomous*, which means that they act without exterior influence. Considering the possibility of human teachers, this is only partially true, but as the agents proactively pursue their inherent goals and perform actions accordingly, their autonomy becomes explicit.

2.2 Architectures for Intelligent Behaviour

After defining common properties of intelligent agents in the last section, this section elaborates ways to implement such properties. Thus the question is: What is inside the agent box of Figure 2.1, that allows the agent to be model-based, rational, proactive, or adopt further properties?

2.2.1 MAPE-K Architecture

Following the idea of the agent environment cycle, an answer to the question must close the cycle within the agent, i. e. the agent performs actions based on perceptions of the environment. Researchers from the field of autonomic computing developed the so-called *MAPE-K cycle*. MAPE is an acronym for monitor, analyse, plan, and execute, while K stands for knowledge that is centrally accessible from all four parts.

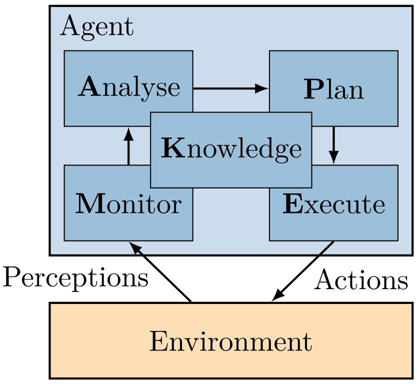


Figure 2.3: MAPE-K cycle [119]

The application in the original publication [119] of the MAPE-K cycle had the autonomous management of servers as application scenario in mind. The goal was to increase the robustness of servers by monitoring their performance, analyse the measured values, plan for any necessary adaption of the behaviour of the servers, and finally execute the planned adaptions. In Figure 2.3, the MAPE-K cycle is deployed in the agent in an agent environment cycle. Here, the environment of the agent replaces the server, compared to the original application scenario.

In case of a domestic service robot, the modules of the MAPE-K could be mapped to specific modules in the robotic architecture. While moving around in the household, the service robot should adhere to certain constraints like avoiding collisions, being quiet in certain parts of the house, and not disrupting conversations. Therefore the robot continuously measures distances to walls, furniture and other obstacles with sensors like laser scanners, RGB-D cameras, ultrasonic sensors, or stereo cameras. These values create a time series of measurements that are analysed in order to create and update the model of the environment. This model is part of the knowledge module of the MAPE-K cycle. The next step is to plan a path towards the destination. Here the knowledge about the constraints and the environment plays a crucial role. In order to follow the planned path, the execution module generates control commands from the given path and sends them to the motion.

2.2.2 Robotic Three-Layered Architecture

In 1970, the first autonomous mobile robot, known as Shakey [150], operated already with an approach similar to the MAPE-K cycle. It is denoted as the Sense-Plan-Act cycle in the robotics community and has the Analyse module partially merged into the Measure and Plan module. Further developments in the robotics research area created the three-layered architecture [128]. The reason for this development is that a simple loop, like the MAPE-K and the Sense-Plan-Act cycle, is not flexible enough to address the need for different reaction times. Some actions need to start fast and in direct reaction to the environment of the robot (see reactive agents, Section 2.1.2), while others are much more complex and require time-consuming algorithms to generate plans.

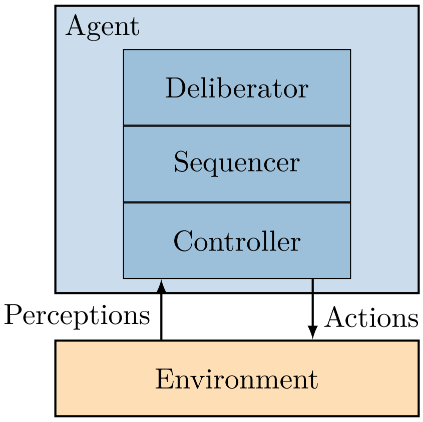


Figure 2.4: Three Layer Architecture [128]

The three layers, as shown in Figure 2.4, are the Controller, Sequencer, and the Deliberator. The first layer is already sufficient to create reactive agents, as it can directly map sensor inputs to actuator commands. The Sequencer layer is responsible for adapting the controller according to the plans it is following. If a robot, for example, reaches a waypoint, the next waypoint is given to the controller. In the third layer, the most complex and therefore often the most time-consuming computations take place. Here, for example, the decision is made whether a goal (see goal-based agent, Section 2.1.2) is still attainable through the current plan, or whether replanning needs to be triggered. Special attention is necessary when designing the interaction between the different layers. Deadlocks and race-conditions often happen, because of the asynchronous workflows and different control frequencies between the layers. The three-layered architecture is also adopted within the area of adaptive systems, which is closely related to the area of autonomic computing, and thereby extends or replaces the MAPE-K cycle [49, 52]. Concerning the three-layered architecture in robotics, the approach for modelling knowledge representation and reasoning processes presented in this thesis is more engaged with the deliberation and sequencing layers than with the control layer.

2.2.3 BDI Architecture

Another architecture that is often implemented as a computational approach is the BDI architecture. The architecture bases on the philosophical model of Bratman [143], where BDI is the acronym for beliefs, desires, and intentions.

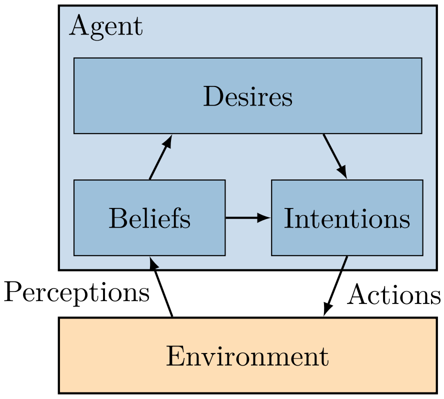


Figure 2.5: BDI Architecture [143]

The BDI architecture also includes a single cycle (see Figure 2.5) and therefore, follows the sense plan act idea from the early days of autonomous robots. Nevertheless, the focus is shifted to a more explicit representation of knowledge and goals. The beliefs module incorporates every knowledge that the robot believes to be true. The beliefs of the robot do not need to be true and will change by continuously sensing the environment or by communicating with other robots. The desires of a robot include the abstract purpose of the robot, for example, to assist humans. Finally, the intentions of a robot are formed by the combination of desires and beliefs. For example, if the robot believes that a human needs to rest and the robot has the desire to assist humans, then the robot could establish the intention to guide the human to a chair. Desires and beliefs create specific goals the robot wants to achieve. Based on these goals and its beliefs, the robot creates intentions that, when followed, create plans. So a plan in this example could describe the way how to approach the human, offer to guide her to a chair, and drive to the next free chair.

2.2.4 Cognitive Capabilities

The explained architectures are the most commonly known in the research areas of autonomous robotics and adaptive systems and form a representative sample of approaches. Interestingly, all approaches endow agents with at least two of the following three cognitive capabilities: reasoning, planning, or learning. Within this thesis, these three capabilities are understood as the key to achieving intelligent behaviour.

Reasoning

Reasoning, as understood in this thesis, is the process of creating new knowledge by applying inference algorithms to existing knowledge. Therefore, reasoning requires some knowledge to start reasoning. The only architectures that represent knowledge explicitly are the MAPE-K architecture with its knowledge module and the BDI architecture with its belief module. However, reasoning takes place in all presented architectures. Details how reasoning is implemented and understood in the context of this thesis are given in Chapter 4.

Planning

The capability of planning can be seen as a special case of reasoning. In particular, planning takes place when the reasoning process is concerned with sequences of actions that shall achieve certain goals. As a result, planning creates new knowledge in the form of sequences of actions, denoted as plans. Even Shakey, whose software architecture implemented the Sense-Plan-Act cylce, executed some simple planning algorithm known as STRIPS [158]. Both, the MAPE-K and the sense plan act architecture have an explicit planning module, but also the three-layered and BDI architectures create plans within their deliberation layer and by translating intentions into actions, respectively. Although planning is not in the focus of this work, for the evaluation scenarios (see Chapter 11), the agents have to carry out some planning, nonetheless.

Learning

Learning, as understood in this thesis, especially means to adapt one's own behaviour according to new knowledge (see Section 2.1.2). In BDI architectures, this capability is only possible to some degree. New beliefs can change which goals and plans are rationally pursued, and as a result, the behaviour of the agent is adapted. In the MAPE-K architecture, learning is even more limited, as goals are not explicitly represented and therefore hard to change. Learning in computer science, in general, is often associated with machine learning, a subfield of the research area of artificial intelligence. The outcome of machine learning algorithms are models that generalise from given data and provide suitable responses to data that have not been given to the system before. Machine learning algorithms are often designed to require a minimal amount of apriori knowledge in order to create novel knowledge during their training phase [17, 38]. The approaches applied in machine learning often belong to the category of connectionist approaches and are not part of this work. Instead, the inherent property of symbolic reasoning systems of being teachable at runtime [115] is utilised in the presented work. Concerning the Chinese room thought experiment, it means that humans can rewrite the rule book through teaching an agent.

2.3 Multi-Agent Systems

Making the step from a single agent to a multi-agent system is often a natural development or sometimes even necessary for particular domains. The reason for this is the variety of advantages that a multi-agent system provides. In a logistics scenario, for example, the amount of automated guided vehicles (AGV) needs to scale with the size of the warehouse. Further, adding redundant systems is a common way to increase the reliability of a system of systems. Finally, also in a domestic household scenario, splitting up capabilities, like to vacuum-clean, tidy up, and to do the dishes, among different kinds of domestic service robots that are easier to construct than a single robot that can accomplish all these things. Without giving further examples, the advantages of a multi-agent system over a single agent system are increased modularity, flexibility, scalability, availability, as well as potential load sharing and lower costs. Although there are many more aspects related to multi-agent systems [127], this foundational section is confined to aspects most relevant for this thesis, i. e. organisational structure of the agents, communication patterns, and the assignment of tasks to agents.

Generally speaking, the advantages mentioned above come at the cost of increased complexity, potential heterogeneity, real parallelism, and the need for communication activities. A critical factor for tackling these challenges and gaining the benefit of multi-agent systems is the chosen form of organisation for the system. In the literature, multifaceted forms and use cases can be found [109], but for this thesis, we focus on a *team* of domestic service robots. This assumption, for example, implies that the agents have a physical embodiment and do not migrate between systems. Therefore, the research area of mobile software agents is only weakly related to this thesis and especially security issues like the unsolvable malicious host problem [130] are not relevant for this work.

A further implication of forming a team is that the individual interests of the team members do not conflict with each other, and they act in order to achieve a common goal. As a result, communication protocols suitable for auctions, negotiations, and argumentations are less common than cooperative forms like the performatives in speech act theory [160]. Two famous attempts to standardise the communication languages between agents were the Knowledge Query and Manipulation Language (KQML) [138] and its follow up, the Agent Communication Language of the Foundation for Intelligent Physical Agents (FIPA-ACL) [129]. The FIPA-ACL standard was criticised for its requirement that an agent compatible with the standard, needs to comply with a particular behaviour when it is receiving or sending a corresponding speech act performative. A general agent communication standard should not influence the behaviour of an agent [67, 132]. Therefore, similar to the well-known agent framework Jade [125], the work presented in this thesis adopts some of the speech act performatives of the FIPA-ACL standard, for example, in order to share and request information between the agents. Independently from the concrete performative, there are three aspects of a speech act, the locution, the illocution, and the perlocution. The locution is the utterance of the speaker, which is equivalent to sending a message in the domain of domestic service robots. The meaning intended by the speaker of the locution is called illocution. The agents in a team are expected to have the same semantics regarding the content of a message, i. e. they have access to the same ontology or extended rulebook, in terms of the Chinese Room thought experiment (see Section 2.1). The perlocution, finally, is the action that follows as a result of the locution. Details about the communication patterns in this work are given in Chapter 9.

The common goal that a team of agents is trying to achieve is often separable into sub-tasks which need to be assigned to a subset of agents in the team. This is denoted as task assignment problem. The literature [113, 1] identifies different instances of the task assignment problem along three dimensions. The first dimension distinguishes whether an agent can be assigned to only one task, denoted as single-task robots (ST), or whether an agent can be assigned to several tasks at a time, denoted as multi-task robots (MT). The second dimension switches tasks and agents and distinguishes between tasks that require only one agent for their execution, single-robot tasks (SR), and tasks that require several robots to contribute, multi-robot task (MR). The last dimension distinguishes between task assignments that are instantaneous assignments (IA) and task assignments that extend over time (TA). Instantaneous assignments allow as the name suggests, only assignments for the present situation, without any planning for future task allocations.

Time-extended assignments, instead, rely on a model of all tasks and how they are related in time. The task assignment problem is solved for this work by the ALICA Framework, as explained the following chapter.

ALICA Framework 3

The ALICA Framework, originally developed by Skubch in his PhD thesis [77], allows to model the behaviour of multi-agent teams and to coordinate the execution of this behaviour. The acronym ALICA [aliːsa] stands for ***A** **L**anguage for **I**nteractive **C**ooperative **A**gents* and focuses on the most important point about the ALICA Framework – the formal language it defines. The ALICA Language, as Skubch [77] defines it, is intended to provide a formal operational semantics for ALICA programs in order to allow anyone to implement a corresponding runtime engine for executing an ALICA program. Furthermore, Skubch et al. provided a graphical modelling tool, termed Plan Designer, for describing ALICA programs as well as an engine, that is written in the GNU/Linux' version of C# (Mono[1]), intended as a reference implementation. Therefore, we recognise three major parts of the ALICA Framework: The formal language, the modelling tool, and its runtime engine. In the following sections, we explain the language (Section 3.1 and 3.2) together with its modelling tool (Section 3.3), elaborate the architectural key properties of the runtime engine (Section 3.4), and point out the limitations (Section 3.5) that the ALICA Framework suffered from before we extended it for this thesis.

3.1 Propositional ALICA

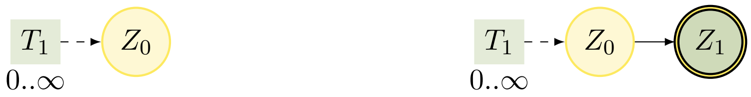
Skubch [77] describes two versions of ALICAPropositional ALICA, and its extension General ALICA. We follow the distinction between Propositional and General ALICA as the extensions made by this thesis mainly refer to the features of General ALICA. The most intuitive way of understanding ALICA and programs described with ALICA is by going through the modelling elements of ALICA in a bottom-up way, starting with the notion of finite-state machines (FSM) according to ALICA.

3.1.1 Finite-State Machines

An FSM in ALICA includes a non-empty set of states and a set of transitions, each connecting two states in a directed way. An FSM has exactly one initial state, that is annotated with a task, a minimal, and a maximal cardinality. A task symbolically represents the abstract purpose of the FSM as intended by the designer. Therefore, tasks allow ALICA Agents to decide whether they are capable of executing an FSM and how good they would perform. Further details about the task assignment in ALICA gives Section 3.1.5. The minimal cardinality is the minimal number of ALICA agents required

[1] Mono - http://www.mono-project.com/ [last accessed on October 25th, 2020]

to execute the annotated FSM successfully. Accordingly, the maximal cardinality is the maximal number of ALICA agents that are allowed to execute the annotated FSM at the same time. The counterparts of the initial states are the terminal states. Terminal states are crucial for the distinction between successful or faulty execution of FSMs. Therefore, they are sub-categorised in success and failure states.



(a) Simple Finite-State Machine

(b) Finite-State Machine with Success State

Figure 3.1: Finite-State Machines with and without Success State

Figure 3.1 shows one FSM without a success state (Figure 3.1a) and one with a success state (Figure 3.1b). The FSM without a success state is never successfully executed. Its execution state is always denoted as *running* because agents entering the FSM will remain in State Z_0 until the execution of the FSM is terminated externally. The FSM with a success state is successfully executed when a certain number of agents have reached the success state. The number of agents required to reach the success state is one if the minimal cardinality of the task is zero. Otherwise, the minimal cardinality is also the number of agents that must reach the success state in order to make the FSM executed successfully. In contrast to the success semantics, an FSM fails when at least one agent reaches a failure state. Finally, it is important to note that whenever an FSM succeeds or fails, the same holds for its annotated task.

3.1.2 Behaviours and Conditions

The semantics of transitions and states in ALICA are switched compared to common concepts like Kripke structures [161], Petri nets [163], or Timed Automata [137]. States in ALICA include a set of behaviours and plans[2] that describe what the agents should do when they are in a state. They execute these behaviours and plans until a precondition guarding an outgoing transition holds. Depending on the domain, a condition describes a state of the world the agents are interacting with. In the aforementioned concepts, these semantics are switched, because, within these concepts, states describe the state of the world and transitions embody changes in the world, induced by agents executing actions.

ALICA considers behaviours and conditions as domain-specific elements. Therefore, these elements are, to some extent, black boxes for ALICA. The ALICA runtime engine can start and stop the execution of behaviours according to the given ALICA program, and it knows when behaviours succeed or fail. Furthermore, ALICA behaviours are annotated with three conditions: a precondition, a runtime condition, and a postcondition. Like the

[2] For plans, please read Section 3.1.3.

conditions that guard transitions, these conditions are black boxes. The ALICA runtime engine can ask whether conditions hold and the results must be *true* or *false*. From the modelling perspective, these different types of conditions are the same. The only difference is how the ALICA runtime engine reacts to their truth value. Preconditions, e. g., must hold before behaviours can be executed or a corresponding transition can be passed. Runtime conditions must hold during the execution of behaviours; otherwise, their execution is cancelled. Finally, postconditions are expected to hold after the successful execution of behaviours. Postconditions, therefore, do not influence the execution of an ALICA program, but they are considered as annotations useful for planning algorithms that compose plans out of a set of basic behaviours.

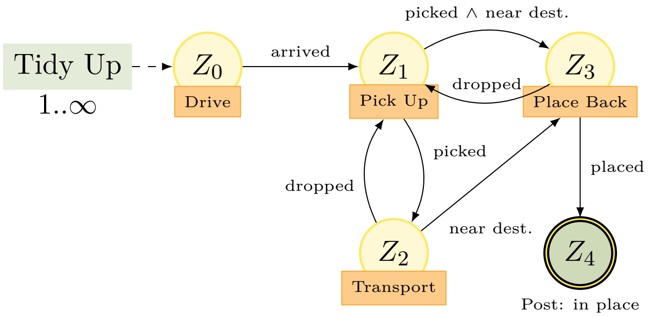


Figure 3.2: Finite-State Machines for Cleaning Up

In Figure 3.2, an FSM for cleaning up by pick-and-place items in a household. While an agent occupies state Z_0, an agent is driving(*Drive*) to an item that is not in place. As soon as it arrives at the item, the precondition of the outgoing transition holds, and it picks up the item. Afterwards, depending on the distance to the destination of the item, the agent needs to transport the item to the destination, or it can directly put it back where it belongs. The preconditions and runtime conditions of behaviours (orange boxes) in the FSM are dropped for the sake of clarity, but the annotations of the transitions represent the preconditions of the transitions. The success state of the shown FSM has a postcondition. Similar to the postconditions of behaviours, it describes the state of the world that is expected to hold after the FSM is executed successfully. Postconditions can be annotated to success and failure states and allow the ALICA runtime engine to react to different reasons for successful or faulty execution of FSMs and tasks.

3.1.3 Plans and PlanTypes

Following the bottom-up way of explaining ALICA, grouping several FSMs into plans introduces the next level of complexity. An ALICA plan is a set of FSMs that work towards a common goal. The FSM in Figure 3.2, e. g., could benefit from another FSM that makes agents searching for items that are not cleaned up.

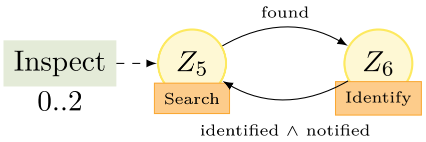


Figure 3.3: Finite-State Machines for Searching Items to Clean Up

The FSM in Figure 3.3 is never successful because of a missing success state. Agents executing the FSM, therefore search and identify items forever, but the number of agents doing this is restricted to two. Both FSMs (see Figure 3.2 and 3.3) have the same purpose, and the agents executing them are intended to work together. The agents searching and inspecting items should send their perceptions to the agents that are cleaning up the items. Nevertheless, Propositional ALICA offers no option to model this kind of communication explicitly. Instead, such kind of communication is considered domain-specific and could be implemented by the domain-specific black box of ALICA behaviours or through synchronised plan variables provided by General ALICA (see Section 3.2).

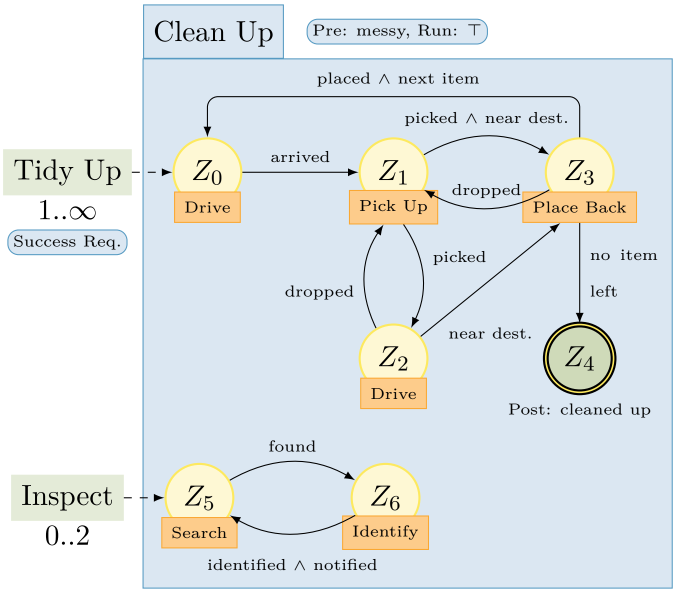
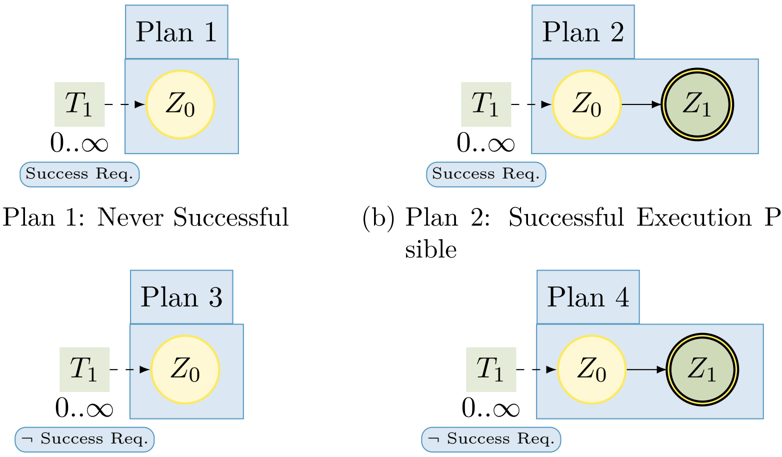


Figure 3.4: Simple Clean Up Plan

The *Clean Up* plan in Figure 3.4 is composed of the *Tidy Up* and the *Inspect* FSM. The *Tidy Up* FSM is slightly modified, compared to Figure 3.2, for cleaning up more than one item. The success state is now reached when no item is left to be cleaned up. Like a behaviour, a plan can have a precondition and a runtime condition. The feature of postconditions for a plan is already available through the postconditions of the terminal states, which allows reacting on different reasons for a successful or failed plan execution. In Figure 3.4, the *Tidy Up* task is declared to be required for a successful plan execution through the annotation below the cardinality interval.



(a) Plan 1: Never Successful

(b) Plan 2: Successful Execution Possible

(c) Plan 3: Always Successful

(d) Plan 4: Always Successful

Figure 3.5: Plans and Their Success Semantics

Similar to an FSM, a plan succeeds, when all FSMs that are required succeed are successfully executed. Please note that this is a universal quantification, i. e. if a plan has no FSMs that are required to be successful for the plan to be successful, then the plan is always considered to be successful. Another implication of the success semantics of a plan is that a plan with an FSM, which does not have a success state but is required to be successful, can never be successful. In Figure 3.5, the four different cases are illustrated for the most simple kind of plans. Only *Plan 2* can change its execution state to be successful, depending on the execution state of the FSM. All other plans are not useful regarding their execution state, because the execution states of fixed independent of the execution state of the corresponding FSMs.

Plans provide a shared context for FSMs that need to be executed in parallel in order to achieve a goal. However, plantypes are sets of plans that have a similar goal. The plans in a plantype are alternatives to each other and are not meant to be executed in parallel. Instead, always only one plan from a plantype is chosen to be executed at a time (see Section 3.1.5).

3.1.4 Plan Hierarchies

Additionally, to behaviours, states can also contain plans and plantypes. Inserting plans and plantypes into states introduces a new hierarchy level in an ALICA program, but before we explain the concept of plan hierarchies in detail, it is necessary to understand the relation between plantypes, plans and behaviours from a theoretical point of view. Note that restricting the content of states to plantypes does not restrict ALICA's expressibility. The expressibility is not restricted, because a state with a plantype containing only a single plan is semantically the same as inserting the plan itself into the state. Furthermore, it is possible to represent a behaviour with the help of a plan, without changing the semantics of the ALICA program. In [77, p. 54], these kinds of plans are denoted as canonical behaviour plans. A canonical behaviour plan is a plan like *Plan 2* in Figure 3.5b with the corresponding behaviour inside State Z_0 and the postcondition of State Z_1 set to the postcondition of the behaviour.

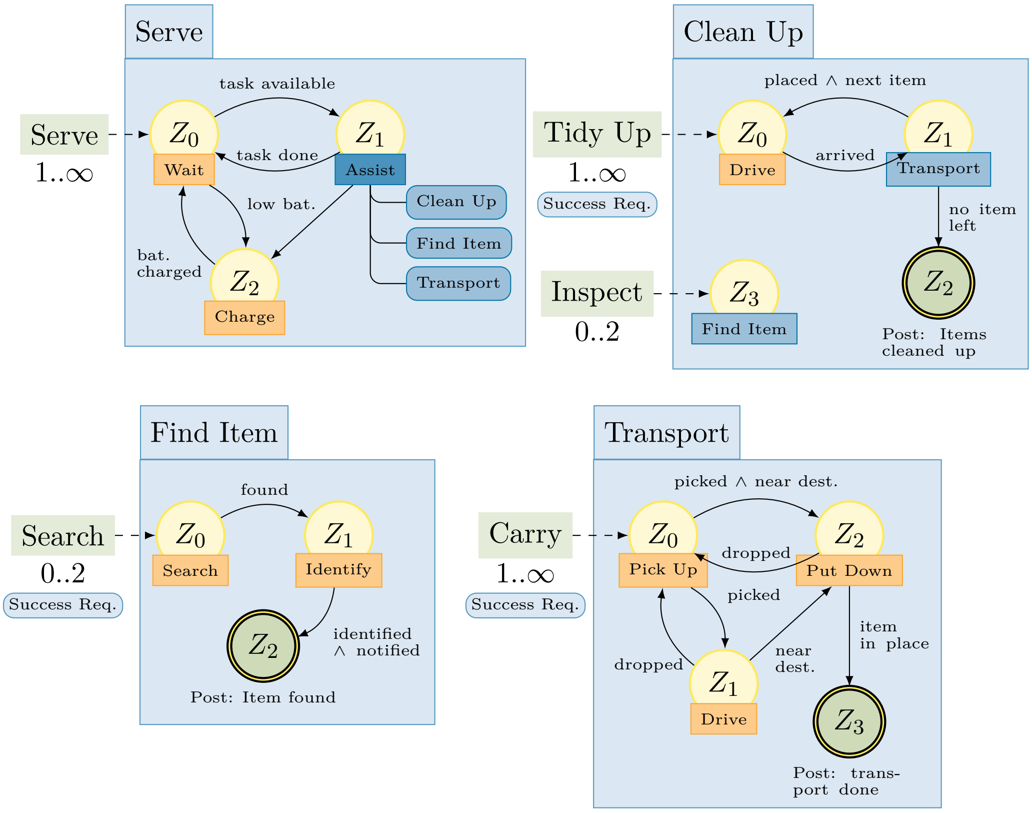


Figure 3.6: Hierarchy of Plans for Service Robots

Figure 3.6 shows a hierarchy of plans from the Service Robotic domain, using modified plans from the aforementioned examples. State Z_1 of the *Serve* plan includes the *Assist* plantype that consists of the three remaining plans in the figure. The *Clean Up* plan directly utilises the plans *Find Item* and *Transport* in the states Z_3 and Z_1, respectively. Furthermore, the *Drive* behaviour is utilised by the *Clean Up* plan in state Z_0 as well as by the *Transport* plan in state Z_1.

In ALICA, plan hierarchies always form a directed acyclic graph (DAG). The DAG for the plan hierarchy of Figure 3.6 is shown in Figure 3.7. The *Serve* plan is the top-level plan where all agents start the execution of the ALICA program. Depending on the assigned tasks and the progress in the corresponding FSM, agents occupy different states and therefore, execute a different set of behaviours and plans at runtime. The edges of the DAG in Figure 3.7 are annotated with the task, state and, if present, the plantype an agent has to follow, in order to execute the plan or behaviour the edge is pointing at.

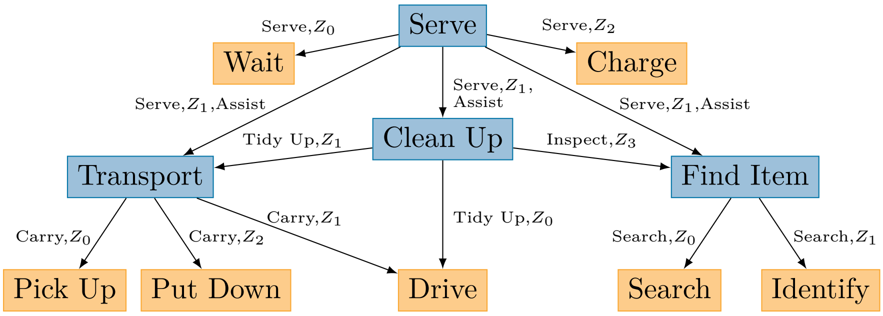


Figure 3.7: Directed-Acyclic Graph for Plans in Figure 3.6

During modelling, on the one hand, the representation of an ALICA program as a DAG enables the reuse of plans, behaviours, and plantypes, which reduces the modelling effort and increases the reusability of parts of the ALICA program. At runtime, on the other hand, the DAG is interpreted as a tree data structure, which means, that the *Transport* plan, for example, is instantiated one time as a child of the *Serve* plan and one time as a child of the *Clean Up* plan. The agents in the different instances behave as agents that do not execute the same plan at all and therefore, do not cooperate with regard to the execution progress of the *Transport* plan. Finally, although the given plan hierarchy does not demonstrate it, an arbitrary number of behaviours, plans and plantypes can be inside a state, meaning they are executed in parallel by the agent occupying the corresponding state.

3.1.5 Roles and Tasks

We already introduced tasks as modelling elements that identify FSMs in the context of plans. According to our literature research, it is still common to assign tasks directly to robots or agents [41, 1, 53]. Nevertheless, ALICA introduces an extra role layer for abstracting agents and assigns tasks to roles instead of agents. Which role an agent is assigned to, depends on the application domain, the capabilities of the agent, and the composition of the team [77, p. 37]. The ALICA runtime engine allows for an extra role assignment module that can implement an arbitrary role allocation algorithm. In the application domains of ALICA, the role of an agent changes less often than its allocated task. A service robot, for example, might be capable of opening doors and carrying items as long as its arm is operable. In case the arm breaks, the agent is no longer assigned to the role requiring these capabilities.

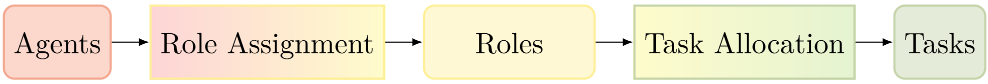


Figure 3.8: Role and Task Abstraction Layers

The benefit of a role abstraction layer is that the designer of an ALICA program does not need to know the capabilities of all agents that potentially will execute the plans of the program. Instead, she describes the roles by specifying the required capabilities for an agent to be assigned to these roles and afterwards describes how suitable a role is to execute a task. As a result, agents unknown at modelling time of an ALICA program are able to execute the program, as long as they can be assigned to any role. The double-layered abstraction through roles and tasks, as illustrated in Figure 3.8, makes ALICA programs highly adaptable and robust with regards to unknown and changing agent capabilities.

3.1.6 Synchronisations

Synchronisations in ALICA programs enforce a group of agents to pass over a set of transitions synchronously. The semantics of synchronisations is similar to that of joint intentions, according to [141].

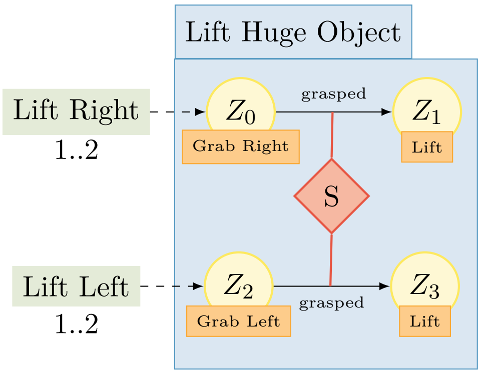


Figure 3.9: A Plan with Two Synchronised Transitions

Figure 3.9 shows a plan for synchronously lifting a huge object. Two robots are necessary to execute this plan, as the minimum cardinality of both tasks is one. The idea of the plan is that one or two robots grab the huge object on the left and the right side. The transitions guarding conditions require that the robots have grasped the object. Before they start to lift the object, they furthermore, need to synchronise their transit over the transitions. A three-way-handshake establishes the common belief that all robots involved in state Z_0 and Z_2 are ready to pass the transitions and that every robot believes in this common belief, i. e. a mutual believe about the intention to pass the transitions is established. With synchronised transitions, it should be possible to lift a huge table without spilling water from the glasses that stand on top of it. Nevertheless, the accuracy of the synchronisation depends on the communication latency between the involved robots.

3.2 General ALICA

Propositional ALICA with its modelling elements, is already expressive enough to describe complex strategies for teams of autonomous robots. Nevertheless, modelling within the constraints of Propositional ALICA introduces limits regarding the generality and reusability of the modelled ALICA programs. Considering the plan in Figure 3.9, the question is: "How to specify the object that needs to be lifted?" So far in Propositional ALICA, two options exist that are explained in the following paragraphs.

The object could be hard-coded into the plan and behaviours, albeit limiting their reusability. Two behaviours for each object (left and right side to grab) would be necessary, and if unknown objects are encountered at runtime, the behaviours and plans need to be generated and planned at runtime, respectively. Another option is to encapsulate the encoding of the object in the black box part of the behaviours. An algorithm that determines the kind of object that needs to be lifted, for example, gets sensor data as input and configures the parameters of behaviours accordingly. A disadvantage of this approach would be the separation of the ALICA program structure from the logic that interacts and reasons about the object. The lift behaviours in state Z_1 and Z_3, for example, would not be able to reference the object that was just lifted. In order to guarantee that the same object is lifted that was just grasped, the black box part of the behaviour would need to establish a connection that already exists in the plan itself, represented through the transition between the *Grab* and *Lift* states.

General ALICA extends Propositional ALICA in a way that allows referencing the same objects from different states and implements several other mechanisms that improve the expressiveness of ALICA programs by combining knowledge about the ALICA program structure and the current situation. The modelling elements and modules necessary for this are explained in the following subsections.

3.2.1 Behaviour Configurations

Behaviour configurations are the simplest way to avoid code duplication induced by copied behaviours. They allow parametrising a behaviour with simple key-value pairs, depending on the context in which the behaviour is executed. A behaviour that drives a service robot to a particular spot in a house, e. g., can be given another spot in each configuration. In this example, behaviour configurations are a viable solution if the house includes only a finite amount of spots for the service robot to drive to. However, if the exact spots are not known at modelling time, behaviour configurations cannot be used at all. Instead, the dynamic approach, including variables, constraint satisfaction problems, and the corresponding solver, explained in the next sections, will be more appropriate in this situation.

3.2.2 Variables

The central modelling elements in order to lift Propositional ALICA to General ALICA are variables. Variables belong to plans, plantypes, or behaviours that provide a context to the variables. There are two kinds of variables available in General ALICA: Agent variables that are bound to a quantified group of agents, e. g., *there is a variable X for each agent A in state S*, and free variables that are not bound by any quantification. The only way to assign values to the variable is through the constraint satisfaction problem solver of ALICA as presented in [77] (see Section 3.2.5).

3.2.3 Variable Bindings

The variables of plans and plantypes in ALICA programs can be bound to variables of other plans, plantypes, and behaviours via a binding. The purpose of bindings in ALICA is to combine different constraints for the same variable in the plan hierarchy of an ALICA program. Details about constraints in ALICA are given in Section 3.2.4 and 3.2.5, but before it is necessary to understand the mechanism of variable bindings. Two objects can include variable bindings in ALICA. States, on the one hand, use bindings to bind the variables of the plan they are part of to variables of plans, plantypes, or behaviours inside themselves. Plantypes, on the other hand, bind their variables to variables of plans inside themselves. The variable bindings are transitive, i. e. that a variable of the top-level plan, e. g., can be bound to variables of plans several levels below in the plan tree structure.

The advantage of the variable bindings is the possibility to constraint the same variable differently on different levels of the ALICA program structure. The Clean Up Plan in Figure 3.10, e. g., may constraint its variable $X_{CleanUp}$ to some object in the house, while the different transport plans constraint their variable X_{Light} and X_{Heavy} differently. Imagine some service robots have a stronger, but more restricted arm, i. e. that they are able to lift heavy objects if the objects have a proper handle attached. Therefore, the Transport Heavy Plan would constraint its variable X_{Heavy} to objects with handles and the Transport Light Plan would constraint its variable X_{Light} to arbitrary light objects. Both transport plans together form the Transport Plantype in State Z_1 of the Clean Up Plan. In order to make $X_{CleanUp}$ reference the same object as X_{Heavy} and X_{Light}, a parametrisation in State Z_1 of the Clean Up Plan needs to bind $X_{CleanUp}$ to the variable $X_{Transport}$ of the Transport Plantype, and a binding in the Transport Plantype needs to bind $X_{Transport}$ to the variable X_{Heavy} and X_{Light} accordingly.

Although both transport plans constraint their variables differently, solutions to the constraint satisfaction problem also need to reference objects inside the house, due to the constraint of the Clean Up Plan and the variable binding as shown in Figure 3.11. It is important to note that variable bindings, as described in this section, can only be applied to free variables. The reference implementation of ALICA [77], e. g., does not allow to

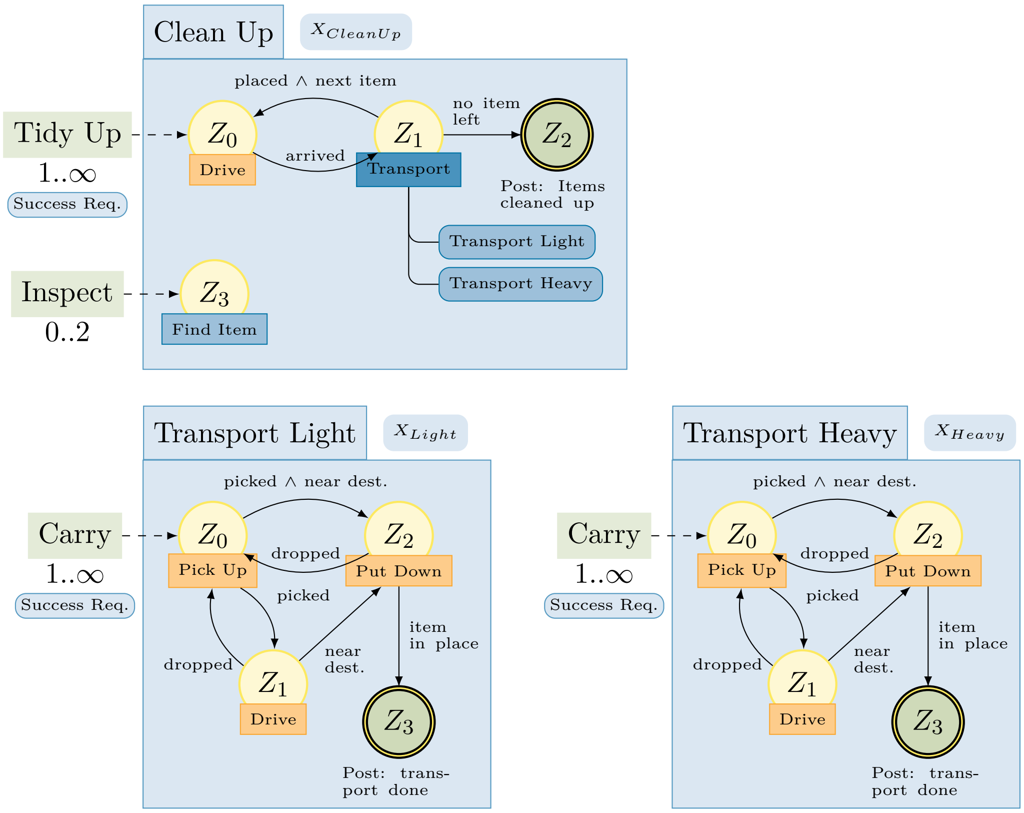


Figure 3.10: Plan Hierarchy with Variables

$$\begin{tikzcd} & & X_{Light} \\ X_{CleanUp} \arrow[r, Rightarrow] & X_{Transport} \arrow[ur, Rightarrow] \arrow[dr, Rightarrow] & \\ & & X_{Heavy} \end{tikzcd}$$

Figure 3.11: Variable Binding

bind quantified variables to variables of plans that are lower in the plan hierarchy. The Clean Up Plan in Figure 3.10, e. g., makes much more sense when the variable $X_{CleanUp}$ is quantified for all agents participating in the plan, i. e. each agent has its own variable that refers to an object that should be cleaned up. Without the quantification, all agents would consider the same object, and all objects would need to be cleaned up one after another. Although theoretical possible the quantified variable $X_{CleanUp}$ cannot be bound to X_{Heavy} or X_{Light}.

3.2.4 Constraint Satisfaction Problems

A constraint in ALICA limits the range of values assignable to variables, but constraints only exist in the context of preconditions and runtime conditions (not postconditions) of ALICA programs. Therefore, each precondition and runtime condition may have an optional set of associated constraints, and each precondition and runtime condition defines which variables may be referenced by its constraints. Furthermore, the conditions guard the constraints in a way that solutions to constraints are asserted (not proven), if and only if its guarding condition holds. The conditions are denoted as weak guards for the constraints, while a strong guard would require to prove the existence of a solution for the constraints. In ALICA the weak guard interpretation is preferred because proving the existence of a solution to a constraint satisfaction problem (CSP) can be intractable, depending on the expressiveness of the utilised constraint formalism.

Considering the example of the Heavy Transport Plan from Figure 3.10, the variable X_{Heavy} is constraint to refer to objects that have a proper handle attached, in order to make the heavy lifting possible for the arm of the agent. This constraint is attached to the runtime condition of the plan, meaning that as long as the plan is executed and the runtime condition holds, the constraint is asserted and therefore influences the behaviour of the agent. As a result of the weak guard semantics, it might happen that temporarily there is no solution for the constraint available, but the agent would never lift an object without a handle. Under the weak guard semantics, only behaviours query for solutions to a subset of their variables. The Drive behaviour, e. g., queries for a solution to the $X_{CleanUp}$ variable and changes its motion commands, in order to drive towards the assigned object.

Triggered by the query of a behaviour, the ALICA runtime engine constructs the CSP, including all constraints that relate to the queried variables and presents the CSP to its CSP solver (see Section 3.2.5). The CSP construction algorithm is, according to the author, the most complex part of the ALICA runtime engine. One prerequisite of the algorithm is the tracking of active constraints according to their guarding condition. The runtime condition of a plan, e. g., must hold during the execution and therefore also activates the corresponding constraints. The same holds for all other conditions in ALICA programs, except for preconditions of transitions. Once an agent passed a transition, because it considers the attached precondition to hold, it also activates the attached constraint. The activate does not revert, even if the agent follows another transition back to its original state. In order to keep track of the activated constraints, each plantype, plan, and behaviour manages its own constraint store. The CSP construction algorithm starts from the querying behaviour with the given subset of variables and traverses the plan hierarchy up towards the top-level plan of the executed ALICA program. On each level, it collects all activated constraints that refer to the queried variables from the constraint stores of the plantypes, plans, and behaviours its agent is executing, while considering the variable bindings of plantypes and states. Thus, all plans and behaviours the agent is executing can contribute to the constraints of the resulting CSP. The only limitation is the locality principle of ALICA (see Section 3.5.1) that forbids references between different branches of the tree of executed plantypes, plans, and behaviours.

