A Self-Organising Multi-Agent Knowledge Base

Dissertation by

Stefan Jakob

in partial fulfillment of the requirements

for the academic degree

Doktor der Naturwissenschaften (Dr. rer. nat.)

Submitted to the Faculty of

Faculty of Electrical Engineering and Computer Science

University of Kassel, Germany

December, 2021

1st Reviewer:	Prof. Dr. Kurt Geihs
2nd Reviewer:	Prof. Dr. Ralf Steinmetz
Date of Defense:	December 21, 2021
Committee:	Prof. Dr. Kurt Geihs, Prof. Dr. Ralf Steinmetz,
	Prof. Dr. Jan Marco Leimeister, Prof. Dr. Gerd Stumme

Contents

Zusammenfassung

Digitale Städte sind zunehmend auf IT- und Kommunikationsinfrastruktur angewiesen. Sie können als große verteilte Systeme betrachtet werden, die aus heterogenen und autonomen Teilnehmern bestehen. Besonders in Notfall- und Krisensituationen muss kritisches Wissen, beispielsweise über verletzte Personen oder beschädigte Infrastruktur, zuverlässig auch unter eingeschränkter Kommunikation erreichbar sein. In einer solchen Umgebung würde eine zentrale Verwaltung des kritischen Wissens einen Engpass verursachen. Daher wird in dieser Dissertation das Design und die Umsetzung einer selbst organisierenden Multi-Agenten-basierten verteilten Wissensbasis vorgestellt. Sie unterstützt nicht-monotones Schlussfolgern (Reasoning) und die verteilte Speicherung von semantisch annotiertem Wissen sowie Fehlertoleranz bei Teilausfällen des Systems.

Neben der dezentralen Speicherung ist die Anwendung von Allgemeinwissen essenziell für die Interaktion von Mensch und Maschine. Dies beruht auf der Art, wie Menschen auf natürliche Weise miteinander kommunizieren, in der die meisten Details aufgrund des allgemeinen Hintergrundwissens normalerweise weggelassen werden. Um eine solche Kommunikation mit einer Maschine bzw. einem Roboter zu ermöglichen, muss das System mit einer entsprechenden Wissensrepräsentation ausgestattet sein. Ontologien könnten ein geeigneter Ansatz sein. Derzeitige Ontologie-Frameworks weisen jedoch keine dynamische Anpassungsfähigkeit auf, sind monoton und sind nicht für große Datenmengen ausgelegt. Daher stellt diese Dissertation einen Formalismus vor mit dem nicht-monotone Ontologien basierend auf einer Allgemeinwissensquelle dynamisch erstellt werden können. Die generierten Ontologien werden anschließend auf Inkonsistenzen geprüft und innerhalb der verteilten Wissensbasis zur Annotation der gespeicherten Wissenselemente verwendet.

Ein weiterer zentraler Punkt in der dezentralen Verwaltung von Wissen in einem lose gekoppelten Netz ist die Entdeckung von Wissen (Knowledge Discovery). Der Fokus liegt dabei auf semantischen Informationen. Dies ist besonders wichtig, wenn Rettungskräfte mit der verteilten Wissensbasis interagieren, da sie hauptsächlich an dem gespeicherten Wissen und nicht dem Speicherort interessiert sind. Klassische Mechanismen des IP-basierten Routings müssen dazu Flooding (Fluten des Netzes mit Nachrichten) einsetzen, da sie im schlechtesten Fall jeden Knoten abfragen müssen, um entsprechende Informationen zu finden. Dies führt zu einer zusätzlichen Belastung der potenziell beschädigten Kommunikationsinfrastruktur. Deshalb wird in dieser Dissertation zusätzlich ein semantischer Routing-Mechanismus vorgestellt, der auf lose gekoppelte Netzwerke zugeschnitten ist, die unstrukturiert sind und sich dynamisch ändern können.

Abstract

Future digital cities increasingly depend on IT and communication infrastructure. They can be viewed as large-scale distributed systems consisting of heterogeneous and autonomous participants. Particularly in emergencies and crises, critical knowledge, such as knowledge about injured people or damaged infrastructure, must be reliably accessible even with limited communication. In such an environment, a centralised knowledge management system would create a bottleneck. Therefore, the design and implementation of a self-organizing multi-agent-based distributed knowledge base are presented in this dissertation. It supports non-monotonic reasoning, the distributed storage of semantically annotated knowledge, and fault tolerance for partial system failures.

In addition to decentralised storage, the use of commonsense knowledge is essential for human-machine interaction. This is due to the way people naturally communicate with each other, where most of the details are usually left out, relying on commonsense knowledge. To enable such communication with a machine or a robot, the system must be equipped with corresponding knowledge representation and reasoning mechanisms. Ontologies could be a suitable approach. However, current ontology frameworks do not support dynamic adaptation of the ontologies, are monotonic, and are not designed for large amounts of data. Therefore, this dissertation presents a mechanism that generates dynamic and non-monotonic ontologies based on a commonsense knowledge source. The generated ontologies are then checked for inconsistencies and used within the distributed knowledge base to annotate the stored knowledge items.

Another central aspect of a decentralised knowledge management system in loosely coupled networks is discovering knowledge focussing on semantics. This is especially important when rescuers interact with the distributed knowledge base, as they are primarily interested in the stored knowledge, not in its location. Classic mechanisms of IP-based routing have to use flooding since they have to query every node in the worst case to find the relevant information. This leads to an additional burden on the potentially damaged communication infrastructure. Therefore, a semantic routing mechanism is presented in this dissertation, which is tailored to loosely coupled networks that are unstructured and can change dynamically.

Acknowledgements

In this section, I would like to thank everyone who accompanied and supported me during my thesis.

To begin with, I want to thank my supervisor Professor Kurt Geihs, who encouraged me to take my doctoral degree. He always believed in my research and supported me during my studies. Additionally, I would like to thank Professor Ralf Steinmetz, who provided insightful comments on my thesis.

During my years at the University of Kassel, I was part of the RoboCup team Carpe Noctem Cassel. Being a member of this team enabled me to grow scientifically and personally. Furthermore, it led me to the decision to write this thesis. Therefore, I would like to thank everyone who was part of Carpe Noctem Cassel. Most notably, these members are: Hendrik Skubch, Eduard Belsch, Jewgeni Beifuß, Kai Liebscher, Florian Seute, Tim Schlüter, Till Amma, Tobias Schellien, Lukas Will, Dennis Bachmann, Thore Braun, Michael Gottesleben, Lisa Martmann, and Yannick Schlamm.