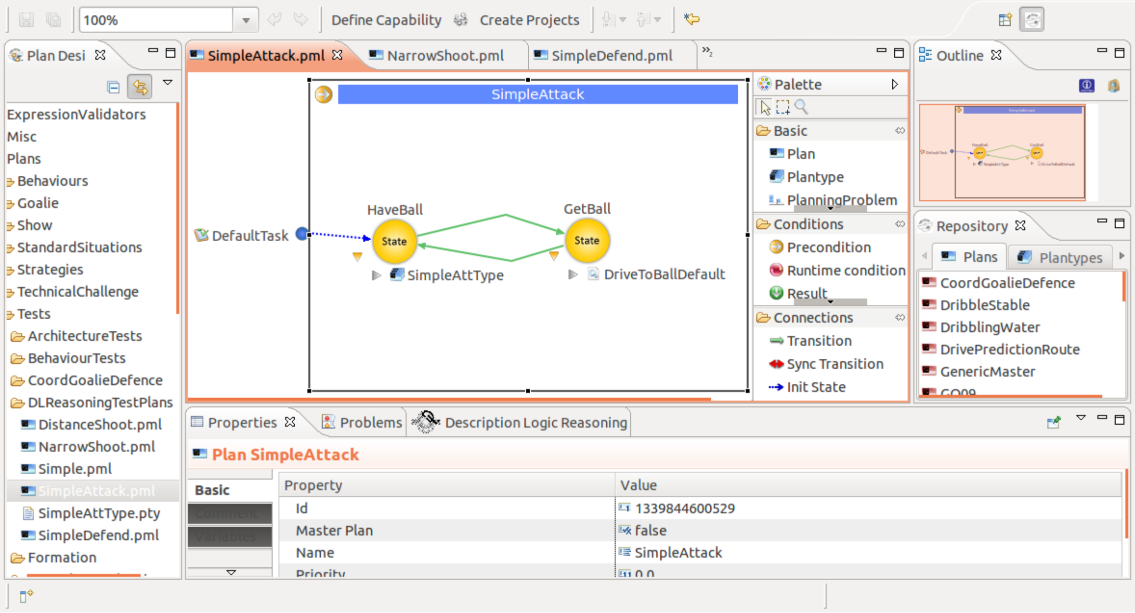
3.2.5 Constraint Satisfaction Problem Solver

The CSP solver presented in [77] solves instances of the problem domain of continuous non-linear CSPs. A continuous non-linear CSP (CNLCSP) is defined as a triple (ϕ, X, C), where ϕ is a propositional formula with variables P, X is a set of variables ranging over $\mathbb{R}$, and every $p_i \in P$ identifies a constraint $c_i \in C$ such that $c_i = f_i(x) \circ_i 0$, where $\circ_i \in \{<, >, \leq, \geq, =, \neq\}, x \subseteq X$, and all f_i are arbitrary functions $\mathbb{R}^k \mapsto \mathbb{R}$.

The given definition of the problem domain implies that variables in ALICA can only be assigned to values in $\mathbb{R}$ and arbitrary logic symbols or strings are excluded. This limitation also applies to ALICA in general, because the CNLCSP solver is the only way to assign values to variables in ALICA as presented in [77]. The solver implements a two-step algorithm: First, it automatically differentiates the functions f_i, and second, it applies the RPROP Optimization Algorithm [123] for finding a local optimum that fulfils the given constraints. The performance of the solver is similar to other state-of-the-art CNLSCP solvers, but the incompleteness property of the RPROP algorithm also applies to the solver [78].

3.3 Plan Designer

The Plan Designer belongs to the ALICA Framework and is the graphical modelling tool necessary to describe an ALICA program. Furthermore, the Plan Designer is able to generate code stubs that are necessary for the ALICA runtime engine to execute the modelled ALICA program.



Property	Value
Id	1339844600529
Master Plan	false
Name	SimpleAttack
Priority	0.0

Figure 3.12: The Original Graphical User Interface of the Plan Designer

Figure 3.12 shows the graphical user interface of the Plan Designer before we redesigned it for this thesis. The software architecture is based on the Eclipse Rich Client Platform [57]. It explicitly uses three distinct frameworks that rely on the Rich Client Platform system: The back-end model of the Plan Designer is described within the Eclipse Modeling Framework [101], the graphical editing is based on the Graphical Editing Framework [82], and the code generation module utilises the Open Architecture Ware system[3](see Figure 3.13).

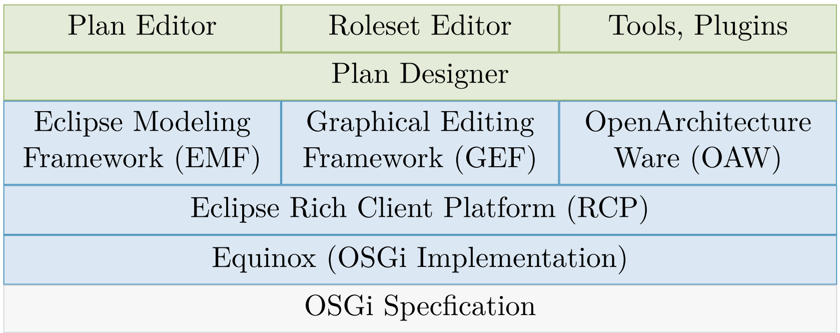

Plan Editor
Roleset Editor
Tools, Plugins
Plan Designer
Eclipse Modeling Framework (EMF)
Graphical Editing Framework (GEF)
OpenArchitecture Ware (OAW)
Eclipse Rich Client Platform (RCP)
Equinox (OSGi Implementation)
OSGi Specfication

Figure 3.13: Plan Designer Architecture

The advantage of this combination of frameworks is their compatibility with each other. The Graphical Editing Framework, e. g., follows the Model-View-Control (MVC) Pattern [135] and can directly interact with models described with the Eclipse Modeling Framework (EMF). Moreover, the template language of the Open Architecture Ware system can also generate code for a given EMF model. As a result, a lot of the functionality of the Plan Designer comes from the utilised frameworks. However, the maintenance effort for the Plan Designer was significant, because the utilised frameworks tended to introduce breaking changes in each major release and finally, the support for Open Architecture Ware reached its end of life.

Figure 3.14 shows the components that interact while creating an ALICA program. The users are interacting with the graphical user interface (GUI) of the Plan Designer. Following the MVC-Pattern, the GUI represents the View Part of the MVC. The controller includes the logic and, e. g., changes the model according to commands issued by the user. If the model changes, the GUI is notified through listeners registered on the model and adapts accordingly. The Plan Designer has two outputs: The EMF model serialised to files in XML format and the generated source code. EMF supports the serialisation of EMF models to XML files out of the box. In order to generate the source code, the EMF Model, together with a source code template, is passed to the OAW system. The ALICA runtime engine parses the XML files and instantiates a runtime model accordingly. Furthermore, the generated source code contains protected regions like method bodies, allowing the

[3] The development on this framework stopped in 2008 and is not available anymore.

user to program the domain-dependent black box parts of the ALICA program. Finally, the compiled source code is linked to the runtime model the engine instantiated, and the ALICA program is executed.

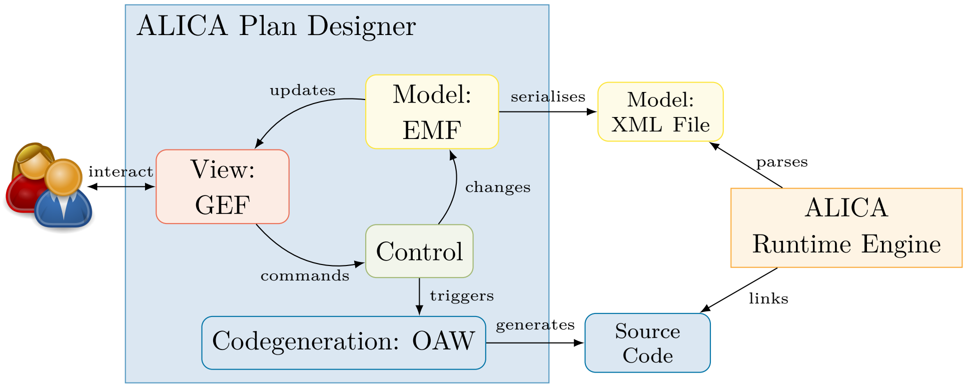


Figure 3.14: Components Interacting while Creating an ALICA Program

3.4 Runtime Engine Architecture

The architecture of the ALICA runtime engine is fully distributed, i. e. a completely independent runtime engine is running on each agent participating in the team. Each runtime engine includes several distinguished modules that belong to different architecture layers, according to their functionality. In Figure 3.15, an overview of the most important modules of the runtime engine is given, and the following sections will explain the different layers bottom-up.

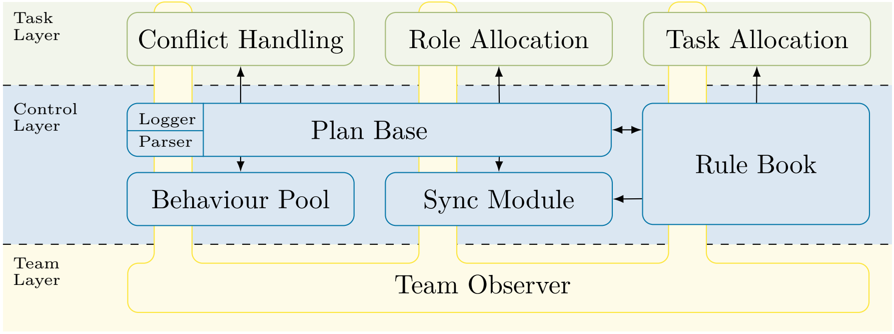


Figure 3.15: ALICA Runtime Engine Architecture

3.4.1 Team Layer

The Team Layer only includes the Team Observer module. Its purpose is to know which agent of the team is active by tracking when the last message arrived for each agent. The

Team Observer is available to all modules in the runtime engine because every module needs to know the current team configuration.

3.4.2 Control Layer

The Control Layer is the most complex layer of the runtime engine because its function is to control the progress of the agent in its ALICA program while coordinating with other teammates. The central module is the Plan Base module. It holds the runtime representation of the ALICA program, including information about which agent is currently occupying which state in the plan tree structure. The Plan Base also controls that the other modules have access to the runtime representation of the ALICA program in the right order. The Behaviour Pool module is similar to a thread pool, including one thread per behaviour. The Plan Base commands the Behaviour Pool to start or stop certain behaviours according to the execution state of the agent in the ALICA program. When a behaviour succeeds or fails, it sends a signal back to the Plan Base accordingly. The Sync Module is responsible for coordinating the passing of synchronised transitions (see Section 3.1.6).

The Rule Book module is the central part of the operational semantics of the ALICA runtime engine. Given a runtime representation of a plan of the ALICA program, the Rule Book module decides about the next step in this plan, according to its rules and the priorities among the rules. The following paragraphs list the rules with decreasing priority and give a summary of the meaning of each rule.

Init The Init rule is triggered if the agent is starting to execute an ALICA program or if the runtime representation of the ALICA program is empty, due to failed plan execution. As a result, the agent is believed to occupy the initial state of the top-level plan, and it is deemed necessary to trigger the task allocation for the initial state.

RoleAlloc In case the team composition changes, because agents join, leave, or change their capabilities, the RoleAlloc rule triggers the reallocation of roles to the team.

BSuccess Every successful execution of a behaviour triggers the BSuccess rule. The rule stops the execution of the behaviour and asserts the postcondition of the behaviour.

TSuccess The TSuccess rule handles the successful execution of a task by the local agent. Therefore, the successful execution is memorised in the Plan Base module and communicated to all other team members, because the successful execution, informally speaking, reduces the minimum cardinality of the task by one, as long as the plan is not restarted or stopped. Primarily, this influences the task allocation module of each agent (see Section 3.4.3) and makes it possible for the agent to get assigned to another task of the same plan.

STrans In case the Sync Module established a mutual belief to pass a synchronised transition, the STrans rule applies the transition over the synchronised transition by stopping the execution of every plan, plantype, or behaviour in the current state, moving the agent and its teammates to the state over the transition and demands a task allocation in this new state.

Trans The Trans rule is similar to the STrans rule, but it does not require to establish a mutual belief with other teammates. Only the precondition of the transition needs to hold, and other agents, that also occupied the former state are expected to move along the transition, too. This assumption reduces asynchronous runtime representations of the ALICA program within the team, because the runtime engine makes optimistic assumptions about the progress of other teammates, instead of relying on potentially delayed communication.

Alloc Whenever it is deemed necessary to make an initial task allocation within the context of state, e. g., because an agent just entered the state by following a transition, the Alloc rule starts the Task Allocation module, in order to recursively assign tasks with the state and all reachable child plans.

Adapt A crucial role for a task allocation is its utility according to the current situation. The Adapt rule recurrently revaluates the utility of all task allocations and triggers a new task allocation, if the agent believes that another allocation is of higher utility.

Apart from the RoleAlloc rule, all rules described so far are denoted as operational rules, have higher precedence than the following rules, and are sufficient for the execution of an ALICA program. The following rules and the RoleAlloc rule are denoted as repair rules. As the name implies, the repair rules handle faulty execution of plans, behaviours, and task allocations. To some extent, the repair rules are domain-dependent, because restarting a behaviour to grasp some object when it already broke by falling to the floor, e. g., is not the best strategy.

The general repair strategy implemented in the ALICA runtime engine is to restart the behaviour, plan, or task allocation up to a configurable number of retries and, in case it was not possible to repair runtime representation on the current plan level, to propagate the failure up the plan hierarchy.

The BAbort rule aborts the execution of behaviours if the behaviour signals that it failed. As a result, the fail counter of the behaviour is increased, and the BRedo rule restarts the execution of the behaviour. If the fail counter reaches its threshold, the BProp rule propagates the failure to the context the behaviour is running, i. e. the fail counter of the plan that includes the behaviour is increased by one. For behaviours, these three rules are simple, because behaviours are atomic from the perspective of the engine and therefore can either be executed or stopped.

In case of plans, the corresponding rules PAbort, PRedo, and PProp are more complex, and the rules PReplace and PTopFail handle other exceptional cases. The PAbort rule stops the execution of the plan and all behaviours, plans, and plantypes running in the current state of the plan. Further, it increases the plans fail counter waiting for other rules to react. While the PAbort rule handles all kind of failures, e. g., failed preconditions, failed runtime conditions, task allocations not fulfilling some tasks minimum cardinalities, the PRedo rule is only triggered if the agents reach a failure state for the first time and all other requirements for the execution of the plan are still met. The PRedo rule simply restarts the execution of the task the agent was working on before and increases the plans fail counter by one. If the fail counter is not zero, the PRedo rule is not applied.

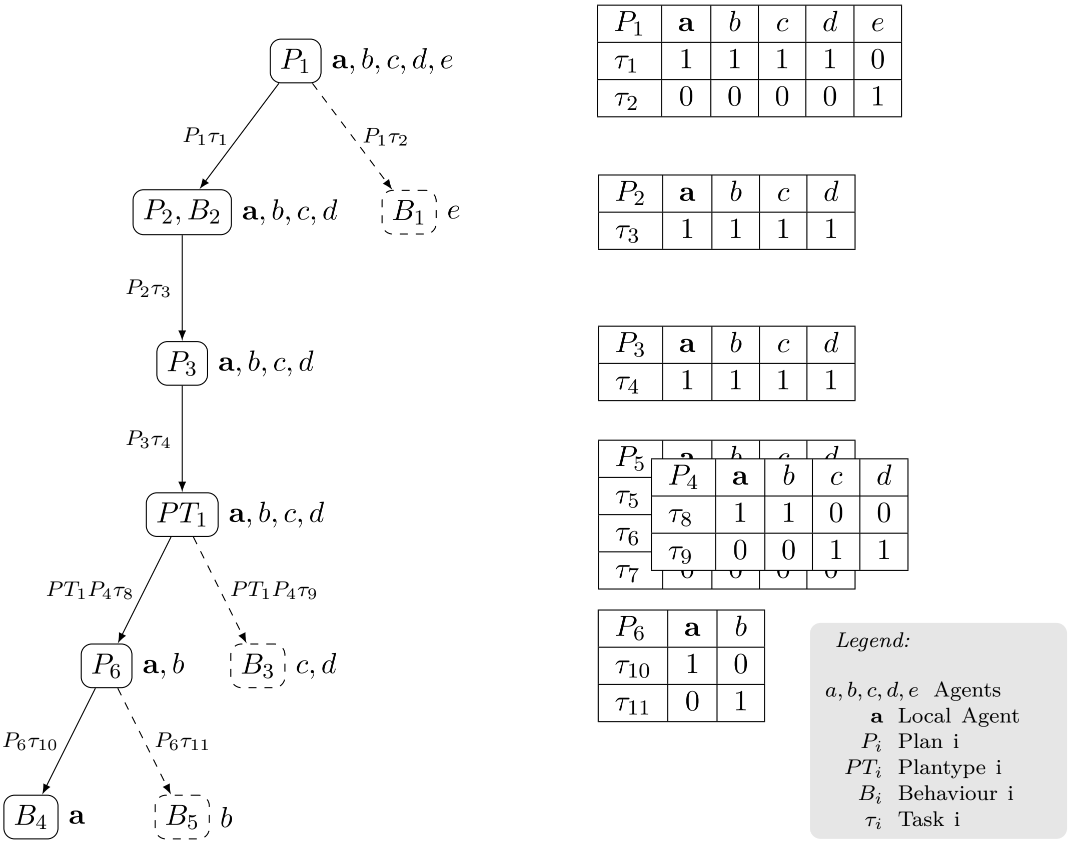
With regard to the propagation of failures, PProp and PReplace offer two different options. The PReplace rule triggers the calculation of a new task allocation within the context of the state the failed plan is running in. The PProp increases the fail counter of the parent plan and aborts it as PAbort would do for the parent plan. In case the fail propagation reaches the top-level plan, the PTopFail rule is triggered and simply calls the Init rule for the top-level plan. The NExpand rule handles the case when a valid task allocation is not possible to find. Therefore, the rule increases the fail count of the plan.

The order of precedence for the repair rules is chosen with regard to their effort to repair the plans and behaviours, starting with the least amount of effort: BAbort, BRedo, BProp, PTopFail, PRedo, PAbort, PReplace, PProp, NExpand.

3.4.3 Task Layer

The Task Layer incorporates all modules that influence the assignment of tasks to agents: The Task Allocation, the Conflict Handling, and the Role Allocation module. The purpose of the Role Allocation module is to assign each agent in the team to a role that fits best to the capabilities of the agent. A role allocation for the team is only recalculated, if a new agent joins the team, an agent is dropped out of the team, or the capabilities of some agent have changed, e. g., due to some upgrade or partial hardware failure.

The Task Allocation module handles the assignment of tasks to agents if a state is entered or some failure occurred during the execution of a plan. There are different classes of task allocation problems, differing with regard to the number of tasks an agent can be assigned to, the number of agents a task can be assigned to, and the duration for which the task assignment lasts. In the case of the ALICA Framework, the task allocation problem is categorised as a multi-task (MT), multi-robot (MR), and instantaneous assignment (IA) problem [1] that is considered as an NP-hard problem.



P_1	**a**	b	c	d	e
τ_1	1	1	1	1	0
τ_2	0	0	0	0	1

P_2	**a**	b	c	d
τ_3	1	1	1	1

P_3	**a**	b	c	d
τ_4	1	1	1	1

P_5	**a**	b	c	d
τ_5	[illegible]	[illegible]	[illegible]	[illegible]
τ_6	[illegible]	[illegible]	[illegible]	[illegible]
τ_7	0	0	0	0

P_4	**a**	b	c	d
τ_8	1	1	0	0
τ_9	0	0	1	1

P_6	**a**	b
τ_{10}	1	0
τ_{11}	0	1

Legend:

a, b, c, d, e	Agents
a	Local Agent
P_i	Plan i
PT_i	Plantype i
B_i	Behaviour i
τ_i	Task i

Figure 3.16: Example of a Recursive Task Allocation

The task allocation problem in ALICA is defined as an optimal assignment problem of agents to tasks. Figure 3.16 illustrates an example instance of this problem. Plan P_1 is the top-level plan and the agents a, b, c, d, e need to be assigned recursively to tasks in the given plan hierarchy. An assignment for one plan is represented as a table, or stack of tables in case of plantypes, where each column corresponds to an agent and each row corresponds to a task in the plan. 1 means the agent is assigned to the task in that row, and 0 means it is not. A valid task assignment only assigns one task per plan or plantype to an agent, i. e. each column includes only one 1. Agent a is the local agent that solves the task allocation problem and afterwards compares its result with the results from its teammates. In case of a conflict, the Conflict Handling module takes over and resolves the conflict. Apart from one task per agent, several other requirements need to be fulfilled for a task assignment of a plan to be valid. The precondition and the runtime condition of the plan must hold under the given task assignment, the minimum cardinality of each task must be fulfilled, and it must be allowed for the role of each agent to execute its assigned task. In Figure 3.16, the dashed branches of the plan hierarchy are not calculated by the local agent a because it is not assigned to the corresponding tasks.

In order to find the optimal assignment of tasks to agents, in ALICA an A* search algorithm evaluates the utility function of each plan. The utility functions are of the form $\mathcal{U}_{p_i} = w_{pri}\, f_{pri} + w_{sim}\, f_{sim} + w_0 f_0 + \ldots + w_n\, f_n$ and include at least the priority summand and the similarity summand. The priority summand considers the aptitude of roles for certain tasks and the similarity summand prioritises task assignments that are similar to currently existing assignments, in order to avoid too many fluctuations in the assignments. The other summands are considered domain-dependent and can be designed to add arbitrary criteria. The A* algorithm is recursively triggered for each hierarchy level along the branch of the local agent. According to the assigned tasks, the amount of agents to be assigned is reduced on each level, and the algorithm terminates if it reaches a leaf of the plan hierarchy. In case it is not possible to find a valid task assignment at a certain level, the algorithm tracks back its search and evaluates the next best assignment on the level before.

As the domain-dependent summands of the plans utility functions often consider sensor values that differ between the agents of the team, it can happen that some agents consistently calculate a different solution for the same task allocation problem. In this case, the Conflict Handling module detects the conflict, identifies the involved agents and, e. g., overwrites the different solutions with the solution of the agent with the lowest ID.

3.5 Discussion

The ALICA Framework is the base for this thesis. This section discusses the advantages and disadvantages of ALICA and summarises the parts that need to be improved for this thesis.

3.5.1 Advantages

There are non-functional and functional reasons for choosing the ALICA Framework as the base for this thesis. One of many non-functional reasons is that the ALICA Framework is open source and published under the BSD License. It allows anyone to use it, change it, and republish it so that anyone can use the results of this thesis. Other frameworks solely created for academic purposes are often not licensed and therefore will not contribute to any other means than serving as academic proof of concept. The ALICA Framework was developed for the domain of robotic soccer[4] and is therefore geared towards a highly dynamic environment, but it was also applied in research projects and international competitions for extraterrestrial multi-agent exploration scenarios [51, 54]. These two different

[4] The team Carpe Noctem Cassel participated at international RoboCup competitions from 2006 to 2018, utilising the ALICA Framwork since 2009.

application scenarios, although similar, already indicate a certain amount of domain independence. Furthermore, the ALICA Framework is continuously subject, and tool for innovative research [30, 18, 48] and is therefore under active development and well tested.

The architecture of the runtime engine and the fundamental design principles of the ALICA Framework provide functional arguments for utilising ALICA for this thesis. The cooperation between ALICA agents, for example, is working under highly degraded network conditions [83], because of its low network traffic and optimistic assumptions about the progress of teammates in the plan structure. Furthermore, the ALICA runtime engine operates fully distributed, by running an independent engine instance on each agent and avoiding any single point of failure.

The ALICA Framework further follows a locality principle, which is the reason why ALICA is scaling up to about 100 agents in a local network. The locality principle states that the utility and the constraints of a plan, as well as the truth value of conditions, must depend only on the local branch of the plan tree. A precondition of a plan, for example, never references the execution state of a sibling in the plan tree.

3.5.2 Disadvantages

The following disadvantages of the original ALICA Framework are either general disadvantages or specific for the application domain of domestic service robots. A more general disadvantage of the ALICA Framework is the fact that it was implemented in Mono, the GNU/Linux' version of C#. The garbage collector of Mono freezes all threads of a process when cleaning up memory and thereby introduces significant peaks in runtime. Applications with hard real-time requirements, for example,therefore cannot be implemented in Mono. Another minor problem with Mono is that most software in the robotic research community is written in C++. As ALICA is generating method stubs in the programming language, that the engine is written in, a wrapper written in Mono was necessary to make use of any software not written in Mono. Especially in robotic applications, it is an advantage to implement in a programming language that qualifies itself to be used in embedded programming, for example, due to the ease of integrating sensors and actuators. Also, a relatively informal survey among roughly 20 members of the European RoboCup Middle Size League in 2014 clearly stated, that other RoboCup teams would like to use the ALICA Framework if it was written in C++ and therefore supports the thesis that Mono was the wrong choice for implementing a framework like ALICA.

Although its domain independence is listed as an advantage of the ALICA Framework in Section 3.5.1, two critical parts of the original framework limited its domain independence. On the one hand, its communication module was based on the ROS middleware (see Section 9.1.1), and its timing mechanisms could only use the ROS clock. On the other hand, the CSP solver (see Section 3.2.5) was the only solver that could assign values to

ALICA variables.

The disadvantages of utilising the ROS middleware itself are explained in Section 9.1.1, but being hardwired to a single communication middleware also limited the flexibility of the original ALICA Framework in general. Furthermore, through the dependency on the ROS middleware and clock the original ALICA Framework was limited to hardware that is supported by ROS, i. e. amd64, arm32, and arm64 processor architectures.

The CSP solver as described in Section 3.2.5 is made for the domain of continuous non-linear CSPs with variables ranging over the set of natural numbers $\mathbb{R}$. Therefore, application domains that require cognitive capabilities like symbolic reasoning and planning (see Section 2.2.4) cannot rely on this solver alone. Unfortunately, the CSP solver was hardcoded into the original ALICA Framework and therefore the only solver that could be used.

Another disadvantage of the original ALICA Framework concerns the original Plan Designer, as described in Section 3.3. Apart from several bugs, that had to be fixed for being comfortable to use; it depended on several large frameworks that partially had reached their end of life. Further, although the GEF Framework follows the model-view-control pattern, the original Plan Designer did not. On the contrary, its software design forced the developer to make changes in model classes that are autogenerated by the EMF Framework. Whenever the model was changed, these changes have to be repeated manually after the autogeneration step. In summary, the maintenance effort for the Plan Designer was so high that it was less effort to reimplement it.

Finally, the complexity of the plan structure of an ALICA program is high. Verifying an ALICA program, for example, requires an expressiveness of the language used to describe the verification model of the ALICA program, that excludes simple formalisms like Boolean algebra and even descriptions logics like $\mathcal{SROIQ}$ [63, 19].

3.5.3 Improvements for this Thesis

Summarising the disadvantages of the ALICA Framework given in Section 3.5.2, three significant improvements need to be made for this thesis.

1. The runtime engine and the Plan Designer need to be rewritten.
2. The runtime engine need, to be independent of the used communication middleware.
3. The framework needs the ability to use arbitrary problem solvers suitable for the application domain in question.

Rewriting the Plan Designer and the runtime engine of the ALICA Framework might be cumbersome, but implementing the second and third improvement already imply extensive changes in the framework. In order to make the framework compatible with arbitrary problem solvers, the runtime engine, as well as the Plan Designer, need to be adapted. The Plan Designer generates method stubs for describing the CSPs of the original CSP solver. In order to make this code generation process adaptable to any other problem solver, the Plan Designer needs a plug-in interface. Such a plug-in needs to provide model classes that describe the problem description for the corresponding solver, and it needs to provide templates for the generation of the method stubs, callable by the runtime engine. In order to work with arbitrary solvers, the runtime engine must be able to provide arbitrary problem descriptions to the corresponding solvers and handle the produced solutions. As described in Section 3.2.3, CSPs can span the hierarchy of ALICA programs, which makes it necessary for the engine to compose the complete CSP by traversing the hierarchy before presenting it to the CSP solver. Especially this problem composition mechanism needs to be improved to work with arbitrary problem solvers.

In order to make the runtime engine middleware-independent, all places where the engine sends messages need to make use of a generic communication interface. A wrapper could then implement this interface for the corresponding middleware. For the independence of the timing mechanism, a clock interface is necessary, following the same approach as for the generic communication interface.

How the aforementioned improvements and further adaptations of the ALICA Framework are implemented is described in Chapter 6.

Knowledge Representation and Reasoning 4

The following sections will define basic terms from the research area of knowledge representation and reasoning (KRR) as they are understood in this thesis. Starting with the concepts of knowledge and reasoning, classical reasoning problems provide arguments for utilising answer set programming as basic reasoning formalism to reason about symbolic knowledge. With this, the focus is on requirement R3 and R4 from Section 1.2: Handling dynamic environments and facilitating human interaction.

4.1 Knowledge

There are four concepts distinguished in literature and in language in general: data, information, knowledge, and wisdom [142, 105]. It is commonly accepted that they are of increasing abstraction in the mentioned order. Data, in terms of this thesis, is created by sensing through sensors or receiving messages. In order to transit from data to information, data needs to be analysed and given some meaning. 'Therefore, the difference between data and information is functional, not structural, ...' [142]. In the process of transforming data into information, the data can be condensed and therefore, the resulting information is often statistical.

Knowledge, compared to data and information, is much more elusive, and in classical epistemology, knowledge can be interpreted in three ways [105]. It is possible to know your neighbour (acquaintance), to know that his car is broken (propositional), and to know how to repair it (procedural).

The interpretation of knowledge in Figure 4.1 is that of procedural knowledge – when your understanding is at the level of knowledge, you are able to create or do something. Within the context of this work, knowledge is either interpreted as procedural or propositional knowledge, but not as knowledge by acquaintance. The concept of wisdom is very abstract and, although it is often discussed in close relation to the other three concepts, it is not relevant in the context of this thesis.

4.1.1 Gaining Knowledge

After roughly describing what knowledge is in comparison to data and information, the question is how domestic service robots are able to gain knowledge. The transition from data to information is made when some meaning is added to data. The number 3, for ex-

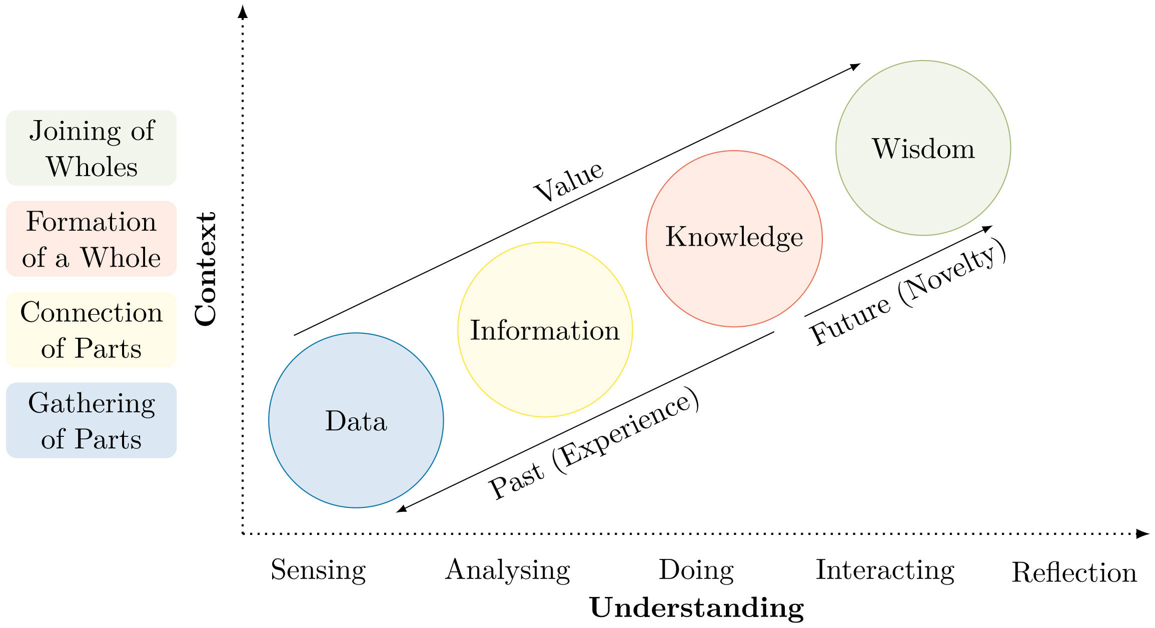


Figure 4.1: Relation between Data, Information, Knowledge, and Wisdom

ample, is just data, but interpreting it as the X coordinate of a robot on a map, transforms it into information. As said before, knowledge is more abstract than data and information, so knowledge can be created through further processing and abstracting information. There are many different approaches to realise such a process. One category of approaches is called machine learning algorithms. Machine learning algorithms like artificial neural networks, for example, create models that generalise from given data and provide suitable responses to data that have not been given to the system before. Another category is the family of reasoning algorithms. They create conclusions drawn by rules implemented in their algorithms. Ignoring the difference between models and conclusions, developers from both categories often denote the outcome of their algorithms as novel knowledge. While machine learning algorithms are often designed to require a minimal amount of a priori knowledge in order to create novel knowledge during their training phase [17, 38], the design of reasoning algorithms allows systems to be taught at runtime [115]. Another difference is that the training phase is often very cumbersome, as it takes many training data in order to create appropriate models, while the rules or facts that are told to the reasoning systems just need to be integrated into existing knowledge via a single reasoning step. The choice between machine learning and logical reasoning, in order to endow service robots with the capability to generate novel knowledge, is the choice between long learning periods and the tedious work of teaching a reasoning system everything it needs to know.

Regarding the process of transforming information into knowledge, if at all, only machine learning algorithms create knowledge without a priori existing knowledge. However, from the perspective of a service robot, it makes no difference, because the service robot is gaining knowledge through reasoning and machine learning either way. Albeit, crucial for the application of domestic service robots is that this process can happen at runtime because it is impossible for the developer of a service robot, to know everything that it needs to know and program this into the service robot. Furthermore, the process at runtime must finish in a reasonable amount of time, because otherwise the knowledge about the environment, for example, is getting invalid through changes in the environment. Due to these requirements, most of the existing machine learning algorithms cannot be applied in a straight forward manner.

Instead, the approach followed in this work is to utilise reasoning algorithms, because new knowledge can be integrated fast and reasoning systems are designed to be taught at runtime [115]. A fortunate circumstance of the domain of domestic service robots is that an excellent source of knowledge is always available – humans. The humans, living in a household, know best what and how service robots should do something for them, and they also know the items in the household and where they belong. Therefore, it is crucial for our approach to make it easy for humans to teach the service robots everything they need to know.

4.1.2 Representing Knowledge

When the service robots get taught knowledge, they need to remember it and therefore, need to represent it in some form in their memory. Revising the Chinese Room thought experiment, the process of storing knowledge could be envisioned as extending the rule-book with additional rules through a special kind of input from outside the room. By this process of transforming knowledge into some representation, it gets symbolised, meaning the symbols stored in memory represent that knowledge, that without this representation would be tacit knowledge, inaccessible for the robots. Zeleny [112] argues that this process transforms knowledge back into information because, in his opinion, all knowledge is tacit. Following the argumentation, that representing knowledge renders knowledge to information, a subsequent question whether this information has any intrinsic semantics or whether it is just data, leads back to the symbol grounding problem, mentioned in Section 2.1.

In the context of the symbol grounding problem, Taddeo and Floridi [110] formulate the Zero Semantical Commitment Condition (z-condition) as a pre-requisite for any approach to solve the symbol grounding problem. The z-condition includes three requirements for an agent to solve the symbol grounding problem: No semantics may already be part of an agent, no semantics are allowed to be added from outside the agent, and the agent must have the capabilities to ground its symbols, for example, through computation, actuators, or sensors. As stated in Section 2.1, our approach does not intend to solve the symbol grounding problem, but regarding the semantics of the symbols, it relies on the semantics persisting in the brain of the humans that interact with the service robots and is thereby violating the second requirement of the z-condition.

Another violation of the z-condition happens when a developer of an agent chooses the form of knowledge representation itself, instead of letting the agent choose itself. It is a violation of the z-condition because any form of knowledge representation always includes some form of semantic commitment. Davis, Shrobe and Szolovits [140] denote this ontological commitment, which is one of the five roles of a knowledge representation, that needs to be considered when choosing or evaluating one. According to Davis, Shrobe and Szolovits [140], the adequacy for fulfilling the different roles gives a knowledge representation its own profile. The five roles are [19]:

1 **Surrogate** Each knowledge representation surrogates things of the real world. In order to measure the adequacy of a knowledge representation as a surrogate, questions related to the surrogated real thing have to be answered. What should be surrogated? What aspects and properties of the real thing are represented and which are omitted? How exact is the representation?

2 **Ontological Commitment** It is essential to control and to know which aspects of the real thing are omitted by the corresponding knowledge representation, as it is impossible to represent the real thing perfectly. Omitting aspects of the real thing guides humans and robots to focus on aspects which are considered to be relevant. The set of aspects represents a commitment to a certain point of view and is denoted as the ontological commitment of the corresponding knowledge representation.

3 **Fragmentary Theory of Intelligent Reasoning** In order to define the knowledge representation's role as the fragmentary theory of intelligent reasoning, three main questions should be answered. What is intelligent reasoning? What inferences are sanctioned? Which inferences are recommended? The answer to the first question cannot be given easily. A rephrased version of the question facilitates to answer: Which conclusions should be drawn?

4 **Medium for Efficient Computation** Knowledge is useless if it is impossible to reason about it. Therefore, the representation should facilitate a computational efficient reasoning process. The representation's structure could be especially appropriate to be processed by reasoning mechanisms or the recommendation of some inferences aim to favour conclusions, which are drawn more efficiently than others.

5 **Medium of Human Expression** A knowledge representation defines the language we use in order to communicate with machines and each other. It should be expressive enough to say everything we want, and it should be easy enough to speak and understand without too much effort.

Some of the roles are contradicting each other. Efficient computational properties and exceptional expressivity, for example, are hard to combine. According to Levesque and Brachman [149], a general trade-off between these two goals exists. Nevertheless, Role 3 and 4 make it clear that representing knowledge is inextricably connected with reasoning about it and Role 5 aligns with Requirement 4 of Section 1.2 and further emphasises the need for a human accessible knowledge representation.

4.2 Reasoning

According to the five roles of knowledge representation from Davis, Shrobe and Szolovits [140], the way knowledge is represented influences the way it is possible to reason about it. In the domain of domestic service robots, the dynamic of the environment poses additional requirements and problems to the applied reasoning mechanism. The elements in the environment that the service robots need to know and reason about, are unknown at design time and change at runtime. The door to the kitchen, for example, can be open, closed, or locked and based on the capabilities of the service robot, it can deduce that things inside the kitchen are accessible for it, or not. When the state of the door changes, deduced propositions about the accessibility of things change to be invalid. Reasoning mechanisms or logics that are able to withdraw conclusions that, for example, are not valid anymore, are denoted as non-monotonic logics and necessary for domestic service robots.

In general reasoning, formalisms can deduce that propositions are true, false, or unknown. In a partially unknown environment, like a household, it is sometimes impossible to deduce a preposition to be true or false without making some assumptions. Therefore, two contrary assumptions exist that reasoning mechanisms can apply, the Closed World Assumption (CWA) and the Open World Assumption (OWA). Under the CWA, everything that is not known to be true is considered to be false. The state of being unknown does not exist under the CWA. Nevertheless, the OWA allows things to be unknown and does not conclude prepositions to be false, just because they cannot be proven. An example for the OWA from Russell and Norvig [47] asks the question, how many computer science courses exist when a board at the computer science department lists CS0815 and CS555. A typical answer from a reasoning formalism that follows the OWA would be 1 to infinite courses. On the one hand, under the OWA, the courses listed on the board might not be the only courses that exist, so an upper limit cannot be given. On the other hand, CS0815 and CS555 could be the same course denoted by different names. Formalisms that apply the CWA often also apply the Unique Name Assumption (UNA) and therefore would answer two. Furthermore, the existence of other courses cannot be proven, so it is assumed that they do not exist. This mechanism of inferring something to be false, when it cannot be proven to be true, is denoted as Negation-as-Failure (NAF) and is a common way to apply the CWA.

Additionally, to utilising a non-monotonic logic and being able to apply different assumptions while reasoning about knowledge in a dynamic domain, it is also essential to formalise changes in the environment through actions. Actions have preconditions that should hold before actions can be executed and have the intended effect on the environment. Qualifying these preconditions is denoted as the qualification problem. The example of crossing a river with a boat, from McCarthy [153], illustrates the problem. In order to cross the river with a boat, the boat must not leak, must have two oars, the oars must suit each other, the rudder must not be broken, and so on. The list of things that need to be in place, fixed, and work as intended is endless, and so are the corresponding axioms. As a result, specifying all the necessary preconditions is impossible. McCarthy argues that humans usually conjecture that everything is working as intended unless it is proven otherwise. Therefore, they only consider a limited set of preconditions before crossing a river on a boat. The solution McCarthy proposes to realise the conjectural reasoning is denoted as circumscription and relies on non-monotonic reasoning by augmenting first-order logic. The Yale Shooting scenario [147] demonstrated that his proposed solution does not work in some cases. Nevertheless, some existing solutions still utilise circumscription in another way, and most existing solutions depend on non-monotonic reasoning, too.

The frame problem [151, 159] and the ramification problem [144] are similar to the qualification problem. In the context of logical reasoning, the frame problem describes the problem of specifying the things that actions do change under the assumption that everything else remains the same. The latter assumption is also denoted as the commonsense law of inertia – everything remains unchanged unless stated otherwise. The ramification problem describes the problem of determining the indirect effects of an action. Independent of the problem, most solutions require non-monotonic reasoning because, in order to imitate human reasoning, it is necessary to include some default reasoning in order to jump to conclusions based on some assumption. Exactly this assumption must be revocable in case of an exception; thus, non-monotonic reasoning is necessary.

Within this and the former section, we collected several properties and capabilities that a reasoning formalism suitable for the domain of service robots needs to fulfil. Therefore, we review some of the existing reasoning formalisms in the next section with regard to these requirements.

4.2.1 Reasoning Formalisms

In general, there is no single reasoning formalism that suits all domains and scenarios. Even the choice for a symbolic reasoning formalism restricts the applicability of this work to specific domains. Therefore, we described in Section 3.5.3, why the ALICA Framework needs to be extended with a general solver interface that allows using any formalism that suits the domain in question. Nevertheless, in the following paragraphs, we review standard symbolic reasoning formalisms with regard to the aforementioned requirements for representing, gaining, and reasoning about knowledge.

The most common classical logics are the propositional logic (PL) and the first-order logic (FOL). The expressiveness of the propositional logic is somehwat restricted (Role 5, Section 4.1.2), but fast satisfiability solvers (SAT solvers) for problems formulated in that logic exist (Role 4, Section 4.1.2). The annual SAT competition[1] contributed to the improvement of existing and development of new algorithms for SAT solvers. Especially the DPLL algorithm [162] that many SAT solvers nowadays utilise in some form initiated a jump in the performance of SAT solving. First-order theorem provers implement algorithms for proving theorems formulated in first-order logic like SAT solvers do for theorems formulated in propositional logic. In contrast to propositional logic, first-order logic is much more expressive, but at the same time undecidable. Kurt Gödel proved in his paper *On Formally Undecidable Propositions of Principia Mathematica and Related Systems* [165] that any sufficiently expressive system allows formulating statements that cannot be proven by the system itself. This prove alone, does not rule out the application of first-order theorem provers in the domain of domestic service robots, because the statements needed to be formulated for that domain could all belong to the provable part of possible statements. The fact that the propositional and the first-order logic are both monotonic logics makes them impractical for the dynamic service robot domain.

Although much more expressive than propositional logic and nevertheless decidable, description logics (DL), the formal underpinning of the semantic web languages, are also monotonic. Furthermore, description logics do not support the closed world and unique name assumptions. Compared to other rule-based reasoning formalisms, several further restrictions in the expressiveness of description logics exist [108, 19]. Schreiber [100] states that ontologies emphasise the first, second, and fifth role in Section 4.1.2. Additionally, the well-defined semantics of the description logics provide precise answers to the questions concerning the fourth role. Compared to other knowledge representation and reasoning techniques, only the computational efficiency of description logic reasoning seems to be neglected.

The history of research in non-monotonic reasoning started in the seventies, among others, with the publication from Doyle about truth maintenance systems [157]. Since then, several different approaches to non-monotonic reasoning have been created. The most common among them are believe revision, autoepistemic logics, default logic and answer set programming.

Truth maintenance systems (TMS) and believe revision (BR) are general approaches that address the problem of managing the consistency of a knowledge base full of inferred

[1] SAT Competition: http://satcompetition.org/ [last accessed on October 25th, 2020]

information. While truth TMS distinguish between base information and derived information, in the most simple case of a believe revision systems, all information are equal. Both approaches expect to be confronted with new information that is inconsistent with the current knowledge base and provide operations to integrate the new information while maintaining consistency, often by simply dropping conflicting information. The research following these approaches is focused on finding better operations and rules for maintaining knowledge, and all findings have their own semantics that is different from the others. As a result, standardisation is missing, and real-world applications of such systems are rare. Furthermore, an improvement of the computational efficiency of the reasoners, as it was done for SAT solvers through their corresponding SAT challenges, did not happen. Current research is trying to implement belief revision and TMS on the base of answer set programming [25–27, 69, 97].

Compared to belief revision and TM systems, autoepistemic logics (AL) and the default logic (DefL) of Raymond Reiter [154] have more thorough semantic foundations. Default logic is, informally speaking, an extension of some classical logic with *defaults* that allow deducing some information without proof.

The most common example of applying default reasoning is that of the flying bird Tweety. The first-order default $\frac{Bird(X):Flies(X)}{Flies(X)}$ states that whenever something is a Bird $Bird(X)$ and the justification that it can fly : $Flies(X)$ is consistent with the knowledge base, it is deduced that it can fly indeed $Flies(X)$. Considering the knowledge base $Bird(Tweety)$, it is deduced that Tweety can fly. Adding $Penguin(Tweety)$ and $Penguin(X) \rightarrow \neg Flies(X)$ to the knowledge base will forbid the application of the aforementioned default, and it is assumed that Tweety cannot fly. In contrast to pure first-order logic, this also allows default logic to apply the closed world assumption. The propositional default $\frac{:\neg F}{\neg F}$ means that everything that is not justified by the knowledge base is assumed to be false. A disadvantage of the default logic is that the defaults are handled separated from the axioms of the underlying logic. Furthermore, the application order of the defaults is not deterministic, and influences which of two conflicting defaults can be applied.

The original autoepistemic logic as defined by Moore [148] is a propositional logic extended by the modal operator $\mathcal{B}$. As being a modal logic, the semantics of this autoepistemic logic stems from the semantics of possible worlds [161]. B stands for belief, and therefore, BX means that it is believed that X holds. Through the believe operator of autoepistemic logics, it is possible to formulate axioms similar to defaults in default logic. In fact, some variants with weak extensions are identical to autoepistemic logic [116]. The relation of default logic and autoepistemic logic to answer set programming is that of a predecessor. According to Bidoit and Froidevaux [145], defaults can be expressed as answer set programming rules and the semantics of negation as failure in answer set programming is the same as that of the negated believe operator in autoepistemic logic [146]. Therefore, it is valid to say that answer set programming is, among others, the result of several decades of research in non-monotonic reasoning and is therefore explained in detail in the next section.