I would like to thank my colleagues and friends Stephan Opfer, Alexander Jahl, Harun Baraki, Andreas Witsch, and Stefan Niemczyk for their inspiring collaboration, discussions, hours of work before approaching deadlines, and many table soccer games.

Furthermore, I would like to thank Eric Douglas Nyakam Chiadjeu, who supported me in implementing and evaluating my ideas.

This work has been performed in the context of the LOEWE center emergenCITY.

Additionally, I would like to thank my parents Heidi and Robert Jakob that supported and encouraged me during my studies and time as a PhD candidate. Last but not least, I would like to thank my fiancée Olga Groh, who accompanied and supported me during this time.

Part I

Preliminaries

1 Introduction

In recent years, cities have expanded their digital infrastructure. In general, these Smart Cities rely on information and communication technologies to collect data from different devices, e. g., IoT (Internet of Things) devices distributed in the city. On the one hand, this supports city officials in managing different assets like energy, water, waste, transport, health, public safety or the general city administration. On the other hand, these technologies enable the provision of services to citizens, for example, efficient navigation or suggestions of free parking spaces.

Around the globe, there are several examples of Smart Cities. At the time of writing, the Smart Nation Singapore is the globally top-ranked Smart City[1]. Based on the *Smart Nation Initiative* [134], the Singaporean government introduced information and communication infrastructure to achieve better living standards, stronger communities, and more participation opportunities to all citizens. The initiative encompasses several platforms and services. These include an open-data platform, which is used to share machine-readable datasets from various agencies. Additionally, they provide a map service that enables the incorporation of data by citizens and, thus, the crowdsourcing of spatial data. For example, the resulting data is then used to optimise navigation services. Furthermore, the cooperation of research institutions, public agencies, and private companies is fostered by corresponding platforms. Additional data is provided by the Smart Nation Sensor Platform[2], which is a nationwide network of sensors integrated into lampposts (Lamppost-as-a-Platform). The goal of the Smart Nation Sensor Platform is to equip more than 95.000 street lamps with LEDs and to expand their functionality with cameras as well as additional sensors including temperature, humidity, noise, or pollutants. During the COVID-19 pandemic, (semi-)autonomous mobile robots have been introduced to support officials during their operations. Therefore, the four-legged platform SPOT[3] by Boston Dynamics has been chosen[4]. It is used to broadcast messages to visitors of parks and delivers goods to patients in isolation. Finally, the complete system is administered by a central agency.

[1]IMD Smart City Index 2019, https://www.imd.org/research-knowledge/reports/imd-smart-city-index-2019/, Accessed April 14, 2021.

[2]Smart Nation Sensor Platform, https://www.smartnation.gov.sg/what-is-smart-nation/initiatives/Strategic-National-Projects/smart-nation-sensor-platform, Accessed April 15, 2021.

[3]SPOT, https://www.bostondynamics.com/spot, Accessed April 15, 2021.

[4]Pillars of Smart Nation, https://www.smartnation.gov.sg/docs/default-source/default-document-library/dgb-public-document_30dec20.pdf?sfvrsn=626dc45f_2, Accessed April 15, 2021.

Systems like the Smart Nation Singapore can be considered as large-scale distributed systems, which consist of heterogeneous participants that rely on communication to exchange information and knowledge. A centralised managing instance, however, may lead to several disadvantages. For example, relying on a central instance to store information and knowledge in such large-scale systems introduces a single point of failure and a potential bottleneck to the system. This is especially the case during emergencies or natural disasters. For example, a strong earthquake can have dramatic consequences for a Smart City. In this case, people could be injured, buildings could collapse, and parts of critical infrastructure like gas or water pipelines as well as the communication infrastructure could be destroyed. To respond to the crisis, rescuers need access to essential information and knowledge. Since the communication infrastructure could be affected, a stable connection to central knowledge management platforms or knowledge bases is not guaranteed. If this central instance fails, critical information and knowledge like free hospital beds or available supplies can no longer be accessed.

To prevent these problems and to increase the resilience of large-scale distributed systems like Smart Cities, critical information and knowledge should be managed decentrally. Since the critical information and knowledge is distributed on multiple nodes over the network, failures of single nodes no longer lead to a failure of the system. However, additional computational power and communication are needed to maintain a decentralised knowledge management platform or a decentralised knowledge base. Several approaches have been proposed in recent years that rely on distributed databases, which are discussed in detail in Chapter 3. In contrast to these approaches, we will create a decentralised knowledge base utilising a Multi-Agent System (MAS). MAS are typically decentralised, are loosely coupled, and can act in unstable heterogeneous environments. Thus, they provide the necessary features to create a distributed knowledge base that can be used to manage knowledge in highly dynamic domains like Search & Rescue. The following section describes the problem addressed in this thesis and the challenges that arise in detail.

1.1 Problem Statement

The core objectives of this thesis can be summarised as follows:

> *The development of the conceptual foundations of a distributed, multi-agent-based knowledge base, which is capable of*
>
> - *symbolically representing and reasoning about commonsense knowledge,*
> - *detecting and preventing semantic inconsistencies,*
> - *discovering and managing semantically annotated knowledge,*
> - *providing scalability and failure tolerance*
>
> *within a loosely coupled network of heterogeneous participants.*

This research goal encompasses several aspects, which have to be considered. The following paragraphs elaborate on these aspects and form the scope of this thesis.

Knowledge

One of the main goals of this thesis is the distributed management of symbolically represented and annotated knowledge. Therefore, a clear definition of the term *knowledge* is needed. Ackoff presents in [1] a hierarchy that defines the differences between data, information, and knowledge. On the lowest level of the hierarchy is the term *data*, which is gathered by observing the environment. By processing *data*, *information* is created. Subsequently, *knowledge* is formed by the relations between pieces of *information* as well as the application of *data* and *information*. A complete description of this hierarchy is given in Section 2.3.

The distributed knowledge base requires an appropriate knowledge representation and reasoning language. To support both humans and machines as users of the knowledge base, a suitable knowledge representation language should rely on symbols since they can be interpreted by both and can be easily expanded with semantics. Furthermore, the incorporation of commonsense knowledge shall, on the one hand, ease the organisation of the knowledge based on its semantics and, on the other hand, support the manipulation of knowledge by users. However, commonsense knowledge is prone to contain inconsistencies. Therefore, the knowledge representation has to provide mechanisms to detect and possibly prevent them. Finally, the knowledge base must be able to cope with adaptations of the knowledge at runtime since users can propose additional knowledge, inconsistencies can arise, or knowledge can be outdated.

Environment

Environments like Fog Computing (see Section 1.2.1) or Search & Rescue scenarios (see Section 1.2.2) introduce several challenges, which have to be considered by a distributed knowledge base. One of these challenges is the dynamically changing number of participants. Especially in Search & Rescue scenarios, communication between the involved participants (unmanned aerial vehicles, mobile robots, stationary smart infrastructure, injured persons, rescuers, etc.) may be unreliable. One of these scenarios is the recovery of injured persons after an earthquake. To ensure a reliable exchange of knowledge between the actors, the distributed knowledge base has to cope with unstable communication. Furthermore, it must provide mechanisms to handle partial failures and rebuild its management.

The unstructured nature of the networks gives another challenge in both environments (Fog Computing and Search & Rescue). Centralised management of data, information or knowledge in such environments would introduce a single point of failure and a potential bottleneck. Therefore, the knowledge base presented in this thesis should not rely on any central instance. Instead, our goal is to employ autonomous agents, which can form and maintain a decentralised knowledge base capable of coping with changes in the underlying network structure.

Knowledge Discovery

Typically, knowledge discovery in loosely coupled networks relies on flooding since no central instance or registry is given, as discussed in the previous paragraph. During flooding, queries are forwarded to every reachable node, which results in a high message complexity. While this process ensures that a query is resolved in the best way possible, it introduces additional stress on a loosely coupled network with potentially limited bandwidth. Thus, extensive flooding can reduce the functionality and response times of the system and, in the worst case, lead to a collapse of the network. To prevent these problems, the decentralised knowledge base proposed in this thesis has to restrict the application of flooding.

Requirements

Based on the problem statement presented and discussed in the previous sections, the following requirements are derived:

R1: Handling Dynamic Environments - The solution presented in this thesis has to be able to handle dynamic environments populated with heterogeneous participants.

R2: Efficient Management of Knowledge - The solution has to manage and store semantically annotated knowledge efficiently, decentrally, and has to provide mechanisms to access the knowledge.

R3: Handling Semantic Inconsistencies - The solution must be able to detect and prevent semantic inconsistencies in the stored knowledge.

R4: Efficient Knowledge Discovery - The solution must provide efficient mechanisms to discover knowledge in dynamic environments.

1.2 Scenarios

In the following sections, we introduce the research projects and scenarios that influenced the creation of this thesis. The presented approaches are not limited to these scenarios and can be applied in other environments. The remainder of this section is structured as follows. Section 1.2.1 introduces a service replacement scenario, which is a regular part of Service-Oriented Architectures (SOAs). Subsequently, Section 1.2.2 presents the LOEWE[5] Center *emergenCITY*, its vision, and research areas.

[5]LandesOffensive zur Entwicklung Wissenschaftlich-ökonomischer Exzellenz, Hessen

1.2.1 Service Replacement in Service-Oriented Architectures

Service-Oriented Architectures consist of loosely coupled services. To enable cooperation, services exchange messages via a network protocol. In this way, a service mesh is created, which can deliver the same functionality as large monolithic software applications. This software design methodology has many advantages in comparison to monolithic approaches. First, the application is split into several small services, which focus on single business functionalities. Thus, each service can be maintained by a small team that does not need to know the complete specification of the original software monolith. Further advantages are scalability and flexibility. If a service is overloaded, additional instances can be created instead of duplicating the complete application. However, SOAs introduce some disadvantages to the system. One of them is the additional communication between the services. For example, messages can be duplicated or lost, which has to be handled by the services. A further problem arises due to frequent changes of SOAs, which are caused by updates, context changes, or failures. The number of these adaptions increases when microservices are used, which are more fine-grained than service modules, run independently, and are managed decentrally. Hence even small changes can affect many interdependent microservices. To handle this issue, an automatic management system for changes or service replacements is needed.

In cloud computing environments, a central service registry typically solves this problem. However, this approach is not suited for dynamic domains like Fog Computing environments. In Fog Computing, computations are forwarded to the edge of the network where Fog Nodes can process the requests using their computational resources [98]. Generally speaking, Fog Nodes are ubiquitous and decentralised devices that cooperate via communication to perform tasks or to store data, information, and knowledge [132]. Therefore, a decentralised approach is required, which can provide alternative services by comparing its application programming interface (API), semantics, or behaviour.

The requirements introduced by Fog Computing influence the design of the distributed and multi-agent-based knowledge base presented in this thesis. It utilises autonomous agents, which can be deployed on the Fog Nodes. Furthermore, it relies on a semantic description of the stored knowledge to resolve queries.

1.2.2 emergenCITY

Emergency Responsive Digital Cities[6] (emergenCITY) is an interdisciplinary research centre operated by the Technical University of Darmstadt, the University of Kassel, the Philipps-University of Marburg, the Federal Office of Civil Protection

[6]Emergency Responsive Digital Cities, https://www.emergencity.de/, Accessed December 29, 2021.

and Disaster Assistance, as well as the City of Darmstadt. The research goal of emergenCITY is summarised as follows: *How can the functionality of cities with digitally networked infrastructures be ensured in extreme situations, crises, and disasters?*[6] To achieve this goal and to create a resilient as well as reliable infrastructure in the case of an emergency or a natural disaster, modern information and communication technologies are applied. Additionally, historical, legal and social aspects, as well as urban planning, are considered. In total, emergenCITY consists of four program areas. The following sections briefly introduce the areas.

City and Society (SG) - *SG* focuses on the historical, political, social, and legal aspects, which are required for the design of resilient technologies that are applied during a crisis or a natural disaster.

Information (INF) - The focus of *INF* is set on the provision of information and communication services utilising the available resources. They include the modelling of relevant data as well as data streams.

Communication (KOM) - *KOM* enables basic communication services under any circumstances. Therefore, this program area aims at designing resilient communication systems.

Cyber-Physical Systems (CPS) - *CPS* focuses on (semi-)autonomous robotic platforms to respond to emergencies and to support the recovery process. Furthermore, *CPS* aims at establishing communication backbones without heavily relying on present infrastructure. The goals include the deployment of (semi-) autonomous robotic platforms and unmanned aerial vehicles (UAVs) to support rescuers or to serve as communication relays. Last but not least, *CPS* aims at providing a decentralised world model and knowledge base.

The Self-Organising Multi-Agent Knowledge Base presented in this thesis is part of the research efforts of *CPS*. It relies on the UAVs, robotic platforms, and available infrastructure to create a distributed knowledge base and knowledge management backbone. Since *KOM* provides the underlying communication, this thesis focuses solely on the design and the creation of the knowledge base, the necessary knowledge representation, and the required protocols and mechanisms.