Formalism	Non-Monotonic	Decidable	Expressiveness	Intelligible
FOL	✗	✗	++	0
PL	✗	✓	− −	+
DL	✗	∅	0	++
BR	✓	∅	∅	+
TMS	✓	∅	∅	+
AL	✓	∅	∅	+
DefL	✓	∅	∅	+
ASP	✓	✓	++	++

Table 4.1: Implementation Independent Properties of Reasoning Formalisms (✗: no, ✓: yes, ∅: depends, Likert: ++,+,0,–,– –)

Table 4.1 summarises properties of the discussed reasoning formalisms that are independent of the implementation of a corresponding reasoner. Whether a formalism is non-monotonic is essential for dealing with dynamic knowledge, being narration tolerant, and providing to the Frame, Qualification, and Ramification Problem. The decidability of a formalism is a hint for the potential efficiency of a solving algorithm. Further, the undecidability of a formalism is adverse to its expressiveness, because the full expressiveness cannot be used when it is required that the solving algorithm always terminates. The expressiveness of a formalism measures what can be expressed with axioms of that formalism and is therefore relevant for roles 1, 3, and 5 of Section 4.1.2. Whether a formalism is intelligible by humans is crucial for Role 5 of Section 4.1.2 and one of the main requirements (R4) in Section 1.2.

The decidability and expressiveness in case of belief revision (BR), truth maintenance systems (TMS), autoepistemic logics (AL), and default logics (DefL), depend on the underlying logic and its semantic. Default logic, for example, was first developed as an extension to first-order logic (FOL) and therefore, would offer a greater expressiveness than FOL, but being undecidable as well. The decidability of description logics are studied in-depth and as a result, it is very well known which 'piece of expressiveness' would turn the corresponding logic to be undecidable[2].

Table 4.2 summarises properties of the discussed reasoning formalisms that dependent on the implementation of corresponding reasoners. The Implementations category follows the Likert scale, and judges, how many ready to use reasoners are available. In the case of belief revision (BR) and truth maintenance systems (TMS), a few implementations are available, but they mostly differ with regard to their underlying logic. Implementa-

[2] Evgeny Zolin created a complexity navigator for description logics: http://www.cs.man.ac.uk/ ezolin/dl/ [last accessed on January 16th, 2020]

Formalism	Implementations	Efficiency	Multi-Shot
FOL	++	+	✗
PL	++	++	✗
DL	++	–	✗
BR	0	0	∅
TMS	0	0	∅
AL	-	0	∅
DefL	-	0	∅
ASP	++	++	✓

Table 4.2: Implementation Dependent Properties of Reasoning Formalisms (✗: no, ✓: yes, ∅: depends, Likert: ++,+,0,–,– –)

tions for autoepistemic logics and default logics are very rare and mostly made available through other subsuming reasoning systems, like an ASP reasoner. Benchmark problems and competitions improve the efficiency of reasoners in case of FOL, PL, DL, and ASP. Tableau algorithms [121] that are used in most description logic reasoners are relatively slow compared to the DPLL algorithm [162] used in SAT-based reasoners like it is the case for most PL and ASP reasoners.

Multi-shot solving is a feature of reasoners that, generally speaking, allows to keep data structures and inferred information over several queries partially. Although monotonic logics are not suitable for multi-shot solving, in 2017 the SAT competition[3] offered a track for incremental SAT solvers. However, only three solvers participated, and the track did not continue in the following years. In the case of the non-monotonic formalisms, it depends on the underlying logic and corresponding implementation. Nevertheless, to the best of our knowledge, currently, all available non-monotonic multi-shot reasoners are based on state-of-the-art ASP reasoners like Clingo [50].

4.3 Answer Set Programming

In Section 4.2.1, several classical and non-monotonic reasoning formalisms are reviewed with regard to their suitability for representing knowledge in dynamic domains. Although answer set programming (ASP) suits the best to the given criteria, we want to emphasise that there will never be a single solver that fits all requirements perfectly. Therefore, ASP is only one of many possible symbolic reasoning formalisms and the general solver interface, developed as part of this thesis (see Section 6.2), allows to choose other formalisms, too.

Answer set programming is, as the name implies, not only a reasoning formalism that allows deducing new information from given information. It is also a logical declarative programming approach. Programming languages like Java or C++ are imperative and

[3] SAT Competition http://sat2018.forsyte.tuwien.ac.at/ [last accessed on January 16th, 2020]

algorithms written in these languages are descriptions of how to get a result by processing an input. In declarative programming languages like ASP, possible results are described, and the solving algorithm is able to generate them by processing the descriptions together with some input. It does not matter in which order the required properties of the results are described.

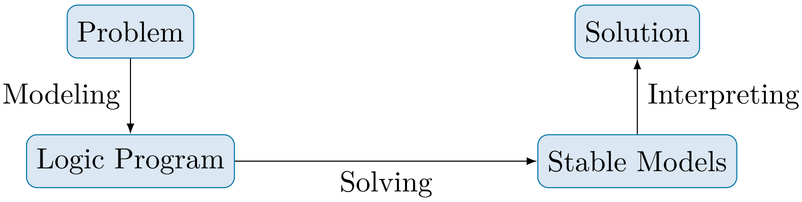


Figure 4.2: Declarative Programming Procedure Using the Example of ASP [42]

Figure 4.2 shows the procedure of the declarative programming approach using the example of ASP. Starting from a given problem, it is necessary to model the problem in a logic program written in ASP syntax. This logic program is presented to the ASP solver, which produces results, in ASP denoted as stable models. Finally, these models often need to be interpreted in order to get a real solution.

The language constructs we describe in Section 4.3.1 are that of the input language of the ASP solver Clingo [59], which is used in this thesis. Therefore, some features will only work with Clingo and no other ASP solver. Especially the ability to define *Externals* is heavily used within this thesis and only available within Clingo.

4.3.1 Language Constructs

Generally speaking, logic programs in ASP are comprised of a set of rules. Rules usually consist of a head, a body, and end with a period. However, the simplest form of a rule does not include a body and is denoted as fact. In Listing 4.1, some examples of facts are given. Facts only consist of a single ground literal. A literal is a positive or negative atom, i. e. preceded by a classic negation. Different forms of negation are explained in the following paragraphs. An atom, however, is a predicate symbol with an optional list of arguments in parentheses. The arguments are called terms, which in general are either numeric constants, symbolic constants, variables, or composite terms. Their first character can distinguish the different types of terms: numeric constants only contain numbers, symbolic constants start with a lower case letter, variables begin with an upper case letter. Composite or complex terms look like they include predicates with arguments in parentheses itself, but in this case, the predicates are not atoms but denoted as symbolic functions.

```
serviceRobot(leonardo).
cup(X).
```

```
3 time(123).
4 holds(color(cup,blue),time(123)).
```

Listing 4.1: Atoms

In Listing 4.1 all rules only contain a single positive literal and terms are green. Rule 2 is not a fact, because the atom contains the variable X. All other rules only contain symbolic constants (Rule 1), numeric constants (Rule 3), or composite terms (Rule 4). It is important to note that Rule 3 is a literal named `time`, while the second argument of Rule 4 is a composite term with the symbolic function named `time` and are therefore entirely different entities. A literal is ground if it does not contain variables as arguments; thus, only Rule 1, 2, and 4 are facts.

```
1 serviceRobot(leonardo) :- #true.
2 serviceRobot(leonardo).
```

Listing 4.2: Simplification of Facts

In general, rules consist of two parts, the head and the body which are separated by an implication (`:-`). The direction of the implication is from the body on the right-hand side to the head of the rule on the left-hand side. In the case of facts, the body is simply true (`#true`) and therefore, the body, including the implication, is dropped for simplicity. Therefore, the two rules in Listing 4.2 are semantically the same. The opposite of the boolean constant `#true` is `#false`.

```
1 room(r1).
2 room(r2).
3 oven(aeroFryerDelux).
4 contains(r2,aeroFryerDelux).
5 kitchen(X) :- room(X), oven(Y), contains(X,Y).
```

Listing 4.3: Normal Rules

Usually, rules can be understood as definitions. In Listing 4.3, for example, Rule 5 defines the concept of a kitchen in terms of being a room that contains an oven. The body of this rule contains a comma-separated list of literals which forms a conjunction of literals that must hold, in order to deduce the head of the rule. During the solving process, the ASP solver will replace the variables of the literals in Rule 5 with the terms of ground instances of the same literals. As a result, it is deduced that `r2` is a kitchen.

```
1 :- kitchen(X), bathroom(X).
```

Listing 4.4: Integrity Constraints

Rule 1 from Listing 4.4 is a rule without head, denoted as integrity constraint. Similar to facts, integrity constraints are semantically the same as rules that only contains the boolean constant #false in their head. Whenever the boolean constant #false is deduced, the ASP solver does not return a result for the corresponding logic program. Therefore, Rule 1 means that nothing can be a kitchen and a bathroom at the same time.

```
1 room(r1).
2 window(w1).
3 contains(r1,w1).
4 is(X,dark) :- room(X), not contains(X,Y), window(Y).
```

Listing 4.5: Default Negation

Rule 4 in Listing 4.5 is similar to Rule 5 in Listing 4.3, but the `contains(X,Y)` literal is preceded by `not`. The keyword `not` is used to formulate a default negation and follows the semantics of Negation-as-Failure (see Section 4.2). Therefore, Rule 4 means that if it can not be proven that a room contains a window, it is considered to be a dark room; thus, `r2` is a dark room, and `r1` is not. Through the extension by default negation, the body of a rule is not anymore a conjunction of literals. Instead, it is a conjunction of literals that can either be a normal (`room(X)`) or be an extended literal (`not contains(X,Y)`).

```
1 room(r1).
2 window(w1).
3 -is(r1,dark).
4 is(X,dark) :- room(X), not contains(X,Y), window(Y).
```

Listing 4.6: Classic Negation

Additional to the default negation, ASP also offers the possibility to use classic negation, as shown in Rule 3 of Listing 4.6. In contrast to Listing 4.5, Room r1 does not contain a window anymore and is therefore deduced to be dark. At the same time, it is explicitly stated that Room r1 is not a dark room by the classic negation. As a result, the logic program in Listing 4.6 is unsatisfiable. The reason for that is, although `-is(r1,dark)` and `is(r1,dark)` are handled as completely independent literals, in ASP an integrity constraint for each pair of positive and negative literal is implicitly added. In this case it is `:- -is(r1,dark), is(r1,dark).`, which is not part of the logic program explicitly.

```
1 room(r1).
2 -is(r1,dark).
3 is(X,dark) :- room(X), not contains(X,Y): window(Y).
```

Listing 4.7: Conditional Literals

With regard to Listing 4.6, it may be questionable why Window w1 is mentioned at all when it has no relation to Room r1. Here, it is important to note that the literals in the conjunction of Rule 4 must all hold independent of each other. In other words, the semantics of the condition expressed by the body of Rule 4 is that there must be some room, some window, and there must be no proof that the room contains this window. Nevertheless, this other window must exist! The feature of conditional literals (see Rule 3 in Listing 4.7) allows avoiding this existential quantification of a window. Here the `contains` and the `window` literal form a single literal. Conditional literals are, informally speaking, rules by themself. The literal on the left side of the colon is the head, and the comma-separated list on the right side is the body. If a conditional literal is not the last literal in the body of a rule, it must be terminated by a semicolon, instead of a comma, in order to make the end of the condition distinguishable from the next atom in the body of the rule.

So far, we introduced most of the basic language construct of ASP, namely facts, rules, integrity constraints, classic negation, default negation, and conditional literals. Clingo supports a lot of built-in arithmetic integer expressions like plus, minus, multiplication, division, modulo, exponentiation, as well as bitwise and, or, exclusive or, invert. Evaluated arithmetic and lexicographical expressions can further be compared with the following built-in comparison predicates: =, !=, <, <=, >, >=. The language constructs explained in the following paragraphs, make use of the aforementioned language constructs and built-in predicates. Due to their complexity, we only explain their basic usage with simple examples. For a full-fledged explanation, consider the following exhaustive references [8, 11].

```
1 room(r1;r2;r3;r4).
2 window(w1;w2;w3).
3 contains(r2,w3).
4 contains(r3,w2).
5 contains(r3,w1).
6 is(X,dark) :- room(X), not contains(X,Y): window(Y).
7 darkApartment :- 1 < #count { X : is(X,dark), room(X) }.
```

Listing 4.8: Aggregates

Listing 4.8 includes an aggregate in Rule 7 and follows the running example of classifying rooms. Within the first five rules, the logic program describes an apartment. The aggregate rule states that the apartment is dark if there is more than one dark room. This instance of an apartment is deduced to be a dark apartment because Room `r1` and `r4` do not contain any window. The form of the aggregate in Rule 7 is relatively simple. At first, there is a simple conditional literal within curly braces including two conditions. Further conditional literals could be added, separated by semicolons. The curly braces imply a set semantic, i. e. same instantiations of `X` count only once. The aggregate in front of the set evaluates the set into an integer that can be compared with a lower and upper bound. In this case, it is the `#count` aggregate, and the upper bound is dropped. Alternative aggregates are `#min`, `#max`, `#sum`, and `#sum+`, which all require an extra weight for each conditional literal in order to be applicable.

```
 1 {apartment(a1); apartment(a2); apartment(a3)} = 1.
 2 room(r1;r2;r3;r4;r5;r6;r7).
 3
 4 contains(a1,r1).
 5 contains(a1,r2).
 6 contains(a2,r3).
 7 contains(a2,r4).
 8 contains(a2,r5).
 9 contains(a3,r6).
10 contains(a3,r7).
11
12 is(r1,dark).
13 is(r3,dark).
14 is(r5,dark).
15 is(r7,dark).
16
17 darkness(X,Z) :- #count{ Y : contains(X,Y), is(Y,dark) } = Z;
18 ↪ apartment(X).
19
20 #minimize{ Z@2, X : darkness(X,Z), apartment(X) }.
```

Listing 4.9: Optimisation

A special form of integrity constraints are optimisation expressions, also denoted as weak constraints. Like integrity constraints, weak constraints are demanded not to be broken, but if no solution that fulfils the constraint, the ASP solver does not conclude that the logic program is unsatisfiable. Instead, the solution that violates the given weak constraints the least is returned. The logic program in Listing 4.9 tries to find the apartment that has the fewest number of dark rooms. Three rules are necessary for this purpose. Rule 1 restricts the solution to include only one apartment. Rule 18 calculates the number of dark rooms per apartment and remembers them in `darkness(X,Z)` literals. Rule 20 minimises the number of dark rooms per chosen apartment. The syntax of the minimise expression is, as the aggregate literals, based on a set defined through conditional literals. The term tuple `Z@2,X` has three parts. `Z` is the value that should be minimised, `@2` states that the value should be minimised with a priority of 2, and `X` makes the term tuple unique in case two apartments have the same number of dark rooms. It is also possible to maximise the value (`#maximize`). Each result of the logic program is rated by a cost function. The cost function summarises the prioritised contributions of each fulfilled weak constraint. In the case of the apartment example, apartment `a1` contributes 1, `a2` contributes 2, and `a3` contributes 1. Therefore, apartments `a1` and `a3` are two equal optimum solutions.

4.3.2 Stable Model Semantics

We explained most of the language constructs supported by the ASP solver Clingo [59] in the last section and explained rather informal what the results of the shown logic programs are. In this section, we explain the process of calculating the results, denoted as answer sets, without going into details about the algorithms. More details about the algorithms implemented in different ASP solvers are discussed in [73, Chapter 4 and 6], [124, Chapter 3 and 4], and [61, Chapter 7].

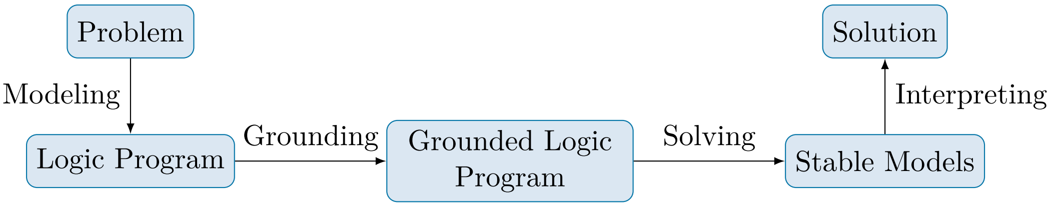


Figure 4.3: Two-Step Process of Determining Stable Models in ASP [42]

The workflow shown in Figure 4.3 is an extended version of the declarative programming workflow in Figure 4.2. Instead of a single solving step, the logic program is ground first, and the grounded logic program is solved afterwards. This extra step is typical for state-of-the-art ASP solvers and essential for the performance of the solving process.

We mentioned that facts are grounded literals, i. e. the arguments of the atoms are terms without variables. A ground logic program is a logic program whose rules are free of variables, too. During the grounding step, each variable is instantiated with all grounded terms of the logic program, creating an exponential blow-up of the grounded logic program, which is free of any variable. Therefore, an intelligent grounding step drops as many irrelevant rules and body parts without altering the possible answer sets of the logic program. As a result, the grounding step drops information that is not relevant for the answer sets of the current logic program, but which could be relevant if further rules extend the logic program. The grounding step makes ASP less narration tolerant, which can be partially compensated by externals (see Section 4.3.3). Before we explain externals, we sketch the process of creating answer sets and explain the peculiarities of the stable model semantics.

An answer set, also denoted as a stable model of an ASP logic program, is a set of ground literals, that follows the stable model semantics. According to [61], the stable model semantics can be informally characterised by the following three principles:

1. *Satisfy the rules of the logic program. In other words, believe in the head of a rule if you believe in its body.*
2. *Do not believe in contradictions.*

3. *Adhere to the "Rationality Principle" that says, "Believe nothing, you are not forced to believe."*

The formal definition of the stable model semantics, as given by [61], includes three parts: Satisfiability, answer sets for logic programs without default negation, and answer sets for general logic programs.

Satisfiability: *A set S of ground literals satisfies*

1. *l if $l \in S$;*
2. *not l if $l \notin S$;*
3. *rule r if, whenever S satisfies r's body, it satisfies r's head.*

Answer Sets (Part I): *Let Π be a program not containing default negation (i. e., consisting of rules of the form*

$$l_0 \leftarrow l_1, ..., l_m$$

An answer set of Π is a consistent set S of ground literals such that

- *S satisfies the rules of Π and*
- *S is minimal (i. e., there is no proper subset of S that satisfies the rules of Π).*

Answer Sets (Part II): *Let Π be an arbitrary program and S be a set of ground literals. By Π^S we denote the reduct of program Π obtained by*

1. *removing all rules containing 'not l' if '$l \in S$';*
2. *removing all other premises containing 'not'.*

The first two informal principles are mapped straight forward to the three definitions. However, it is noteworthy that the rationality principle is expressed by the minimum requirement for answer sets. Answer sets can be interpreted as the belief state of an agent that has its knowledge encoded in the corresponding logic program. The minimality of answer sets makes the belief state of an agent more stable with regard to changes in its knowledge. Furthermore, answer sets do not contain literals that are not supported by rules of the logic program.

Finally, the three definitions do not describe any way of creating the mentioned set S of ground literals. Algorithms that can find such sets are described in the literature, mentioned at the beginning of this section.

4.3.3 Externals and Program Sections

The grounding step, which is done before solving the ground logic program afterwards, drops information encoded in the logic program if they are not relevant for potential answer sets of the current logic program. Especially, the creation of a reduct with regard to some set of ground literals, as described in the second part of the definition of answer sets, removes parts of rules or complete rules.

```
1 robot(leonardo).
2 mobile(X) :- robot(X), not -mobile(X).
3 ...
4 %Added later
5 -mobile(leonardo).
```

Listing 4.10: Non-Monotonic Reasoning with ASP

Listing 4.10 shows a simple example about the mobility of robots. In a domestic service robot domain, it is reasonable to assume that a robot is mobile, as long as it is not stated otherwise. This assumption is expressed in Rule 2 with the default negated literal `not -mobile(X)`. Rule 1 states that there is a robot called leonardo, and Rule 5 should be ignored, as it is added to the knowledge base at a later time point. If the first two rules in Listing 4.10 are grounded and solved, the only answer set is {`robot(leonardo)`, `mobile(leonardo)`} and the literal `not -mobile(X)` is removed (see Definition Answer Set Part II), as it cannot be fulfilled. At a later time point, it is recognised that `leonardo` is broken and the fact `-mobile(leonardo).` is added to the knowledge base. Again, now with all three rules, the logic program is grounded and solved. Unfortunately, the logic program is now unsatisfiable, because the information about the default negated literal is dropped from the body of Rule 2, and the semantics of the program states that every robot is mobile, but `leonardo` the robot is not mobile. As a result, `mobile(leonardo)` and `-mobile(leonardo)` are deduced, and therefore, no consistent answer set can be found.

At this point, reconsidering the non-monotonic reasoning capabilities of ASP is essential. Although ASP is based on decades of research in non-monotonic reasoning and includes benefits from findings in default logic and autoepistemic logic, it is the implemented algorithm that, although being state-of-the-art, limits the non-monotonic reasoning capabilities of ASP. A straight forward solution to the demonstrated problem, would be a complete reset of the solver and solving the three rules altogether, again. However, this way, every monotonic formalism mentioned in Section 4.2.1 would be able to handle non-monotonic reasoning, too, because the ASP solver would not take any advantage of ASP being a non-monotonic logic.

Another solution, provided by the ASP solver Clingo, is not based on standard language constructs of ASP. Instead, Clingo is to the best of our knowledge, the only solver available, that supports the features of *program sections* and *externals*.

```
#program robot(n).
robot(n).
mobile(n) :- robot(n), not -mobile(n).
```

```

#program room(n).
room(n).
is(n,dark) :- room(n), not contains(n,X) : window(X).
```

Listing 4.11: Program Sections

On the one hand, program sections are able to divide a logic program into different sections that can be grounded independently. On the other hand, can program sections be grounded several times with different parameters. Listing 4.11, for example, allows to ground knowledge about the concept of robots for different instances. The `n`, although adhering to the syntax of terms is a constant which is replaced by a given term when grounding the program section.

```
1 #program robot(n).
2 #external -mobile(n).
3 robot(n).
4 mobile(n) :- robot(n), not -mobile(n).
```

Listing 4.12: Externals

Externals, as shown in Rule 2 of Listing 4.12, allow protecting literals from simplifications happening during the grounding. The literal `-mobile(n)` in Rule 4, for example, is not dropped anymore, because it is declared as an external in Rule2. Instead, the truth value of an external literal can be set through the API of Clingo after the grounding of its corresponding program section. The default truth value of an external is `#false`, therefore the answer set of Listing 4.12, when grounded with `n` set to `leonardo`, is {`robot(leonardo)`, `mobile(leonardo)`}. After setting the truth value of `-mobile(leonardo)` to `#true`, the grounded logic program can be solved again, without an additional grounding step, and the answer set would be {`robot(leonardo)`, `-mobile(leonardo)`}.

4.3.4 Module Property

With the features of program sections and externals, it is possible to define independent portions of knowledge in the form of program sections and to set specific properties of the entities described by the sections after grounding. When using these features, several portions of knowledge, defining different concepts and their properties will be modelled. In the literature, portions of knowledge are denoted as modules, and the module property defines whether two such modules are compositional. Informally speaking, a violation of the module property will again induce problems like dropped default negated body literals.

The module property is defined several times in literature, but each time in a different context, because the module property is essential for all flavours of logic programs and different operations applied to these logic programs. The module property, for example, is essential in the context of combining logic programs [93], splitting logic programs [139], but it is also important in the context of *forgetting* in the sense that an agent should forget invalidated knowledge from its knowledge base [10]. However, the following definition of the module property, as published in [20], is based on the lecture materials and publications of Thorsten Schaub [99, 2], head of the Potassco Research Group, which is developing the ASP solver Clingo, utilised in this thesis.

A Module $\mathcal{P}$ is defined as a triple of sets (P, I, O). P is a ground program over the ground literals, denoted as ground(A), and both, I and O, are disjoint subsets of ground(A). Furthermore, all literals appearing in P are either part of I or O, and all rule heads are part of O. I and O are denoted as input and output, respectively. Given this definition of a module, two modules $\mathcal{P}$ and $\mathcal{Q}$ are compositional, meaning their join will not violate the Module Property if the following two conditions hold. The first condition is that the output sets of both modules are disjoint, meaning that they do not have any literal in common. The second condition relies on strongly connected components [73, 152]. A strongly connected component is a subset of a directed graph, in which every vertex is reachable by any other vertex in this subset. In order to check this condition, all strongly connected components of the union of $\mathcal{P}$ and $\mathcal{Q}$ denoted as SCC, have to be considered. If any strongly connected component in SCC has a non-empty intersection with one output set ($\mathrm{O}(\mathcal{P}) \cap \mathrm{SCC} \neq \emptyset$ or $\mathrm{O}(\mathcal{Q}) \cap \mathrm{SCC} \neq \emptyset$) and at the same time introduces a recursion between both modules, the second condition is violated.

4.3.5 Conclusion

Although we already gave our assessment of ASP with regard to the criteria in Tables 4.1 and 4.2 in Section 4.2.1, we would like to review this assessment in this section with more detail after we have now explained ASP.

Implementation Independent Properties		**Implementation Dependent Properties**	
Non-Monotonic	✓	Implementations	++
Decidable	✓	Efficiency	++
Expressiveness	++	Multi-Shot	✓
Intelligible	++		

Table 4.3: Assessment of Answer Set Programming (✗: no, ✓: yes, ∅: depends, Likert: ++,+,0,–,– –)

For the sake of convenience, Table 4.3 repeats the assessment of ASP from Section 4.2.1. Among the implementation-independent criteria, like non-monotonicity, decidability, expressiveness, and intelligibility, the non-monotonicity is an indisputable property of ASP. With regard to the decidability of ASP, among other things, the restriction of recursive function is crucial for maintaining decidability [86, 104]. In practice, the issue of maintaining decidability is well covered by loop detection mechanisms and further checks implemented in the ASP solvers. In case of a potential undecidable logic program, the program is rejected with appropriate feedback given by the solvers. Therefore, we marked ASP to be decidable in Table 4.1 in the sense of always terminating or rejecting undecidable logic programs.

Except for undecidable formalisms like FOL, the expressiveness of ASP is outstanding. At the same time, ASP rules are almost human-readable and not aggravated through nested quantifiers as it is usual in FOL. However, as FOL is more expressive than ASP, it is natural that FOL can, except for the non-monotonicity of ASP, mimic the behaviour of ASP. Therefore, the extension of FOL with a certain amount of non-monotonic reasoning capabilities [80], would make FOL almost as suitable for the domain of domestic service robots as ASP. Especially when we reconsider the limitations of ASP, due to the module property. With regard to the module property, it is worth noting that, according to personal conversations with developers of the ASP solver Clingo [3], the limitations of the module property exist for practical and efficiency reasons, because most ASP solvers implement the conflict-driven clause learning algorithm. Therefore, the limits of the module property can be overcome for the cost of efficiency. In the case of Clingo, these limitations are partially overcome due to the feature of Externals, which allows Multi-Shot solving. Within this thesis, the limitations of ASP due to the module property is further reduced to automatic renaming and generation of additional rules, as described in Chapter 7.

PartII

Solution

Architecture 5

In the first chapter about our solution, we want to give an overview of the general architecture design that has been adopted for the autonomous agents in this work. Afterwards, the concrete architecture is described. The architecture is based on the experience from a large body of preliminary works [96, 30, 62, 33, 81, 48, 64].

5.1 Design Principles

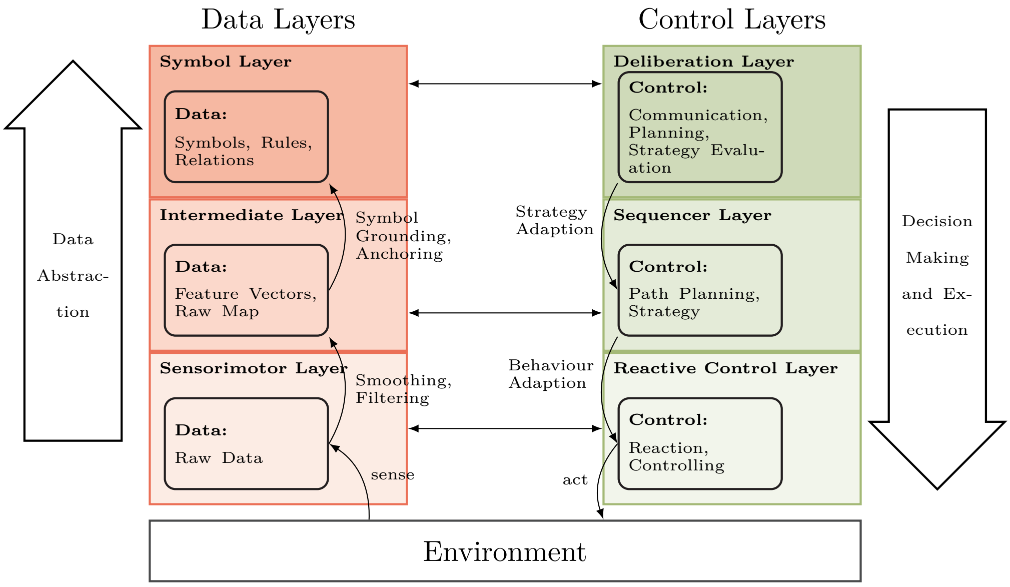


Figure 5.1: Abstract Control Loop Architecture

In Figure 5.1, an abstract robot software architecture is shown, as it is envisioned for the design of the solution presented in this work. It combines properties of a three-layered architecture with principles of a MAPE-K, Sense-Plan-Act, or BDI-based architecture (see Section 2.2). An assignment of parts, like the Monitoring module of MAPE-K or the Intentions of BDI, to specific layers in general, would be wrong, since the architecture parts can be part of several layers of the hierarchy.

The Data Layers on the left-hand side represent algorithms utilised to gain information and knowledge from raw sensor data. During the data processing, the data is continuously filtered, condensed, and abstracted until symbols and relations can be integrated into a symbolic knowledge representation. The algorithms on the right-hand side of Figure 5.1 require this knowledge. The Deliberation Layer, including algorithms for planning, communication, and strategy evaluation, computes its decisions based on the knowledge of the symbolic knowledge representation. The results are processed by the Sequencer Layer and transformed into commands for controlling actuators of the robot.

Nevertheless, the data abstraction and decision-making cycle are not strictly followed in all situations. Often the abstraction of data up to the level of symbols and relations is not necessary. Moreover, it is not efficient to make everything a decision on the topmost available layers. We think of a robot control architecture as of a human neural system. Many processes in a human body are handled without having the brain explicitly thinking about it, e.g., breathing, heartbeat, reflexes on the knee, sneezing, blinking. For a robot, the corresponding things could be obstacle avoidance, motor controlling, avoiding rough terrain such as stairs. Things like that should also in a robot control architecture be handled on lower levels and not bothering algorithms for planning or communication.

However, it can be difficult or even impossible to assign a specific task to a certain level in the architecture, shown in Figure 5.1. Obstacle avoidance, for example, is sometimes part of all three layers: While driving along a path, the robot in an office environment avoids humans or other robots by directly reacting to its 2D distance scans from its laser scanner (lower layers). The path itself is planned in order to avoid already known walls and finding the shortest path (intermediate layers). Planning this path is influenced by some knowledge about the environment, e.g., the robot knows that there is currently a meeting going on in some room and as the motion of the robot is relatively noisy, it knows that it should avoid meetings like a virtual obstacle, in order not to disturb the meeting (top layers).

The effort for abstracting data rises with each necessary processing step, therefore creating symbolic knowledge is often very costly. On the one hand, this argues for putting everything as on the lowest levels. On the other hand, the data on the lowest levels is often extensive in terms of memory consumption. One often has to ponder, whether it is more efficient to abstract the data and thereby reduce it to the smallest possible size or whether it is more efficient to directly operate on millions of raw camera pixel values without explicitly representing objects in the environment. Sometimes technical limitations also enforce a particular direction. If robots, for example, need to communicate about objects, it is probably impossible to send whole images of the objects to each other, due to bandwidth limitations. Instead, the objects have to be abstracted to symbols or coordinates in order to match bandwidth restrictions.

Although the focus of this work is on symbolic knowledge representation, we designed a structured architecture that is open for the integration of arbitrary approaches and principles, in order to be adaptable to any requirements while following the aforementioned arguments.

5.2 Implementation

In contrast to the abstract architecture design in Figure 5.1, the architecture in Figure 5.2 shows the concrete architecture of an autonomous agent. The design principle of a three-layer architecture is not visualised in Figure 5.2 because the focus is on the ALICA Framework. With regard to the architecture design in Figure 5.1, the ALICA Engine would be part of the Sequencer Layer, because the ALICA Engine is sequencing the execution of behaviours.

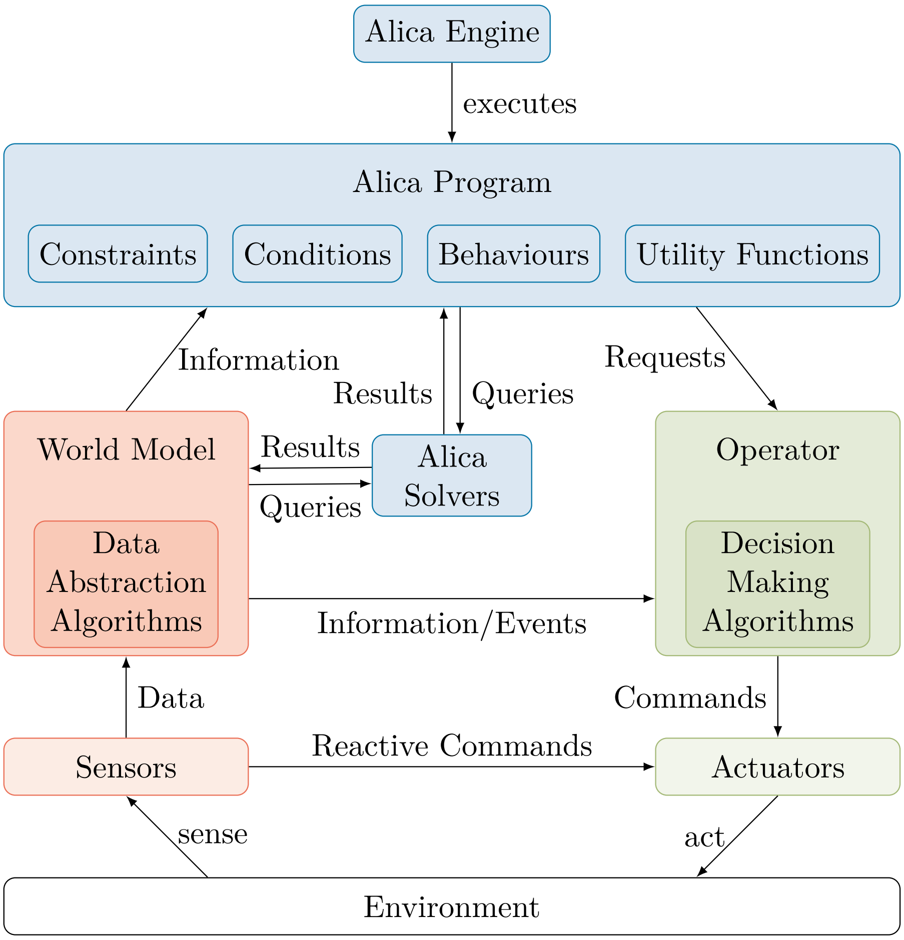


Figure 5.2: Components of the Architecture

Two essential design principles of the architecture are flexibility and reactiveness. Flexibility supports domain independence (R1, Section 1.2.4) and reactiveness is necessary for dynamic environments (R3, Section 1.2.4). In the following, we explain how these two principles are reflected in the architecture. At the lowest level of abstraction, we allowed the software components to receive the raw sensor data to send commands directly to the software components that control the actuators of a robot. In the robotic soccer domain, for example, this allows the RGBD camera driver on the goalkeeper to directly control ball catching devices in case of an approaching ball. In the domestic service robot domain, it allows the 2D laser scanner driver to directly stop the motion of the robot in case it would crash into a human. The reactiveness, achieved through the short cycle of sensing, reactive commands, and acting, mimics the reflexes of the human nervous system. The software components that represent the sensors and actuators are typically running in their own processes, which provide extra guarantees of reactiveness when the system is running in a real-time operating system.

The next level of reactiveness is slower because it is based on processed and therefore, more abstract data. The processing cycle, in that case, adds the World Model and the Operator component in between the sensors and actuators. The data layers and control layers from Figure 5.1 are comprised within this World Model and Operator component, respectively. The world model, together with its associated data abstraction algorithms, forms a directed processing graph whose root nodes are raw sensor data received from the sensor processes.

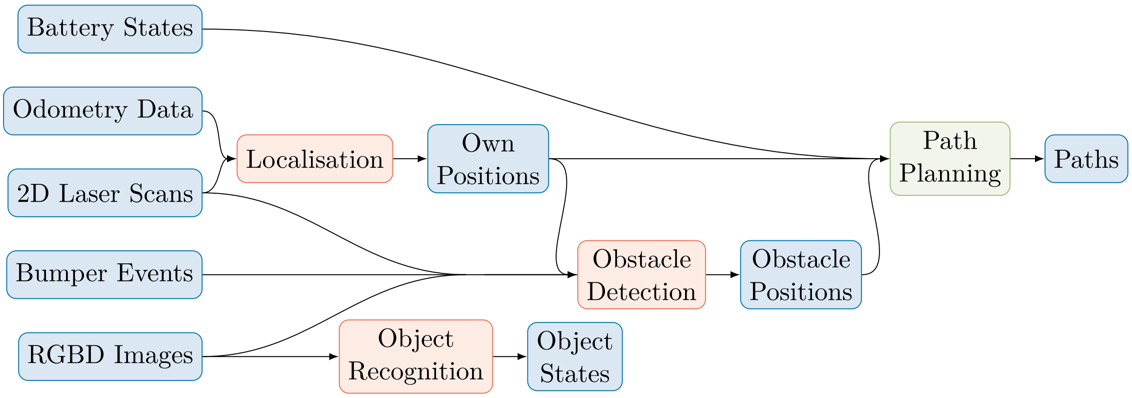


Figure 5.3: Example for a Part of a Directed Processing Graph

In Figure 5.3, a part of such a directed processing graph is shown as an example. Blue nodes are information, red nodes represent data abstraction algorithms, and green nodes represent decision-making algorithms. Each kind of information is stored in a ring buffer that can be accessed through the World Model component. The ring buffers have a configurable length, depending on the length of history that downstream decision making and data abstraction algorithms require. When and how often the algorithms are triggered, is highly adaptable. It is possible to set a fixed rate at which a processing algorithm is triggered, to trigger a processing step whenever one of the input ring buffers got a new information element, and to trigger a processing step on demand, i. e. a downstream processing algorithm queries for the most current information. Such flexibility of the data processing is necessary to create a more reactive and at the same time resource-efficient behaviour.

While the World Model is responsible for storing information in its ring buffers and offers a standardised way to access this information, the Operator is more like an interface for accessing planning and decision-making algorithms. Nevertheless, if the results of the planning and decision-making algorithms are required for a subsequent processing step, the results are stored in ring buffers of the Operator as well. The information stored in the World Model and the Operator, as illustrated in Figure 5.2 and 5.3, are based on sensor values only. However, within this work, we consider two more sources of information and knowledge. One source is the communication with humans and other collaborating agents, and one source is the use of commonsense knowledge databases. The detailed integration of both sources is explained in Chapter 8. For now, we consider communication with humans and other agents to be another sensor that perceives data about the environment of the agent. A commonsense knowledge database is static and therefore, can be considered as a single ring buffer element in the World Model that can be queried whenever needed.

In Figure 5.2, the ALICA Engine and the executed ALICA program is depicted separated from the Operator, although the ALICA Engine is mainly a decision-making algorithm. There are four different parts in the ALICA program, that require information from the World Model: constraints, conditions, behaviours, and utility functions. The purpose of those parts is explained in Chapter 3, here we explain their interaction with the rest of the architecture. Conditions and utility functions only need information from the World Model, in order to evaluate their truth and utility value, respectively. Constraints are, with regard to the need for information, just like conditions and utility functions, but their evaluation depends on the corresponding ALICA solvers. Following the declarative programming approach, as explained in Section 4.3.2 for the instance of ASP, the constraints describe the problem and the solvers provide the solving algorithms.

Behaviours are, from the perspective of the ALICA program, the junction between the World Model and the Operator. This junction exists because a usual behaviour includes three steps that are borrowed from the sense-plan-act cycle from the early days of artificial intelligence (see Section 2.2.2). A behaviour that should safely navigate the robot to a certain point of interest, for example, first acquires all necessary information from the World Model (sense). It then instantiates the path planning algorithm with the gained information (plan), and finally creates the abstract drive commands and sends them through the interfaces of the Operator to the motion of the robot (act). A behaviour could also use one of the ALICA solvers by querying for a solution of a specific constraint. The interaction of behaviours with constraints and ALICA solvers, makes the ALICA solvers belong to the Operator component, which comprises decision-making algorithms. However, it is also possible to trigger ALICA solvers as part of the information processing in the World Model component. Therefore, ALICA solvers do not belong to the World Model component, nor the Operator component.

6 ALICA – Even More General

The advantages and disadvantages of the ALICA Framework in its former version (Propositional and General ALICA), as well as the improvements that are necessary to use ALICA for this thesis, are discussed in Section 3.5. We consider the advantages of the ALICA Framework to outweigh its disadvantages and therefore chose the ALICA Framework as the basis for this work. Therefore, we explained our general system architecture and highlighted the unique roles of the ALICA Framework and ALICA solvers in Chapter 5. In the following, we describe the improvements of the ALICA Framework that have been realised within the context of this thesis, summarised in three sections. Section 6.1 is about the improved domain independence through the reimplementation of the ALICA Framework in C++ and its independence from any underlying communication middleware and clock. In Section 6.2, we go into details of the new general solver interface of the ALICA Engine, which further improves the domain independence of the ALICA Framework. Finally, the new architecture of the reimplemented ALICA Plan Designer, together with its plugin support for arbitrary reasoning formalisms, is presented in Section 6.3.

6.1 Improved Domain Independence

One major issue with the former version of the ALICA Framework was that its engine was implemented in the GNU/Linux' version of C#. C# is running in a virtual machine, just like Java, and depends on a garbage collector that can only be partially controlled from an application running in that virtual machine. Therefore, the ALICA Engine could never be deployed in a scenario that requires an application to be executed in real-time. The ALICA Engine, neither the old nor the version developed in this thesis, is designed to be executable in real-time, because no runtime guarantees are given for behaviours or the main thread of the engine. However, since the engine is just a library, the C# version of this library could have never been integrated within a real-time application. Furthermore, any scenario or domain that requires a certain amount of reactiveness suffered from the intervention of the garbage collector. In the past, this problem was recognised, for example, when the robotic soccer team Carpe Noctem Cassel tried to implement a shooting behaviour that scores a goal while rotating. Measured delays of 60 ms, induced by the garbage collector, where big enough to make the robot miss the goal. Similar situations may appear in the context of domestic service robots, the main application domain of this thesis. A service robot, for example, needs to synchronise its execution of a grasping behaviour with a human, in order to receive an item without dropping it.

As a result, we reimplemented the ALICA Engine in the C++ programming language and thereby paved the way for additional application domains. The new ALICA Engine is not just a simple reimplementation; it also has significantly fewer dependencies to external libraries. The new version is written in Version 14 of the C++ programming language, which provides its own threading and synchronisation capabilities and therefore avoids dependencies to bulky libraries like Boost[1].

Another dependency that we could drop for the new version is the Robot Operating System version 1 (ROS-1). The communication middleware, implemented as part of the ROS-1 libraries, was used to communicate between different instances of the ALICA Engine. In Section 9.1.1, the ROS-1 middleware is explained in detail. In the context of improving the domain independence of the ALICA Framework, it is essential to know that the ROS-1 middleware is only designed for interprocess communication on a single robot system. Although the ROS-based communication was hard-coded into of the former ALICA Engine, it did not take care of sending messages across system boundaries and therefore, it was still necessary to send the local ROS messages via an external proxy to other systems. The new version of the ALICA Engine only requires some communication middleware to implement a message-based communication interface, which is part of the source code of the engine. This way, it is possible to use any message-based communication middleware by developing a thin wrapper that implements the communication interface of the ALICA Engine. Theoretically, it would even be possible to let ALICA-based agents communicate via E-Mail. The flexibility to use the communication middleware that fits best to the application domain improves the domain independence of the ALICA Framework significantly.

A feature of the ROS ecosystem inspires another improvement of the new ALICA Engine. ROS provides its own time interface. Instead of measuring the time directly through an available system clock, applications that use ROS could measure the time through a dedicated ROS clock. The advantage of the ROS clock is that another application can control it. In case of a simulated robot scenario, for example, the simulated physics might be so resource-intensive that it cannot run in real-time. In that case, an agent that is operating in real-time might not be simulated correctly. Therefore, the execution of an agent that is using the ROS clock could be slowed down and thereby match the slower simulated real-time. In order to keep that feature and making the ALICA Engine independent from ROS nevertheless, we introduced a clock interface in the ALICA Engine, which works just like the communication interface.

We made the reimplementation of the ALICA Engine independent from a specific communication middleware and independent from a specific clock and thereby improved its domain independence. However, it is also noteworthy that the availability of C++ com-

[1] https://www.boost.org/ - [last accessed on February 11th, 2020]

pilers for arbitrary system architectures also makes ALICA applicable on these system architectures, while the support of GNU/Linux-based C# virtual machines on system architectures different from x86 and x86-64 are only community based and therefore potentially incomplete.

6.2 General Solver Interface

The requirement to extend the ALICA Framework by a general solver interface is due to two issues. First, the CSP solver (see Section 3.2.5) was the only way to assign values to variables of ALICA programs in the former version of ALICA and therefore limited the applicability of the ALICA Framework to specific domains. Second, the ASP reasoning formalism (see Section 4.3.5) has, although providing non-monotonic semantics, certain limitations to represent dynamic knowledge in a symbolic form. Therefore, we conclude that the ALICA Framework needs to support arbitrary reasoning formalisms for a maximum of domain independence.

Concerning the interaction of the ALICA Engine and an arbitrary solver, the provided solution is following the same approach as described in Section 6.1 for the interaction with an arbitrary communication middleware and clock. However, a simple ALICA solver interface definition is not enough in this case. A behaviour usually triggers the determination of a value for a specific variable; the process itself is described in Section 3.2.4. During this process, the problem description parts, that are distributed in the hierarchy of the ALICA program, are collected and composed of a single problem description that is presented to the corresponding solver. These problem description parts reference the variables whose values should be determined and reference the expressions that constrain the range of possible values. Therefore, three domain-specific entities exist in this process: the solver, the variables, and the expressions that constrain the variables.

In our implementation, these three entities are denoted as *ALICA Solver*, *Solver Variable*, and *Solver Term*. An ALICA Solver is implementing the ALICA solver interface and wraps the actual solver accordingly. Therefore, it must implement a method for determining a solution to a given problem description and a method to determine whether a solution for a given problem description exists. The complexity of determining a solution and determining whether a solution exists is often the same. However, if it is required to find an optimal solution, the complexity rises. The original CSP solver of the former ALICA version allowed to specify utility functions that should be optimised during the solving process. Therefore, we decided to keep the feature of expressing optimisation expressions in Solver Terms and the two different solver methods.

An instance of an arbitrary solver can be created once and given to the ALICA Engine from outside. The *Solver Variables* and *Solver Terms*, however, often depend on the current situation and information stored in the ring buffers of the World Model component.

Therefore, they need to be described during the collection of the actual problem description parts. The *Solver Variables* are created by the solver itself on-demand of the ALICA Engine. The *Solver Terms* are created through the implementation of method stubs that are auto-generated by the ALICA Plan Designer. The method bodies include an annotated region that protects its content from being overwritten during a new generation of the method stubs. A more detailed description of the auto-generation process and the corresponding architecture of the new ALICA Plan Designer is given in Section 6.3.

That way, the ALICA Engine can interact with arbitrary solving mechanisms and create the corresponding problem descriptions without any compile-time dependencies. The only limit here is that the solving mechanism must work with some variable and term objects, in order to describe related problems. According to the current state, four different solvers were wrapped in compliance with the described general solver interface. At first, there is the original CSP solver. Second, we developed a solver that is assigning values to variables without any further optimisation or problem description. Third, the PROViDE middleware [48] was developed in a way that it could be used to determine values of variables through the general solver interface as well. Finally, the fourth solver is the ASP solver Clingo, whose integration is described in detail in Chapter 7.

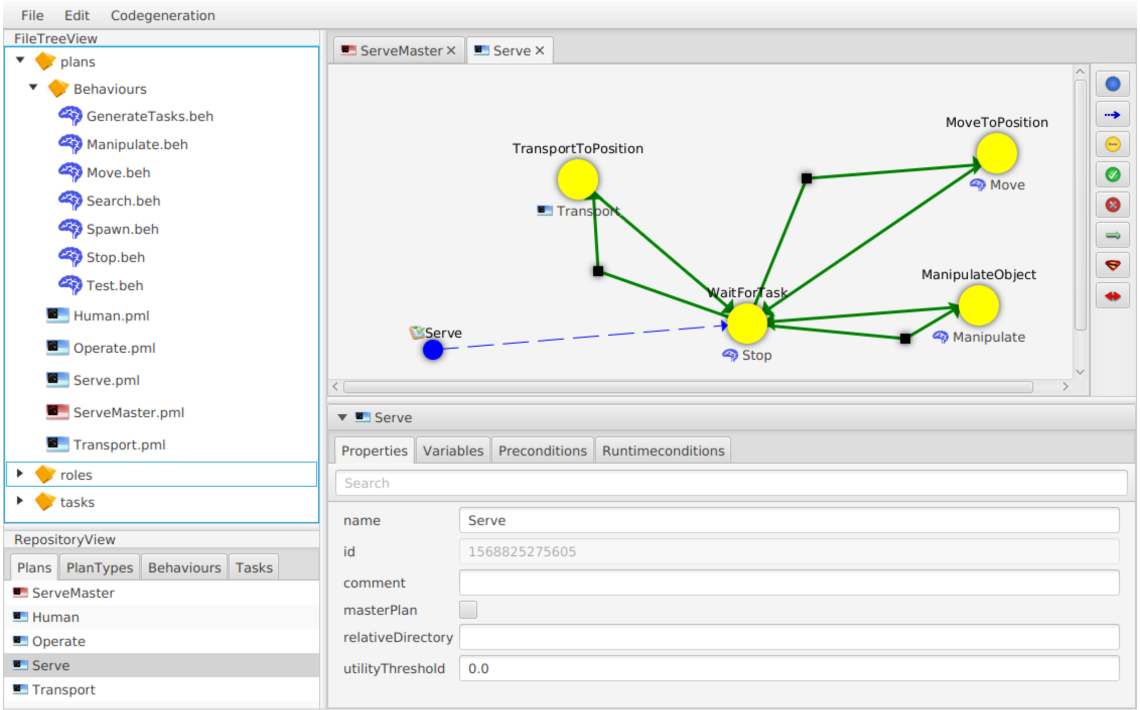
Depending on the problem, it is desirable to come to a common solution for the whole team. The ALICA Framework achieves such a consensus through three measures: caching solutions, exchanging solutions periodically among team member, and clustering similar solutions. These measures only work for CSPs. Stable models of answer set programs, for example, are not suitable to be clustered. Instead, the consensus algorithms of the PROViDE middleware [48] can be used to synchronise the resulting stable models among the team members. However, in our experiments, we rely on the direct exchange of knowledge (see Section 9.2) and thereby achieve consistency for relevant parts of the knowledge base of each agent. Therefore, deduced results of different agents in our experiments do not contradict each other.

6.3 The New Plan Designer

The new ALICA Plan Designer has been completely rewritten for this thesis and significantly improves the usability of the whole ALICA Framework. In order to convey the idea of its usage, we shortly explain the user interface in Section 6.3.1. In Section 6.3.2, we explain the modular design of the sophisticated underlying architecture and highlight further improvements and developments that have been realised based on its modular design. Further, in Section 6.3.3, we describe the plugin system, necessary for the new Plan Designer to generate code for arbitrary solving mechanisms (see Section 6.2). Finally, general improvements are summarised in Section 6.3.4.

6.3.1 User Interface

At a first glance, the user interface of the new Plan Designer (see Figure 6.1) is similar to the one of the former version, in order to ease the migration for the users. However, we also made the components of the user interface generally more lightweight and accessible.



name: Serve
id: 1568825275605
comment:
masterPlan: □
relativeDirectory:
utilityThreshold: 0.0

Figure 6.1: Overview of the UI of the new ALICA Plan Designer

The primary user interface, as shown in Figure 6.1, is partitioned into four segments. A classic file tree view is located in the upper left corner. The plans folder of file tree view comprises ALICA programs including behaviour, plan, and plantype files, each with a distinct icon that is reused consistently throughout the whole user interface. The roles and tasks folder is for specifying role sets and task repositories, respectively. In the file tree view, it is possible to move or copy files, which also triggers the movement or copy of corresponding auto-generated source code files. Maintaining the consistency between modelled ALICA programs and corresponding auto-generated source code files is a significant improvement compared to the old Plan Designer (see Section 6.3.4). Furthermore, the file tree view and the file menu are now the only two options for creating new files.

The old version had extra tools for creating conditions, plans, behaviours, and plantypes, that polluted the tools palette of the Plan Editor and were somewhat unintuitive to use. Instead, the palette of the Plan Editor of the new Plan Designer, as shown in Figure 6.2, only includes tools essential for editing the finite state machines of a plan. The tools from top to bottom are for creating entry points, assigning states to entry points,

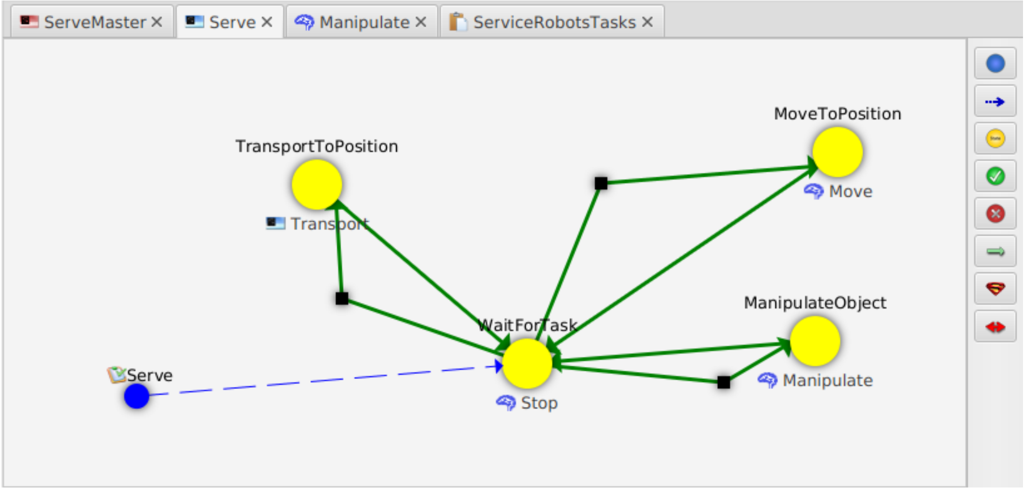
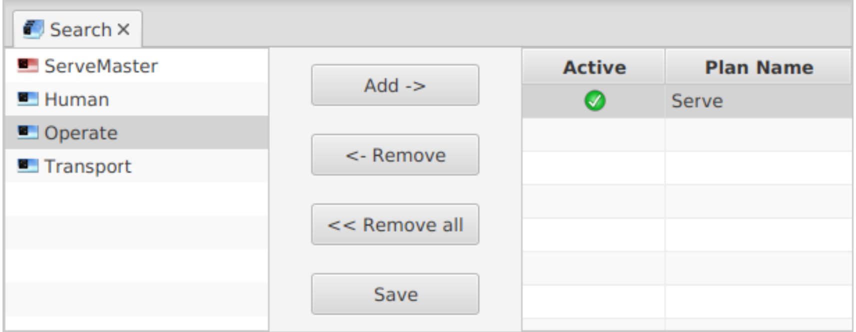


Figure 6.2: Plan Editor

creating states, creating success states, creating failure states, creating transitions, creating synchronisations, and assigning transitions to synchronisations. While editing state machines, the background canvas grows automatically, which allows the user to create state machines on a virtually unlimited canvas. The outline window of the old Plan Designer is dropped in the new Plan Designer because it had little use according to the users of the old Plan Designer.



ServeMaster
Human
Operate
Transport

Active	Plan Name
✓	Serve

Figure 6.3: Plantype Editor

Similar to the Plan Editor, though less complicated, editors for plantypes, task repositories, and behaviours can be opened in a separate tab in the upper right segment of the main user interface. The Plantype Editor basically remained to be the same as for the old Plan Designer (see Figure 6.3). On the left-hand side, a list with all available plans is given and on the right-hand side a list with all plans that belong to the plantype, including their activation state, is presented.

The Taskrepository Editor, as shown in Figure 6.4, presents the list of all available tasks, just like the Repository View does in the Tasks tab, but also provides controls to create additional tasks. Via the context menu of each task, it is also possible to list all

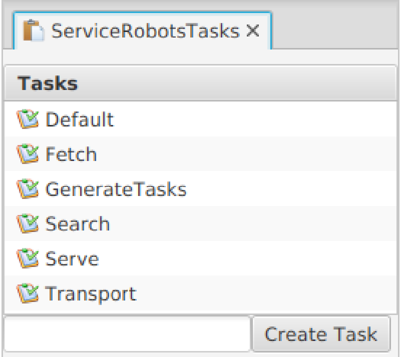


Figure 6.4: Task Repository

plans were the task is used in a pop-up window.

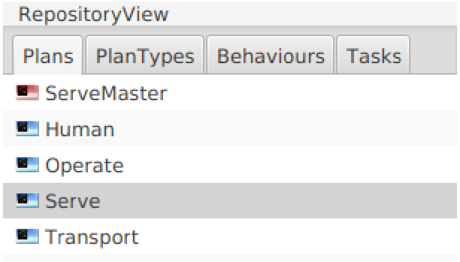
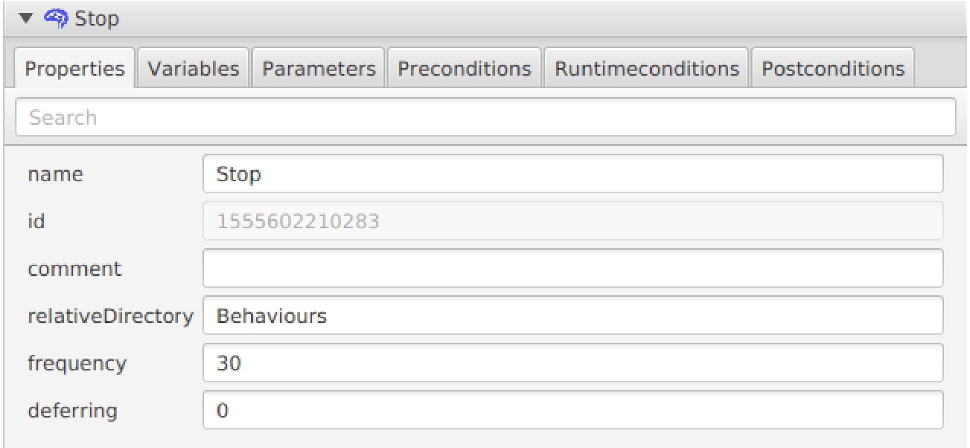


Figure 6.5: Repository View

Assigning tasks to entry points and inserting behaviours, plans, or plantypes into states is done by a simple drag and drop operation from the Repository View, shown in Figure 6.5. The Repository View, located in the lower-left segment of the new Plan Designer, includes an extra tab for each kind of reusable plan element and allows the deletion of these elements via its context menu. If the element is still used somewhere, the deletion is aborted, and an interactive pop-up window presents a list of places where the corresponding element is used. This feature of showing the usage of a repository element can also be directly triggered via the context menu of the elements. Thus, the reuse of plan elements and clean up of unused plan elements is made a lot easier, compared to the old Plan Designer.

The fourth segment of the main user interface is the Properties View, located in the lower right segment (see Figure 6.6). Whenever an element within the Plan Editor, Plantype Editor, or Task Repository Editor is selected, its properties are shown in the Properties View. The user interface of the Properties View is based on the Property Sheet of the controls-fx library[2] and shows different tabs, like the variables and precondition tab, depending on the selected element. The new Plan Designer provides an extra editor tab for all reuseable plan elements except behaviours. In case of behaviours, the editing is done through the Properties View, which can be opened in an extra tab just like all other editor

[2] https://github.com/controlsfx/controlsfx [last accessed on February 13th, 2020]

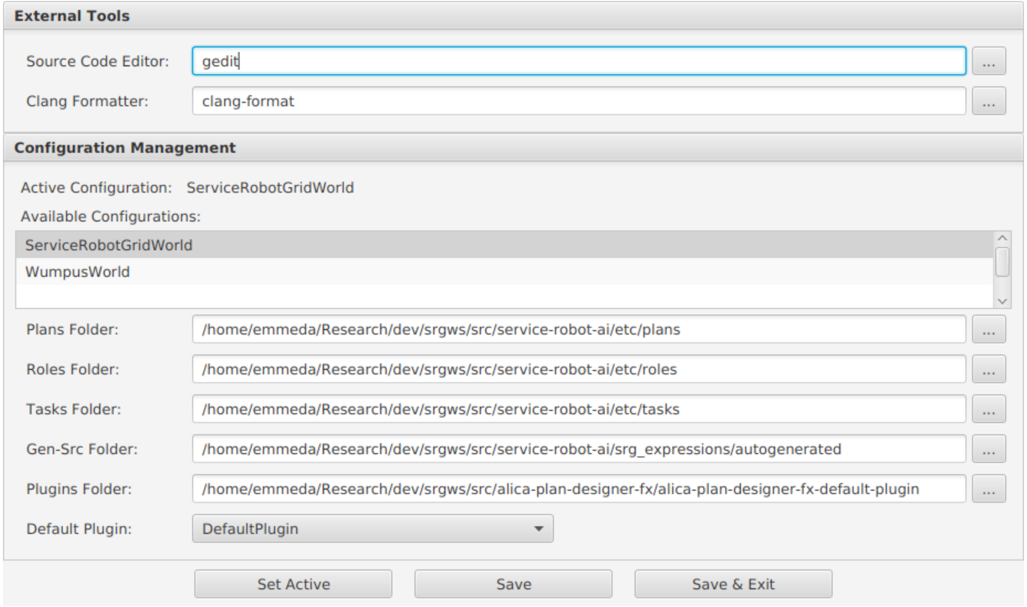


name: Stop
id: 1555602210283
comment:
relativeDirectory: Behaviours
frequency: 30
deferring: 0

Figure 6.6: Properties View

tabs. Figure 6.6, for example, shows the Properties View of the Stop behaviour.

In general, the user interface of the new Plan Designer is based on the Java-FX Framework only. Until version 8, Java-FX was part of Java itself, and since then it is available as an open-source library[3]. The former Plan Designer based its user interface on the Graphical Editing Framework (GEF)[4], which fails to support Java-FX completely in current versions and heavily relies on further libraries of the Eclipse Foundation (see Section 3.3).



Source Code Editor: gedit
Clang Formatter: clang-format
Active Configuration: ServiceRobotGridWorld
Available Configurations:
ServiceRobotGridWorld
WumpusWorld
Plans Folder: /home/emmeda/Research/dev/srgws/src/service-robot-ai/etc/plans
Roles Folder: /home/emmeda/Research/dev/srgws/src/service-robot-ai/etc/roles
Tasks Folder: /home/emmeda/Research/dev/srgws/src/service-robot-ai/etc/tasks
Gen-Src Folder: /home/emmeda/Research/dev/srgws/src/service-robot-ai/srg_expressions/autogenerated
Plugins Folder: /home/emmeda/Research/dev/srgws/src/alica-plan-designer-fx/alica-plan-designer-fx-default-plugin
Default Plugin: DefaultPlugin

Figure 6.7: Configuration Window

Another feature of the new Plan Designer that support the designer to work concurrently on different domains is the support for different configurations. Accessible via the Edit

[3] https://openjfx.io/ [last accessed on February 13th, 2020]
[4] https://www.eclipse.org/gef/ [last access on February, 13th 2020]

menu, the Configuration Window, as shown in Figure 6.7, allows managing configurations of plans, roles, and tasks folders, as well as target folders for the code generation and plugin folders for loading different plugins at runtime. The active configuration is changed and automatically applied to all user interface segments by selecting a configuration and pressing the Set Active button.

6.3.2 Modular Design

Approaching the Plan Designer via the "Separation of Concerns" principle uncovers the different purposes that the Plan Designer has. As described in Section 6.3.1, the Plan Designer should provide a graphical user interface that allows designing ALICA programs. Another purpose is to serialise and deserialise the model that represents the designed programs into a machine-readable format. A third functionality is the code generation that eases the development of domain-specific program parts that are compatible to and therefore executable by the ALICA Engine.

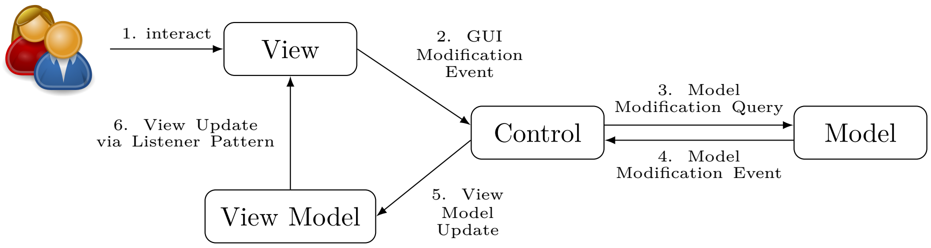

1. interact
View
2. GUI Modification Event
Control
3. Model Modification Query
Model
4. Model Modification Event
5. View Model Update
View Model
6. View Update via Listener Pattern

Figure 6.8: Model-View-Control-Viewmodel Pattern

In order to implement the "Separation of Concerns" principle, a variant of the well known Model-View-Viewmodel pattern is realised within the architecture of the new Plan Designer. The Model-View-Viewmodel pattern is itself a variant of the Model-ViewControl pattern, which the former Plan Designer failed to enforce (see Section 3.3 and 3.5.2). The view model is a model that wraps the state of the actual model, but also includes the state of the view that should not be part of the model. The control component is merged into the view model of the Model-View-Viewmodel pattern. In the pattern realised in the architecture of new Plan Designer, this merge is reverted for an even more apparent separation of concerns and is therefore denoted as Model-View-Control-Viewmodel (MVCVM).

A typical chain of events and queries, following the MVCVM pattern, is shown in Figure 6.8. At first, the user interacts with the user interface, for example, by changing a label. The affected view element is capturing the change via a listener pattern and creates a GUI Modification Event, which is sent to the control component. The GUI Modification Event is then interpreted by the control component and translated into a Model Modification Query, which asks the model to rename a plan. In case the request triggers a modification of the model, the model, in turn, generates a Model Modification Event that is sent back to the control component. At a fifth step, the view model is updated by the control component. The view is registered on the properties of the view model via a listener pattern and is therefore updated by the modification of the view model.

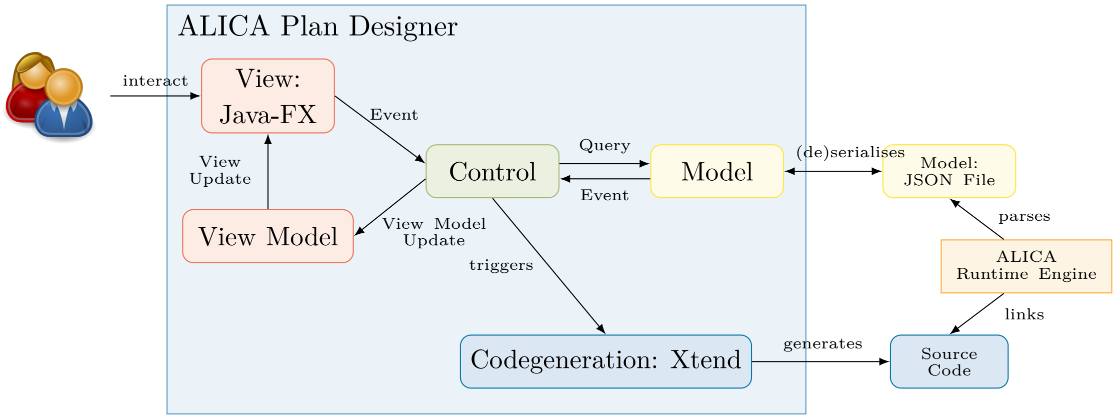


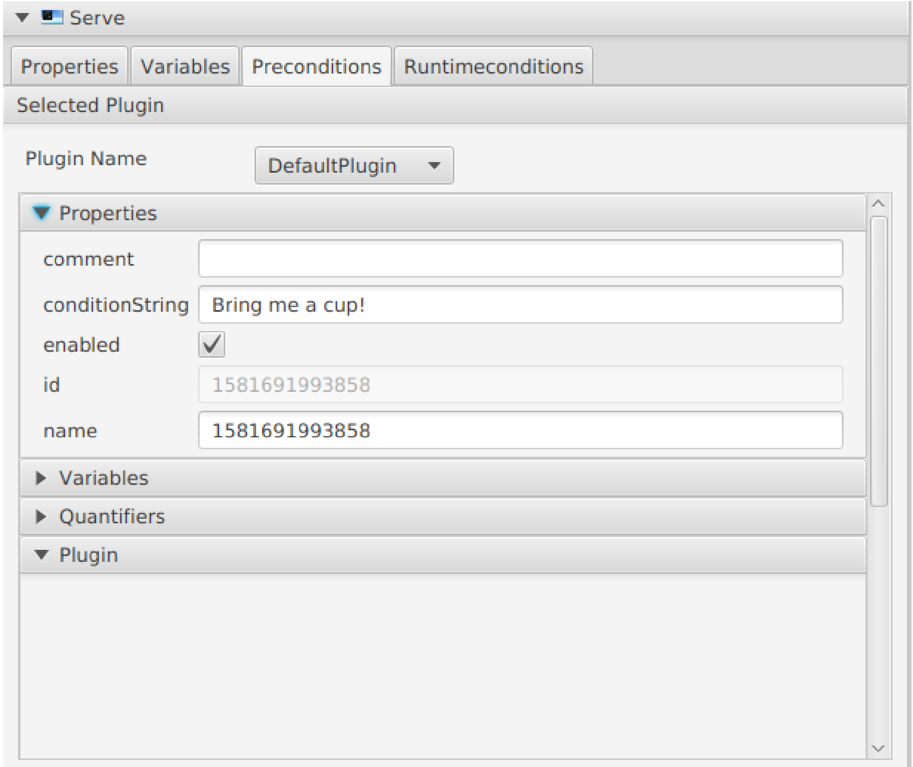
Figure 6.9: Components Interacting with the new Plan Designer

The three separated purposes of the Plan Designer are visualised in Figure 6.9. The view and view model, responsible for the graphical editing of ALICA programs, are shown in red. The model and its files which are (de)serialised are yellow. The third purpose is the generation of code, which is represented by the blue code generation and source code components. The central control component (green box) combines the three "separated concerns" and avoids dependencies between them. The benefit of the modular design is manifold. The new Plan Designer includes fewer lines of code, the code is easier to maintain, and the Plan Designer is more comfortable to extend. It is even possible to trigger the code generation from a simple console program, without starting the whole user interface. Another functionality that would make sense as a stand-alone program, is the usage analysis of reusable plan elements (see Repository View in Section 6.3.1). All this is just possible, because of the successful realisation of the "Separation of Concerns" principle in the new Plan Designer.

Additionally, the improved design of the new Plan Designer already paved the way for further extensions that are not part of this thesis. The new Plan Designer is, for example, used to visualise the execution state of a team of ALICA agents for debugging purposes. Further, there was only little effort necessary to separate the code generation from the new Plan Designer in order to support different programming languages. Aside from C++, the code generation is currently extended to support Java and Python.

6.3.3 Plugin System

While the CSP solver was a fixed component of the former ALICA Engine, there was no need to generate method stubs different from those suitable for the CSP solver. For the new Plan Designer and ALICA Engine, this has changed. Although the empty method stubs, generated by the original code generation, are general enough to implement problem descriptions for arbitrary formalisms by hand, it is desirable to support the modelling process further. A plugin system is implemented in the new Plan Designer, in order to improve the support for different formalisms, like propositional logic or ASP.



Plugin Name: DefaultPlugin

comment:

conditionString: Bring me a cup!

enabled: ☑

id: 1581691993858

name: 1581691993858

Figure 6.10: Plugin User Interface of the new ALICA Plan Designer

Each plugin needs three components. At first, it needs to provide a graphical user interface that can be integrated into the Plan Designer. In Figure 6.10, the Properties View with the Precondition tab of a plan is shown. In the drop-down menu, all loaded plugins are listed, and it is, therefore, possible to choose the formalism that should be used to describe the constraint attached to the corresponding condition. In the case of Figure 6.10, the Default Plugin is selected and mimics the original behaviour of the former Plan Designer. The empty area in the lower part of Figure 6.10 is reserved for the possibility to provide a formalism specific user interface, for example, to describe boolean formulas.

The second component of a plugin allows to (de)serialise the modelled constraints as part of the ALICA program. In case of the Default Plugin, nothing is modelled at runtime, and the constraints only exist as handwritten problem descriptions in the protected regions of the auto-generated method stubs. The plugin system brings the creation process of an ALICA program closer to a model-driven approach, where the executable code is generated from a model instead of written by hand. For this purpose, the third component of a plugin is the extension of the code generation of the new Plan Designer. The code generation is based on the language Xtend[5], which supports the extension of code generation templates at runtime.

In summary, a plugin needs a graphical user interface, must be able to (de)serialise the modelled constraints, and must extend the code generation, in order to avoid unnecessary handwritten source code. The integration of the plugin system into the new Plan Designer further improves the domain independence of the ALICA Framework and eases the development of ALICA program by utilising the advantages of the model-driven engineering approach.

6.3.4 General Improvements

In this final section about the new Plan Designer, we will summarise improvements that are on the one hand more technical and on the other hand, induce small changes in the general model for ALICA programs.

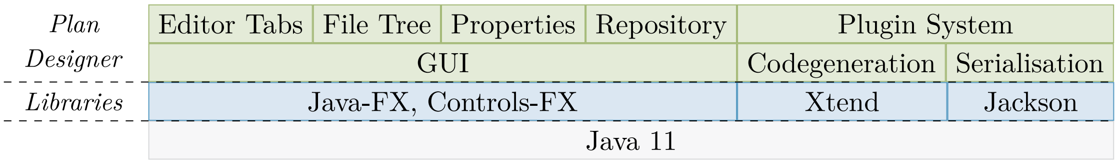


Figure 6.11: New Plan Designer Architecture

Generally speaking, the dependencies of the new Plan Designer are significantly fewer than the dependencies of the old Plan Designer. The architecture overview in Figure 6.11 lists the library dependencies of the new Plan Designer: Java-FX, with its extension Controls-FX, Xtend, and Jackson. The latter is used to serialise the model to JSON files. The JSON files are generally smaller and easier to read because the serialisation of the former Plan Designer used XML as its serialisation format. Another advantage of Jackson is that it can (de)serialise plain old Java objects (POJOs) and generate JSON schemas from them.

With the old Plan Designer, the coupling of model files and generated source code was rather loose. The deletion of model files, for example, did not remove the corresponding source code. After some time of using the old Plan Designer, several unused source code artefacts slowed the build process and their manual removal was very cumbersome because

[5] www.xtend-lang.org [last accessed on February 14th, 2020]

it was hard to check whether any model file belonging to the source code files still existed. Therefore, we put extra effort in synchronising the model and generated source files during copy, move, delete, or rename operations. Moreover, most of these operations did not work correctly in the old Plan Designer, due to the growing maintenance effort.

An improvement that also concerns ALICA programs is the addition for behaviours to have their own pre-, runtime-, and postconditions. Although already envisioned by the ALICA Language, the old Plan Designer and ALICA Engine did not support conditions on behaviours. Instead, it was necessary to wrap the behaviour in a plan that was annotated with the conditions instead. This workaround is denoted as canonical behaviour plans in [77, Section 5.7]. The necessity of this workaround aggravated the application of planning approaches in the context of the ALICA Framework, as described in [18].

Finally, we extended the support for behaviour configurations and allow to configure plans and plantypes as well.

6.4 Summary

Although the former ALICA Framework already provided a certain amount of domain independence, it was necessary to extend it. Therefore, we analysed the current state of the framework, as written in Section 3.5, and improved the framework where necessary. At first, we reimplemented the original ALICA Engine, written in C#, in C++. Afterwards, the restriction of a single CSP solver was removed and we enabled the use of arbitrary solving and reasoning formalisms, by introducing a general solver interface to the ALICA Engine. This improvement is accompanied by the redesign of the ALICA Plan Designer. Aside from a modular architecture, the introduction of a plugin system allows to model arbitrary problem specifications, just like the general solver interface allows to use arbitrary solvers. This collection of changes to the ALICA Framework improves its reactiveness (R3, Section 1.2.4), domain independence (R1, Section 1.2.4), and makes the framework an easy to use and state-of-the-art multi-agent framework.

7 Answer Set Programming as Dynamic Knowledge Base

In Chapter 5, we described the design principles of our system architecture. Following these principles, we improved the domain independence of the ALICA Framework, as shown in Chapter 6. Thanks to the revision of the ALICA Framework, we are now able to integrate the ASP solver Clingo as ALICA-compatible solver and create a dynamic knowledge base that adheres to the design principles of our architecture. While Chapter 4.2 gives the reason for choosing ASP over other formalisms, our understanding of a dynamic knowledge base is that of a central module for storing and reasoning about knowledge. Thereby, the knowledge is continuously adapted at a high frequency, due to updates from sensor value processing or planning modules. From a runtime perspective, the knowledge base is always on, and no restarts are necessary. In summary, the properties of our dynamic knowledge base are similar to those of a truth maintenance system.

7.1 Integrating Clingo

The ASP solver Clingo [9] is developed by the Knowledge Processing and Information Systems Group of Professor Thorsten Schaub at the University of Potsdam. The reason why we use Clingo for our knowledge base is that among all available ASP solvers Clingo is the only solver that supports Externals as described in Section 4.3.3. Externals allow to change truth values of facts at runtime without reinitiating the complete reasoning process. The Clingo system is a combination of the grounder Gringo and the ASP solver CLASP [60].

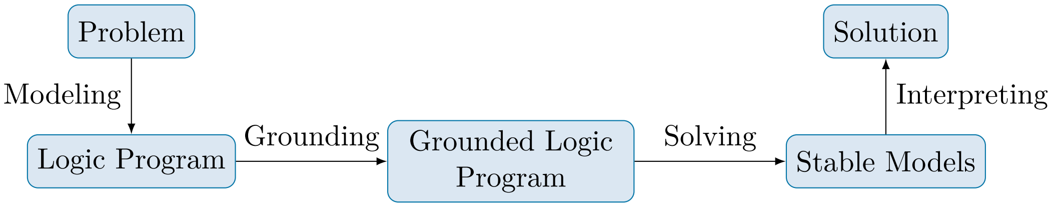


Figure 7.1: Two-Step Process of Determining Stable Models in ASP [42]

Concerning the two-step process of determining stable models in ASP, as described in Section 4.3.2, Gringo is responsible for the grounding step and CLASP for the solving step. The API of Clingo therefore provides a method to ground an ASP program segment, which is then part of the grounded ASP program internally represented in Clingo. Another method of the API allows to solve the internally ground ASP program and returns the stable models. The wrapper that integrates Clingo into our framework and makes it compatible with the general solver interface of ALICA needs to expose both methods separately. The grounding step must be called several times for different program sections. Afterwards, the solving step is executed one time, once the proposed rule and fact composition is constructed, in order to deliver the stable models. Before a program section can be grounded, it needs to be added to Clingo. The rules that need to be added are forwarded as simple strings, which are parsed by Clingo and transformed into its own data structures. A straight forward interaction with Clingo consists of three steps: composing the non-grounded program by adding them, grounding the non-grounded program, and solving the grounded program.

As proposed in Figure 7.1, two more steps without direct interaction with Clingo are necessary. At first, it is necessary to model the rules that need to be added. Second, the resulting stable models need to be interpreted. When we denote the first step as *modelling the input*, and the second step as *interpreting the output*, the API of Clingo is asymmetric about its input and output. While the input is expected to be represented as strings, the recommended way to perceive the resulting stable models requires the wrapper to handle the internal data structures of Clingo. The easiest way to relate the output with the input, is to translate the strings of the input into internal data structures of Clingo, too. For this purpose, Clingo offers a parse method that returns the internal data structures of Clingo for a given string.

```
#program tasks.
task(t1).
type(t1, search).
task(t2).
type(t2, transport).
task(t3).
can(X, move) :- can(X, transport).
-can(X, transport) :- -can(X, move).

#program robots.
robot(sr1).
can(sr1, move).
robot(sr2).
can(sr2, transport).
robot(sr3).
-can(sr3, move).

#program query.
{ assign(X,R) : robot(R) } = 1 :- task(X).
{ assign(X,R) : task(X) } = 1 :- robot(R).
:- assign(X,R), type(X, transport), -can(R, transport).
```

```
22 :- assign(X,R), type(X, search), -can(R, move).
```

Listing 7.1: Task Assignment Example: Input Program

The ASP program in Listing 7.1 is the starting point for an exemplified interaction with Clingo as it could happen within our knowledge base. At first the rules and facts from the program sections `tasks`, `robots`, and `query` are added into Clingo. The second step is to ground these program sections.

```
-can(sr3,transport).
can(sr2,move).
:-assign(t1,sr3).
:-assign(t2,sr3).

#delayed(1).
#delayed(2).
#delayed(3).
#delayed(1) <=> 1<=#count{0,assign(t1,sr1):assign(t1,sr1); 0,assign(t1,sr2):assign(t1,sr2); 0,assign(t1,sr3):assign(t1,sr3)}<=1
#delayed(2) <=> 1<=#count{0,assign(t2,sr1):assign(t2,sr1); 0,assign(t2,sr2):assign(t2,sr2); 0,assign(t2,sr3):assign(t2,sr3)}<=1
#delayed(3) <=> 1<=#count{0,assign(t3,sr1):assign(t3,sr1); 0,assign(t3,sr2):assign(t3,sr2); 0,assign(t3,sr3):assign(t3,sr3)}<=1
```

Listing 7.2: Task Assignment Example: Additional Rules after Grounding

The grounded program will include all facts from Listing 7.1, as well as the rules given in Listing 7.2. The first two facts state that robot `sr3` cannot `transport` and robot `sr2` can `move`. Both facts where deduced from Rule 7 and 8 of the original program and are no longer part of the grounded program. The next two rules state that robot `sr3` cannot be assigned to task `t1` nor to task `t2`, which is deduced from the constraints formulated in Rule 21 and 22. Both constraints where also dropped during grounding.

The last six rules which, which where added during grounding, include a special `#delayed()` predicate, which means that the actual assignment of tasks to robots, as requested by Rules 19 and 20, will be decided during the solving step.

```
assign(t1,sr2) assign(t2,sr1) assign(t3,sr3)
assign(t1,sr1) assign(t2,sr2) assign(t3,sr3)
```

Listing 7.3: Task Assignment Example: The two incomplete Answer Sets

The result of the solving step is shown in Listing 7.3. The two answer sets, one per line, represent the two valid assignments of tasks according to the input program in Listing 7.1.

As already known through the grounding step, robot `sr3` can only be assigned to task `t3`. Therefore, the only difference between the two answer sets is the assignment of the tasks `t1` and `t2` to the robots `sr1` and `sr2`. Now, the final step for this example is the interpretation of the answer sets. Therefore it is essential to note, that the answer sets given in Listing 7.3 are incomplete. The complete answer sets would both also include all facts from the initial program as well as all facts that where added during the grounding step. We dropped these additional facts for a moment, to make the result more easy to understand.

```
-can(sr3,move) -can(sr3,transport) can(sr1,move)
    ↪ can(sr2,transport) can(sr2,move) robot(sr1) robot(sr2)
    ↪ robot(sr3) task(t1) task(t2) task(t3) type(t1,search)
    ↪ type(t2,transport) assign(t1,sr2) assign(t2,sr1)
    ↪ assign(t3,sr3)
-can(sr3,move) -can(sr3,transport) can(sr1,move)
    ↪ can(sr2,transport) can(sr2,move) robot(sr1) robot(sr2)
    ↪ robot(sr3) task(t1) task(t2) task(t3) type(t1,search)
    ↪ type(t2,transport) assign(t1,sr1) assign(t2,sr2)
    ↪ assign(t3,sr3)
```

Listing 7.4: Task Assignment Example: The two complete Answer Sets

The two complete answer sets are shown in Listing 7.4 and need to be interpreted in order to retrieve the actual assignment of tasks to robots. The interaction with Clingo through our framework is based on queries which, among other things, facilitate the interpretation of answer sets. How this is achieved, in case of a simple filter query is described in the next section.

7.2 Filter Queries

Following the definitions from Section 4.3.1 and 4.3.2, an answer set is a set of literals that are deduced to hold under the answer set semantics of a given answer set program. As shown in the last example from Section 7.1, at least it is necessary to filter those answer sets for the literals that are of a particular interest. In the introduction of this chapter, we mentioned that we understand a dynamic knowledge base as a central module that gets queried from all kind of modules in the system of an agent. Therefore, which literals are of particular interest depends on the module that is querying the knowledge base. So in the simple case of a filter query, it is not necessary to change the knowledge in the knowledge base, but only to filter the set of deduced literals.

Our proposed solution works with a set of template literals that are part of the filter query itself. These template literals need to be matched to the literals given in the answer set. The elegance in our matching solution is that we are not comparing strings, but work with the native Clingo data structures and are still able to extend these data structures with a wildcard functionality on the template side, by handling a special wildcard term differently during matching.

```
% Answer Set
a(1) b(1) c(2) d(3) a(b(3)) a(b(3),b(b(2))) b(1,a(b(3)))

% Template Literals
a(wildcard) b(wildcard) b(1, wildcard)
```

Listing 7.5: Filter Query Example

The answer set given in Listing 7.5 demonstrates that a literal can also include composite terms of any nesting level and arity. Our filter algorithm matches the wildcard term to any kind of term, independent from its arity of nesting level. Therefore, in this example, `a(wildcard)` matches to `a(1)` and `a(b(3))`, but not to `a(b(3),b(b(2)))`, because here the arity of `a` does not match. For the same reason `b(wildcard)` matches to `b(1)` and `b(b(2))` and only `b(1, wildcard)` matches to `b(1,a(b(3)))`.

7.3 Extension Queries

The filter query can filter the given answer sets with flexible template-based filtering, but it does not change the knowledge in the knowledge base. However, there are use cases where it is required to not only filter specific literals but also to deduce them. The task assignment example from Listing 7.1 is an excellent example of such a use case. Filtering all tasks can be done with the filter template literal `task(wildcard)`, but, how is it possible to filter for all search tasks?

```
filteredTask(X) :- task(X), type(X, search).
```

Listing 7.6: Rule to Filter all Search Tasks

The rule in Listing 7.6 naturally captures this additional constraint on the type of tasks that need to be filtered. Straight forward we would add this rule to the knowledge base and afterwards use a filter query with the template literal `filteredTask(wildcard)`. Unfortunately, this would pollute our knowledge base with hundreds of useless or even contradicting filter rules on the long run, and as we are designing a knowledge base that is continuously running and updated, this solution is not acceptable. Nevertheless, it is necessary to add this rule in some form to the knowledge base, in order to deduce the `filteredTask(wildcard)` literal.

The solution provided by the extension query adds the additional rule only temporarily to the knowledge base. The key to achieving this is the Externals feature of Clingo, as described in Section 4.3.3.

```
#program query1.
#external extQuery1.
filteredTask(X) :- task(X), type(X, search), extQuery1.
```

Listing 7.7: Add Rules Temporarily with Externals.

In Listing 7.7 the additional rule is guarded by the external `extQuery1`, which is set to `#true` while the additional rule should be part of the knowledge base and is set to `#false`, otherwise. The encapsulation of the additional rule is done automatically by the extension query in a way that we can pass the original rule from Listing 7.6 to it. Further, the number of the external and program section is automatically increased for every new extension query (`query1`, `query2`, `query3`, ...), in order to make the queries independently controllable.

```
#program tasks.
task(t1).
type(t1, search).
task(t2).
type(t2, transport).
task(t3).
can(X, move) :- can(X, transport).
-can(X, transport) :- -can(X, move).

#program robots.
robot(sr1).
can(sr1, move).
robot(sr2).
can(sr2, transport).
robot(sr3).
-can(sr3, move).

#program query.
{ assign(X,R) : robot(R) } = 1 :- task(X).
{ assign(X,R) : task(X) } = 1 :- robot(R).
:- assign(X,R), type(X, transport), -can(R, transport).
:- assign(X,R), type(X, search), -can(R, move).
```

Listing 7.8: Repetition of Task Assignment Example

The use cases that the filter query and the extension query can address so far are all about filtering the answer sets of a given answer set program. Another use case that is going one step further in the direction of temporarily adding an additional rule is to add exceptions within the context of a query. We repeated the example from Listing 7.1 in Listing 7.8 for revisiting the example without going back and forth on the pages. In the example, Task t3 did not have any type assigned to it. In order to assume that it is also a search task, we need to add the fact type(t3, search) to the program section of the query in Listing 7.7. The extension query, as provided within this work, is even capable of temporarily adding an arbitrary number of additional rules within one query. Temporarily adding rules allows to scale seamlessly between two extremes: Either have everything added to the knowledge base and filter the results via a filter query, or start with an empty knowledge base and add everything via an extreme case of an extension query. This feature makes our integration of Clingo very flexible about the decision what can be considered as static domain knowledge and is therefore added to the knowledge base in advance, and what is depending on the current situation and is therefore added only temporarily. More details on this distinction and how this feature is used in the case of commonsense knowledge are given in Section 8.2.

A general problem that hinders the applicability of extension queries and even Clingo itself to the domain of dynamic knowledge bases is the requirement to fulfil the module property, as described in Section 4.3.4. However, we can mitigate this problem so that we are still able to add and remove knowledge in a reasonable way dynamically. According to the definition, given in Section 4.3.4, adding the fact type(t3, search) would violate the module property, because the type/2 literal is already used within the existing part of the knowledge base and the extension query is again using this part in the filter rule from Listing 7.7. Thus, adding this literal introduces a cycle between new and existing knowledge.

In order to break those cycles, the extension query automatically encapsulates all its facts and head literals. In case of the fact type(t3, search), this means that the rule query1(type(t3, search)) :- extQuery1. would be added to the knowledge base instead.