1.3 Solution Approach

To provide efficient management of semantically annotated knowledge as well as mechanisms to detect and prevent semantic inconsistencies, a sophisticated knowledge representation and reasoning formalism is needed. According to Krötzsch, a declarative and symbolic knowledge representation is suited for these requirements [84]. Especially the combination of Answer Set Programming (ASP) [22, 87] and the solver Clingo [50] provides unique features, which are essential for efficient knowledge management. Among other features, they support non-monotonic reasoning, the definition of defaults and choices, the division of logic programs into reusable sub-programs, and the adaptation of truth values of distinct predicates at run-time. Thus, ASP provides efficient management of knowledge.

While syntactic inconsistencies (predicates like *a* and -*a* appearing in the same knowledge base) can be easily detected, additional knowledge is needed to detect semantic inconsistencies. A way to gain the required knowledge is to incorporate a commonsense knowledge database like ConceptNet5 (CN5) [128]. It provides the necessary background knowledge to detect semantic inconsistencies. Using ASP as a representation for the knowledge extracted from CN5, we create mechanisms that prevent inconsistencies by providing consistent alternative solutions, which can be selected by the corresponding user of the knowledge base.

Multi-Agent Systems (MAS) typically consist of autonomous entities (agents), which cooperate or collaborate to solve tasks. Therefore, they are suited for dynamic environments, as presented in Section 1.2. The agents used in the solution of this thesis adhere to the MAPE-K model (see Section 2.1.1). Furthermore, the development of the agents focuses on efficient resource consumption since the management of the knowledge base should not limit the functionality of participants with limited resources like robots in Search & Rescue scenarios. An additional aspect is the decentralised knowledge base. Since no central instance is given, the agents have to maintain the structure of the knowledge base. Therefore, they act in one of two roles: *Registry Nodes*, which maintain the tree-like structure of the knowledge base. Furthermore, they manage the *Registry Leafs* that focus on the storage of knowledge.

To foster an efficient knowledge discovery in a single group of agents (*Knowledge Group*) or between several *Knowledge Groups*, this thesis introduces a solution based on semantic routing tables. They rely on taxonomies extracted from CN5 to aggregate semantically close entries, thus enabling semantic queries utilising the created aggregates. Furthermore, the tables are only adapted if entries with new semantics arrive. Therefore, limiting the communication to semantic deltas.

The solutions described in the previous paragraphs are evaluated using the scenarios introduced in Section 1.2. The first scenario encompasses the discovery of service replacements in service-oriented architectures. In this case, replacements are selected based on their semantics and their characteristics. This scenario shows the benefits of ASP as the knowledge representation formalism as well as the application of ASP and the created routing tables. The second scenario focuses on a Search & Rescue example, which demonstrates the applicability of the proposed knowledge base in highly dynamic environments.

1.4 Contributions

This thesis contributes to three research fields, which are *knowledge representation and reasoning*, *distributed knowledge management*, and *semantic-based routing*.

The first contribution is in the field of *distributed knowledge management*. Central management of knowledge has the advantage that a single component stores all relevant knowledge. Hence, queries can be resolved considering all existing relevant knowledge. However, this approach is not suited for dynamic environments as shown in Section 1.2 since it introduces a single point of failure to the system. Therefore, this thesis presents an **agent-based knowledge management** (**Knowledge Group**). Agents organise themselves in a **tree structure**. In large scale environments, **several distinct Knowledge Groups** can be applied. Therefore, this thesis presents **mechanisms to foster the cooperation of Knowledge Groups**.

The second contribution is given in the field of *knowledge representation and reasoning*, which is the **(semi-)automatic extraction of ontologies from a graph-based commonsense knowledge source**. Instead of the Web Ontology Language [3], the non-monotonic logic reasoning language **Answer Set Programming (ASP)** [57] is used since it provides sophisticated features for dynamic environments. These include, among others, dynamic adaptation of truth values, the adaptation of the resulting ontologies, and a mechanism to subdivide the ontology into reusable parts. Since commonsense knowledge inevitably contains **semantic inconsistencies**, this thesis presents a **sophisticated method to prevent these semantic inconsistencies**. Again, this method relies on ASP and, thus, can be easily integrated into the generated ontologies.

The third contribution addresses *semantic-based routing*. The **discovery of knowledge** in a loosely coupled network is a challenging task due to the frequently changing network structure and the number of participants. In such scenarios, classical IP-based routing relies on flooding, which introduces a high message load to the network. To tackle this issue, this thesis proposes a **routing mechanism based on the semantics of the stored knowledge**, which relies on **dynamically adaptable routing tables**. The proposed routing mechanism utilises **taxonomies to aggregate semantically related knowledge** to reduce the number of routing entries. Finally, the **routing entries are represented by ASP rules**, which enables the use of its reasoning capabilities to resolve queries.

Summarising, it can be said that the main contribution of this thesis is a **distributed multi-agent-based knowledge base**, which is **tailored for dynamic loosely coupled networks** and **manages semantically annotated knowledge autonomously**.

1.5 Structure of the Thesis

This thesis is composed of three parts. Part I contains the preliminaries. Chapter 2 presents the foundations of this thesis. They include, among others, an introduction to Multi-Agent Systems, Answer Set Programming (ASP), and ontology generation. Chapter 3 gives an overview of related work in several areas. They encompass the application of commonsense knowledge, the use of Answer Set Programming as knowledge representation language, the generation of ontologies, databases and knowledge management, as well as semantic routing.

Part II discusses the solutions for the problem statements given in Section 1.1. Chapter 4 introduces the first main contribution, a distributed knowledge storage and management system (*Knowledge Group*). It consists of a hierarchically organised multi-agent system, which manages semantically annotated knowledge based on ontologies. Chapter 5 presents the second main contribution of this thesis, which is the handling of symbolic commonsense knowledge. It is divided into two components. The first is the generation of ASP-based ontologies used by the *Knowledge Groups*. Furthermore, Chapter 5 introduces a framework that extracts ontologies from a graph-based knowledge source, introduces a schema for inference rules, and supports the user in the refinement of the ontologies by providing a graphical user interface. Inherently, commonsense knowledge contains semantic inconsistencies, for example, contradicting properties or ambivalences. Therefore, the second component is the handling of semantic inconsistencies. It utilises a graph-based knowledge source to generate ASP rules that enable the creation of a semantically consistent knowledge base. In large-scale environments, as presented in Section 1.2, several *Knowledge Groups* can exist in parallel. Therefore, it is expanded with mechanisms that foster collaboration and knowledge exchange between several knowledge bases. Chapter 6 elaborates an adaptive semantic routing approach tailored for dynamic environments to provide efficient routing of queries between the knowledge bases. It uses Answer Set Programming to form dynamic routing tables and utilises the ontologies respective taxonomies introduced in Chapter 5.