Algorithm 1 describes the necessary procedure for making all rules of an extension query safe with regard to the requirements of the module property. The input of this algorithm includes all rules from the extension query, and its output is the rules compatible with the module property. At first, in Lines 1 and 2, the query program section (query1) and the external (extQuery1) are created. In Lines 3 to 6, all head literals of the given rules are extracted and collected in a set of literals. These head literals represent the connection to the knowledge that is already in the knowledge base and are, therefore, the only way how the module property could be violated. In the following loop (Lines 7-16) of the algorithm, all rules get rewritten with respect to these head literals. Therefore, each occurrence of such a head literal gets replaced by its encapsulated version in Line 11 and the body of each rule is extended by the external in Line 14. So far, this alone would already guarantee that the extension query rules do not violate the module property. However, as the head literals got replaced in all the query rules, the body literals of the query rules cannot be deduced by the knowledge that already exists in the knowledge base. In order to allow

Algorithm 1: Automatic satisfaction of the Module Property.

```
Input   : Query Rules qr
Output: Extended Query Rules qrExtended
1  Program Section ps = createUniqueProgramSection()
2  External ex = createUniqueExternal()
3  Literals headLiterals = $\emptyset$
4  foreach Rule rule $\in$ qr do
5      headLiterals.addAll(extractHeadLiterals(rule))
6  end
7  foreach Rule rule $\in$ qr do
8      Rule extendedRule = rule
9      foreach Literal literal $\in$ extendedRule do
10         if literal $\in$ headLiterals then
11             extendedRule.replace('literal','ps(literal)')
12         end
13     end
14     extendedRule.body.add('ex')
15     qrExtended.add(extendedRule)
16 end
17 foreach Literal head $\in$ headLiterals do
18     qrExtended.add('ps(head) :- head, ex.')
19 end
20 return qrExtended
```

this one-way connection between knowledge from the knowledge base and knowledge from the extension query, the algorithm further adds one additional rule per head literal in Line 18.

The opposite connection, where the knowledge deduced from the rules in the extension query would fulfil body literals of rules in the knowledge base, needed to be cut off by this algorithm entirely in order to fulfil the module property. We consider this as a significant restriction of state-of-the-art answer set programming solvers like Clingo with regard to their usability in the context of a dynamic knowledge base because the knowledge represented by an extension query cannot be considered as input for rules already existing in the knowledge base. Only the knowledge that already exists in the knowledge base can be considered as input for the rules in the extension query.

```
#program query1.
#external extQuery1.
% query rule
query1(filteredTask(X)):-query1(task(X)),
    ↪ query1(type(X,search)), extQuery1.

% encapsulated facts
```

```
 7 query1(task(t1)) :- extQuery1.
 8 query1(type(t1, search)) :- extQuery1.
 9 query1(task(t2)) :- extQuery1.
10 query1(type(t2, transport)) :- extQuery1.
11 query1(task(t3)) :- extQuery1.
12 query1(robot(sr1)) :- extQuery1.
13 query1(can(sr1, move)) :- extQuery1.
14 query1(robot(sr2)) :- extQuery1.
15 query1(can(sr2, transport)) :- extQuery1.
16 query1(robot(sr3)) :- extQuery1.
17 query1(-can(sr3, move)) :- extQuery1.
18
19 % encapsulated additional rules
20 query1(can(X, move)) :- query1(can(X, transport)), extQuery1.
21 query1(-can(X, transport)) :- -query1(can(X, move)), extQuery1.
22 { query1(assign(X,R)) : query1(robot(R)) } = 1 :-
      ↪ query1(task(X)), extQuery1.
23 { query1(assign(X,R)) : query1(task(X)) } = 1 :-
      ↪ query1(robot(R)), extQuery1.
24 :- query1(assign(X,R)), query1(type(X, transport)),
      ↪ -query1(can(R, transport)), extQuery1.
25 :- query1(assign(X,R)), query1(type(X, search)), -query1(can(R,
      ↪ move)), extQuery1.
26
27 % knowledge base capturing rules
28 query1(-can(X1,X2)) :- -can(X1,X2), extQuery1.
29 query1(assign(X1,X2)) :- assign(X1,X2), extQuery1.
30 query1(can(X1,X2)) :- can(X1,X2), extQuery1.
31 query1(filteredTask(X1)) :- filteredTask(X1), extQuery1.
32 query1(robot(X1)) :- robot(X1), extQuery1.
33 query1(task(X1)) :- task(X1), extQuery1.
34 query1(type(X1,X2)) :- type(X1,X2), extQuery1.
```

Listing 7.9: Task Assignment Example Converted by Extension Query

Listing 7.9 shows the output of Algorithm 1 in case it had to convert the rules and facts from the task assignment example in Listing 7.8. Rule 4 is the converted version of the actual query rule, Rules 7-17 represent the facts in the original example, Rules 20-25 encapsulate the normal rules from the example, and finally Rules 28-34 are generated by Line 18 of Algorithm 1. As shown by this example, the blow-up of the number of rules for keeping the module property is only linear concerning the number of head literals of the extension query, because only one additional rule is added per head literal.

7.4 Knowledge Base

The three building blocks of the knowledge base that we described in the last sections are the integrated solver at its core, the interpretation of stable models via the filter queries, and the temporal extension of the knowledge base with the help of the extension queries. This section of the knowledge base chapter is focused on the interaction of these building blocks and thereby leads to the topic of the next chapter: How to teach robots that are equipped with this knowledge base. Teaching a robot can help them to handle unknown environments. Therefore, the knowledge base and its capabilities are essential to our approach for handling unknown environments (R2, Section 1.2.4).

As described Section 7.1, there are five steps during a usual interaction with the core of the knowledge base:

1. Modelling the knowledge with ASP rules
2. Composing knowledge rules to one non-grounded ASP program
3. Grounding the non-grounded ASP programs
4. Solving the grounded ASP program
5. Interpreting the resulting stable models

Steps 2-4 are directly interacting with Clingo inside the knowledge base and can be triggered via the knowledge base interface, as shown in Figure 7.2. This, for example, is the case when commonsense knowledge from the world model is inserted into the knowledge base (see Section 8.2).

Interpreting the resulting stable models is supported by the filter and extension queries, which can be registered at the query registry. The registry fulfils a common requirement of modules that query the knowledge base. Typically modules want to be notified on newly deduced literals, in order to check whether they match their template literals. Otherwise, they would need to poll the knowledge base, which is relatively inefficient. The query can further be registered with a lifetime. The lifetime can either be for any number of solving calls to Clingo or until the query is explicitly unregistered again. This feature especially allows for decoupling modules that add knowledge into the knowledge base and modules that process or get triggered by knowledge from the knowledge base. In case of the extension query, the truth value of the encapsulating external is set to `#false`, when the lifetime is 0 or the query is unregistered.

Filter and extension queries are both used by the operator and the world model. The queries internally use the knowledge base interface for their purposes, while the ALICA Engine is interacting with the knowledge base as with any other ALICA-compatible solver (see Section 6.2). Nevertheless, the ALICA solver interface uses the query registry internally, as well. Therefore, all interaction with the knowledge base falls back to steps 2-4, which are captured by the interface module of the knowledge base.

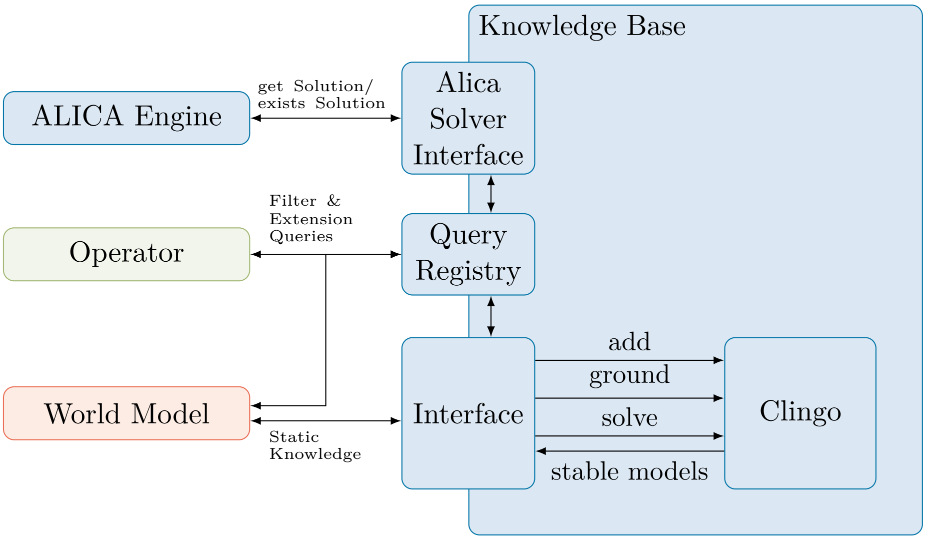


Figure 7.2: The Interaction with the Knowledge Base

Finally, looking at Figure 7.2 it is essential to note that the static knowledge, that is added by the world model, is not static at all. Instead, the term static denotes that this knowledge is always present in the knowledge base, but its truth value can be changed. Whether a service robot is in a particular room, for example, can be true or false, but depending on the application scenario, the statement about the relation between the service robot and the room can be added permanently to the knowledge and is therefore never unknown.

Teaching Robots 8

In an environment that is unknown to a service robot, it is imperative to learn about its environment (R2, Section 1.2.4). The knowledge base, as implemented for this thesis, offers diverse possibilities to add knowledge to it. Further, its symbolic implementation stores knowledge very similar to the way humans formulate their knowledge in natural language. Therefore, equipping a service robot with this knowledge base, paves the way for humans to teach the robot, for example, about the unknown environment.

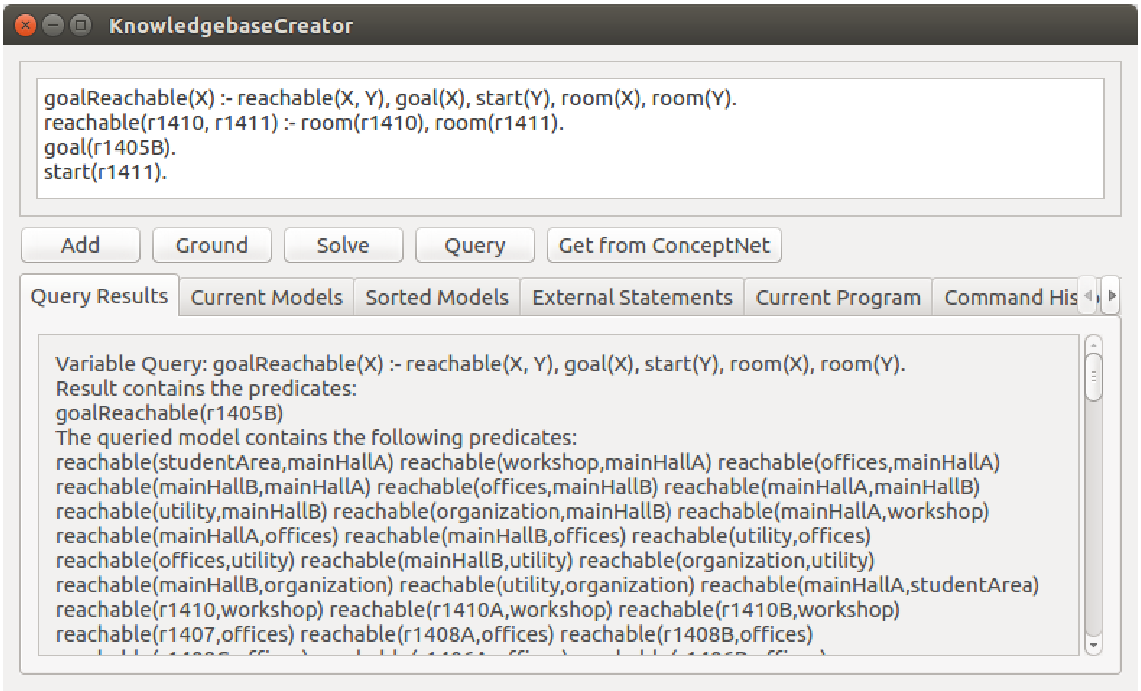
8.1 Human Teachers

Humans that want to teach a service robot, equipped with the knowledge base presented in Chapter 7, must use the interfaces of the knowledge base. In order to make this interaction natural, they need techniques as they are developed in the research area of human-robot interaction. One of the most natural ways for humans to communicate is to talk. Therefore, a speech recognition software such as Sphinx[1] coupled with a part-of-speech tagging system such as spaCy[2] would allow humans to communicate with the robots quite naturally. We consider the gap between the output of such a speech tagging algorithm and an ASP rule to be small compared to other knowledge representations. However, there is still a considerable amount of research to be done before we can automatically transform natural speech into ASP rules. While we do not directly address this research area in this work, Schwitter et al. [29, 23] presented proper solutions for the case of controlled natural language. Based on the referred research and algorithms, we assume that it is possible to convert parts of natural language into ASP rules already and therefore only provided a human user interface that expects already well-formed ASP rules as input.

Figure 8.1 shows this user interface denoted as knowledge base creator. Its supported features directly map to the query registry and knowledge base interface, as described in Section 7.4. With the knowledge base creator, it is possible to create and interact with a connected knowledge base. Humans, for example, can add rules to the knowledge base of a service robot that reflects their preferences for getting up, how they want their coffee, or they could also teach the robot the expected location of different items in the household in order to make the service robot clean up on its own. As such information depends on the user, a developer cannot program it into a service robot in advance, and as they may change over time and are subject to exceptions, it is complicated, for example, to apply

[1] CMUSphinx Speech Recognition Engines - https://cmusphinx.github.io/wiki/ [last accessed on June 29th, 2020]

[2] spaCy - https://spacy.io/ [last accessed on 29th, June 2020]



```
goalReachable(X) :- reachable(X, Y), goal(X), start(Y), room(X), room(Y).
reachable(r1410, r1411) :- room(r1410), room(r1411).
goal(r1405B).
start(r1411).
```

```
Variable Query: goalReachable(X) :- reachable(X, Y), goal(X), start(Y), room(X), room(Y).
Result contains the predicates:
goalReachable(r1405B)
The queried model contains the following predicates:
reachable(studentArea,mainHallA) reachable(workshop,mainHallA) reachable(offices,mainHallA)
reachable(mainHallB,mainHallA) reachable(offices,mainHallB) reachable(mainHallA,mainHallB)
reachable(utility,mainHallB) reachable(organization,mainHallB) reachable(mainHallA,workshop)
reachable(mainHallA,offices) reachable(mainHallB,offices) reachable(utility,offices)
reachable(offices,utility) reachable(mainHallB,utility) reachable(organization,utility)
reachable(mainHallB,organization) reachable(utility,organization) reachable(mainHallA,studentArea)
reachable(r1410,workshop) reachable(r1410A,workshop) reachable(r1410B,workshop)
reachable(r1407,offices) reachable(r1408A,offices) reachable(r1408B,offices)
```

Figure 8.1: The Knowledge Base Creator Interface

machine learning algorithms in this context.

8.2 Commonsense Knowledge

The previous section elaborated on how humans could teach a service robot if we equip it with the knowledge base presented in Chapter 7. However, the task of teaching a service robot everything that it needs to know becomes too complex and time-consuming, if we start with an empty knowledge base. Principally, humans are aware of commonsense knowledge per definition, and they expect it as a common ground during communication. Unfortunately, this is not the case for service robots. Thus, we integrated the commonsense knowledge database Concept Net 5 (CN5) [39] with our knowledge base, in order to relieve humans from teaching service robots commonsense knowledge and facilitate the human-robot interaction (R4, Section 1.2.4).

8.2.1 Concept Net

CN5 comprises extracted knowledge from projects and databases like WordNet, DBPedia, the Open Mind Common Sense project, Wiktionary, OpenCyc. Further knowledge was acquired through games, for example, from the project *Games with a Purpose* [95, 106]. The company behind CN5 is Luminoso Technologies[3]. CN5 itself is an open-source pro-

[3] Luminoso Technolgoies - https://luminoso.com/ [last accessed on July 5th, 2020]

ject, which is available for download in case of self-hosted solutions or accessible through a website[4] hosted by Luminoso Technologies. The original purpose of CN5 is the automatic understanding of natural language, which is also the reason why text comprehension frameworks use CN5 in corresponding challenges [21].

CN5 represents the multi-lingual knowledge as a semantic hypergraph whose nodes are denoted as concepts and whose edges are relations between these concepts. Nodes include information like a term in natural language, a language tag, and a sense label, which distinguishes between nouns, verbs, adjectives, and adverbs. The edges include one of the 34 different semantic relations of CN5, a weight, and the sources that support this relational knowledge. The edge weight is the sum of all weights that belong to the different sources, and it further holds that the higher the weight, the more the source can be trusted. A weight higher than 1, for example, indicates that the source is a verified source like WordNet.

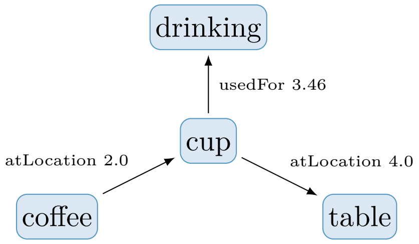


Figure 8.2: Excerpt from Cup-Related Knowledge in the CN5 Hypergraph

Figure 8.2 shows a subgraph of CN5, which includes some commonsense knowledge about cups. Translated back to natural language the shown subgraph says that coffee can be found in cups, cups are often located on tables, and cups are used for drinking. All three relations are annotated with their corresponding weights.

In addition to *usedFor* and *atLocation*, Table 8.1 also shows the 32 other relations and their meanings included in CN5. In total CN5 includes over 21 million relation instances and 1.5 million English concepts. Other knowledge graphs are compared in [45], but they are either not open source, or include similar sources of knowledge as CN5.

8.2.2 Representing Commonsense Knowledge in ASP

The commonsense knowledge from the CN5 database needs to be transformed into ASP rules before it can be added to the ASP-based knowledge base of a service robot. Further, commonsense knowledge is relatively static, because it is about concepts of things in the environment, like cups in general, and not about a specific thing, like an individual cup. However, when it comes to specific things, there are always exceptions to commonsense knowledge. Coffee, for example, can also be inside a glass, instead of a cup, although that

[4] Concept Net 5 - https://concept.io [last accessed on July 5th, 2020]

Relation	**Meaning**
RelatedTo	A is positively related to B (symmetric).
FormOf	A is an inflected form of B.
IsA	A is a subtype or a specific instance of B.
PartOf	A is part of B.
HasA	B belongs to A.
UsedFor	A is used for B.
CapableOf	Something that A can typically do is B.
AtLocation	A is a typical location for B.
Causes	A and B are events, and it is typical for A to cause B.
HasSubevent	A and B are events, and B happens as a subevent of A.
HasFirstSubevent	A is an event that begins with subevent B.
HasLastSubevent	A is an event that concludes with subevent B.
HasPrerequisite	In order for A to happen, B needs to happen.
HasProperty	A has property B.
MotivatedByGoal	Someone does A because they want result B.
ObstructedBy	A is a goal that can be prevented by B.
Desires	A is a conscious entity that typically wants B.
CreatedBy	B is a process or agent that creates A.
Synonym	A and B have very similar meanings (symmetric).
Antonym	A is the opposite of B in some relevant way (symmetric).
DistinctFrom	A and B are distinct member of a set (symmetric).
DerivedFrom	A is a word or phrase that appears within B and contributes to B's meaning.
SymbolOf	A symbolically represents B.
DefinedAs	A and B overlap considerably in meaning, and B is a more explanatory version of A.
MannerOf	A is a specific way to do B.
LocatedNear	A and B are typically found near each other (symmetric).
HasContext	A is a word used in the context of B.
SimilarTo	A is similar to B (symmetric).
EtymologicallyRelatedTo	A and B have a common origin (symmetric).
EtymologicallyDerivedFrom	A is derived from B.
CausesDesire	A makes someone want B.
MadeOf	A is made of B.
ReceivesAction	B can be done to A.
ExternalURL	Instead of relating to ConceptNet nodes, this pseudo-relation points to a URl outside of Concept Net, where further Linked Data about this term can be found.

Table 8.1: All 34 Relations Included in Concept Net 5[5]

is uncommon. Therefore, commonsense knowledge added to the knowledge base statically must allow for exceptions.

As a first step, the knowledge needs to be queried from the database of CN5. For this, we provide six different query types with parameters for concepts and relations, as shown in Table 8.2. Each query retrieves a set of edges from the CN5 database, which then can be translated.

Query	**Returned edges**
Concept	Edges connected to the given concept.
Wildcard(Concept,Concept)	Edges containing both concepts.
Relation(Wildcard,Concept)	Edges with the given relation and pointing to the given concept.
Relation(Concept,Wildcard)	Edges with the given relation and starting from the given concept.
Relation(Concept,Concept)	The defined edge, if it exists in CN5.
Relation(Concept)	Edges containing the given relation and concept.

Table 8.2: Query Types for Retrieving Commonsense Knowledge from CN5

In order to create a CN5 query, the occurrences of *Concept* and *Relation* need to be replaced with concrete concepts and relations, respectively. *AtLocation(cup, table)*, for example, is instantiated from the query *Relation(Concept, Concept)*. Fortunately, CN5 supports wildcards similar to the wildcard literals of the filter query described in Section 7.2. *Wildcard(cup, table)* results in a set of all edges that point from the concept cup to the concept table. Finally, the unary query *Relation(Concept)* is a shortcut for the union of edges retrieved by *Relation(Concept, Wildcard)* and *Relation(Wildcard, Concept)*.

```
#program commonsenseKnowledge.
cs_CapableOf(cup,hold_liquids,90).
cs_AtLocation(cup,table,4).
cs_AtLocation(coffee,cup,2).
cs_AtLocation(cup,shelf,2).
cs_UsedFor(cup,drinking_out_of,1).
...
```

Listing 8.1: Extract of Commonsense Knowledge about Cups.

The commonsense knowledge, which is represented by the retrieved CN5 edges, is converted into ASP facts, as shown in Listing 8.1. Each relation is prefixed by `cs_` in order to distinguish commonsense knowledge from knowledge about specific things, as described in Section 8.2.3. The transformation into ASP facts preserves the weight of the CN5 edges as the third argument of the relation literal. The weights allow us to define priorities, for example, among different *atLocation* edges. In Listing 8.1, the most likely location of a

[5] https://github.com/commonsense/conceptnet5/wiki/Relations [last accessed on July 21th, 2020]

cup is a table, followed by a shelf. Whenever some background knowledge about newly encountered things is required, the CN5 database is queried, and the retrieved commonsense knowledge is added to the `commonsenseKnolwedge` program section.

The presented encoding in ASP rules does not allow to retract commonsense knowledge from the knowledge base, and the program section is monotonically growing. Over time, this might lead to inconsistencies, because the CN5 database is not semantically consistent and commonsense knowledge taught by a human can contradict itself. Therefore, we make use of the antonym relation of CN5 and thereby handle semantic inconsistencies explicitly through the rules that apply commonsense knowledge to the current situation of the service robot [5].

8.2.3 Applying Commonsense Knowledge

As explained in the last sections, we can query commonsense knowledge from the CN5 database and insert it into the knowledge base of a service robot by transforming it into a set of ASP facts. Now that this commonsense knowledge is available in the knowledge base, the service robot needs to make use of it. For this, it is essential to distinguish between three different abstraction levels in the knowledge base: commonsense, situational, and sensor knowledge. Commonsense knowledge does not need any further explanation (see Section 8.2.2). Sensor knowledge is our term for the input given by frameworks like YOLO [22] or the Google Vision API [28]. Generally speaking, sensor knowledge is the output of the object recognition module of the service robot, but note that humans can also be the source of such information [65].

```
#program commonsenseKnowledge.
cs_AtLocation(cup,table,4).
...
#program sensorKnowledge.
#external is(blueCup,cup).
#external is(kitchenTable,table).
...
#program situationalKnowledge(n,m).
#external -atLocation(n,m).
atLocation(n,m,W):- not -atLocation(n,m),is(n,cup),is(m,table),
    ↪ cs_AtLocation(cup,table,W).
...
```

Listing 8.2: Three Different Kinds of Knowledge

Listing 8.2 shows how our knowledge base represents sensor knowledge. All sensor knowledge is grouped in the `sensorKnowledge` program section, and the assignment is done by the `is/2` literal. Please note, that, just for this example, `blueCup` and `kitchenTable` are human-readable identifiers for individual objects in the environment. Usually, these are just unique numbers that identify recognised objects. Further, sensor values can often change, which is why the assignment of an object type to a specific object is modelled as an external. This way, it is possible to switch the truth value for the assignment of an object type to an object on demand.

Situational knowledge is more about combining commonsense knowledge and sensor knowledge in order to apply default reasoning while still taking exceptions into account. The `situationalKnowledge` program section in Listing 8.2 has two parameters that need to be set to two specific objects when grounding this program section. The objects `blueCup` and `kitchenTable`, for example, could be used to instantiate the parameters `n` and `m` for grounding, respectively. In that case, it is deduced that the `blueCup` might be on the `kitchenTable` only if it is not explicitly stated that this is not the case. By setting the external in Rule 9 to true, for example, it can be stated that the `blueCup` is not on the `kitchenTable`.

After the explanation of the different types of knowledge in the knowledge base, we will give some examples of how the knowledge base can be used for different tasks.

Example: Location of a Cup

Imagine the human owner of a service robot asks the robot to bring her some cup. If the robot already knows about the location of a cup, it does not need to query its knowledge base and can directly bring it to her. If the robot knows about a cup but does not know its location, the robot can query the knowledge base. Therefore, it needs to ground the `situationalKnowledge` program section with this cup and any other object that it knows. This only needs to be done once, and the result of the grounding will persist in the knowledge base afterwards. Finally, it can query its knowledge base with the template literal `atLocation(knownCup, wildcard, wildcard)` and even add an optimisation statement for retrieving the most likely location according to CN5.

Example: Searching for Cups

In another case, where the robot does not know about any cup, it still can make use of the commonsense knowledge and query for potential locations of cups in general with the template literal `cs_AtLocation(cup, wildcard, wildcard)`. The difference, compared to the latter example, is that in this example it is possible to add an exception to the query that states that, for example, there are no cups on a particular table: `-atLocation(X,kitchenTable) :- is(X,cup)`. This exception, which might be added by the human owner of the service robot, avoids that the body of Rule 10 does hold for any cup on the `kitchenTable`.

Example: Searching for Cups - Extended

No matter whether the service robot knows about a specific cup or not, in case it does not know any table or kitchen shelf, the simple examples before will fail to guide the robot to a proper location. In such cases, the robot can keep on querying the knowledge base for locations of the retrieved concepts, until it either queried the complete transitive hull of the *atLocation* relation in CN5, or it encountered some concept of which it knows an instance with a corresponding location. Such a sophisticated interaction with the knowledge base allows the robot to get the most out of its knowledge and improves its search for objects in the household.

Knowledge-Based Cooperation 9

Interaction between humans and service robots is one aspect that motivates this thesis and the knowledge that we provide to the service robots, as described in the last chapter, facilitates this interaction. However, in addition to the interaction with humans, service robots can also share their knowledge to cooperate with each other. For this knowledge-based cooperation, different communication protocols and patterns are necessary than those supported by the communication middleware in previous ALICA application scenarios such as robot soccer. Hence, we developed our own middleware (see Section 9.1) that overcomes the limitations of the middleware of the Robot Operating System and borrowed communication patterns from common agent communication languages and speech act theory (see Section 9.2).

9.1 Communication Middleware

Sharing knowledge is a form of cooperation, and for sharing knowledge, service robots need to communicate with each other. In the former ALICA application scenarios and for ALICA itself, the middleware of the Robot Operating System (ROS) was used. We explain how the ROS middleware works and why it does not fulfil the requirements of the domain of domestic service robots in Section 9.1.1. As a result, we created a new communication middleware, called Cap'n Zero (see Section 9.1.2), that fulfils the demand for a distributed and flexible message-based communication system.

9.1.1 Limits of the Robot Operating System Middleware

ROS is commonly used in robotic research and therefore, is supported by a large community of researchers and practitioners. The ROS middleware is written in C++ and designed for interprocess communication in a single robot application. It is message-based and follows the publish-subscribe communication pattern.

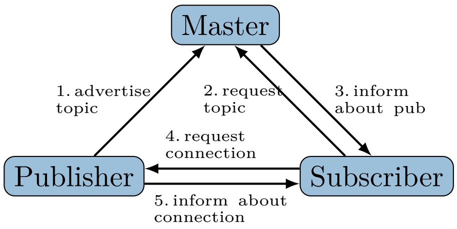


Figure 9.1: Establishment of the Connection Between two ROS Nodes

In Figure 9.1, the procedure for establishing a connection is shown. The Master node is the central registry of the ROS ecosystem. Without the Master, publishers and subscribers are not able to communicate, and as such, the Master is a single point of failure. The procedure in Figure 9.1 utilises XML/RPC as communication protocol, but for the actual communication between publishers and subscribers, ROS utilises raw TCP connections. In case of communicating between different computers, among further efforts, the address of the Master must be apriori known to every publisher and subscriber and must be reachable at all times. These properties make the ROS middleware inapplicable to the domestic service robot domain, where robots can leave the communication range of each other or of any central communication infrastructure.

In former ALICA applications, like robotic soccer, this issue was addressed by a simple UDP proxy that sends relevant ROS messages of the local robot to all others via multicast communication. Another instance of the same proxy, running on a remote robot, receives the ROS messages and injects them back into its local system. As this solution only offers UDP multicast, it is only suitable for scenarios where exchanged knowledge is always relevant for all systems in communication range, i. e. it is not suitable for the domain of domestic service robots.

9.1.2 Cap'n Zero

The name Cap'n Zero stems from the fact that it is a combination of the serialisation library Cap'n Proto[1] and the transport library ZeroMQ[2]. As such, Cap'n Zero avoids to reinvent the wheel and relies on mature open source libraries that provide implementations in ∼18 different languages, among which are C++, Java, and Python. Cap'n Zero itself glues Cap'n Proto and ZeroMQ together within 315 lines of C++ code.

Like the ROS middleware, Cap'n Zero provides publishers and subscribers. The main difference is that they do not need a central registry to establish a connection between each other and that connections are based on addresses, instead of topics.

Table 9.1 summarises the differences between ROS and Cap'n Zero. The publishers and subscribers of Cap'n Zero offer more flexibility than those of ROS, because of their ability to send and receive to and from multiple addresses and send and receive arbitrary message types. Especially, to receive and distinguish different message types is only possible, because Cap'n Proto provides message reflection at runtime. A comparison between ROS and Cap'n Zero regarding their performance is given in Section 12.4.

[1] Cap'n Proto - https://capnproto.org/ [last accessed on October 24th, 2020]

[2] ZeroMQ - https://zeromq.org/ [last accessed on October 24th, 2020]

	ROS Pub/Sub	Cap'n Zero Pub	Cap'n Zero Sub
Addresses	localhost only	sends to multiple addresses at once	receives from multiple addresses
Message Types	one ROS message type	arbitrary Cap'n Proto message types	arbitrary Cap'n Proto messages types
Topics	one topic	sends each message to a different topic	one topic
Protocols	one protocol at once (TCP, UDP)	only one protocol at once (IPC, TCP, UDP Multicast)	only one protocol at once (IPC, TCP, UDP Multicast)

Table 9.1: Comparison between Cap'n Zero and ROS

9.2 Speech Act Performatives

After we explained the technical aspect of our knowledge-based cooperation in the last section, i. d. the communication middleware Cap'n Zero. We focus on the semantics of our speech act and dialogue system, by pointing out similarities and differences to common agent communication languages.

An agent communication language, as the research area of multi-agent systems defines it, is a *lingua franca* for agents. Two famous attempts to standardise the communication languages between agents are the Knowledge Query and Manipulation Language (KQML) [138] and its follow up, the Agent Communication Language of the Foundation for Intelligent Physical Agents (FIPA-ACL) [129]. Both follow cooperative communication protocols that use performatives based on the theory of speech acts [160]. However, the FIPA-ACL standard is criticised for its cognitive semantics, which requires a specific behaviour from the agent when it receives or sends a corresponding speech act performative. Further, researchers argue that an open agent communication language should follow social semantics instead of cognitive semantics and that commitments between the communicating agents are a crucial element of such potential language [67, 132]. For the same reason, the common agent framework Jade [125] only labels its messages with FIPA-ACL conformant performative names, but cannot implement the corresponding mental semantics.

In contrast to Jade, the presented work only supports three speech act performatives, and their names do not imply any compliance to the FIPA-ACL standard, although similar names are used. At first, there is the *Command* performative, which humans use to command a service robot to do something. The utterance "bring me a cup", for example, will be represented as a speech act message of type *Command* and the receiving service robot, as implemented in the service robot demonstrator (see Section 11.2) of this thesis, will start to search a cup and bring it to the demanding human.

The *Query* performative is another supported performative, that can be used by humans and service robots as well. It expresses a query to the knowledge base of a service robot and, for example, can be used to ask whether a service robot knows the location of a cup. The service robots in the service robot demonstrator use this performative for asking each other about the locations of cups they are searching. The expressiveness of the query performative is the same as of those queries supported by the knowledge base described in Chapter 7 and forms the basis for our knowledge-based cooperation between robots.

The last of the three supported performatives is the *Inform* performative. It is used by humans to teach service robots, as described in Chapter 8. A human could inform a robot, for example, that the cups are stored under the sink. *Inform* is the only performative that directly changes the content of the knowledge base of a service robot. The *Query* performative only retrieves knowledge from the knowledge base, and the *Command* performative extends the knowledge base only indirectly, for example, when the robot is searching for a cup, it will remember the position of things it encounters during the search.

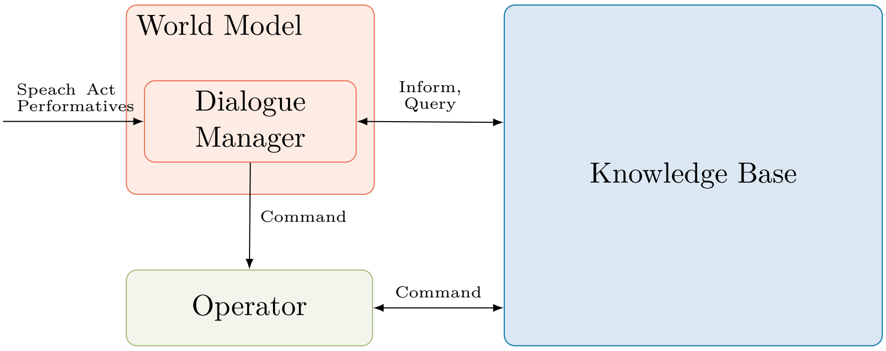


Figure 9.2: The Dialogue Manager Utilising the Knowledge Base

The handling of the speech act performatives is done by the Dialogue Manager, which is a sub-module of the world model. As shown in Figure 9.2, the *Command* performative is forwarded to the operator, which itself queries the knowledge base for the information that is necessary to fulfil the command. The dialogue manager itself handles the other performatives. Further, all speech act messages include a unique id and, if they are sent in response to another message, include the id of the former message. The relation between messages allows us to keep track of multiple dialogues with robots and humans at the same time.

The *Command* and the *Inform* performatives are received from humans, while robots themselves issue the Query performative. Our current implementation expects the description of tasks that are received as part of *Command* performatives, to mention the kind of knowledge that is relevant to solve the task. When a robot, for example, should bring some object, it is hard-coded that the robot asks other robots for the location of such objects. This limited flexibility exists because the robots do not reason about the meaning of "bringing something" and therefore do not relate the location property of objects with the task to transport them. In a similar line of thought, our current approach expects that a problem description also includes what kind of knowledge robots need to exchange in order to find a common solution for a cooperatively solved problem.

Related Work 10

In Section 4.2.1 of Chapter 4, we have already analysed common logical reasoning formalisms concerning their suitability for the domain of domestic service robots. In this chapter, an overview over related frameworks with the focus on practicable relevance will be given. We divide the chapter into three sections. The behaviour modelling frameworks, presented in Section 10.1, form the related work for the ALICA Framework. Afterwards, the knowledge representation and reasoning frameworks described in Section 10.2 have proven their applicability in practical domains utilising symbolic reasoning formalisms. Finally, Section 10.3 provides an overview of current approaches applying service robots with ASP-based reasoning capabilities.

10.1 Behaviour Modelling Frameworks

The PhD thesis of Skubch about the ALICA Framework from 2012 [77] already included a comprehensive discussion about related work. Therefore, within this section, we mainly focus on more current work that relates to the ALICA Framework as published in [13].

The Extensible Agent Behaviour Specification Language (XABSL) [107] was developed in the context of RoboCup Soccer Competitions, and just like ALICA, it utilises hierarchical finite-state machines (HFSM) to cope with the complexity of modelling sophisticated behaviours. It is also able to model behaviours for teams of robots, but its execution engine does not support teams in the sense of coordinating the team, assigning tasks, or exchanging solutions to domain-specific constraint problems. In contrast to ALICA it is, to some extend, able to exchange behaviours at runtime, because the behaviour specification language is mainly XML and interpreted at runtime. Recently, a new version of XABSL was published [36]: C-based Agent Behaviour Specification Language (CABSL). It follows the same approach as XABSL, but the behaviour specification language is C-based. As a result, the execution engine has fewer dependencies, has a smaller footprint, and supports all kinds of C datatypes, which allows referencing data structures from external C libraries directly. Among a set of minor shortcomings, the most significant disadvantage of CABSL compared to XABSL is that behaviours cannot be exchanged at runtime.

The Cognitive Robot Abstract Machine (CRAM) is a toolbox for the design, implementation and deployment of cognition-enabled autonomous robots [87]. One of its tools is the CRAM Plan Language (CPL), which is expressive enough to formulate single-agent behaviours, similar to BDI-based approaches (see Section 2.2.3). The advantage of CPL over classic BDI-based approaches is, according to the authors, that it features concurrency, action synchronisation, failure handling, loops and reactiveness. The ALICA Framework also includes all these features, supports teams of agents and offers a comfortable modelling tool, i. e. the ALICA Plan Designer. The modelling in CPL is, just like it is for the ALICA Framework, hierarchical and therefore allows for better scalability and reusability than non-hierarchical modelling approaches.

Live Robot Programming (LRP) is a nested-state-machine-based approach for modelling the behaviours of autonomous agents [7]. It allows changing the finite-state machine (FSM) at runtime, in order to give the developer direct feedback. Written in Smalltalk, the application of LRP in robotic domains demands for an extra effort to make LRP behaviour specifications interact with external libraries, which are often written in C/C++. Nevertheless, the live feedback for developers is an advantage that the ALICA Framework is missing. The ALICA Framework is designed for teams of autonomous agents that are typically executed on physically separated machines. Therefore, the concurrent execution, update and deployment of modified ALICA programs on distributed systems is a challenging task.

The Robotics and Mechatronics Center (RMC) of the German Aerospace Center developed a flowchart-based behaviour specification formalism, denoted as RMC Advanced Flow Control (RAFCON) [40]. According to the authors, the difference between state machines, as used in ALICA programs, and flowcharts, as used in by RAFCON, is that transitions in state machines are event-driven and transitions in flowcharts depend on the outcome of a state. Furthermore, the implementation of RAFCON is intentionally limited to a concise set of features, in order to avoid error-prone behaviour specifications that are hard to understand. Its applicability has been shown during the SpaceBotCamp 2015, it allows the exchange of behaviours at runtime, concurrency, and offers a graphical modelling tool. Unfortunately, it does not address teams of robots.

Kim et al. introduce a new approach to allow concurrency in hierarchical state machines. According to the authors, a hierarchical and concurrent finite state machine requires assumptions, that are theoretic and not suited for high-level behaviour coordination. Instead, the introduced inter-level concurrency loosens these requirements by running each state machine in a thread on its own. The new approach denoted as HFSM-IC does not allow to exchange behaviours at runtime and does not include graphical modelling, which, however, should be straight forward to implement. Finally, HSFSM-IC does not support teams.

Table 10.1 summarises the related behaviour modelling frameworks and their properties, except for the underlying runtime structure. Hierarchical finite (concurrent) state machines are used by ALICA, HFSM-IC [43], CABSL [36], XABSL [107], CPL [87], and LRP [7], while RAFCON [40] relies on hierarchical flow control. The remaining properties, as shown in Table 10.1, are *Exchange*, *Concurrency*, *Teams*, and *Tool*. Exchange considers the ability to change behaviours at runtime, which is supported by [40, 7, 107] since they rely on reflection and interpreted languages. Concurrency is the capability to run multiple

	Exchange	Concurrency	Teams	Tool
ALICA	✗	✓	✓	✓
CABSL[36]	✗	✗	✗	✓
XABSL[107]	✓	✗	✗	✓
CPL[87]	✗	✓	✗	✗
LRP[7]	✓	✗	✗	✓
RAFCON[40]	✓	✓	✗	✓
HFSM-IC[43]	✗	✓	✗	✗

Table 10.1: Comparison of Behaviour Specification Frameworks

behaviours simultaneously. Most frameworks support a graphical modelling tool. Finally, the explicit support for teams of agents is, among the presented frameworks, only provided by the ALICA Framework [13].

10.2 Knowledge Representation and Reasoning Frameworks

In this section about knowledge representation and reasoning frameworks, frameworks are discussed that have either shown their practical applicability in real-world scenarios similar to the domain of domestic service robots or have adopted a strong focus on symbolic reasoning in real world applications. During this review, the differences and commonalities between the work presented in this thesis and each framework are pointed out.

10.2.1 KnowRob

In Section 10.1, we mentioned the CRAM Plan Language, which is part of the CRAM Framework [44]. Like CRAM several other frameworks where developed by the Artificial Intelligence Research Group, lead by Michael Beetz. Among the frameworks are RoboEarth [85], openEASE [56], and KnowRob [14], which are all applied in the domestic service robot laboratory of the research group. Their research is focused on everyday household manipulation tasks in general and therefore addresses, in most cases only single robot applications.

Figure 10.1 shows a robot from the lab, making breakfast. Such complex manipulation tasks require enormous knowledge which is mostly commonsense knowledge from a human perspective. The framework that makes the required knowledge useable for the robot is KnowRob. The first version of KnowRob was developed by Tenorth [84]. The architecture of the second version is shown in Figure 10.2. The reasoning in the knowledge base of KnowRob is ontology-based. The utilised reasoner is implemented in Prolog and applies the closed world assumption, instead of the more commonly used open-world assumption. This design complies with the results of [70], which states that the open-world assumption is inappropriate in some robotic domains and in that case, it is even harder to close the

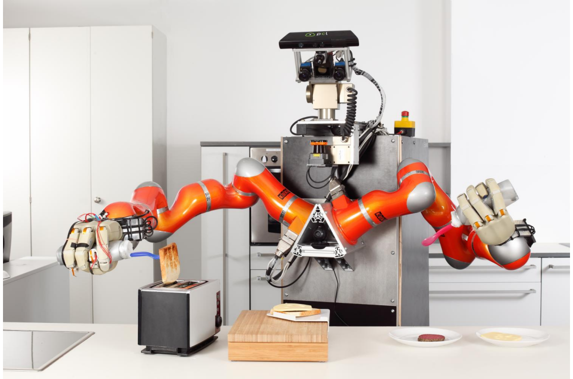


Figure 10.1: Service Robot Preparing Breakfast[1]

open world of common reasoners with additional axioms.

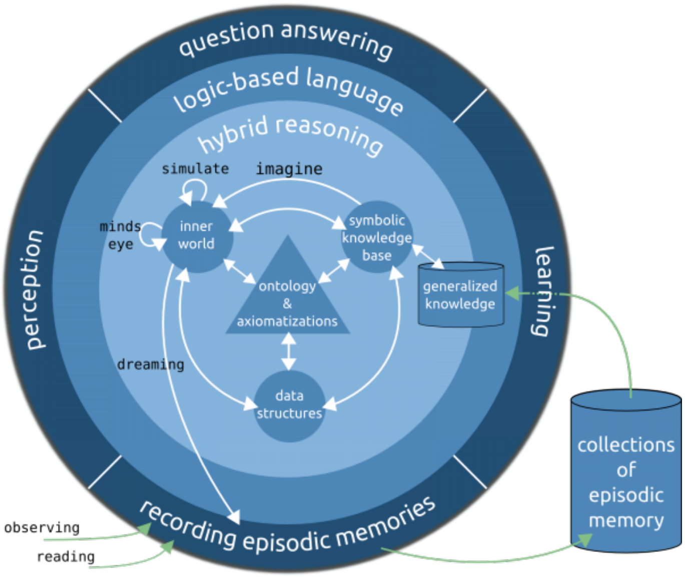


Figure 10.2: Architecture of KnowRob 2.0[2]

KnowRob also provides techniques for acquiring knowledge and grounding symbols. Especially the second version of KnowRob simulates the execution of tasks in a simulated environment in order to test the validity of planned action sequences. This approach also serves to ground symbols, following an understanding of symbol grounding that is different from the definition given in Section 2.1. The symbols in the knowledge base of the robot are part of the simulation, and the simulation is designed to be as close as possible to the real environment. The idea is that the symbols are grounded if the simulation is close enough to the real environment. The simulation is denoted as an *inner world* in

[1] https://techxplore.com - Pancake-making PR2 spells teachable future in robotics [last accessed on January 31th, 2020]

[2] http://knowrob.org [last accessed on January 31th, 2020]

Figure 10.2.

Another concept in KnowRob is that of computables. The Prolog-based ontology reasoning serves as glue logic between various problem solvers interacting with the knowledge base of KnowRob. A computable is an atomic symbol in the ontology that is calculated on demand. The truth value of the symbol `grasped(spatula)`, for example, is grounded through the feedback of the kinematic system of the arm of the robot. Like the general solver interface for ALICA described in Section 6.2, computables allow for the integration of arbitrary formalisms into KnowRob. Although the demonstrated capabilities of the robots equipped with KnowRob are very impressive and state of the art, the scenarios are relatively static and allow almost unlimited time for sensor data processing and reasoning. Therefore, the requirements for handling dynamic environments (R3) and facilitating human interaction (R4) from Section 1.2 are only partially addressed.

10.2.2 Artificial Cognition for Social Human-Robot Interaction

A series of research, which puts the requirement to facilitate human interaction (R4) from Section 1.2 into the focus, is done under the direction of Rachid Alami [31]. Similar to the research presented in the last section, several different frameworks are developed and integrated with each other, in order to make an autonomous service robot capable of interacting with humans, based on advanced cognitive capabilities.

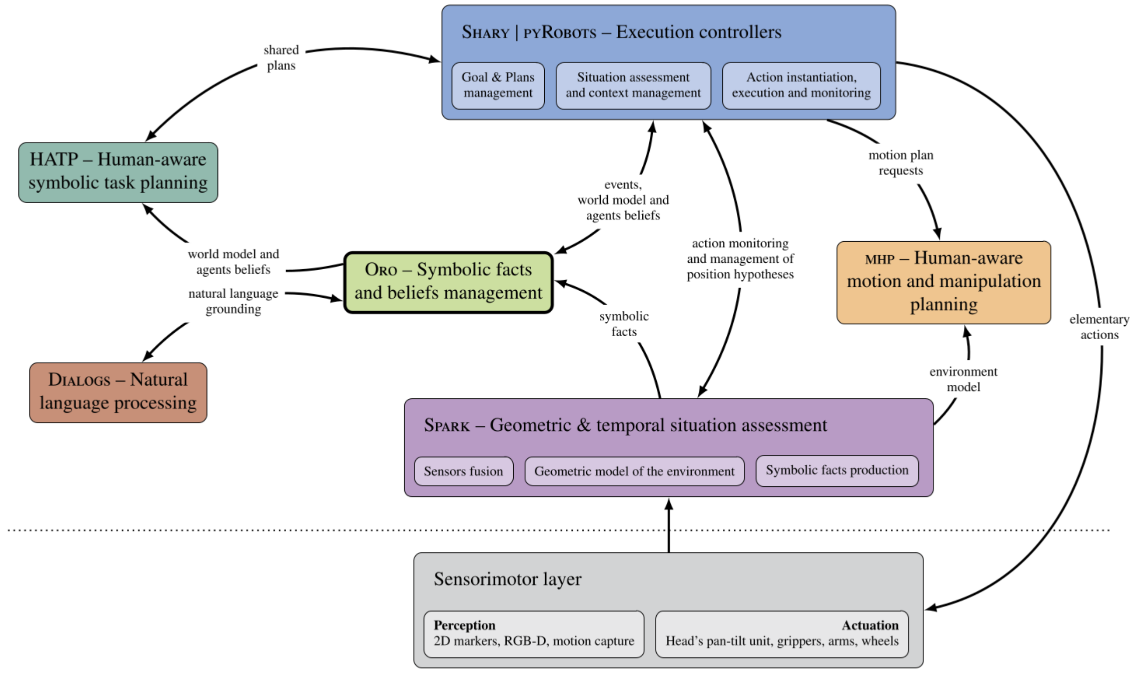

Shary | pyRobots – Execution controllers
Goal & Plans management
Situation assessment and context management
Action instantiation, execution and monitoring
shared plans
HATP – Human-aware symbolic task planning
events, world model and agents beliefs
action monitoring and management of position hypotheses
motion plan requests
MHP – Human-aware motion and manipulation planning
elementary actions
world model and agents beliefs
natural language grounding
Oro – Symbolic facts and beliefs management
symbolic facts
Dialogs – Natural language processing
environment model
Spark – Geometric & temporal situation assessment
Sensors fusion
Geometric model of the environment
Symbolic facts production
Sensorimotor layer
Perception
2D markers, RGB-D, motion capture
Actuation
Head's pan-tilt unit, grippers, arms, wheels

Figure 10.3: Architecture Overview of the Cognitive Framework by Alami et al. [31]

The architecture overview in Figure 10.3 shows the different components that provide the robot with these advanced cognitive capabilities. Like in most robot software architecture, a sensorimotor layer represents the interface between software and hardware. The commands sent to the sensorimotor layer are generated by execution controllers that belong to the two libraries Shary and PyRobots. Shary is the main controller and as such responsible for the control of the robot. Triggered through events from ORO, the symbolic facts and beliefs management of the framework, Shary supervises the creation of tasks, human-robot interaction, and recognition of human actions. This is done through the interaction of Shary with four human-centred components: Spark, MHP, HATP, and Dialogs. Spark is responsible for the assessment of the situation and therefore feed with sensor values from the sensorimotor layer. MHP creates human-aware motion and manipulation plans. HATP is responsible for more abstract, but also human-aware symbolic task planning. Finally, DIALOGS [75] is the natural language processing module of the architecture.

Central to the architecture and of particular interest from our perspective is ORO [89], the symbolic facts and beliefs management. It is realised as a server with JSON-based API. The knowledge is represented in the form of RDF triples following the semantics of the description logic $\mathcal{SROIQ}(\mathcal{D})$, also known as the semantic underpinning of OWL 2. Our assessment of advantages and disadvantages of utilising description logics as knowledge representation and reasoning formalism, as described in Section 4.2.1, are shared by the authors of ORO [31, Section 2.2.1] and therefore further supports former results [108, 19]. The authors also explicitly state that the monotonicity of OWL 2 is problematic and that whenever a single RDF triple in the knowledge base is changed, the whole ontology needs to be reclassified. Further, the size of the utilised ontology, the performance of the ontology reasoner, and the dynamic of the represented knowledge (requiring a high reclassification frequency), create recognisable delays in the reaction of the robot.

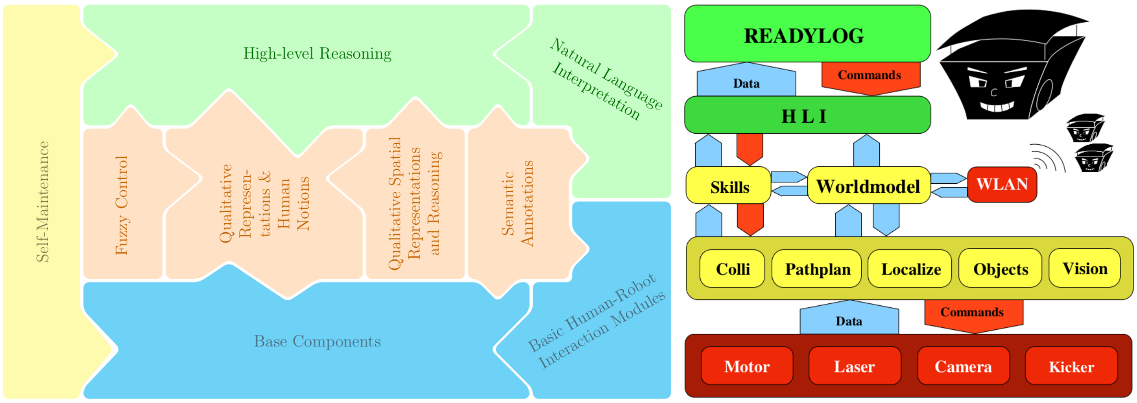
There are three sources of knowledge for ORO. The processing of sensor values, the created human-aware plans, and commonsense background knowledge. The latter one is considered to be static and includes existing ontologies like OpenCyc, WordNet, DBPedia, or RoboEarth. In order to interact with humans, a large amount of commonsense knowledge is necessary, and therefore, the reuse of large ontologies is an acceptable reason to choose description logics as knowledge representation mechanism. However, due to the advantages of ASP over description logics, we made the knowledge represented in ConceptNet 5 available to our ASP-based knowledge base, as described in Section 8.2. Furthermore, ConceptNet 5 already includes knowledge from most of the aforementioned ontologies and several results on combining ontologies with ASP reasoners exist [98, 118, 90, 35].

10.2.3 ReadyLog

The Knowledge-Based Systems Group, under the direction of Gerhard Lakemeyer, has a strong research history in real-world robot applications. Among others, they successfully participated in international robotic competitions in the domain of domestic service robots [76], robotic soccer [103], and logistics. In order to provide their robots with cognitive capabilities, they utilise the logic programming language Golog (alGOl in LOGic) [133], which is based on the Situation Calculus [122, 102]. The Situation Calculus is defined in terms of First-Order Logic axioms and is designed to describe the change of situations through the execution of actions. Starting from a single situation S_0, the application of an action a_0, denoted by $do(a_0, S_0)$, induces the new situation S_1. Planning in the Situation Calculus is regression-based, and a plan is a chain of the form $do(a_i, do(a_{i-1}, do(a_{i-2}, \ldots do(a_{i-i}, S_0)))$. The effort for evaluating an individual property of the world depends on the operating time of the robot because the chain of actions increases over time. In order to reduce this effort to only small conditions at specific decision points, Golog combines the declarative Situation Calculus with imperative control statements like loops and if-then-else constructs.

The experience gained due to the application in real-world scenarios induced several improvements of the knowledge representation in case of highly dynamic domains and many Golog dialects where developed. One of Lakemeyer's PhD students, Alexander Ferrein, presents ReadyLog in his thesis [103]. ReadyLog is a combination of several Golog dialects into one framework. ReadyLog, therefore, includes features to model concurrency, continuous change, probabilistic actions, and passive sensing. Especially the latter feature relates to our work because it shares the same motivation. Ferrein states that "When sensor values must be updated very frequently, acquiring world information through sensing actions is not feasible. The agent is busy with executing sensing actions most of the time" [103, Section 4.2.2]. This reason is one among others, which lead to the "passive sensing" approach, which moves the task of continuous sensing the environment to a parallel thread and enables the reasoner to run independently. As a result, it is necessary to have an explicit world model, which is updated by low-level sensor values continuously. The reasoner queries the most current values from the explicit world model on demand. Although developed independently, our architecture, as described in Chapter 5, follows the same approach.

In Figure 10.4, two robot software architectures of the Knowledge-Based Systems Group are depicted. Figure 10.4a shows the different modules of their service robot architecture, which includes similar components as the architecture in Figure 10.3. While the service robot architecture only sketches the interaction between its components, the soccer robot architecture in Figure 10.4b clearly shows a three-layered design. Similar to our architecture in Chapter 5 and the three-layered architectures, mentioned in Section 2.2, the



(a) Domestic Service Robot Architecture [37]

(b) Soccer Robot Architecture [103, Chapter 6]

Figure 10.4: Robotic Architectures from the Knowledge-Based Systems Group

lowest layer (red) is concerned with sensorimotor control of the robotic hardware. The processing and abstraction of data are done by components in the middle layer (yellow) and updates a central world model that is partially synchronised with other teammates. At the highest abstraction layer (green), ReadyLog interacts with the rest of the architecture through a ReadyLog-specific high-level interface (HLI). Commands are then again sent and processed downstream through the architecture to the sensorimotor layer.

The focus of ReadyLog is on actions and planning, and as such, it is more concerned with the high-level control of a robot, instead of providing a general knowledge base. This approach represents, to some extent, the opposite of the approach followed in this thesis. Regarding its purpose, ReadyLog is equivalent to the ALICA Framework, but ReadyLog is logic-based, enriched with imperative control elements, and queries on a world model, while the ALICA Framework follows an imperative approach, has operational semantics, and is enriched with declarative constraints, which could be formulated, for example, in ASP. Nevertheless, according to Lee and Palla [74], it is also possible to formulate the Situation Calculus, as well as the Event Calculus [126], in ASP. Following this approach would mitigate the lack of non-monotonicity, but the continuously growing runtime due to the growth of recursions, like this $do(a_i, do(a_{i-1}, do(a_{i-2}, \ldots do(a_{i-i}, S_0)))$, over time, is a severe disadvantage for long-term operating robots, such as domestic service robots.

10.2.4 Flux

Flux [114, 111] is an agent description language that is, like ReadyLog, focused on actions and resulting consequences. Its formal semantics is based on the Fluent Calculus [131], a variant of the Situation Calculus that remedies the disadvantage of growing recursions of actions applied to situations. This is achieved by a progression-based planning approach, which starts the search for plans at the current situation, independent from the history of former situations. In [91], an approach for solving the symbol grounding problem, based on the Fluent Calculus is presented, and the Fluent Calculus also provides solutions to the Frame, Ramification, and Qualification Problem (see Section 4.2).

The Flux Framework and the underlying Fluent Calculus was developed by Michael Thielscher, which changed his research focus to the domain of General Game Playing. Therefore, no implementation of the Flux Framework geared towards robotic applications is available anymore. Nevertheless, it was shown that a real Flux-based robot could deliver packages in an office-like environment. However, Fichtner et al. stated that "robots which follow this approach in highly dynamic environments would be overwhelmed with constantly calculating all changes that happen around them." [117, Section 2.2]. As a partial solution to this problem, information was automatically dropped after a certain lifetime. About the assessment of reasoning formalisms in Section 4.2.1, it is noteworthy that the Fluent Calculus, like the Situation and Event Calculus, is based on a decidable fragment of First-Order Logic, but is nevertheless monotonic. Therefore, the reasoner must restart its deduction process after every change of the knowledge base. Finally, there is no explicit support for teams of robots within the Flux Framework.

10.3 Applications of Answer Set Programming in Robotic Scenarios

The assessment of related work in the following paragraphs is partially published in [12]. It includes a discussion of ASP-based solutions for human-robot interaction and domestic service robots that often utilise a source of general and commonsense knowledge.

Erdem et al. present in [72] a hybrid planning approach utilising ASP, Prolog, ConceptNet 4 (CN4), and a continuous motion planner. Thereby, ASP is used to model the task of tidying a house, including possible actions of robots. The commonsense knowledge is extracted from CN4 and handled as ASP predicates represented as external predicates within a Prolog program(not to be confused with the `#external` keyword of input language of the ASP solver Clingo). Only two relations are utilised from CN4: The first relation is *AtLocation*, denoting possible locations of an object in a given room. The second relation is *HasProperty*, describing, e. g., the fragility of objects. Both relations are then used to formulate queries to ConceptNet, resulting in possible locations of objects and information about which object has to be treated carefully. The extracted commonsense knowledge then influences the presented continuous motion planning.

Apart from more current versions of ConceptNet and the ASP solver Clingo, our approach has a lot in common with the work of Erdem et al. Nevertheless, our approach integrates the commonsense knowledge extracted from the CN5 directly into the ASP knowledge base of the robots, instead of representing it as external Prolog predicates.

Prolog is not pure declarative and an unnecessary additional formalism, therefore utilising only ASP reduces the complexity of the system. Furthermore, with our approach, it is possible to retract the commonsense knowledge from the ASP knowledge base, in case it causes inconsistencies or is overwritten by human input. Erdem et al. rely on only two commonsense knowledge relations and restrict their queries to a sufficiently small number of answers in order to avoid inconsistencies. Finally, our focus is geared towards human-robot interaction, in the sense that the reasoning system, presented in this work, allows humans to interact with the knowledge base of robots and to teach robots temporary exceptions valid only for specific queries or tasks.

The goal of Lu et al. [32] is to make robots understand instructions formulated in natural language in the context of task planning. Therefore, they extract objects and verbs with the help of the semantic dictionaries Frame-Net and OMICS and transform the instructions into their own meta-language. Taking the expressions in their meta-language as input, they propose an automatic transformation into ASP rules. The purpose of the ASP solver is to generate action plans based on domain-specific background knowledge. Although Frame-Net is, compared to ConceptNet, a relatively small dictionary, they were able to significantly improve the number of solved tasks compared to the state-of-the-art OKPlanner [66].

Both mentioned approaches [72, 32] utilise an intermediate representation of the commonsense knowledge and handle it to some extent, separated from the ASP problem specification. Our understanding of the ASP solver and its problem specification is that of a continuously running knowledge representation and reasoning module, that is available to all modules of the robotic control architecture at all times. Therefore, we actively deal with adding and retracting parts of the knowledge from the knowledge base, instead of restarting the solving process for each problem instance over again.

Since, we are concerned about human-robot interaction, the application of ASP in natural language processing is of particular interest, too. The work of Chen et al. [88, 58] explicitly address human-robot interaction by processing limited segments of natural language. For the automatic translation into ASP, the language segments are restricted to if-then sentences. In contrast to our approach, other forms of commonsense knowledge cannot be added at runtime. In order to avoid inconsistencies during long-term human-robot interaction, Chen et al. make the truth values of all ASP rules depend on a monotonically increasing and discrete time step.

The body of research from Schwitter et al. [29, 23] investigates the automatic translation of natural language into ASP rules and back to natural language. In their approach, they use a bidirectional grammar for constraint natural language (CNL). Within this CNL, their approach is even capable of resolving anaphora. The intent of the authors is to allow domain experts without knowledge about ASP to investigate and create ASP programs. This supports our conclusion that ASP is the right choice for representing knowledge with regard to Rule 5 in Section 4.1.2 and allows for human-robot interaction in the domain of domestic service robots.

PartIII

Assessment

Demonstrators 11

In addition to conducting experiments for several separate components, we also developed two more complex demonstrators. These demonstrators allow evaluating the performance of the system as a whole. At first, there is the Wumpus World demonstrator which focusses on the knowledge-based cooperation between agents. The second demonstrator addresses the main motivating scenario for this thesis - domestic service robots. In the following sections, we explain the setup and mechanics of these two demonstrators, which prepares the discussion of the experiments in Chapter 12.

11.1 Wumpus World

The Wumpus World is based on the 70s computer game *Hunt the Wumpus*, and it is a common toy scenario for intelligent agents [47]. The world, in this scenario, is a quadratic grid of underground caves, where adjacent caves are connected. In the beginning, the agent enters a random cave through a ladder, by which it is also required to leave the world after finding and grabbing the gold. The agent is controlled through an interface, which allows to move the agent, perceive the environment and manipulate certain things in the environment.

Apart from the agent and the gold, the world also includes the wumpus. The wumpus is a monster, which never leaves its own cave, but it eats the agent right away when the agent enters its cave. Luckily for the agent, the wumpus stinks in such a way that the agent can smell it in all adjacent caves. Other dangers for the agent are trap doors that are spread all over the world. Again, the agent can perceive a trap door close by, because there is always a breeze in adjacent caves. The difference between a wumpus and a trap door is that the agent can shoot the wumpus. The agent has exactly one arrow that it can shoot in any direction. In case the wumpus got hit, the agent will hear it screaming before it dies and its stench will vanish.

The objective is to control the agent in such a way that it finds the gold, grab it, and leaves the world back through the ladder with as few actions as possible. Although being a toy scenario, the Wumpus World already offers enough complexity to properly evaluate knowledge-based cooperation between two agents that solve the Wumpus World as a team. Further, the Wumpus World is commonly known and facilitates the comparison of future approaches with the work presented in this thesis.

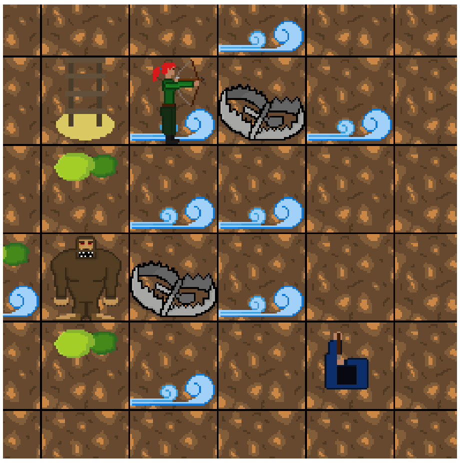


Figure 11.1: Part of a Wumpus World with Two Agents

11.1.1 The Classic Wumpus World

Figure 11.1 shows a part of a wumpus world example. In this example, already, two agents (blue and green) are present, which is an extension compared to the classic Wumpus World scenario. Before the extensions made for this thesis are described in Section 11.1.2, let us first formalise the rules and mechanics of the classic Wumpus World.

Action	**Effects**
Turn Left	Orientation of the agent changes 90° counter-clockwise.
Turn Right	Orientation of the agent changes 90° clockwise.
Move	Agent moves towards its orientation. Either enters an adjacent cave or bumps into a wall.
Shoot	Agent shoots its arrow towards its orientation. Either it hits the wumpus, which will scream and die, or it hits the wall.
Grab	Agent picks up the gold, if it is present in the cave of the agent.
Climb	Agent leaves the underground, if it is at the location of the ladder.

Table 11.1: Possible Actions of Wumpus Agents

The possible actions of agents in the Wumpus World are listed in Table 11.1. Each action has a specific effect and will change the current state of the world.

Matching the actions, Table 11.2 lists all possible perceptions of agents in the Wumpus World. The caves are rather dark so that the eyesight of an agent can only perceive the glitter of gold right in front of it. However, it cannot see what is inside an adjacent cave, nor can it perceive whether there is an adjacent cave at all. Therefore, the agent needs to explore the world and bumps into walls from time to time. Senses like hearing, smelling, and feeling work rather well in the dark and allow the agent to smell the wumpus close

Perceptions	**Conditions**
Breeze	The agent is adjacent to at least one trap door.
Glitter	The agent is inside the cave of the gold.
Stench	The agent is adjacent to the wumpus.
Scream	The wumpus got killed by an arrow.
Bump	The agent walked against a wall.

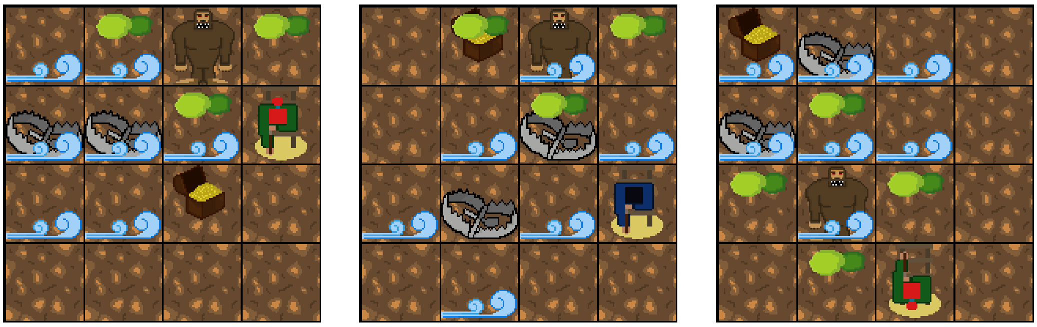
Table 11.2: Possible Perceptions of Wumpus Agents

by, hear it screaming when it got hit by an arrow, and feel a breeze caused by trap doors close by. In addition to the mechanics of actions and perceptions in the world, the classic Wumpus World also prescribes constraints on the initial state of the world:

1. The world has a rectangular shape of equal width and height.
2. There is only one wumpus.
3. There is an arbitrary number of trap doors.
4. There is only one agent.
5. The agent has only one arrow.
6. There is only one gold.
7. The gold is not in a cave with a trap door.
8. There is only one ladder.
9. The ladder is not in a cave with a trap door or a wumpus.

Although the location of the gold and the ladder seem to be secured by Rule 7 and 9, it is still not always possible to grab the gold and leave the caves unharmed. Especially Rule 3 allows the creation of worlds, where a series of trap doors separate the location of the ladder and the gold. We distinguish between three kinds of worlds with regard to their solvability: solvable worlds, unsolvable worlds, and brave worlds. In a solvable world, the agent can grab the gold and leave the caves without any actions that are not known to be safe. Safe actions will not get the agent killed, while unsafe actions will get the agent potentially get killed. In a brave world, the agent has to conduct unsafe actions in order to solve it. Finally, in an unsolvable world, the agent will always die before reaching the gold.

Figure 11.2 shows three examples, with one for each kind. Each world has the same size, one wumpus, and the same number of trap doors. The only difference is their arrangement and the starting position of the agent. The world in Figure 11.2a is solvable because the agent can explore a path to the gold without even shooting the wumpus. In the world of Figure 11.2b, the agent must be brave and conduct several unsafe actions in order to reach the other side of the trap doors. The agent, for example, cannot know whether there is a



(a) Solvable World (b) Brave World (c) Unsolvable World

Figure 11.2: Different Kinds of Wumpus Worlds

trap door in the lower-left corner of the world or not, because the perception of breezes does not indicate towards the direction of the trap door neither do they indicate the number of adjacent trap doors. Therefore, the agent in Figure 11.2b might be unlucky and get killed by choosing the wrong brave action. In the unsolvable world from Figure 11.2c, the agent cannot reach the gold, even if it has killed the wumpus. The agent cannot pass the trap doors close to the gold. It is also notable that the rules, mentioned above do not exclude trivial cases for brave worlds, like a ladder adjacent to a trap door or wumpus. The same holds for solvable cases, where the ladder is right on top of the gold.

11.1.2 Extension of the Wumpus World

In order to evaluate the work presented in this thesis, we extended the classic Wumpus World. Multiple agents are now allowed to enter the world and solve it together. Each agent has its own arrow, but for the sake of fairness, we also allowed multiple wumpi to exist in the caves.

Due to these two extensions, the mechanics and rules of the world need further clarifications. The constraints on the initial state of the world are rewritten as follows:

1. The world has a rectangular shape of equal width and height.
2. There is an arbitrary number of wumpi, trap doors, and agents.
3. Each agent has only one arrow.
4. Each agent has its own ladder.
5. There is an infinite pile of gold in one cave.
6. The gold is not in a cave with a trap door.
7. The ladders are not in a cave with a trap door, wumpus, or another ladder.

The meaning of the rules is essentially the same as for the classic Wumpus World. Only some remarks on the multiple spawning positions of agents and wumpi are added. The effects of the actions Move, Shoot, Grab, and Climb, however, are extended as written in Table 11.3.

Action	Effects
Move	An arbitrary number of agents can be in one cave.
Shoot	Arrows cannot kill other agents, but will kill all wumpi standing in the direction of flight.
Grab	Agents pick only one peace of gold and will always leave enough gold for all other agents.
Climb	Agents can only use their own ladder.

Table 11.3: Additional Remarks on Actions in the Extended Wumpus World

About the perceptions of agents, there is only to mention that agents cannot distinguish between one or several screaming wumpi.

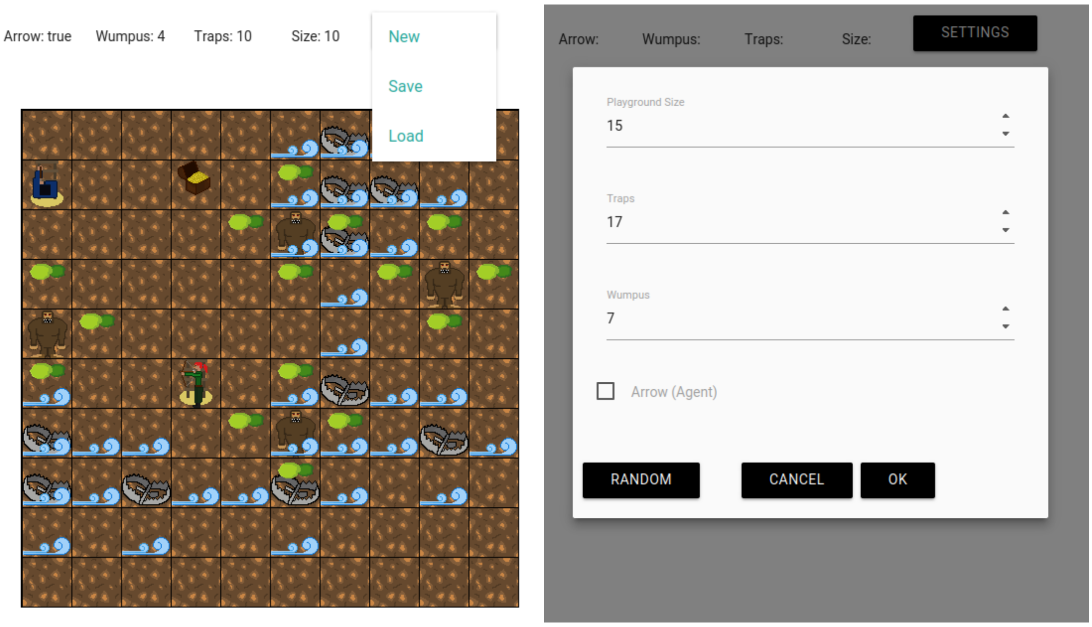
While in the classic Wumpus World scenario, there are three different kinds of Wumpus Worlds, in the extended case, we further distinguish between completely and partially solvable worlds. A world is considered *completely solvable* when all agents can grab some gold and leave, while in a partially solvable world, not all agents can grab some gold and leave. However, in both cases, completely and partially solvable, brave actions might or might not be necessary. Therefore five different kinds of worlds can be distinguished in the extended Wumpus World scenario.

11.1.3 Wumpus World Simulator

Although there are multiple Wumpus World simulator available, none supports the aforementioned extensions that would allow for a proper evaluation of the work presented in this thesis. We, therefore, developed our own open-source Wumpus World simulator [16].

The user interface of the simulator, shown in Figure 11.3, is minimalistic. The main window, shown in Figure 11.3a, visualises the current state of the world as well as its configuration parameters at the top. Via the *New* menu item, the dialogue for generating new worlds is opened, as shown in Figure 11.3b. Here it is possible to specify the size of the world, the number of traps, the number of wumpi, and finally whether agents have an arrow. In addition to generating new worlds, it is also possible to save and load existing worlds. This feature is handy for thorough investigating complicated worlds and for comparing the performance of different intelligent agents.

The application programmable interface (API) of the simulator utilises the publish and subscribe pattern (see Section 9.1.1). At first, each agent requests to spawn, which is acknowledged with the corresponding response message by the simulator (see Figure 11.4). After the spawning phase, the simulator notifies the first agent to send its action request. As a reply, the simulator executes this action and sends the potential new perceptions back via an action response message. Afterwards, the simulator notifies the next agent.



Arrow: true Wumpus: 4 Traps: 10 Size: 10

Arrow: Wumpus: Traps: Size:

Playground Size
15

Traps
17

Wumpus
7

□ Arrow (Agent)

(a) World Visualisation

(b) World Generation

Figure 11.3: Wumpus World Simulator User Interface

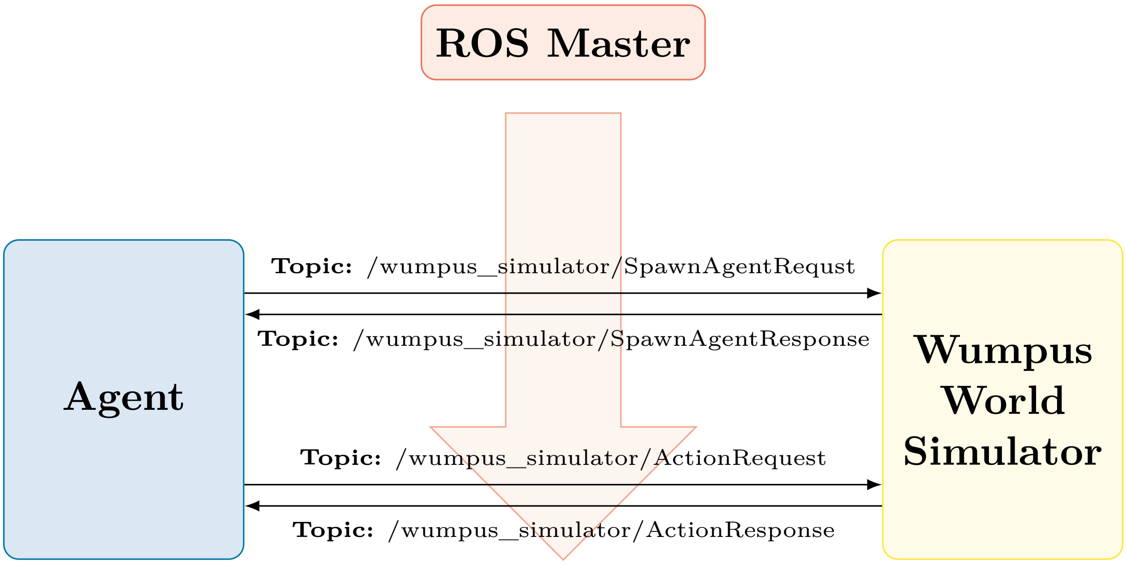

ROS Master
Agent
Topic: /wumpus_simulator/SpawnAgentRequst
Topic: /wumpus_simulator/SpawnAgentResponse
Topic: /wumpus_simulator/ActionRequest
Topic: /wumpus_simulator/ActionResponse
Wumpus World Simulator

Figure 11.4: ROS-based Connections between Wumpus Simulator and Agent

Actions send by agents that are not notified, are ignored by the simulator, which makes the simulator strictly turn-based and therefore allows for solving the same world in a reproducible way.

11.2 Service Robots

The primary motivation for this demonstrator is to show the capabilities of the presented framework concerning requirements R3 and R4 given in Section 1.2. Namely, the handling of dynamic environments and interaction with humans. Therefore, one of the main differences compared to the Wumpus World demonstrator is its asynchronous execution environment. However, both demonstrators use a grid world representation for their environment. In case of the Service Robot demonstrator, this allows, for example, to abstract from the continuous domain of path planning, inverse kinematics, and processing raw image data, without losing the characteristics of a real service robot environment that are relevant for this work to be evaluated.

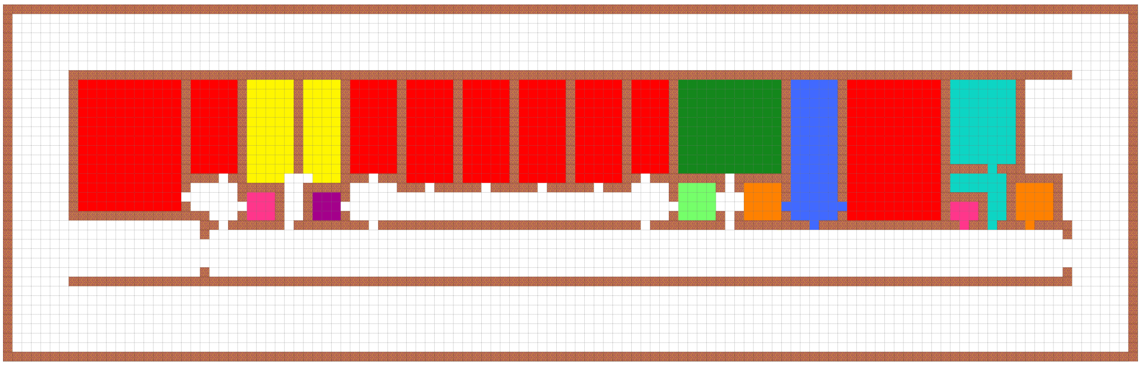
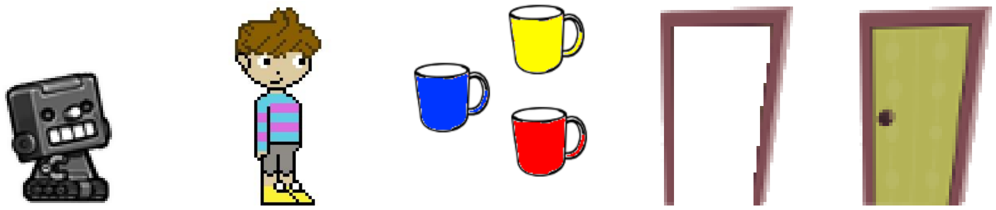


Figure 11.5: Grid World Representation of the Distributed Systems Department.
Legend: ● Office, ● Workshop, ● Storage Room, ● Server Room, ● Kitchen, ● Conference Room, ● Utility Room, ● Reception, ● Bathroom

The environment, shown in Figure 11.5, is a grid representation of the Distributed Systems Department of the University of Kassel. The grid world of the department is constructed to scale and the legend of Figure 11.5 explains the different kinds of rooms. Agents operating in this environment already know everything that is shown in this figure. However, there are several dynamic objects in the environment which need to be perceived before the agents know anything about them.

At first, there are the service robots (Figure 11.6a) itself that move around the department to fulfil human requests. Further, there are humans (Figure 11.6b) that walk around and can change the environment. In our simple scenario, there are two further objects. Cups (Figure 11.6c) of different colour and doors (Figure 11.6d) that can either be open or closed. The service robots have a sight limit of 10 grid cells and cannot look through walls,



(a) Service Robot (b) Human (c) Cups (d) Doors

Figure 11.6: Dynamic Objects

closed doors, or around edges. Therefore, they need to sweep the environment thoroughly when they search for a particular object. Further, the humans often request the robots to bring them a cup of coffee, but we made the scenario more realistic by making the humans a little bit picky about their cups[1]. As a result, humans always request a specific kind of cup, when they ask for coffee. The details on the tasks, which will be generated by the humans are given for the actual experiments, described in Section 12.2. The actions that agents can execute are described in Table 11.4.

Action	**Effects**
Move	The agent moves up, down, right, or left.
Open	The agent opens a door.
Close	The agent closes a door.
Putdown	The agent drops an object to its current location.
Pickup	The agent picks up an object.

Table 11.4: Possible Actions and their Effects in the Service Robot Simulator

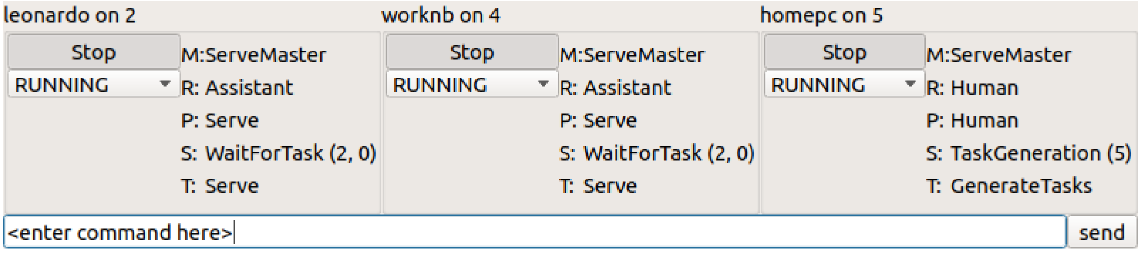
As described in Table 11.4, the agent can move in any direction without turning in advance. The only precondition is that a wall or a closed door does not occupy the target grid cell. The latter one can only be opened or closed by the agent when the agent is occupying a cell which is adjacent to the door. The same also holds for picking and dropping an object. Further, the object must not be taken by another agent already.

11.2.1 Control Panel

We also developed a user interface, denoted Control Panel, that facilitates the control of the ongoing experiment and allows for interaction with the service robots in the simulation environment.

Figure 11.7 shows the Control Panel interface with three agents visible. For each agent, humans and service robots, it shows the current operating mode, as well as basic status

[1] Similarities to employees in the Distributed Systems department are completely and entirely unintentional.



leonardo on 2
M:ServeMaster
R: Assistant
P: Serve
S: WaitForTask (2, 0)
T: Serve

worknb on 4
M:ServeMaster
R: Assistant
P: Serve
S: WaitForTask (2, 0)
T: Serve

homepc on 5
M:ServeMaster
R: Human
P: Human
S: TaskGeneration (5)
T: GenerateTasks

<enter command here>

Figure 11.7: Control Panel for the Service Robot Demonstrator

information with regard to the ALICA program they are currently executing. This includes their role, the name of the top-level plan, as well as the name, state, and task of the lowest level plan they currently execute.

11.2.2 Runtime Architecture

The foundation for the runtime architecture of the service robot simulator is similar to any other state-of-the-art robotic simulator, like Gazebo[2] or Webots[3]. The simulator is running asynchronously to any external events or incoming messages at a rate of 30 Hz. Each main loop iteration includes the following abstract steps: process control messages from the agents, update the environment and send generated perceptions to the agents.

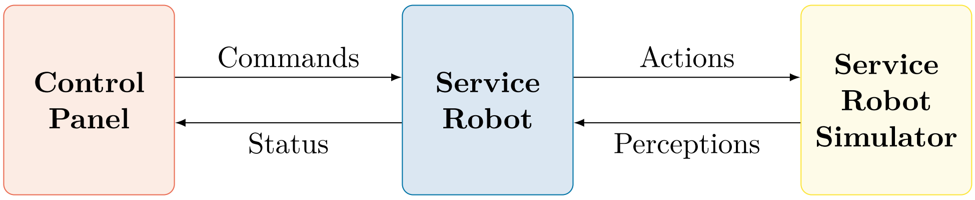


Figure 11.8: Cap'n Zero-based Connections between Service Robot Simulator, Agent, and Control Panel

Instead of ROS, the communication is based on the lightweight middleware Cap'n Zero described in Section 9.1.2. Compared to the communication based on ROS, Cap'n Zero is more efficiently and robustly transferring messages, facilitates the communication across system borders, and supports UDP multicast. The processes of the agents can, therefore use their own machines as long as they are all connected in the same local network. Nevertheless, Cap'n Zero also follows the message-based publish and subscribe pattern, just like ROS. Figure 11.8 only visualises the connection for one agent. During an experiment, there will be several agents, each with the same connections to the control panel and the service robot simulator. Among the agents themselves, depending on the experiment, there will also be two further connections. One for sending requests and one for sending specific answers to the different kind of requests.

[2] Gazebo Simulator - http://www.gazebosim.org/ [last accessed on June 19th, 2020]
[3] Webots Simulator - https://www.cyberbotics.com/ [last accessed on June 19th, 2020]

Evaluation 12

In addition to already published results, in this chapter we present results of further experiments. These experiments include experiments in the Wumpus World environment, the service robots environment, and a comparison between the communication middleware Cap'n Zero and the communication layer of ROS. Further, we will summarise some key experiments and results that we have already published, to create a holistic view of the conducted experiments. In Chapter 13, this holistic view will allow us to discuss all advantages and possible future work of the presented thesis in depth without requiring the reader to consult other publications.

12.1 Wumpus World

In the Wumpus World experiments, we did not follow our approach on an open and adaptable architecture design, as described in Chapter 5. Instead, we utilise our ASP knowledge base as the only decision-making entity of the agent. We inserted all continuously changing information about the environment into the knowledge base, and all conducted actions of the agent are solely deduced by the reasoning process of the knowledge base. In general, we do not recommend this approach because it has limited applicability by design. However, by following this approach, we were able to investigate the properties of our knowledge base in an extreme use case.

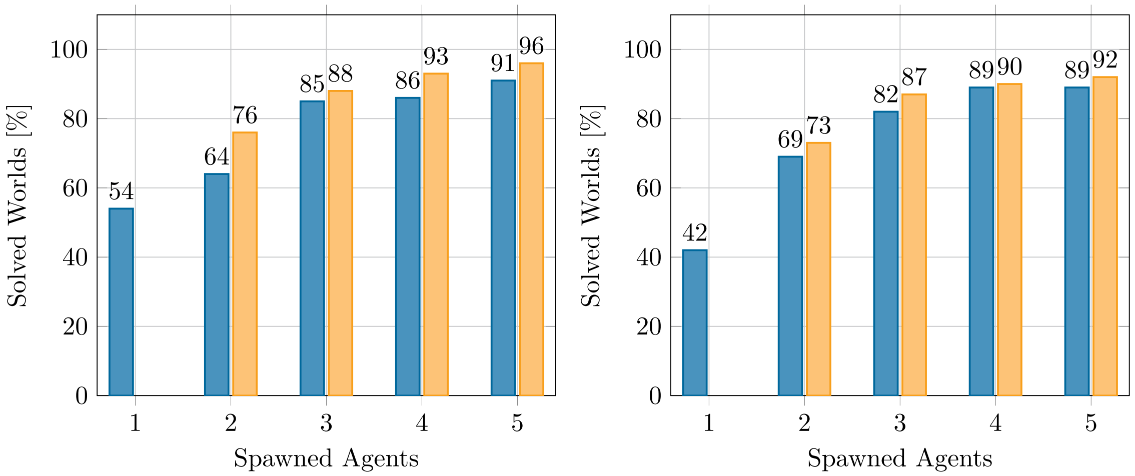
Setup Property	**Description**
CPU	Intel i7-4710HQ @ 2.5GHz
Memory	16GiB S0DIMM DDR3, 1600MHz
Operating System	Ubuntu 16.04 LTS
Kernel	4.15.0-112-generic
ASP Solver	Clingo 5.3.1 - Configuration "handy"

Table 12.1: Setup of the Wumpus World Experiments

Table 12.1 lists the detailed hardware specifications of the laptop, the installed operating system, and the utilised ASP solver with its configuration under which we conducted all Wumpus World experiments. Clingo can apply different configurations to its search strategies. We used the *handy* configuration for our Wumpus World experiments because it is recommended for large problem instances by the Clingo developer team.

During the experiments, we evaluated two different sizes of rectangular Wumpus Worlds: 5x5 and 6x6. For each size, we tried to solve worlds with 1 to 5 agents, and for each number of agents, we evaluated 100 different worlds. Each world was solved once with communication between the agents and once without communication. In total, this sums up to 1800 different runs. Due to the turn-based execution model of the Wumpus World simulator, these experiments take relatively long to perform. However, the focus of these experiments is not on measuring the total runtime, but on a commonly used metric for the Wumpus Worlds. This metric measures the number of actions an agent executes before it can leave the world with the gold and how many worlds it could solve in total. These two key performance indicators are hardware independent and make it possible to compare our results with other approaches.

The random world generator was configured to spawn 2 to 5 wumpi randomly and 1 to 4 traps in case of 5x5 worlds and 2 to 7 wumpi and 2 to 5 traps in case of 6x6 worlds. The randomly spawned agents are all equipped with one arrow. For the randomly chosen spawning positions of agents, it is not allowed to chose caves with other agents, wumpi, traps, or the gold inside.

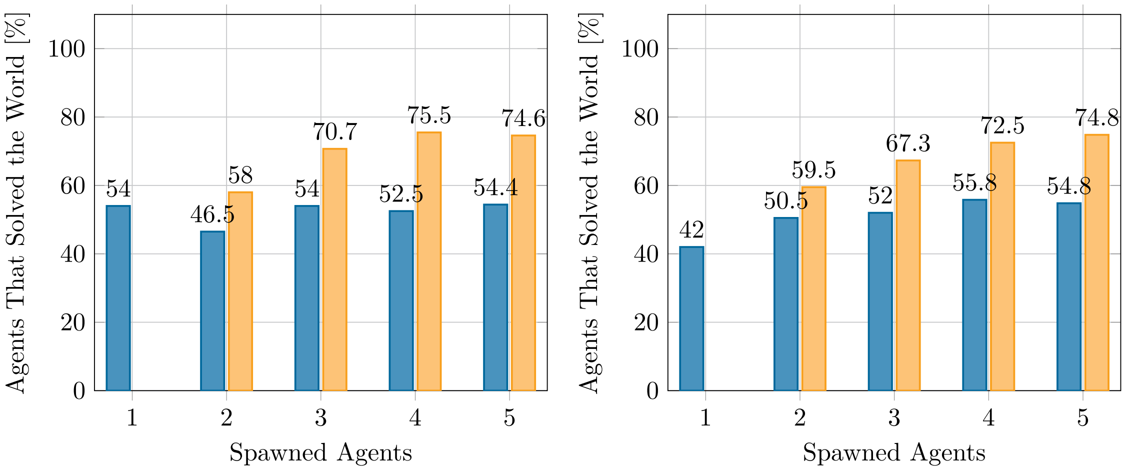


(a) 5x5 Worlds

(b) 6x6 Worlds

Figure 12.1: Solved World Percentage Without (●) and With (●) Knowledge-Based Cooperation

Figure 12.1 shows the percentage of solved worlds separated by world size (Figure 12.1a and 12.1b), spawned agents, and enabled or disabled knowledge-based cooperation. The bar for enabled knowledge-based cooperation in case of a single agent is missing for obvious reasons; however, it functions as a reference to the cases with multiple agents spawned. The results show the advantage of knowledge-based cooperation over the attempt of agents to solve worlds independently. The difference between the number of worlds solved with and without knowledge-based cooperation decrease with the number of spawned agents because we consider a world as solved when a single agent leaves the world with the gold. Increasing the number of agents while keeping the worlds at the same size increases the probability of spawning one agent near to the gold and therefore to solve the world easily. Further, there are worlds and spawning positions that require cooperation between agents to kill wumpi and solve the world. However, these worlds are improbable to be generated by a random world generator. Instead, the reason for the clear advantage of knowledge-based cooperation over the attempt to solve worlds independently is another one. The additional knowledge that is received from other agents reduces the probability to die in a brave world (see Figure 11.2b in Section 11.1.1) by conducting unsafe actions.