Part III presents the assessment. Chapter 7 shows the results of the evaluation and discusses them. Chapter 8 summarises the contents of this thesis and concludes the evaluation results. Finally, future work is discussed.

2 Foundations

In this section, the foundations of this thesis are presented. Section 2.1 provides an introduction to agents and Multi-Agent Systems. One of these systems is the ALICA (A Language for Interactive Cooperative Agents) framework introduced in Section 2.2. Section 2.3 illustrates the field of Knowledge Representation and Reasoning. In Section 2.4, the non-monotonic reasoning formalism Answer Set Programming [22] is presented that is used to represent knowledge in the proposed knowledge base. Subsequently, Section 2.5 presents the state-of-the-art ASP solver Clingo. Furthermore, Section 2.6 introduces ontologies, which are used to provide semantics to the knowledge stored in the distributed knowledge base. Finally, ConceptNet 5 is shown in Section 2.7, which offers commonsense knowledge for the generation of ontologies and taxonomies.

2.1 Multi-Agent Systems

Multi-Agent Systems (MAS) are distributed systems combining two key aspects [135]. The first aspect is the notion of agents, or especially in the scope of this work, of intelligent autonomous agents. The second aspect is their cooperation. By cooperating, agents form Multi-Agent Systems.

2.1.1 Intelligent Autonomous Agents

The term agent has been extensively discussed in the literature [44, 121, 135], but so far, there is no generally accepted definition [135]. There is only a small consent in the literature, stating that an agent has to act *autonomously* in its environment [135]. Thus, an agent can be seen as a computer program, a robot, or even a human being. Furthermore, an agent perceives its environment via sensors and can interact with it using actuators. According to Russel and Norvig [121], the environment can be classified by four dimensions. The first dimension is accessibility. An accessible environment provides complete and accurate data, which cannot be guaranteed for inaccessible ones. Furthermore, an environment can be deterministic or non-deterministic. While actions have a guaranteed effect in a deterministic environment, the results of an action in a non-deterministic environment are uncertain. Additionally, static and dynamic environments have to be distinguished. In a static environment, the actions of an agent do not change it. In a dynamic environment, the actions of an agent can cause changes that the agent cannot control. Finally, the environment can be discrete or continuous. Thus, the most complex environment is inaccessible, non-deterministic, dynamic, and continuous [121].

To cope with such complex environments, agents have to act intelligently. Since the definition of intelligence is a research topic on its own, research in the field of intelligent agents focuses on the properties and capabilities of agents [135]. According to [136], three essential capabilities are required for an intelligent autonomous agent. The first capability is *reactivity*. It means that an intelligent autonomous agent must react to changes in its environment in an acceptable time frame and according to its design goal. For example, an autonomous car has to react to an appearing obstacle in a very short time to prevent a collision. Such reflex-like behaviour alone cannot be considered intelligent but is an important capability of an autonomous agent. The second capability is *proactivity*. Proactive or goal-oriented agents have to be able to take the initiative and act according to their goals. For example, an autonomous car could plan a collision-free path to its goal while dynamically reacting to obstacles during the execution. The third capability is essential for agents, which are part of a MAS. Intelligent autonomous agents have to have some sort of *social ability*. This means that agents have to interact with each other to achieve their goals. A car, which had an accident on a highway, could proactively inform other vehicles. These cars could react to the information and plan new paths to their destination. Further capabilities are discussed in [44] and summarised in Table 2.1.

Capability	**Alias**	**Meaning**
reactive	sensing and acting	agents respond timely to environmental changes
autonomous		agents have control over own actions
goal-oriented	proactive purposeful	agents plan their actions to achieve their design goal
temporally continuous		agents are continuously running processes
communicative	socially able	agents communicate and share their information with other agents
learning	adaptive	agents change their behaviour according to their experience
mobile		agents are able to relocate themself
flexible		agents do not rely on scripted actions
character		agents have an emotional state and some kind of personality

Table 2.1: Agent Capabilities [44]

To enable the presented capabilities, several agent architectures have been proposed. One of these architectures is the MAPE-K cycle [80]. While MAPE is short for Monitor, Analyse, Plan, and Execute, K represents the central Knowledge component that can be accessed by the other four components. Figure 2.1 shows a schematic representation of the MAPE-K cycle.

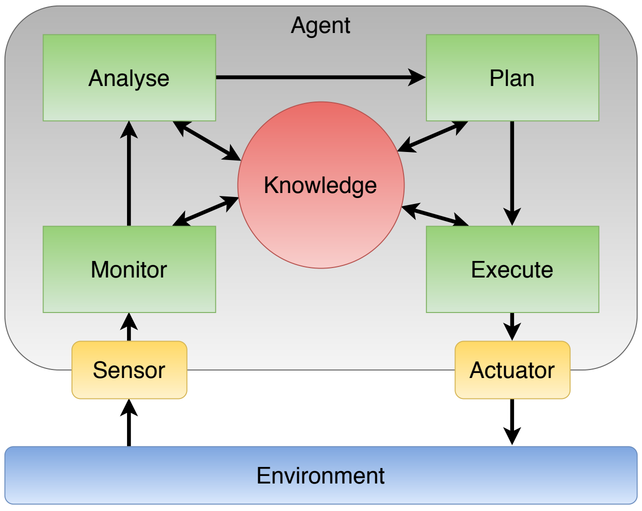


Figure 2.1: MAPE-K Agent Architecture [80]

The monitoring component of the agent receives raw data via sensors from its environment. The corresponding component then analyses this data. The analysed information is used to create a plan, which is executed by the actuators of the agent. All components are further connected via a knowledge component, which can store data, information, and knowledge. The knowledge component can contain background knowledge about the environment and can be used by any component of the cycle. Furthermore, this enables shortcuts between the components. In an autonomous car, sensors could be cameras or laser scanners, which are used to detect obstacles. The monitoring component retrieves the raw data, preprocesses it, and forwards it to the analyse component. This component combines the preprocessed information into knowledge, which could be the shape, size, and distance of an obstacle. This knowledge is then passed to the planning component to avoid it. Based on the background knowledge stored in the knowledge component and the knowledge about the obstacle, the planning component calculates a path to avoid the obstacle. Finally, the execute component conducts the calculated obstacle avoidance via its actuators.

A second architecture is the Belief-Desire-Intention (BDI) architecture [113]. The goal of this architecture is to model different mental attitudes of an agent. Figure 2.2 depicts the BDI schema.