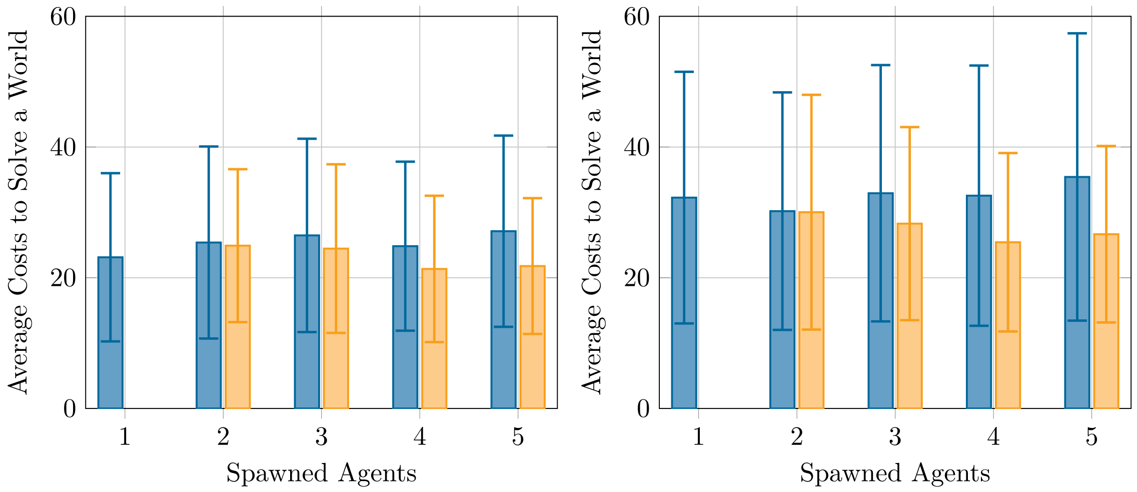
(a) 5x5 Worlds

(b) 6x6 Worlds

Figure 12.2: Percentage of Agents That Solved Their World Without (●) and With (●) Knowledge-Based Cooperation

Figure 12.2 shows the same scales as Figure 12.1, with an exception in the case of the ordinate. Here, the percentage of agents that can solve the world by leaving it with the gold is shown. The shown comparison between enabled and disabled knowledge-based cooperation does not depend on the number of agents as it did in Figure 12.1. Therefore, the advantage of knowledge-based cooperation is more significant and increases with the number of agents. The increase is induced by the extra amount of knowledge that every agent gains when an additional agent is spawned, and its knowledge is shared.

Figure 12.3 shows yet another aspect of the conducted experiments. The ordinate in Figure 12.3 is about the costs that agents had to solve a world. Each executed action increases the costs by one. Therefore, the average costs of 23 for a single agent to solve a 5x5 world (first blue bar in Figure 12.3a) means that an agent executed 23 actions to solve a world on the average. The error bars show the standard deviation of the costs. Compared to the average costs, the standard deviations are significant because the costs heavily depend on the spawning positions of the agents and the randomly generated worlds. Further, the standard deviations increase from 5x5 worlds to 6x6 worlds because the possible spawning positions and world configurations are larger in 6x6 worlds. The advantage of knowledge-based cooperation with regard to the costs is not as significant as it is for the percentage of agents that can solve the world. The reason for the minor

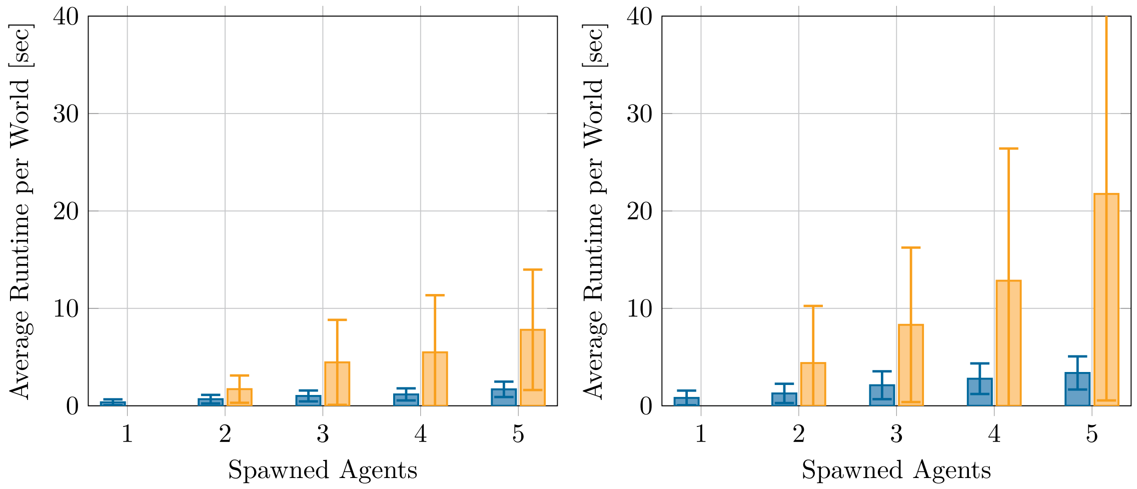


(a) 5x5 Worlds

(b) 6x6 Worlds

Figure 12.3: Average Costs to Solve a World Without (●) and With (●) Knowledge-Based Cooperation

impact on the costs is that even if agents share their knowledge, still all agents need to walk to the gold and leave the cave on their own. The only actions that can be avoided when agents share their knowledge are unnecessary explorative actions.



(a) 5x5 Worlds

(b) 6x6 Worlds

Figure 12.4: Average Runtime per World Without (●) and With (●) Knowledge-Based Cooperation

We analysed the advantages of knowledge-based cooperation in the case of the Wumpus World demonstrator and visualised them in Figure 12.1, 12.2, and 12.3. However, as there is no such thing as free lunch [134, 136], every advantage comes at a cost. Figure 12.4 shows the average time that it takes to execute one run in a world independent of whether it was solved or not. The average runtime and the corresponding standard deviation for runs with knowledge-based cooperation are significantly higher. The steps that are responsible for the increased times include the transmission of the knowledge to all other agents, the integration of this knowledge by other agents into their knowledge base. Each turn of each agent, these steps are executed and therefore, the runtime increases.

In our experiments, the programs of the agents were executed on the same machine (see Table 12.1) which implies that the runtime for sending messages is neglectable and most of the increased time is accounted for integrating the additional knowledge into the knowledge base. This observation coincides with our previous results [34, 20] from analysing the scalability of incorporating dynamic knowledge into our knowledge base by utilising the Externals feature of Clingo (see Section 4.3.3). From the scalability results, we concluded that 1800 Externals in the knowledge base still produce a reasonable performance for finding a path via the transitive closure of the `reachable/2` literal in a graph. The ASP encoding for the Wumpus World, however, is much more complex. The knowledge representation of sensor input about stench, glitter, and breeze, as well as the knowledge about positions of traps, wumpi, agents and the gold for every field in the world, already demands a considerable number of externals. Together with knowledge about visited fields, shot arrows, and current objectives of agents, the knowledge base includes 1500 Externals without querying any knowledge.

Nevertheless, we tried to keep the number of registered queries as low as possible because every registered query adds further externals to guarantee the module property. The path planning, for example, can use the same query all the time because we adjust the start and goal cave by two externals per cave. Apart from the increased number of Externals, the Wumpus World demonstrator also requires encoding of search problems with optimisation expressions. While ASP programs without optimisation expressions are already able to express problems in NP^{NP} [79], optimisation expressions increase the complexity exponentially. Finally, it is essential to note that our encoding of the Wumpus World may not be optimal and improvements might decrease the runtime significantly, but the limited scalability with respect to Externals cannot be overcome in general.

12.2 Service Robots

The purpose of the Service Robot demonstrator is to provide a test scenario that is close to the domain of domestic service robots, as this domain motivates the presented work the best. In contrast to the Wumpus World demonstrator, we also implemented the software architecture as described in Chapter 5. Further, the experiments are conducted on a more powerful system (see Table 12.2).

While the runtime benefits from the modern CPU, the 64GiB memory are not necessary during the execution of the experiments. Instead, they were only necessary during

Setup Property	Description
CPU	Intel i7-9750H CPU @ 2.60GHz
Memory	64GiB S0DIMM DDR4, 2667MHz
Operating System	Ubuntu 18.04 LTS
Kernel	5.4.0-42-generic
ASP Solver	Clingo 5.5.0 - Configuration "default"

Table 12.2: Setup of the Service Robot Experiments

the deployment of the ConceptNet 5 database because of the memory-intensive database creation process. At runtime the experiments require only 3-4 GiB memory. Further, we use Clingo version 5.5 in its default configuration for the knowledge base.

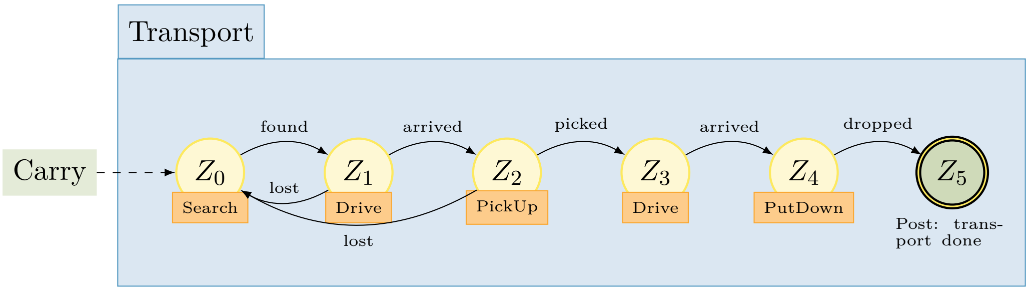


Figure 12.5: ALICA Plan for Transporting Items

We spawned three robots in the service robot experiments to fulfil the queried transport tasks. The tasks always have the same form, according to which the robot should bring a cup of a specific colour to a specific cell. We modelled the transport task in five steps, as shown in Figure 12.5. At first, the robot needs to search for a cup of the requested colour. When it found one, it needs to approach it to be in range for the Pick-Up action. Afterwards, it drives to the destination and puts the cup down into the cell. However, in the scenario, humans and other robots can pick up the cup that the robot was aiming at and therefore, make the cup unrecognisable at any time. In such a case, the robot looses the cup and goes back to execute the Search behaviour in Z_1 state again.

Since we run the simulation and the robots in our experiments on the same system, the simulation of the employees of the Distributed Systems Department would have a significant influence on the results. We, therefore, simulate the interaction of humans with the environment of the robots by randomly beaming cups around. On average, the simulator beams a random free cup to a random cell every 1.5 seconds. Further, the chosen cells are not equally distributed on the map shown in Figure 12.6. Instead, in 30% of all beams, the cup will be placed on a cell of the kitchen, in 50% on a cell of an office, and in 20% on any other cell on the map. We chose this distribution because most employees take their cup with them into their office were it stands most of the time and only for the short moment after the cups are cleaned and wait for someone to use them

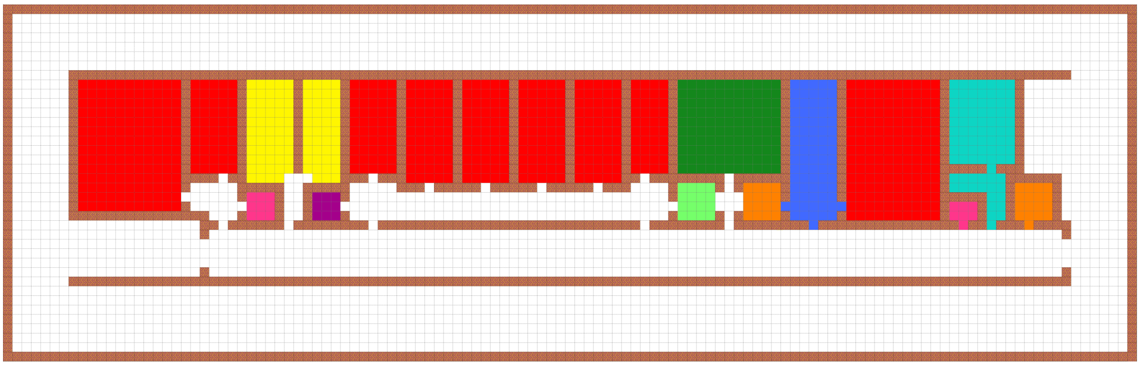


Figure 12.6: Service Robot Demonstrator Scenario.
Legend: ● Office, ● Workshop, ● Storage Room, ● Server Room, ● Kitchen, ● Conference Room, ● Utility Room, ● Reception, ● Bathroom

again, they are located in the kitchen. We also chose the destinations of the transport tasks according to the same distribution for the same reasons.

Additional to the three robots, we spawned eight cups on the map: three blue cups, three red cups, and two yellow cups. This restricted number of cups makes the transport tasks more challenging for the robots. It is, for example, possible that two robots are carrying a yellow cup while the third robot is searching for another yellow cup. However, it will not find any until another robot puts its yellow cup down again. We evaluated four different configurations within the cup transportation scenario. Three thousand tasks are requested to the robots per configuration summing up to 12000 cup transportations evaluated with the Service Robot demonstrator.

In the first configuration, we restricted the knowledge of the robots to their sensor values only. This configuration serves as a reference for the other three configurations in which we stepwise added another source of knowledge to each configuration. The first additional source is the commonsense database ConceptNet5. We developed a dedicated database query for finding all rooms that typically are a place for a given object.

The pseudocode of this database query, as shown in Algorithm 2, searches over the graph data structure of CN5. The input of the algorithm includes the type of the object and the type of the location for which the algorithm should search in the database. In the case of the cup transport tasks, the object type is *cup*, and the location type is *room*. We highlighted the steps that query CN5 in blue. The first step in the algorithm is to find all concepts that are equivalent to the given object type. Therefore, we queried all edges that are connected with the object type concept, in our case the *cup* concept, via the *Synonym*, *SimilarTo*, or *InstanceOf* relation and initialise the set of search paths with the results. A search path is a path in the CN5 graph that connects CN5 concepts via CN5 relations. The rest of the algorithm includes three steps that are repeated in the while-loop. At first, the end of the next path, taken from the list of search paths, is checked whether it

Algorithm 2: Querying CN5 for Typical Locations of Objects

```
Input   : Concept objectType, Concept locationType
Output: ConceptSet locations
SearchPathSet searchPaths = $\emptyset$
searchPaths.add(createSearchPath(objectType))
for Edge equivalentEdge : queryEquivalentEdges(objectType) do
    searchPaths.add(createSearchPath(equivalentEdge))
end
while searchPaths $\neq \emptyset$ do
    SearchPath searchPath = searchPaths.pop()
    if queryIsAConcepts(searchPath.lastConcept).contains(locationType) then
        locations.add(searchPath.lastConcept)
    end
    if searchPath.length == 2 then
        continue
    end
    for Edge atLocationEdge : queryAtLocationEdges(searchPath.lastConcept) do
        searchPaths.add(createExpandedPath(searchPath, atLocationEdge))
    end
end
return locations
```

is of the same type as the input of the algorithm requests. Therefore, we verify that a CN5 edge between the last concept and the location type exists. In such a case, we add the last concept to the list of found locations. The second step limits the depth of the search to the length of 2 *AtLocation* edges. The analysis of search paths longer than 2 showed that the reliabilities of the connections are too small to be meaningful. The third step queries for *AtLocation* relations that connect the last concept in the search path with new concepts. We expand the search path with found connecting relations and add the expanded search path back into the search path set.

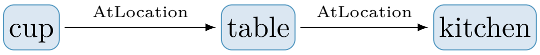


Figure 12.7: Example for a Concept Path

Figure 12.7 shows an example of a path of concepts found by Algorithm 2. The result of the algorithm is a list of all rooms where, according to the CN5 commonsense knowledge database, cups are likely to find. This knowledge is then automatically transformed into ASP rules, as described in Section 8.2.3. Together with the knowledge about the map, the service robot is now able to compare see which type of rooms it knows and in which it should look for cups first. In our scenario, this makes the robots at first search for cups in the kitchen of the department.

Another additional source of knowledge is coming from humans that communicate with the service robots. Although the commonsense knowledge database is very comprehensive, it does not take the situation of the simulated Distributed Systems department into account. The number of cups compared to the number of employees is too small and therefore makes it unlikely to find cups in the kitchen. Instead, most of the time, the cups are in the offices of the employees. The employees of the department are aware of the problem and tell the service robots that they should also look for cups in the offices before they start to sweep the whole department. In this configuration, the ASP-based knowledge base of the service robots includes knowledge about the map, their sensor values, the commonsense knowledge from CN5, and the knowledge taught by the employees. All this knowledge is shared between the service robots in the last of our four configurations. From the perspective of a single service robot, the knowledge of other service robots is another source of knowledge. This shared knowledge, in case of the Service Robot demonstrator, only includes positions of cups they have seen while moving through the department because the other knowledge sources are already available to every service robot by the design of the demonstrator scenario.

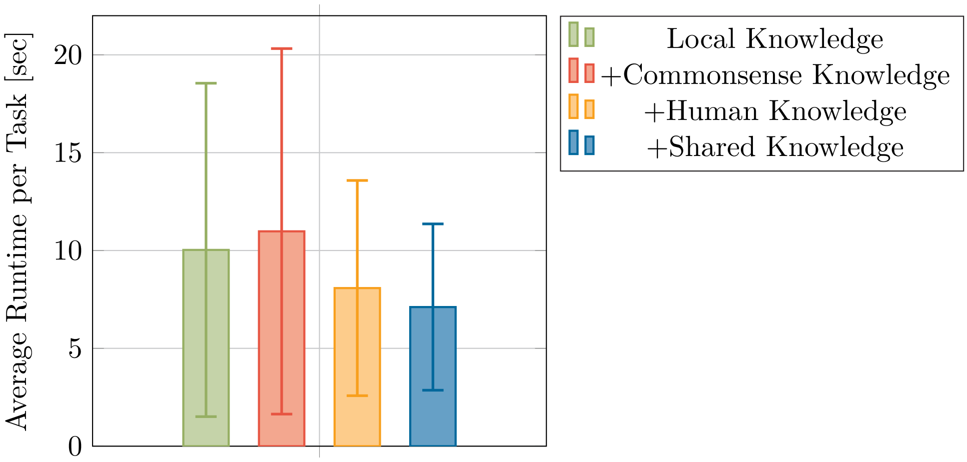


Figure 12.8: Average Runtime per Task With Different Sources of Knowledge

Figure 12.8 summarises the results of the different configurations. The green bar is the reference for the other three configurations and only include local knowledge. The runtime to finish a transport task takes 10.03 seconds in this configuration. The standard deviation of the runtime is 8.52 seconds, which means that the runtime of transport tasks can be very different. In one instance, the service robot is already standing right next to the right cup and only needs to bring it to a cell right next to its current position. In another instance, a service robot cannot find the right cup, because the other robots picked them all up. In this case, the robot sweeps the whole department multiple times until the right cup is put down again.

The second configuration, represented with a red bar, has an increased average runtime of 10.98 seconds with a standard deviation of 9.34 seconds. This shows the disadvantage of relying on commonsense knowledge in the case of the Distributed System department. However, the additional human knowledge compensates this disadvantage and achieves an average runtime of 8.08 seconds with a standard deviation of only 5.5 seconds illustrated in the orange bar. The blue bar represents the last configuration, where all knowledge sources are combined. In the case of combining all sources, an average runtime of 7.11 seconds with a standard deviation of 4.25 seconds was measured.

The results of the experiments show that utilising commonsense knowledge does not always increase the performance of a service robot. The scenario, for example, can be different from the assumptions made by the available commonsense knowledge. However, we showed that the presented system could make appropriate exceptions, without retracting the commonsense knowledge or resetting the knowledge base, when additional knowledge is available.

12.3 Implicit Human Service Requests

In the experiments conducted with the Service Robot demonstrator, we utilised commonsense knowledge to deduce common locations of objects that the service robots should search. We consider this to be a simple example of the application of commonsense knowledge. In an experiment published on the International Conference for Social Robotics [12], we demonstrated the application of commonsense knowledge in a more multifaceted and subtle human-robot interaction scenario. Humans always communicate with each other under the assumption that other humans have similar commonsense knowledge in their mind. The following dialogue, for example, shows this implicit assumption. *Alice: "I am tired." Bob: "Do you want to sit down?"* Although Alice is just uttering a fact, Bob interprets this as an implicit request and offers his help. Imagining that Bob is a service robot, we show that, with the knowledge available in CN5, Bob can understand implicit human service requests and can offer a variety of possible assisting actions.

Algorithm 3 uses similar queries as Algorithm 2 but with different relations of the CN5 database. Its input is a concept of human condition such as being tired. The first step is to create the initial search paths with *MotivatedByGoal* and *CausesDesire* relations connected to the human condition concept. The second step extends the search paths with *Synonym*, *SimilarTo*, or *InstanceOf* relations. Afterwards, *UsedFor* relations are appended to all search paths. Finally, the resulting search paths are extended with *Synonym*, *SimilarTo*, or *InstanceOf* relations again.

Table 12.3 shows the potential of Algorithm 3. The human conditions in the first column are examples for inputs to the algorithm. The concepts in the *Activity* column are the last concepts of the search paths in the equivalentSearchPaths variable after Line 14 of

Algorithm 3: Querying CN5 for Assisting Actions or Objects

```
Input   : Concept humanCondition
Output: ConceptSet assistingActionsOrObjects
SearchPathSet initialSearchPaths = $\emptyset$
for Edge motivatedByGoalEdge : queryMotivatedByGoalEdges(humanCondition)
 do
    initialSearchPaths.add(createSearchPath(motivatedByGoalEdge))
end
for Edge causesDesiresEdge : queryCausesDesireEdges(humanCondition) do
    initialSearchPaths.add(createSearchPath(causesDesiresEdge))
end
SearchPathSet equivalentSearchPaths = $\emptyset$
while initialSearchPaths $\neq \emptyset$ do
    SearchPath searchPath = initialSearchPaths.pop()
    for Edge equivalentEdge : queryEquivalentEdges(searchPath.lastConcept) do
        equivalentSearchPaths.add(createExpandedPath(searchPath,
          equivalentEdge))
    end
end
SearchPathSet usedForSearchPaths = $\emptyset$
while equivalentSearchPaths $\neq \emptyset$ do
    SearchPath searchPath = equivalentSearchPaths.pop()
    for Edge usedForEdge : queryUsedForEdges(searchPath.lastConcept) do
        usedForSearchPaths.add(createExpandedPath(searchPath, usedForEdge))
    end
end
equivalentSearchPaths = $\emptyset$
while usedForSearchPaths $\neq \emptyset$ do
    SearchPath searchPath = usedForSearchPaths.pop()
    for Edge equivalentEdge : queryEquivalentEdges(searchPath.lastConcept) do
        equivalentSearchPaths.add(createExpandedPath(searchPath,
          equivalentEdge))
    end
end
ConceptSet assistingActionsOrObjects = $\emptyset$
for SearchPath searchPath : equivalentSearchPaths do
    assistingActionsOrObjects.add(searchPaths.lastConcept)
end
return assistingActionsOrObjects
```

Conditions	Activities	Assisting Actions or Objects
tired	rest, sleep	sitting down, sleeping at night, going to sleep, bed, hotel
thirsty	drink, drink water	water, ice, beer mug, faucet, glass
hungry	cook, eat food, cook meal	fork, forks, indian restaurant, stove, kitchen
hot	cool off, swim	fan, pool, river, stream, ocean
cold	light fire, start fire	match, matches, flint, kindling, lighter
bored	watch television, play games	living room, playroom, computer, family room, basketball court
ill	go to doctor, lie down	illness, bed
smell	take shower, take bath	shower head, shower stall, portable shower head, bathtub, tub
stink	bathe	bars of soap, bathtub, wash cloth, separate shower
lonely	call friend	telephone

Table 12.3: Excerpt from Suggested Actions for Basic Human Needs

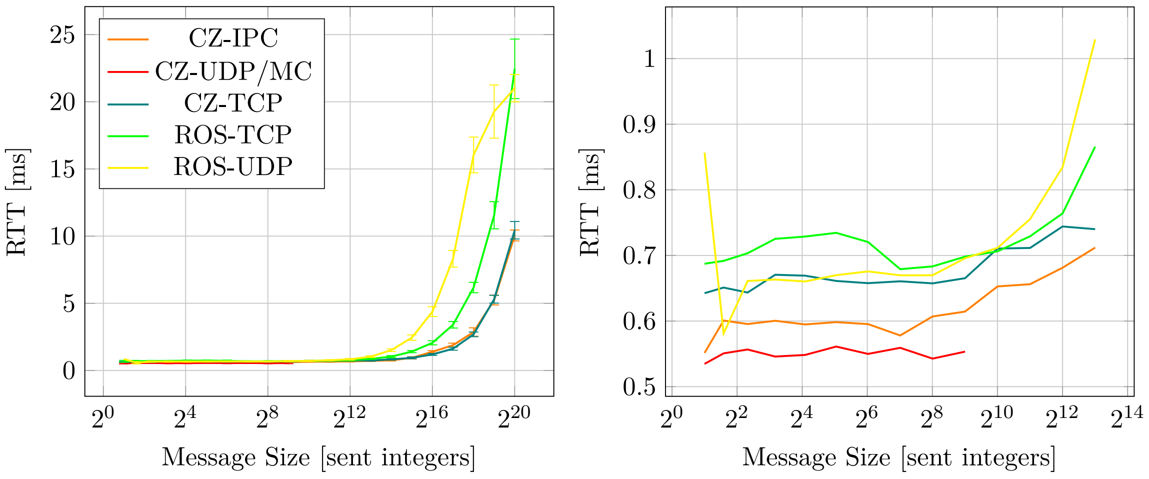
the algorithm. These concepts are activities that address the implicit needs implied by the negative human conditions in the first column. The third column lists the output of Algorithm 3, which are actions or objects that help to execute the activities in the second column. A thirsty human, for example, is suggested to drink. However, instead of merely giving this advice, the service robot further analyses objects that are typically used for the suggested activity. A human that should drink, for example, needs a glass or water.

12.4 Cap'n Zero

The experiments to compare the performance of Cap'n Zero and ROS measure the Round Trip Time (RTT) between two local processes and between two processes on computers that are connected over a 100 Mbit/s access point. In order to make the experiment free of external disturbances, the computers use their own wired connections.

The messages sent include a 32-bit integer as message ID and an array of 32-bit integers as payload. The payload started at a size of 1 integer and is doubled until bandwidth limitations are reached. Each message with a specific payload size is sent 1000 times with a frequency of 30 Hz.

Figure 12.9a includes the average RTT, including the standard deviation over the full range of message sizes. IPC in this context means interprocess communication. The plots show that the RTT for all protocols of both communication libraries grow exponentially, which could be expected, as the message size is increased exponentially, too. The RTT of ROS is growing faster than the RTT of Cap'n Zero, although, in case of UDP, the growth



(a) All Message Sizes

(b) Small Message Sizes

Figure 12.9: Local RTT of ROS and Cap'n Zero

of the RTT of ROS slows down. A reason for this behaviour could be the fact that ROS, when it is using UDP, is starting to drop messages at a smaller message size than Cap'n Zero. Please consider the tables in Appendix G for more details about lost messages and bandwidth limitations of the experiment.

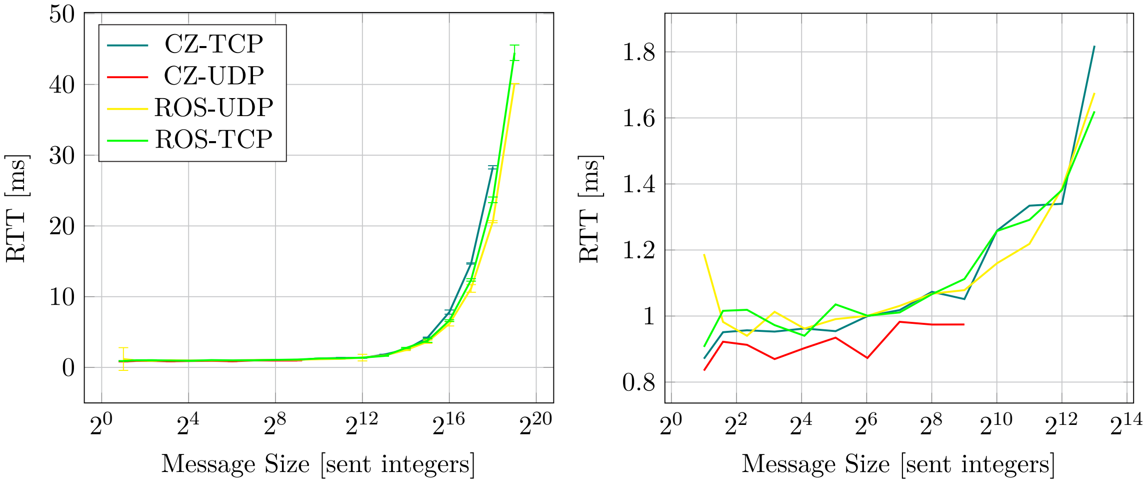
Figure 12.9b shows a close up of the data shown in Figure 12.9a, with a focus on small messages. While the RTT of Cap'n Zero using UDP Multicast and IPC is generally smaller than the RTT of ROS, the RTT of Cap'n Zero using TCP is almost the same as ROS using UDP and TCP. Here it is important to note that ROS is using UDP Unicast and Cap'n Zero is using UDP Multicast. Unfortunately, ZeroMQ, the transport library of Cap'n Zero, is suffering a limitation in message size for UDP Multicast and cannot transmit messages large than 8191 Bytes [1]. Therefore, the corresponding plot ends much earlier.

Figure 12.10a and 12.10b show the RTT of ROS and Cap'n Zero via a 100 MB/s access point. The plots are similar to the local case, except that the RTT is 0.3 ms higher for all configurations. Another difference is that Cap'n Zero using TCP shows a slightly worse performance than ROS, while Cap'n Zero still performs better than ROS when using UDP Multicast.

Finally, it is worth noting, that both communication libraries drop messages during the establishment of connections (ROS slightly more than Cap'n Zero) and when the bandwidth limits are reached ROS drops much more messages than Cap'n Zero when using UDP.

In summary Cap'n Zero outperforms ROS in almost all experiments, offers cross system communication which ROS does not, and provides a more versatile publish-subscribe API.

[1] ZeroMQ UDP Message Size Limitation - https://github.com/zeromq/libzmq/issues/2009 [last accessed on January 8th, 2020]



(a) All Message Sizes

(b) Small Message Sizes

Figure 12.10: RTT of ROS and Cap'n Zero over Network

Conclusion 13

The research goal of this work is the development of conceptional foundations for a team of autonomous robots that is capable of symbolically representing knowledge about the environment, communicating about its symbolic knowledge, and reasoning about the knowledge, while the environment changes dynamically. We defined this research goal in Chapter 1 after looking at different application domains for autonomous robotic teams such as disaster scenarios, space missions, and domestic service robots. The latter scenario motivates the formulated research goal the most and also provides several requirements that a solution must fulfil. In Section 13.1, we summarise our developed system and give an overview of subsequent developments enabled by this thesis. In Section 13.2, we revisit the requirements from the problem statement and elaborate on how our solution addresses them. Finally, we conclude this work with possible future research that is along the lines our research goal.

13.1 Summary

We build several components that, fused together in one system, fulfil all identified requirements and therefore, solve our research goal. The components include the rewritten ALICA Framework, an ASP-based dynamic knowledge base, and the lean communication middleware Cap'n Zero. Our flexible software architecture allowed us to combine these components into a reactive multi-robot system that features knowledge-based cooperation. Further, the system eases human-robot interaction, by integrating a significant amount of commonsense knowledge, supporting speech act performatives, and the capability of understanding implicit requests.

Figure 13.1 includes a simplified but complete overview of the presented system. While we integrated the existing ASP solver Clingo and commonsense knowledge database ConcepNet 5 into our knowledge base, all other components of the system are either created from scratch or reimplemented and adapted for the purpose of this thesis.

With regard to our improved ALICA Framework, we want to mention that Rapyuta Robotics[1] decided to base their warehouse automation solutions on our work. Further, since we published our work under the MIT License, Rapyuta Robotics already contributed several additional features to our improved ALICA Framework:

- A discovery component now allows agents to cooperate without knowing each other beforehand.

[1] Rapyuta Robotics - https://rapyuta-robotics.com [last accessed on November 14th, 2020]

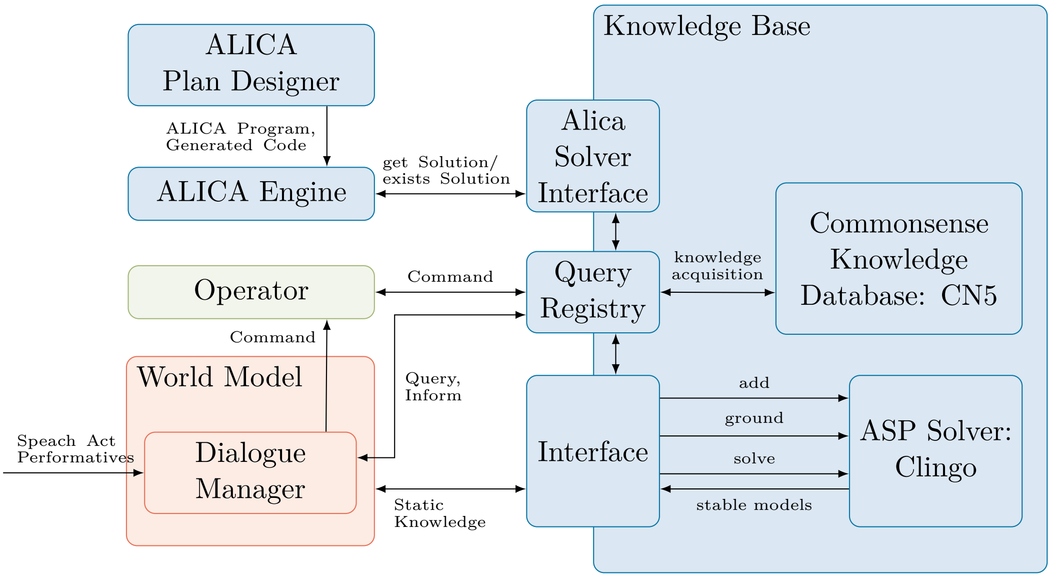


Figure 13.1: Solution Overview

- A dependency on an outdated configuration library is replaced by an integrated JSON-based configuration handling.
- The ALICA engine is extended by a facade pattern to provide a clean interface to users.

The most current version of the ALICA Framework, including all improvements provided by this thesis and the contributions by Rapyuta Robotics are as well published under the MIT License[2]. An additional testimonial for the improvements by this thesis, provides the growing number of companies, already including two Fortune 500 companies, that utilise the ALICA Framework in their production systems.

13.2 Requirements Revisited

In the introductory chapter of this thesis, see Section 1.2, we summarised the variety of collected requirements under four main requirements. The system, as shown in Figure 13.1, addresses all requirements as described in the following sections.

R1: Domain Independence

Developed conceptual solutions should apply to many different application scenarios without significant adaption. We, therefore, have chosen ASP for its outstanding expressiveness to represent the knowledge about the domain but also reimplemented the ALICA Framework in C++ and extended it with a general solver interface allowing the integration of other

[2] ALICA Framework - https://github.com/rapyuta-robotics/alica [last accessed on November 14th, 2020]

formalisms as well. Additionally to the ASP-based expressiveness, our solution scales to a significant number of objects, for example, encountered in a usual household. Further, the flexibility of our architecture design allows using highly reactive control layers in combination with more elaborated data abstraction processes for high-level decision making. The ALICA Framework contributes to the domain independence of our solution by being fully distributed, avoiding single-point-of-failures, and being robust against communication delay and packet loss. Each agent can operate completely independent of other agents, and therefore, we make no implicit assumptions on communication infrastructure or team composition.

R2: Handling of Unknown Environments

All three motivating application scenarios for this thesis confront the team of agents with unknown environments. Therefore, it is an inherent requirement for a solution to reliably work in an unknown and unpredictable environment and concurrently maintain the autonomy of the agents. Our solution addresses this requirement under the focus of long-term operating agents by providing the agents with the three key cognitive capabilities of reasoning, learning, and planning. While we did not specifically investigate the latter, the Wumpus World demonstrator showed the planning capabilities of our ASP-based knowledge base. Further, we thoroughly investigated the ASP-based reasoning capabilities of our solution and showed in various experiments and demonstrators that it meets the requirements of all applied scenarios. However, the most crucial capability to operate for long times in unknown environments is to learn. Our knowledge base supports this capability as we designed its symbolic knowledge representation to be taught new knowledge at runtime. As a source of novel knowledge that agents can learn, we focussed on the application scenario of domestic service robots and exploited the fact that humans are always around and know best the facts that the agents need to learn. As a result, we allow humans to teach the agents. Furthermore, we also designed our agents to cooperate based on their knowledge to create a more comprehensive model of the world as a team. For this, we developed an appropriate middleware and implemented a speech-act-based protocol for exchanging knowledge. Finally, our solution also makes only little assumptions about the connections between agents, since unknown environments could require the agents to drop out of the communication range of each other to explore the unknown environment.

R3: Handling of Dynamic Environments

Dynamic environments impose similar requirements to solutions as unknown environments do. However, while in unknown environments, agents discover things, and new knowledge needs to be incorporated, in dynamic environments, already available knowledge can also change over time. In both cases, the three cognitive capabilities of reasoning, learning, and planning are the key to long-term operating in such environments. However, in the case of a dynamic environment, the adoption of knowledge needs to take place before the knowledge changes and might be useless already. In our experiments, we investigated the efficiency of the ASP-based reasoning of the developed knowledge base and could show its sufficiency for the addressed application scenarios. This was also possible because our knowledge base inherits the non-monotonicity of ASP-based reasoning and allows applying default reasoning with exceptions as well. As a result, the knowledge base can run continuously without having to make adjustments to the knowledge or handling exceptions outside of its inherent logic. Additionally, to the internal handling of dynamic knowledge, the semantic of extension queries and their compliance with the module property further allows changing the knowledge in the knowledge base externally. Finally, the ALICA Framework initially developed for highly dynamic domains like robotic soccer, allows our solution to robustly coordinate agents in dynamic environments.

R4: Facilitate Human Interaction

The requirement to facilitate human-robot interaction stems from the focus on the domain of domestic service robots. In contrast to other application domains, service robots can use humans as a source of knowledge to adapt themselves to an unknown environment. Further, service robots must interact with humans since they are built to serve them. A symbolic representation of knowledge, as utilised by our solution, facilitates the interaction between humans and service robots, since the knowledge presented is much more accessible to humans than any connectionist approach such as a neural network. As shown in Chapter 8 and 12, our solution allows humans to teach service robots. However, to mitigate the burden of teaching everything to the service robots, we also demonstrated how to integrate knowledge from a commonsense knowledge database automatically. With the commonsense knowledge accessible by our knowledge base, service robots can even understand implicit human requests and offer intelligent assistance.

The revision of the requirements in the last paragraphs shows that our proposed solution to the problem statement formulated in Chapter 1 comprehensively addresses all requirements and we, therefore, conclude that our work presented in this thesis fulfils the proposed research goal.

13.3 Future Work

During our work on the proposed solutions, we discovered interesting aspects that are not within the focus of this work but deserve scientific attention nevertheless. In this section, we list these aspects and other ideas for further improvements to our proposed solution to provide the basis for further research in the areas that are related to the focus of this thesis.

ALICA Framework One current limitation of the ALICA Framework is that it currently assumes that all agents that execute a plan are relevant for the success of this plan from each agents perspective. Success in this context is to be understood in the sense of the success semantics of the ALICA Framework (see Section 3.1.3). A group of service robots, for example, that are cleaning up a household might end up in the same Transport plan at runtime. However, since each robot is carrying something different, the success of the Transport plan for each robot is independent of the success of all other robots. Therefore, sometimes the success of a plan needs to be decoupled similar as it is done in the planning domain. Here action is a template that needs to be instantiated, for example, *to grasp* is an action and *to grasp a cup* is an instantiation of that action. In the context of the ALICA Framework, we denote this as plan templates. In contrast to action templates, plan templates would still provide the cooperation implied by the success semantics of plans, but limited to the context that instantiated the template.

Another idea is to verify ALICA programs with state-of-the-art model checking techniques in order to guarantee, for example, that the program is deadlock-free or two specific behaviours can never run in parallel. Some preliminary work has been conducted in this direction [46, 24]. However, our idea is more of an integrated development step applicable to every application domain of the ALICA Framework. The key to a proper solution is to check the domain-specific elements of an ALICA program that can contain arbitrary C++ code. First steps towards that direction could be achieved by annotating, for example, some of the C++ function with their expected semantics in a language that can be understood by the applied model checker.

ASP Solver Our solution provides automatic compliance to the module property of ASP programs. However, this compliance limits the inferences that can be deduced when adding additional knowledge to the knowledge base. Therefore, the integration of the automatic module property compliance into an ASP solver itself could allow more inferences to be deduced and could also reduce the number of necessary Externals. While the number of Externals in an ASP program was a significant problem in the older versions of the ASP solver Clingo, the version used in the Service Robot demonstrator significantly improved the performance concerning the number of Externals. Nevertheless, it is still a performance factor that needs to be considered during the design of an ASP program.

Although being much more performant than usual description logic reasoners, ASP generally lacks a product ready API for designing and investigating ASP programs at runtime. A pendant to the OWL-API[3] made for ASP programs would significantly increase the acceptance of ASP in the industry.

[3] OWL-API - https://www.w3.org/2001/sw/wiki/OWLAPI [last access on October 8th, 2020]

Human-Robot Interaction Our interface proposed for humans to interact with service robots only understand valid ASP rules. In future, this interface should also understand (restricted) natural language which could be achieved based on existing work in the field [29, 23]. Another improvement would be the utilisation of Telegram[4] because with a Telegram client running on each service robot, humans could chat with their service robots from all around the world by utilising a smartphone. With these two improvements in place, it would be possible to deploy a naive but real service robot at home and conducting experiments in real environments.

The integration of CN5 as commonsense knowledge database already improved the interaction between robots and humans a lot. However, there is also a significant amount of existing ontologies represented in description-logic-based OWL. The automatic transformation of these OWL ontologies into ASP offers another source of commonsense knowledge for service robots and also allows to apply the performance-wise superior ASP reasoning techniques to these ontologies [4].

Our approach currently expects that the knowledge that needs to be queried from other robots or synchronised in order to come to a common solution, needs to be part of the task description itself. In order to lift this limitation, we would need to start reason about tasks on a more abstract level and make robots answer questions like "What am I doing?", "Why am I doing it?", and "How am I doing it?".

Some of the knowledge encoded in CN5 is hard to translate to ASP. The concept *your own kitchen*, for example, has the capability to *store cups*. Nevertheless, this does not induce the existence of the *atLocation* relation between cup and kitchen. Unifying the concepts represented in CN5, for example, by trimming unimportant parts like *your own* and ignoring the difference between singular and plural (*fork* vs *forks*) would allow the service robots to utilise even more knowledge encoded in CN5.

[4] Telegram - https://telegram.org/ [last access on October 8th, 2020]

PartIV

Appendices

Publications as (Co-)Author

[1] Stefan Jakob, Stephan Opfer, Alexander Jahl, Harun Baraki and Kurt Geihs. 'Handling Semantic Inconsistencies in Commonsense Knowledge for Autonomous Service Robots'. In: *2020 IEEEE 14th International Conference on Semantic Computing (ICSC)*. IEEE Press, 2020, pp. 136–140.

[2] Stephan Opfer, Stefan Jakob and Kurt Geihs. 'Teaching Commonsense and Dynamic Knowledge to Service Robots'. In: *11th Conference on Social Robotics (ICSR)*. 2019.

[3] Stephan Opfer, Stefan Jakob, Alex Jahl and Kurt Geihs. 'ALICA 2.0 - Domain-Independent Teamwork'. In: *42nd German Conference on Artificial Intelligence*. 2019.

[4] Stephan Opfer. *Towards Description Logic Support for ALICA*. AV Akademikerverlag, 2018. ISBN: 978-620-2-20873-4.

[5] Stephan Opfer, Stefan Jakob and Kurt Geihs. 'Reasoning for Autonomous Agents in Dynamic Domains: Towards Automatic Satisfaction of the Module Property'. In: *Agents and Artificial Intelligence*. Ed. by Jaap van den Herik, Ana Paula Rocha and Joaquim Filipe. Springer International Publishing, 2018, pp. 22–47. ISBN: 978-3-319-93581-2.

[6] Stephan Opfer, Marie Ossenkopf and Kurt Geihs. 'Student Competition Teams: Combining Research and Teaching'. In: *47th Annual Conference of the Southern African Computer Lecturers' Association*. 2018.

[7] Stefan Niemczyk, Stephan Opfer, Nugroho Fredivianus and Kurt Geihs. 'ICE: Self-Configuration of Information Processing in Heterogeneous Agent Teams'. In: *Symposium on Applied Computing*. ACM. 2017, pp. 417–423.

[8] Stephan Opfer, Stefan Jakob and Kurt Geihs. 'Reasoning for Autonomous Agents in Dynamic Domains'. In: *9th International Conference on Agents and Artificial Intelligence - Volume 2 (ICAART)*. INSTICC. 2017, pp. 340–351. ISBN: 978-989-758-220-2.

[9] Stephan Opfer, Stefan Niemczyk and Kurt Geihs. 'Multi-Agent Plan Verification with Answer Set Programming'. In: *3rd Workshop on Model-Driven Robot Software Engineering*. ACM. 2016, pp. 32–39.

[10] Stefan Niemczyk, Dominik Kirchner, Andreas Witsch, Stephan Opfer and Kurt Geihs. ‘Distributed Sensing in a Robotic Soccer Team’. In: *CPSWeek 2014-International Workshop on Robotic Sensor Networks*. 2014, p. 3.

[11] Stephan Opfer. ‘Grenzen von SROIQ bei der Unterstützung der Modellierung von Multi-Agenten-Plänen’. In: *Informatik-Spektrum* 37.1 (2014), pp. 42–45. ISSN: 0170-6012. (in German).

[12] Andreas Witsch, Stephan Opfer and Kurt Geihs. ‘A Formal Multi-Agent Language for Cooperative Autonomous Driving Scenarios’. In: *International Conference on Connected Vehicles & Expo (ICCVE 2014)*. Vienna, Austria, Nov. 2014.

[13] Stephan Opfer. ‘Limits of SROIQ in the Context of Reasoning Support for Multi-Agent Plan Modelling’. In: *GI-Informatiktage 2013 - Smart Life - dank Informatik*. Lecture Notes in Informatics. Bonn: Gesellschaft für Informatik, 2013, pp. 55–58.

[15] Stephan Opfer, Hendrik Skubch and Kurt Geihs. ‘Cooperative Path Planning for Multi-Robot Systems in Dynamic Domains’. In: *Mobile Robots - Control Architectures, Bio-Interfacing, Navigation, Multi Robot Motion Planning and Operator Training*. InTech, 2011. Chap. 11, pp. 237–258. ISBN: 978-953-307-842-7.

Bibliography B

[1] A. Khamis, A. Hussein and A. Elmogy. 'Multi-Robot Task Allocation: A Review of the State-of-the-Art'. In: *Cooperative Robots and Sensor Networks 2015*. Vol. 604, pp. 31–51 (cit. on pp. 21, 29, 41).

[2] Torsten Schaub. *Module Composition*. Potsdam, Germany, 2016-10-13 (cit. on p. 65).

[3] Roland Kaminski. *Consequence of violating the module property? E-mail message*. In collab. with Stephan Opfer. 2020-01-26 (cit. on p. 66).

[4] Stefan Jakob, Alexander Jahl, Harun Baraki and Kurt Geihs. 'Generating Commonsense Ontologies with Answer Set Programming'. In: *Under Review for the Proceedings of the 13th International Conference on Agents and Artificial Intelligence*. 2021 (cit. on p. 150).

[5] Stefan Jakob, Stephan Opfer, Alexander Jahl, Harun Baraki and Kurt Geihs. 'Handling Semantic Inconsistencies in Commonsense Knowledge for Autonomous Service Robots'. In: *2020 IEEEE 14th International Conference on Semantic Computing (ICSC)*. IEEE Press, 2020, pp. 136–140 (cit. on p. 102).

[6] Tamim Asfour, Dillmann Rüdiger, Nikolaus Vahrenkamp, Martin Do, Mirko Wächter, Christian Mandery, Peter Kaiser, Manfred Kröhnert and Markus Grotz. 'The Karlsruhe ARMAR Humanoid Robot Family'. In: *Humanoid Robotics: A Reference*. Ed. by Ambarish Goswami and Prahlad Vadakkepat. 2019, pp. 337–368. ISBN: 978-94-007-6045-5 (cit. on p. 1).

[7] Miguel Campusano, Johan Fabry and Alexandre Bergel. 'Live Programming in Practice: A Controlled Experiment on State Machines for Robotic Behaviors'. In: *Information and Software Technology* 108 (2019), pp. 99–114 (cit. on pp. 111, 112).

[8] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele and Philipp Wanko. *Potassco User Guide*. 2019. URL: https://github.com/potassco/guide/releases/ (visited on) (cit. on p. 59).

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann and Torsten Schaub. 'Multi-shot ASP Solving with Clingo'. In: *Theory and Practise of Logic Programming* 19.1 (2019), pp. 27–82 (cit. on p. 86).

[10] Ricardo Goncalves, Tomi Janhunen, Matthias Knorr, Joao Leite and Stefan Woltran. 'Forgetting in modular answer set programming'. In: *Proceedings of th AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 2843–2850 (cit. on p. 65).

[11] Vladimir Lifschitz. *Answer Set Programming.* Springer, 2019. ISBN: 978-3-030-24657-0 (cit. on p. 59).

[12] Stephan Opfer, Stefan Jakob and Kurt Geihs. 'Teaching Commonsense and Dynamic Knowledge to Service Robots'. In: *International Conference on Social Robotics.* 2019, pp. 645–654 (cit. on pp. 118, 140).

[13] Stephan Opfer, Stefan Jakob, Alex Jahl and Kurt Geihs. 'ALICA 2.0 - Domain-Independent Teamwork'. In: *Joint German/Austrian Conference on Artificial Intelligence.* Springer, 2019, pp. 264–272 (cit. on pp. 110, 112).

[14] Michael Beetz, Daniel Beßler, A. Haidu, M. Pomarlan, A. K. Bozcuoğlu and G. Bartels. 'Know Rob 2.0 — A 2nd Generation Knowledge Processing Framework for Cognition-Enabled Robotic Agents'. In: *2018 IEEE International Conference on Robotics and Automation (ICRA).* 2018, pp. 512–519 (cit. on p. 112).

[15] Bruce Drinkwater and Rob Malkin. 'What next for Fukushima?' In: *Physics World* 31.1 (2018), pp. 29–33. ISSN: 0953-8585 (cit. on p. 3).

[16] Stefan Jakob, Stephan Opfer, Olga Groh, Sebastian Copei and Kevin Lichtenberg. *Wumpus World Simulator.* 2018. URL: wumpus-simulator.org (visited on 16/06/2020) (cit. on p. 126).

[17] Hannes Kisner and Ulrike Thomas. 'Segmentation of 3D Point Clouds using a New Spectral Clustering Algorithm Without a-priori Knowledge'. In: *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications.* Ed. by Alain Tremeau, Francisco Imai and Jose Braz. 2018, pp. 315–322. ISBN: 978-3-030-26756-8 (cit. on pp. 20, 47).

[18] Daniel Neuber. 'Planning of Autonomous and Mobile Robots in Dynamic Environments'. Dissertation. Kassel, Germany: University of Kassel, 2018 (cit. on pp. 43, 85).

[19] Stephan Opfer. *Towards Description Logic Reasoning Support for ALICA.* AV Akademikerverlag, 2018. ISBN: 978-620-2-20873-4 (cit. on pp. 44, 49, 52, 115).

[20] Stephan Opfer, Stefan Jakob and Kurt Geihs. 'Reasoning for Autonomous Agents in Dynamic Domains: Towards Automatic Satisfaction of the Module Property'. In: *Agents and Artificial Intelligence.* Ed. by Jaap van den Herik, Ana Paula Rocha and Joaquim Filipe. Springer International Publishing, 2018, pp. 22–47. ISBN: 978-3-319-93581-2 (cit. on pp. 65, 135).

[21] Simon Ostermann, Michael Roth, Ashutosh Modi, Stefan Thater and Manfred Pinkal. 'SemEval-2018 Task 11: Machine Comprehension Using Commonsense Knowledge'. In: *Proceedings of The 12th International Workshop on Semantic Evaluation.* Association for Computational Linguistics, 2018, pp. 747–757 (cit. on p. 99).

[22] Joseph Redmon and Ali Farhadi. ‘YOLOv3: An Incremental Improvement’. In: *Computing Research Repository* abs/1804.02767 (2018) (cit. on p. 102).

[23] Rolf Schwitter. ‘Specifying and Verbalising Answer Set Programs in Controlled Natural Language’. In: *Theory and Practise of Logic Programming* 3-4 (2018), pp. 691–705 (cit. on pp. 97, 119, 150).

[24] Thao Nguyen Van, Nugroho Fredivianus, Huu Tam Tran, Kurt Geihs and Thi Thanh Binh Huynh. ‘Formal Verification of ALICA Multi-Agent Plans using Model Checking’. In: *Proceedings of the Ninth International Symposium on Information and Communication Technology*. 2018, pp. 351–358 (cit. on p. 149).

[25] Theofanis I. Aravanis and Pavlos Peppas. ‘Belief revision in Answer Set Programming’. In: *Proceedings of the 21st Pan-Hellenic Conference on Informatics*. 2017, p. 2 (cit. on p. 53).

[26] Harald Beck. *Reviewing Justification-based Truth Maintenance Systems from a Logic Programming Perspective*. 2017 (cit. on p. 53).

[27] Harald Beck, Thomas Eiter and Christian Folie. ‘Ticker: A System for Incremental ASP-based Stream Reasoning’. In: *Theory and Practise of Logic Programming* 17 (2017), pp. 744–763 (cit. on p. 53).

[28] Shih-Hsin Chen and Yi-Hui Chen. ‘A Content-Based Image Retrieval Method Based on the Google Cloud Vision API and WordNet’. In: *Intelligent Information and Database Systems*. Ed. by Ngoc Thanh Nguyen, Satoshi Tojo, Minh Le Nguyen and Bogdan Trawinski. Springer International Publishing, 2017, pp. 651–662. ISBN: 978-3-319-54472-4 (cit. on p. 102).

[29] Stephan C. Guy and Rolf Schwitter. ‘The PENG ASP System: Architecture, Language and Authoring Tool’. In: 51.1 (2017), pp. 67–92 (cit. on pp. 97, 119, 150).

[30] Dominik Kirchner. ‘Self-Healing in Autonomous Robots Teams’. PhD thesis. Kassel, Germany: University of Kassel, 2017 (cit. on pp. 43, 68).

[31] Séverin Lemaignan, Mathieu Warnier, E. Akin Sisbot, Aurélie Clodic and Rachid Alami. ‘Artificial cognition for social human-robot interaction: An implementation’. In: *Artificial Intelligence* 247 (2017), pp. 45–69. ISSN: 0004-3702. URL: http://www.sciencedirect.com/science/article/pii/S0004370216300790 (visited on) (cit. on pp. 114, 115).

[32] Dongcai Lu, Yi Zhou, Feng Wu, Zhao Zhang and Xiaoping Chen. ‘Integrating Answer Set Programming with Semantic Dictionaries for Robot Task Planning’. In: *26th International Joint Conference on Artificial Intelligence (IJCAI)*. 2017, pp. 4361–4367 (cit. on p. 119).

[33] Stefan Niemczyk, Stephan Opfer, Nugroho Fredivianus and Kurt Geihs. ‘ICE: Self-Configuration of Information Processing in Heterogeneous Agent Teams’. In: *Symposium on Applied Computing*. 2017, pp. 417–423 (cit. on p. 68).

[34] Stephan Opfer, Stefan Jakob and Kurt Geihs. ‘Reasoning for Autonomous Agents in Dynamic Domains’. In: *Proceedings of the 9th International Conference on Agents and Artificial Intelligence*. Ed. by Jaap van de Herik, Ana Paula Rocha and Joaquim Filipe. ScitePress Digital Library, 2017, pp. 340–351. ISBN: 978-989-758-220-2 (cit. on p. 135).

[35] Teeradaj Racharak, Satoshi Tojo, Nguyen Duy Hung and Prachya Boonkwan. ‘Combining Answer Set Programming with Description Logics for Analogical Reasoning Under an Agent’s Preferences’. In: *Advances in Artificial Intelligence: From Theory to Practice*. Ed. by Salem Benferhat, Karim Tabia and Moonis Ali. 2017, pp. 306–316. ISBN: 978-3-319-60045-1 (cit. on p. 115).

[36] Thomas Röfer. ‘CABSL–C-based Agent Behavior Specification Language’. In: *RoboCup 2017: Robot World Cup XXI*. Ed. by H. Akiyama, O. Obst, C. Sammut and F. Tonidandel. Lecture Notes in Computer Science. Springer, Cham, 2017, pp. 135–142. ISBN: 978-3-030-00307-4 (cit. on pp. 110–112).

[37] Stefan Schiffer and Alexander Ferrein. ‘A System Layout for Cognitive Service Robots’. In: (2017), p. 44 (cit. on p. 117).

[38] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai and Adrian Bolton. ‘Mastering the Game of Go without Human Knowledge’. In: *Nature* 550.7676 (2017), p. 354. ISSN: 1476-4687 (cit. on pp. 20, 47).

[39] Robyn Speer, Joshua Chin and Catherine Havasi. ‘ConceptNet 5.5: An Open Multilingual Graph of General Knowledge’. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 4444–4451 (cit. on pp. 7, 98).

[40] Sebastian G. Brunner, Franz Steinmetz, Rico Belder and Andreas Dömel. ‘RAFCON: A Graphical Tool for Engineering Complex, Robotic Tasks’. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 3283–3290 (cit. on pp. 111, 112).

[41] Yichuan Jiang. ‘A Survey of Task Allocation and Load Balancing in Distributed Systems’. In: *IEEE Transactions on Parallel and Distributed Systems* 27.2 (2016), pp. 585–599 (cit. on p. 29).

[42] Benjamin Kaufmann, Nicola Leone, Simona Perri and Torsten Schaub. ‘Grounding and Solving in Answer Set Programming’. In: *AI Magazine* 37.1609 (2016), pp. 25–32 (cit. on pp. 56, 61, 86).

[43] R. Kim, H. T. Kwon, S. Chi and W. C. Yoon. ‘A Coordination Model for Agent Behaviors Using Hierarchical Finite State Machine with Inter-level Concurrency’. In: *International Conference on Information and Communication Technology Convergence (ICTC)*. 2016, pp. 359–364 (cit. on pp. 111, 112).

[44] Lorenz Mösenlechner. ‘The Cognitive Robot Abstract Machine: A Framework for Cognitive Robotics’. Dissertation. München: Technical University of Munich, 2016 (cit. on p. 112).

[45] Maximilian Nickel, Kevin Murphy, Volker Tresp and Evgeniy Gabrilovich. ‘A Review of Relational Machine Learning for Knowledge Graphs’. In: *Proceedings of the IEEE* 104 (2016), pp. 11–33 (cit. on p. 99).

[46] Stephan Opfer, Stefan Niemczyk and Kurt Geihs. ‘Multi-Agent Plan Verification with Answer Set Programming’. In: *Proceedings of the Third Workshop on Model-Driven Robot Software Engineering*. Ed. by Uwe Aßmann, Davide Brugali and Christian Piechnick. ACM, 2016 (cit. on p. 149).

[47] Stuart Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. 3rd ed. Malaysia: Pearson Eduction Limited, 2016. ISBN: 978-0-13-604259-4 (cit. on pp. 14, 50, 122).

[48] Andreas Witsch. ‘Decision Making for Teams of Mobile Robots’. Dissertation. Kassel, Germany: University of Kassel, 2016 (cit. on pp. 43, 68, 76).

[49] Victor Braberman, Nicolas D’Ippolito, Jeff Kramer, Daniel Sykes and Sebastian Uchitel. ‘MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaption’. In: *1st International Workshop on Control Theory for Software Engineering*. Ed. by Antonio Filieri and Martina Maggio. Vol. 1. 2015 (cit. on p. 18).

[50] Martin Gebser, Roland Kaminski, Philipp Obermeier and Torsten Schaub. ‘Ricochet Robots Reloaded: A Case-Study in Multi-shot ASP Solving’. In: *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation*. Ed. by Thomas Eiter, Hannes Strass, Mirosław Truszczyński and Stefan Woltran. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 17–32. ISBN: 978-3-319-14725-3 (cit. on p. 55).

[51] D.L.R. German Aerospace Center. *Carpe Noctem - Team Universität Kassel (SpaceBot Camp 2015)*. 2015. URL: https://www.youtube.com/watch?v=CPW1DsrrEx8 (visited on 15/03/2019) (cit. on p. 42).

[52] Jeff Kramer. ‘Adventures in Adaptation: A Software Engineering Playground!’ In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. Ed. by Paola Inverardi, Bradley Schmerl and Romina Spalazzese. Piscataway, NJ, USA: IEEE Press, 2015, p. 1. ISBN: 2157-2305 (cit. on p. 18).

[53] Hong Liu, Peng Zhang, Bin Hu and Philip Moore. 'A novel approach to task assignment in a cooperative multi-agent design system'. In: *Applied Intelligence* 43.1 (2015), pp. 162–175 (cit. on p. 29).

[54] Daniel Saur and Kurt Geihs. 'IMPERA: Integrated Mission Planning for Multi-Robot Systems'. In: *Robotics* 4 (2015), pp. 435–463. ISSN: 2218-6581 (cit. on p. 42).

[55] Siddharth Srivastava, Shlomo Zilberstein, Abhishek Gupta, Pieter Abbeel and Stuart Russell. 'Tractability of Planning with Loops'. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. Ed. by Blai Blonet and Sven Koenig. 2015 (cit. on p. 1).

[56] Moritz Tenorth, Jan Winkler, Daniel Beßler and Michael Beetz. 'Open-EASE - A Cloud-Based Knowledge Service for Autonomous Learning'. In: *Künstliche Intelligenz* 29.4 (2015), pp. 407–411 (cit. on p. 112).

[57] Lars Vogel. *Eclipse Rich Client Platform: The complete guide to Eclipse application development*. 3rd ed. Vogella. Hamburg: Vogella, 2015. ISBN: 978-3-943747-13-3 (cit. on p. 36).

[58] Xiaoping Chen, Dongcai Lu, Kai Chen, Yingfeng Chen and Ningyang Wang. *KeJia: the intelligent service robot for RoboCup@ Home 2014*. 2014 (cit. on p. 119).

[59] Martin Gebser, Roland Kaminski, Benjamin Kaufmann and Torsten Schaub. *Clingo = ASP + Control: Extended Report*. University of Potsdam, 2014 (cit. on pp. 56, 61).

[60] Martin Gebser, Roland Kaminski, Benjamin Kaufmann and Torsten Schaub. 'Clingo = ASP + Control: Preliminary Report'. In: *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*. Ed. by M. Leuschel and T. Schrijvers. 2014 (cit. on p. 86).

[61] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge, USA: Cambridge University Press, 2014. ISBN: 978-1-107-02956-9 (cit. on pp. 61, 62).

[62] Stefan Niemczyk, Dominik Kirchner, Andreas Witsch, Stephan Opfer and Kurt Geihs. 'Distributed Sensing in a Robotic Soccer Team'. In: *CPSWeek 2014 - International Workshop on Robotic Sensor Networks*. 2014, p. 3 (cit. on p. 68).

[63] Stephan Opfer. 'Grenzen von SROIQ bei der Unterstützung der Modellierung von Multi-Agenten-Plänen'. In: *Informatik-Spektrum* 37.1 (2014), pp. 42–45. ISSN: 0170-6012. (in German) (cit. on p. 44).

[64] Andreas Witsch, Stephan Opfer and Kurt Geihs. ‘A Formal Multi-Agent Language for Cooperative Autonomous Driving Scenarios’. In: *Proceedings of the International Conference on Connected Vehicles & Expo (ICCVE)*. Ed. by Reinhard Pfliegl, Lee Stogner and Yu Yuan. 2014 (cit. on p. 68).

[65] E. Bastianelli, D. D. Bloisi, R. Capobianco, F. Cossu, G. Gemignani, L. Iocchi and Daniele Nardi. ‘On-line Semantic Mapping’. In: *16th International Conference on Advanced Robotics, ICAR 2013, 25-29 November 2013, Montevideo, Uruguay*. Ed. by José Luis Massera. IEEE Press, 2013, pp. 1–6. ISBN: 978-1-4799-2722-7 (cit. on p. 102).

[66] Xiaoping Chen, Jian-Min Ji, Zhiqiang Sui and Jiongkun Xie. ‘Handling Open Knowledge for Service Robots’. In: *23rd International Joint Conference on Artificial Intelligence (IJCAI)*. 2013, pp. 2459–2465 (cit. on p. 119).

[67] Amit K. Chopra, Alexander Artikis, Jamal Bentahar, Marco Colombetti, Frank Dignum, Nicoletta Fornara, Andrew J. I. Jones, Munindar P. Singh and Pinar Yolum. ‘Research Directions in Agent Communication’. In: *ACM Transactions on Intelligent Systems and Technology* 4.2 (2013), pp. 1–23 (cit. on pp. 21, 107).

[68] Silvia Coradeschi. ‘Interview on Symbol Grounding’. In: *Künstliche Intelligenz* 27.2 (2013), pp. 933–1875 (cit. on p. 13).

[69] James Delgrande, Pavlos Peppas and Stefan Woltran. ‘AGM-Style Belief Revision of Logic Programs Under Answer Set Semantics’. In: *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 2013, pp. 264–276 (cit. on p. 53).

[70] Stephan Opfer, Hendrik Skubch and Kurt Geihs. ‘Limits of SROIQ in the Context of Reasoning Support for Multi-Agent Plan Modelling’. In: *Informatiktage 2013 - Fachwissenschaftlicher Informatik-Kongress*. Ed. by Ludger Porada. Lecture Notes in Informatics. Bonn: Bonner Köllen Verlag 2013, 2013, pp. 55–58. ISBN: 3-88579-446-2 (cit. on p. 112).

[71] Robin Speer and Catherine Havasi. ‘ConceptNet 5: A Large Semantic Network for Relational Knowledge’. In: *The People’s Web Meets NLP*. Ed. by Iryna Gurevych and Jungi Kim. Theory and Applications of Natural Language Processing. New York: Springer Heidelberg, 2013, pp. 161–176. ISBN: 978-3-642-35084-9 (cit. on p. 13).

[72] Esra Erdem, Erdi Aker and Vokan Patoglu. ‘Answer Set Programming for Collaborative Housekeeping Robotics: Representation, Reasoning, and Execution’. In: *Journal of Intelligent Service Robots* 5.4 (2012), pp. 275–291 (cit. on pp. 118, 119).

[73] Martin Gebser, Roland Kaminski, Benjamin Kaufmann and Torsten Schaub. *Answer Set Solving in Practice.* Vol. 6. Morgan & Claypool Publishers, 2012. ISBN: 978-1-60845-971-1 (cit. on pp. 61, 65).

[74] Joohyung Lee and Ravi Palla. 'Reformulating the Situation Calculus and the Event Calculus in the general theory of Stable Models and in Answer Set Programming'. In: *Journal of Artificial Intelligence Research* 43.1 (2012), pp. 571–620 (cit. on p. 117).

[75] Séverin Lemaignan, Raquel Ros, E. Akin Sisbot, Rachid Alami and Michael Beetz. 'Grounding the Interaction: Anchoring Situated Discourse in Everyday Human-Robot Interaction'. In: *International Journal of Social Robots* 4.2 (2012), pp. 181–199. ISSN: 1875-4791 (cit. on p. 115).

[76] Stefan Schiffer, Alexander Ferrein and Gerhard Lakemeyer. 'Caesar: An Intelligent Domestic Service Robot'. In: *Intelligent Service Robotics* 5.4 (2012), pp. 259–273. ISSN: 1861-2776 (cit. on p. 116).

[77] Hendrik Skubch. 'Modelling and Controlling Behaviour of Cooperative Autonomous Mobile Robots'. Dissertation. Kassel: University of Kassel, 2012 (cit. on pp. 6, 8, 23, 28, 29, 32, 35, 85, 110).

[78] Hendrik Skubch. 'Solving Non-Linear Arithmetic Constraints in Soft Realtime Environments'. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing.* New York, 2012, pp. 67–73. ISBN: 978-1-4503-0857-1 (cit. on p. 35).

[79] Gerhard Brewka, Thomas Eiter and Mirosław Truszczyński. 'Answer Set Programming at a Glance'. In: *Communications of the ACM* 54.12 (2011), pp. 92–103 (cit. on pp. 7, 135).

[80] Marc Denecker, Joost Vennekens, Hanne Vlaeminck, Johan Wittocx and Maurice Bruynooghe. 'Answer Set Programming's Contributions to Classical Logic'. In: *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday.* Ed. by Marcello Balduccini and Tran Cao Son. Berlin and Heidelberg: Springer Berlin Heidelberg, 2011, pp. 12–32. ISBN: 978-3-642-20831-7 (cit. on p. 66).

[81] Stephan Opfer, Hendrik Skubch and Kurt Geihs. 'Cooperative Path Planning for Multi-Robot Systems in Dynamic Domains'. In: *Mobile Robots - Control Architectures, Bio-Interfacing, Navigation, Multi Robot Motion Planning and Operator Training.* Ed. by Janusz Bedkowski. 2011, pp. 237–258 (cit. on p. 68).

[82] Dan Rubel, Jaime Wren and Eric Clayberg. *The Eclipse Graphical Editing Framework (GEF).* 1st ed. Eclipse Series. Boston, Mass. and London: Addison-Wesley Professional and Addison-Wesley, 2011. ISBN: 978-0-321-71838-9 (cit. on p. 36).

[83] Hendrik Skubch, Michael Wagner, Roland Reichle and Kurt Geihs. ‘A Modelling Language for Cooperative Plans in Highly Dynamic Domains’. In: *Special Issue on Advances in Intelligent Robot Design for the Robocup Middle Size League*. Ed. by M.J.G. van de Molengraft and Oliver Zweigle. Vol. 21. Elsevier, 2011, pp. 423–433 (cit. on p. 43).

[84] Moritz Tenorth. ‘Knowledge Processing for Autonomous Robots’. Dissertation. München: Technical University of Munich, 2011 (cit. on p. 112).

[85] M. Waibel et al. ‘RoboEarth’. In: *IEEE Robotics and Automation Magazine* 18.2 (2011), pp. 69–82. ISSN: 1558-223X (cit. on p. 112).

[86] Mario Alviano, Wolfgang Faber and Nicola Leone. ‘Disjunctive ASP with functions: Decidable queries and effective computation’. In: *Theory and Practise of Logic Programming* 10.4-6 (2010), pp. 497–512 (cit. on p. 66).

[87] Michael Beetz, L. Mösenlechner and Moritz Tenorth. ‘CRAM — A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments’. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 1012–1017 (cit. on pp. 110–112).

[88] Xiaoping Chen, Jianmin Ji, Jiehui Jiang, Guoqiang Jin, Feng Wang and Jiongkun Xie. ‘Developing High-level Cognitive Functions for Service Robots’. In: *Proceedings of th 9th International Conference on Autonomous Agents and Multiagent Systems*. Vol. 1. 2010, pp. 989–996. ISBN: 978-0-9826571-1-9 (cit. on p. 119).

[89] Séverin Lemaignan, Raquel Ros, Lorenz Mösenlechner, Rachid Alami and Michael Beetz. ‘ORO - A Knowledge Management Platform for Cognitive Architectures in Robotics’. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 3548–3553 (cit. on p. 115).

[90] Thomas Lukasiewicz. ‘A Novel Combination of Answer Set Programming with Description Logics for the Semantic Web’. In: 22.11 (2010), pp. 1577–1592 (cit. on p. 115).

[91] Matthias Fichtner. ‘Anchoring Symbols to Percepts in the Fluent Calculus’. Dissertation. Dresden: Technical University of Dresden, 2009 (cit. on p. 118).

[92] Hiroyasu Iwata and Shigeki Sugano. ‘Design of human symbiotic robot TWENDY-ONE’. In: *2009 IEEE International Conference on Robotics and Automation*. Ed. by Antonio Bicchi. 2009, pp. 580–586 (cit. on p. 1).

[93] Tomi Janhunen, Emilia Oikarinen, Hans Tompits and Stefan Woltran. ‘Modularity Sspects of Disjunctive Stable Models’. In: *Journal of Artificial Intelligence Research* 35 (2009), pp. 813–857 (cit. on p. 65).

[94] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2009 (cit. on pp. 11, 14).

[95] Luis von Ahn. 2008. URL: https://www.cmu.edu/homepage/computing/2008/summer/games-with-a-purpose.shtml (visited on 25/10/2020) (cit. on p. 98).

[96] Philipp A. Baer. 'Platform-Independent Development of Robot Communication Software'. PhD thesis. Kassel, Germany: University of Kassel, 2008 (cit. on p. 68).

[97] James Delgrande, Torsten Schaub, Hans Tompits and Stefan Woltran. 'Belief Revision of Logic Programs under Answer Set Semantics'. In: *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning*. Vol. 8. 2008, pp. 411–421 (cit. on p. 53).

[98] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer and Hans Tompits. 'Combining answer set programming with description logics for the Semantic Web'. In: *Artificial Intelligence* 172.12 (2008), pp. 1495–1539. ISSN: 0004-3702 (cit. on p. 115).

[99] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub and Sven Thiele. 'Engineering an Incremental ASP Solver'. In: *International Conference on Logic Programming*. Springer, 2008, pp. 190–205 (cit. on p. 65).

[100] Guus Schreiber. 'Knowledge Engineering'. In: *Handbook of Knowledge Representation*. Ed. by Frank van Harmelen, Vladimir Lifschitz and Bruce Porter. Foundations of Artificial Intelligence. United Kingdom: Elsevier, 2008, pp. 929–946. ISBN: 978-0-444-52211-5 (cit. on p. 52).

[101] Dave Steinberg, Frank Budinsky, Marcelo Paternostro and Ed Merks. *EMF: Eclipse Modeling Framework*. 2nd ed. Eclipse Series. Addison-Wesley Professional, 2008. ISBN: 978-0-321-33188-5 (cit. on p. 36).

[102] Frank van Harmelen, Vladimir Lifschitz and Bruce Porter, eds. *Handbook of Knowledge Representation*. Foundations of Artificial Intelligence. United Kingdom: Elsevier, 2008. ISBN: 978-0-444-52211-5 (cit. on p. 116).

[103] Alexander Ferrein. 'Robot Controllers for Highly Dynamic Environments With Real-time Constraints'. Dissertation. Aachen: Rheinisch-Westfälischen Technischen Hochschule Aachen, 2007 (cit. on pp. 116, 117).

[104] Mantas Simkus and Thomas Eiter. 'FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols'. In: *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasonin*. 2007 (cit. on p. 66).

[105] Chaim Zins. ‘Conceptual Approaches for Defining Data, Information, and Knowledge’. In: *Journal of the American Society for Information Science and Technology* 58.4 (2007), pp. 479–493 (cit. on p. 46).

[106] Luis von Ahn, Mihir Kedia and Manuel Blum. ‘Verbosity: a game for collecting common-sense facts’. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2006, pp. 75–78 (cit. on p. 98).

[107] M. Loetzsch, M. Risler and M. Jungel. ‘XABSL - A Pragmatic Approach to Behavior Engineering’. In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2006, pp. 5124–5129 (cit. on pp. 110–112).

[108] Boris Motik, Ian Horrocks, Riccardo Rosati and Ulrike Sattler. ‘Can OWL and Logic Programming Live Together Happily Ever After?’ In: *The Semantic Web - ISWC 2006*. Ed. by Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold and LoraM Aroyo. Vol. 4273. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 501–514. ISBN: 978-3-540-49029-6 (cit. on pp. 52, 115).

[109] Bryan Horling and Victor Lesser. ‘A Survey of Multi-Agent Organizational Paradigms’. In: *The Knowledge Engineering Review* 19.4 (2005), pp. 281–316 (cit. on p. 21).

[110] Mariarosaria Taddeo and Luciano Floridi. ‘The symbol grounding problem: A critical review of fifteen years of research’. In: *Journal of Experimental and Theoretical Artificial Intelligence* 17.4 (2005), pp. 419–445 (cit. on p. 48).

[111] Michael Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Vol. 33. Applied Logic Series. Springer Science & Buisness Media, 2005. ISBN: 978-1-4020-3068-0 (cit. on p. 117).

[112] Milan Zeleny. *Human Systems Management: Integrating Knowledge,Management and Systems*. World Scientific, 2005. ISBN: 978-981-02-4913-7 (cit. on p. 48).

[113] Brian P. Gerkey and Maja J. Matarić. ‘A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems’. In: *Journal of Robotics Research* 23.9 (2004), pp. 939–954 (cit. on p. 21).

[114] Michael Thielscher. ‘FLUX: A Logic Programming Method for Reasoning Agents’. In: *Theory and Practise of Logic Programming* 5 (2004) (cit. on p. 117).

[115] Ronald J. Brachman and Hector J. Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Series in Artificial Intelligence: Morgan Kaufmann, 2003. ISBN: 978-1558609327 (cit. on pp. 2, 20, 47, 48).

[116] Marc Denecker, Victor W. Marek and Miroslaw Truszczynski. ‘Uniform Semantic Treatment of Default and Autoepistemic Logic’. In: *Artificial Intelligence* 143.1 (2003), pp. 79–122. ISSN: 0004-3702 (cit. on p. 53).

[117] Matthias Fichtner, Axel Großmann and Michael Thielscher. ‘Intelligent Execution Monitoring in Dynamic Environments’. In: *Fundamenta Informaticae* 57.2-4 (2003), pp. 371–392 (cit. on p. 118).

[118] Stijn Heymans and Dirk Vermeir. ‘Integrating Description Logics and Answer Set Programming’. In: *Principles and Practice of Semantic Web Reasoning.* Ed. by Francois Bry, Nicola Henze and Jan Maluszynski. Springer Berlin Heidelberg, 2003, pp. 146–159 (cit. on p. 115).

[119] Jeffrey O. Kephart and David M. Chess. ‘The Vision of Autonomic Computing’. In: *Computer* 36.1 (2003), pp. 41–50 (cit. on p. 16).

[120] Michael Wooldridge. *Reasoning about Rational Agents.* MIT Press, 2003 (cit. on pp. 11, 14).

[121] Franz Baader and Ulrike Sattler. ‘An Overview of Tableau Algorithms for Description Logics’. In: *Studia Logica* 69.1 (2001), pp. 5–40. ISSN: 0039-3215 (cit. on p. 55).

[122] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press, 2001. ISBN: 978-0-262-18218-8 (cit. on p. 116).

[123] Christian Igel and Michael Hüsken. ‘Improving the Rprop Learning Algorithm’. In: *Proceedings of the Second International Symposium on Neural Computation.* ICSC Academic Press, 2000, pp. 115–121 (cit. on p. 35).

[124] Patrik Simons. ‘Extending and Implementing the Stable Model Semantics’. Dissertation. Helsinki: University of Technology, 2000 (cit. on p. 61).

[125] Fabio Bellifemine, Agostino Poggi and Giovanni Rimassa. ‘JADE - A FIPA-compliant agent framework’. In: *PAAM99 Proceedings.* Ed. by Hyacinth Nwana and Divine Ndumu. Vol. 99. 1999, pp. 33–45 (cit. on pp. 21, 107).

[126] Rob Miller and Murray Shanahan. ‘The Event Calculus in Classical Logic - Alternative Axiomatisations’. In: *Electronic Transactions on Artificial Intelligence* 3 (1999), pp. 77–105 (cit. on p. 117).

[127] Gerhard Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* MIT Press, 1999 (cit. on pp. 11, 14, 20).

[128] Erann Gat. ‘On Three-Layer Architectures’. In: *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems.* Ed. by David Kortenkamp, R. Peter Bonasso and Robin Murphy. Cambridge, USA: MIT Press, 1998, pp. 195–210. ISBN: 0-262-61137-6 (cit. on p. 17).

[129] Paul D. O'Brien and Richard C. Nicol. 'FIPA - Towards a Standard for Software Agents'. In: *BT Technology Journal* 16.3 (1998), pp. 51–59. ISSN: 1358-3948 (cit. on pp. 21, 107).

[130] Tomas Sander and Christian F. Tschudin. 'Protecting Mobile Agents Against Malicious Hosts'. In: *Mobile Agents and Security*. Ed. by Giovanni Vigna. Springer Berlin Heidelberg, 1998, pp. 44–60. ISBN: 978-3-540-64792-8 (cit. on p. 21).

[131] Michael Thielscher. 'Introduction to the Fluent Calculus'. In: *Electronic Transactions on Artificial Intelligence* 2.3-4 (1998), pp. 179–192 (cit. on p. 117).

[132] Yannis Labrou and Tim Finin. *Comments on the specification for FIPA '97 AGENT COMMUNICATION LANGUAGE*. 1997 (cit. on pp. 21, 107).

[133] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin and Richard B. Scherl. 'GOLOG: A logic programming language for dynamic domains'. In: *Journal of Logic Programming* 31.1-3 (1997), pp. 59–83 (cit. on p. 116).

[134] David H. Wolpert and William G. Macready. 'No Free Lunch Theorems for Optimization'. In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82 (cit. on p. 134).

[135] Erich Gamma, Richard Helm, Ralph E. Johnson and John Vlissides. *Design Datterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Reading, Mass. and Wokingham: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0 (cit. on p. 36).

[136] David H. Wolpert and William G. Macready. *No Free Lunch Theorems for Search*. 1995 (cit. on p. 134).

[137] Rajeev Alur and David L. Dill. 'A Theory of Timed Automata'. In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. ISSN: 0304-3975 (cit. on p. 24).

[138] Tim Finin, Richard Fritzson, Don McKay and Robin McEntitre. 'KQML as an agent communication language'. In: *Proceedings of the Third International Conference on Information and Knowledge Management*. New York, NY, USA: ACM, 1994. ISBN: 978-0-89791-674-5 (cit. on pp. 21, 107).

[139] Vladimir Lifschitz and H. Turner. 'Splitting a Logic Program.' In: *Proceedings of the 11th International Conference on Logic Programming*. Ed. by P. V. Hentenryck. MIT Press, 1994, pp. 23–37 (cit. on p. 65).

[140] Randall Davis, Howard Shrobe and Peter Szolovits. 'What Is a Knowledge Representation?' In: *AI Magazine* 14.1 (1993), pp. 17–33 (cit. on pp. 49, 50).

[141] Hector J. Levesque, Philip R. Cohen and José H. T. Nunes. 'On Acting Together'. In: *AAAI'90 Proceedings of the eighth National conference on Artificial Intelligence*. 1990, pp. 94–99. ISBN: 978-0-262-51057-8 (cit. on p. 30).

[142] Russell L. Ackhoff. 'From Data to Wisdom'. In: *Journal of Applied Systems Analysis* 16.1 (1989), pp. 3–9 (cit. on p. 46).

[143] Michael E. Bratman, David J. Israel and Martha E. Pollack. 'Plans and Resource-Bounded Practical Reasoning'. In: *Computational Intelligence* 4.3 (1988), pp. 349–355. ISSN: 1467-8640 (cit. on p. 18).

[144] Matthew L. Ginsberg and David E. Smith. 'Reasoning About Action I: A Possible Worlds Approach'. In: *Artificial Intelligence* 35.2 (1988), pp. 165–195. ISSN: 0004-3702 (cit. on p. 51).

[145] Nicole Bidoit and Christine Froidevaux. 'Minimalism subsumes Default Logic and Circumscription in Stratified Logic Programming'. In: *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*. 1987, pp. 89–97 (cit. on p. 53).

[146] Michael Gelfond. 'On Stratified Autoepistemic Theories'. In: *Proceedings of the Sixth National Conference on Artificial Intelligence*. Ed. by Kenneth Forbus and Howard Shrobe. 1987, pp. 207–211 (cit. on p. 53).

[147] Steve Hanks and Drew McDermott. 'Nonmonotonic Logic and Temporal Projection'. In: *Artificial Intelligence* 33.3 (1987), pp. 379–412. ISSN: 0004-3702 (cit. on p. 51).

[148] Robert Moore. 'Possible-World Semantics for Autoepistemic Logic'. In: *Readings in Nonmonotonic Reasoning*. Morgan Kaufmann, 1987, pp. 137–142. ISBN: 978-0-934613-45-3 (cit. on p. 53).

[149] Hector J. Levesque and Ronald J. Brachman. 'A Fundamental Tradeoff in Knowledge Representation and Reasoning'. In: *Proceedings of the Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*. 1984 (cit. on p. 50).

[150] Nils J. Nilsson. *Shakey the Robot*. 1984 (cit. on p. 17).

[151] Patrick J. Hayes. 'The Frame Problem and Related Problems in Artificial Intelligence'. In: *Readings in Artificial Intelligence*. Ed. by Bonnie Lynn Webber and Nils J. Nilsson. San Francisco, CA, USA: Morgan Kaufmann, 1981, pp. 223–230. ISBN: 978-1-4832-1440-5 (cit. on p. 51).

[152] M. Sharir. 'A Strong-Connectivity Algorithm and its Applications in Data Flow Analysis'. In: *Computers & Mathematics with Applications* 7.1 (1981), pp. 67–72. ISSN: 0898-1221 (cit. on p. 65).

[153] John McCarthy. 'Circumscription - A Form of Non-Monotonic Reasoning'. In: *Artificial Intelligence* 13.1-2 (1980), pp. 27–39. ISSN: 0004-3702 (cit. on p. 51).

[154] Raymond Reiter. 'A Logic for Default Reasoning'. In: *Artificial Intelligence* 13.1-2 (1980), pp. 81–132. ISSN: 0004-3702 (cit. on p. 53).

[155] John R. Searle. 'Minds, Brains, and Programs'. In: *Behavioural and Brain Sciences* 3.3 (1980), pp. 417–424. ISSN: 0140-525X (cit. on pp. 12, 13).

[156] Stevan Harnad. *Journal of Behavioural and Brain Sciences*. Cambridge University Press, 1980 (cit. on p. 13).

[157] Jon Doyle. 'A Truth Maintenance System'. In: *Artificial Intelligence* 12.3 (1979), pp. 231–272. ISSN: 0004-3702 (cit. on p. 52).

[158] Richard E. Fikes and Nils J. Nilsson. 'STRIPS: A new approach to the application of the theorem proving to problem solving'. In: *Artificial Intelligence* 2.3-4 (1971), pp. 189–208. ISSN: 0004-3702 (cit. on p. 19).

[159] John McCarthy and Patrick J. Hayes. 'Some Philosophical Problems from the Standpoint of Artificial Intelligence'. In: *Readings in Artificial Intelligence* (1969), pp. 431–450 (cit. on p. 51).

[160] John R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969. ISBN: 978-1-139-17343-8 (cit. on pp. 21, 107).

[161] Saul A. Kripke. 'Semantical Considerations Of The Modal Logic'. In: *Studia Philosophica* 1 (1963) (cit. on pp. 24, 53).

[162] Martin Davis, George Logemann and Donald Loveland. 'A Machine Program for Theorem-Proving'. In: *Communications of the ACM* 5.7 (1962), pp. 394–397 (cit. on pp. 52, 55).

[163] Carl Adam Petri. 'Kommunikation mit Automaten'. Dissertation. Bonn: University of Bonn, 1962 (cit. on p. 24).

[164] Alan Turing. 'Computing Machinery and Intelligence'. In: *Mind* 59.236 (1950), pp. 433–460 (cit. on p. 12).

[165] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Courier Corporation, 1931 (cit. on p. 52).

Index C

List of Figures D

List of Tables E

List of Listings

F

G Network Throughput

In Section 12.4, we did not focus on absolute numbers in terms of throughput or latency. Instead, our experiment tested Cap'n Zero and the ROS middleware under the same circumstances in order to get a relative comparison between both middlewares. The message size was doubled until the number of messages both middlewares dropped were significant, and we could assume that some limit was reached. A theoretical limit would be the bandwidth of the utilised network. In the experiment, where the communicating processes were located on different machines, the network had a theoretical bandwidth of 100 Mbit/s. In case of a usual TCP connection, utilising a maximum transmission unit (MTU) of 1500 byte, the overhead for the involved protocols would sum up to 6% of that bandwidth. Therefore the theoretical limit would be between 2^{21} and 2^{22} Integers per second. Since both libraries also have to serialise the integers and add further overhead with their protocols, we hit the limit much earlier, and a first significant number of message drops are recognised for ROS UDP at 2^{15} Integers and for all other protocols at 2^{20} Integers. The ROS version utilised in this experiment is ROS Melodic with Version 1.14.3 of the roscpp library. Details about the lost messages are given in the following two tables.

Ints per Msg	**CZ-IPC**	**CZ-UDP**	**CZ-TCP**	**ROS-UDP**	**ROS-TCP**
2	0.4	0	0.2	0.9	0.5
3	0	0	0	0	0
5	0	0	0	0	0
9	0	0	0	0	0
17	0	0	0	0	0
33	0	0	0	0	0
65	0	0	0	0	0
129	0	0	0	0	0
257	0	0	0	0	0
513	0	0	0	0	0
1025	0	-	0	0	0
2049	0	-	0	0	0
4097	0	-	0	0	0
8193	0	-	0	0	0
16385	0	-	0	0	0
32769	0	-	0	0	0.1
65537	0	-	0	0	0
131073	0	-	0	0	0
262145	0	-	0	0	0.2
524289	0	-	0	0	0.2
1048577	0	-	0	0	2.8

Table G.1: Percentage of Lost Messages for Local Transmission.

Ints per Msg	**CZ-UDP**	**CZ-TCP**	**ROS-UDP**	**ROS-TCP**
2	0	0.2	0.5	1.3
3	0	0	0	0
5	0	0	0	0
9	0	0	0	0
17	0	0	0	0
33	0	0	0	0
65	0	0	0	0
129	0	0	0	0
257	0	0	0	0
513	0	0	0	0
1025	-	0	0	0
2049	-	0	0	0
4097	-	0	0	0
8193	-	0	0	0
16385	-	0	0.1	0
32769	-	0	95.0	0
65537	-	0	94.7	0
131073	-	0	92.8	0
262145	-	0	1.1	0
524289	-	0	99.9	0
1048577	-	44.8	-	3.6

Table G.2: Percentage of Lost Messages for Network Transmission.