In comparison to the MAPE-K architecture, sensors are again used to get information about the environment. Since the agent cannot perceive the complete state of the environment, the information is updated after every sensing action. The component that manages this information is called the Beliefs. Besides this informative

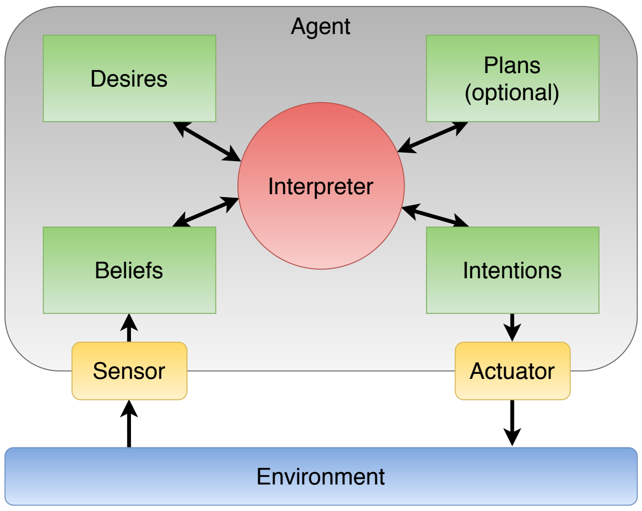


Figure 2.2: BDI Agent Architecture [113]

component, the goals of the agent, their priorities, and their payoffs have to be represented. This is handled by the Desires of the agent, which capture the motivational state of the agent. The Plans component is optional and can store already executed plans if the agent can reuse them. Furthermore, the currently executed plan is represented by the Intentions of the agent. Finally, all components are connected via the Interpreter, which executes the main loop of the agent. In the first step, the agent generates options according to new Beliefs. After selecting one of these options, the Intentions are updated and atomic actions are executed. Finally, the agent drops successful and impossible Intentions and Desires. Coming back to the example of an autonomous car, Beliefs could be the data captured by its sensors, a Desire could be to reach a specific goal, and the Intention is the current execution of the path to the destination. In this case, the autonomous car senses an obstacle and updates its Belief. Since one Desire is to prevent a crash, the action of evading the obstacle is executed. Finally, the Intention is updated.

As presented in this section, agents can cope with dynamic environments and a wide range of problems. Sometimes, a single agent is not sufficient to solve a problem, and agents need to cooperate, coordinate, and communicate to form a Multi-Agent System (MAS). An introduction to these essential aspects of a MAS is given in the next section.

2.1.2 Agent Cooperation

A central aspect of agent cooperation is the communication between them. To ensure that the agents are able to understand each other, a common vocabulary is needed. An ontology can provide such a kind of common vocabulary (see Section 2.6). In summary, an ontology provides a taxonomy of classes or concepts, their relationships, and their properties [135]. One way to communicate is the exchange of messages, for example, by directly passing messages between agents or in a publish-subscribe like fashion. One of the earliest standards is the agent communication language (ACL) provided by the Foundation for Intelligent Physical Agents (FIPA) [101]. Among other message fields, the ACL message supports fields for sender and receiver, a language specification, diverse content, and the intention of the message. It includes the passing and requesting of a message, negotiation, action performing, and error handling. Based on the fields of the message and its intent, agents are able to cooperate.

The cooperation of agents in a MAS has to address two significant challenges [135]. The first challenge is the use of heterogeneous agents, which have varying capabilities, languages, and goals. The second challenge is the autonomy of agents. Each agent decides its actions autonomously at run time. Hence, the agents need to be able to dynamically adapt their actions and activities to other agents in the system. Thus, during recent years many agent coordination frameworks have been proposed [112]. The next section presents a concrete example (ALICA - A Language for Interactive Cooperative Agents), originally published in [127]. Additionally, we have expanded ALICA in [104] to increase its applicability.

2.2 ALICA - A Language for Interactive Cooperative Agents

This section provides a brief introduction to ALICA (A Language for Interactive Cooperative Agents)[7]. The original version of ALICA was developed for robotic soccer in the Middle Size League of the RoboCup[8] and actively applied by the Carpe Noctem Cassel team[9]. In this league, teams consisting of five autonomous robots play soccer against each other using a regular-sized FIFA ball. Furthermore, no central coordination instance is given and, thus, the agents have to coordinate themselves. Additional requirements arise from the soccer domain itself. Soccer is a highly dynamic game and requires agents that can act accordingly. Due to a high amount of transmitted messages, they can be lost or duplicated. Hence, the framework has to cope with unreliable communication. Finally, robots can be damaged and, subsequently, have to be repaired, which requires a dynamic team constellation and task allocation.

Besides robotic soccer, ALICA has already been used in several domains like space exploration and autonomous driving [108]. The ALICA framework consists of three

[7]ALICA, `https://www.uni-kassel.de/eecs/vs/research/alica`, Accessed December 29, 2021.

[8]RoboCupSoccer - Middle Size League, `https://www.robocup.org/leagues/6`, Accessed December 29, 2021.

[9]Carpe Noctem Cassel, `https://www.uni-kassel.de/eecs/vs/research/carpe-noctem-cassel`,Accessed August 13, 2021.

major components [108]. The first one is the ALICA language, which is presented in the following section. It describes the behaviour of the agents by hierarchical plans. The second component is the execution engine presented in Section 2.2.2. It implements the operational semantics of ALICA and executes the plans. The third component is the ALICA Plan Designer (see Section 2.2.3), which supports developers to model plans. For a complete specification, the interested reader is advised to [104, 127].

2.2.1 Language

The central parts of the ALICA language are hierarchical multi-agent Plans, as shown in Figure 2.3. A Plan implicitly models a goal and consists of at least one finite state machine (FSM) [19]. Figure 2.3 shows two Plans *AutonomousDriving* and *Drive*. Each FSM is accessible by an agent via an Initial State. Focussing on the *AutonomousDriving* Plan, S_0 is the Initial State and is annotated by a Task (*Drive*), which denotes the purpose of the FSM. Furthermore, a minimum and maximum cardinality are given, stating that at least one agent is required to execute the FSM and that there is no restriction to the maximum number of agents. Additionally, the Task is denoted as *Success Required*, which states that at least one Success State (S_4) has to be reached to finish the Plan successfully. Normal States, like S_1, S_2, and S_3, can contain an arbitrary number of Behaviours (orange), other Plans (light blue), and Plan Types (dark blue), which provide a set of alternative Plans. Behaviours encapsulate domain-specific code and describe the actions of an agent. They can be started and stopped, and their execution can be either successful or unsuccessful (failed). For example, the Behaviour *Stop* will stop the autonomous car, or the Behaviour *Park* will park the car. Plan Types provide a set of alternative Plans. For example, the *Junction* Plan Type in S_7 of the *Drive* Plan provides Plans for different junction types.

All elements inside a State are executed concurrently and with a fixed frequency, e. g., 30 Hz. This means that when an agent enters State S_1, it executes the *Drive* Plan while using the *AvoidObstacles* Behaviour to prevent a crash. Additionally, ALICA defines three condition types, which are Pre-, Runtime, and Postconditions. Preconditions have to hold before and during the execution of an element of the ALICA language. Runtimeconditions have to hold while the element is executed, and Postconditions are supposed to hold after the execution. For example, after the successful execution of the *AutonomousDriving* Plan, it is expected that the car has reached its goal. To change States, agents have to use Transitions (arrows between states). These Transitions are annotated with a Condition. For example, to move from State S_0 to S_1, the motor of the car has to be started. Like Behaviours, these Conditions allow the use of domain-specific code. Furthermore, they should only use idempotent functions since they are checked periodically by the ALICA Engine.

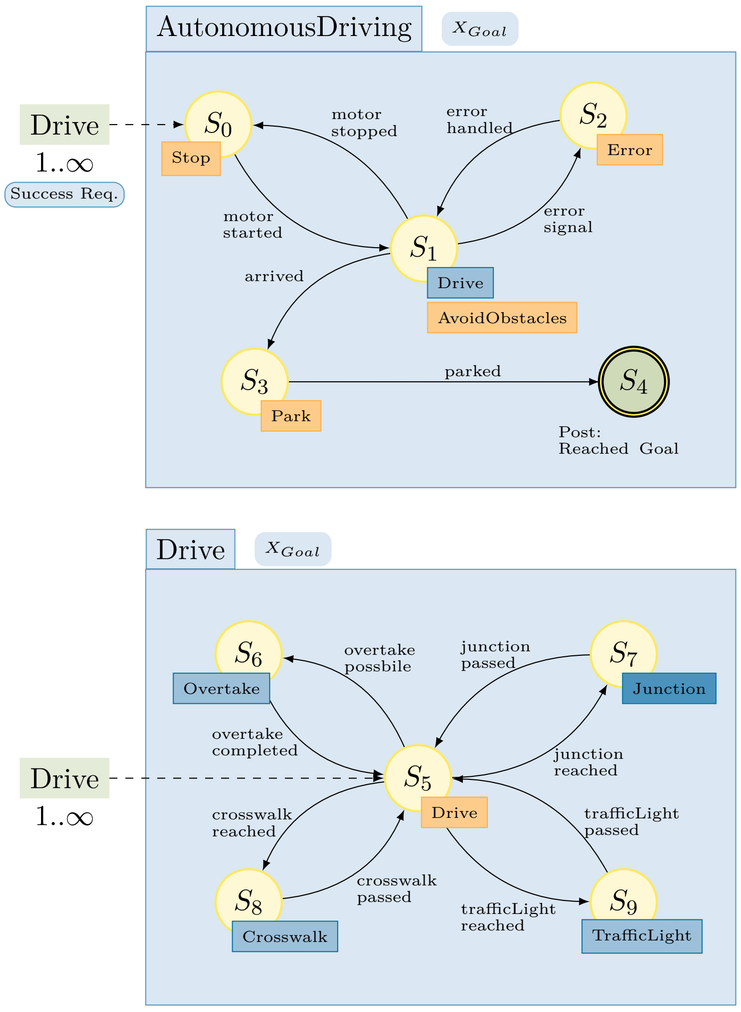


Figure 2.3: Hierarchy of Plans for Autonomous Driving

ALICA supports the use of different solvers and thus enables agents to query them by providing Variables and Constraints. Up to now, three different solvers can be used: a continuous non-linear constraint satisfaction and optimization problem solver, an Answer Set Programming solver (see Section 2.4), and a simple solver, which supports simply setting Variables. In the example in Figure 2.3, both Plans are annotated with the Variable X_{Goal}. While these Variables share their name, they can have different values. In order to pass Variable values between Plans, ALICA supports binding Variables. For example, both instances of the Variable X_{Goal} can be bound to contain the same goal location.

Since ALICA Plans can contain Plans inside their States, endless recursions can occur. To prevent this, the dependencies of Plans need to form a directed acyclic graph (DAG). During the runtime, the DAG is interpreted as a tree. The Plans are the nodes of the tree, and the Behaviours form the leaves. Thus, each time a Behaviour is used, a new instance is created. This prevents agents from influencing the execution of a Behaviour that other agents use in different Plans.

2.2.2 Engine

The second component of the ALICA framework is its Execution Engine. Each ALICA agent uses its own instance of the ALICA Engine, and hence, no central coordinator or registry is needed. Since there is no central component, the ALICA framework is able to cope with package loss, network delay, and failures of agents. Figure 2.4 shows the architecture of the ALICA Engine that consists of three layers.

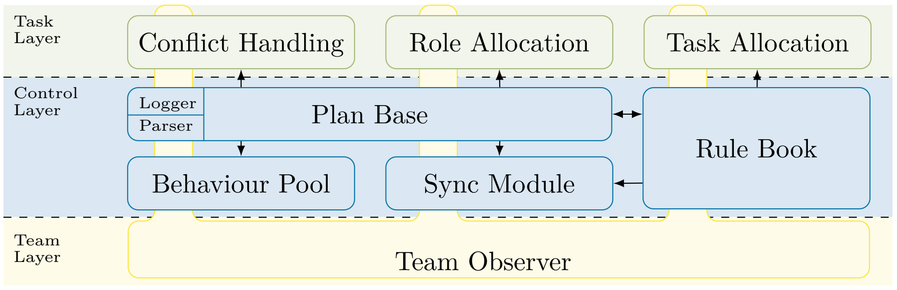
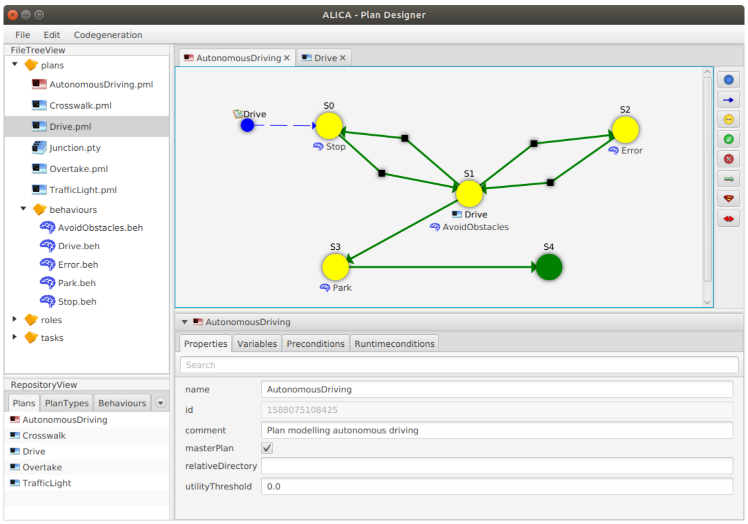


Figure 2.4: ALICA Execution Engine Architecture [104, 108]

The first layer is the Task Layer. It is used to assign Tasks and Roles to agents. Furthermore, it provides a conflict-handling module in case assignments cause a conflict involving multiple agents. For example, in its current implementation, Roles can be configured via a settings file and Tasks are allocated using Utility Functions of the corresponding Plans. Conflicts can occur due to uncertain sensor values or information, which will cause inconsistent assignments by different agents. The second layer is the Control Layer. It parses the JSON files generated by the Plan Designer, which represent the Plans and executes them. Furthermore, this layer provides interfaces for communication, the clock, and a general solver interface. One essential component of this layer is the Rule Book. It implements the operational semantics of ALICA. They define when an agent has to follow a Transition, when a Plan needs to be restarted in case of a failure, when to trigger the task allocation, and when Conditions should be evaluated. A full specification of the operational semantics is given in [127]. The last layer is the Team Layer, which is accessible by the other layers. The central component of this layer is the Team Observer, which is responsible for managing information about other agents. In its original version, ALICA used a fixed team, which is not suitable for every domain. Hence, in the second version of ALICA [108], the Team Observer has been expanded, and a discovery module has been added. Both modules enable ALICA to incorporate unknown ALICA agents dynamically.

2.2.3 Plan Designer

The Plan Designer is a graphical tool, which supports developers creating ALICA Plans. Figure 2.5 depicts the main window of the Plan Designer. It is divided into four major parts. The file tree view provides an overview of the files and the folder structure. The repository view provides fast access to all already created elements of the ALICA language. The main area in the middle of the figure provides tools to model ALICA Plans and the properties view supports different settings. For example, the *AutonomousDriving* Plan is marked as a Master Plan, which indicates that this Plan is intended to be used as the root of a Plan tree.



name	AutonomousDriving
id	1588075108425
comment	Plan modelling autonomous driving
masterPlan	✓
relativeDirectory	
utilityThreshold	0.0

Figure 2.5: Plan Designer [104, 108]

Besides the support during the creation of ALICA Plans, the Plan Designer has two outputs. The first output is a serialisation of the ALICA Plans to a machine-readable format (JSON) enabling the ALICA Engine to read the Plans. The second output consists of source code stubs for the domain-specific parts of the ALICA language: Behaviours, Conditions, and Constraints. Developers can then use these stubs to program the overall behaviour of the ALICA agent. Currently, these stubs are available in C++ and will be provided for Java and Python in [71].

2.3 Knowledge Representation and Reasoning

The research field of knowledge representation and reasoning (KRR) is a central aspect of this thesis. According to [21], KRR can be summarised as follows: *Knowledge representation and reasoning in the area of Artificial Intelligence (AI) is concerned with how knowledge can be represented symbolically and manipulated in an automated way by reasoning programs.* Thus, two main research questions are focussed: how to define and represent knowledge as well as how to draw conclusions from knowledge.

To represent knowledge and to reason about it, it is necessary to define what knowledge actually is. Therefore, Ackoff presents in [1] the data-information-knowledge-wisdom (DIKW) hierarchy. The lowest level of DIKW is data. Data is generated by observing the environment and represents an object and its properties [120]. To generate information, the next level of the hierarchy, the raw data is processed. The system can then use the resulting information. The third level of the hierarchy is knowledge, which is the application of data and information to form instructions. Furthermore, Brachman and Levesque formulate knowledge as a relation between a knower and a proposition [21]. Thus, it can be seen as information interconnected via relations. The fourth level of DIKW is intelligence which is the capability to use knowledge efficiently. The last level of this hierarchy is wisdom that enables improving the effectiveness of a system based on knowledge. While the previous two levels of this hierarchy are abilities, knowledge is the highest level of abstraction of data. To efficiently work with knowledge, a suitable representation is needed. In general, a representation can be seen as a relation between two domains [21], where the first domain is the representation, e. g., a symbol, and the second an object in the domain. To sum up, the field knowledge representation focuses on symbols defining or representing propositions that are known or believed by agents [21].

The second field of KRR is reasoning. To define the process of reasoning, the notion of a knowledge base (KB) is needed. A KB is an accumulation or store of abstract symbols, which are structural representations of knowledge. Furthermore, the KB can dynamically grow during the runtime of the system by adding new knowledge, communication, and logical entailment. Thus, logical reasoning is the ability to draw conclusions from an existing KB by the application of logical entailment, which generates novel knowledge and expands the KB. To build a KB, a formal declarative language that can formulate and model knowledge is needed, for example, Answer Set Programming (see Section 2.4). A language like this, in general, is a set of words or symbols. Additionally, syntax, semantics, and pragmatics are needed, which are essential for declarative languages that are used to build a KB [21]. The syntax defines how the symbols are aligned and what functional characters can be used in a sentence. The semantics expresses what syntactically correct sentences mean. Finally, pragmatics describe how syntactically and semantically correct sentences should be used.

In order to build a KB, a language fitting to the application domain has to be selected. Therefore, several assumptions have to be considered. The first is the Domain Closure Assumption [117], which states that there are no other individuals besides the ones given in a KB or a database. For example, considering the KB, which contains the three sentences:

“Bob is a parent of John;”
“Alice is a parent of John;”
“John is the child of Alice.”

In this case, only three individuals (Alice, Bob, and John) exist in the KB. Under the Domain Closure Assumption, no other individual can and will exist.

The following assumptions, which are the Closed- and Open-World Assumptions [116], contradict each other. The Closed-World Assumption states that a sentence is true if it is supported by the KB. Furthermore, every sentence which is not supported by the KB is considered to be false. Thus, this assumption is suited for domains in which complete or nearly complete knowledge is given. Considering the example KB presented above, the sentence “John is the child of Alice” is true since it is part of the KB. The sentence “John is the child of Bob” would be false since it is not explicitly stated that *child* is an inverse relation to *parent*. To enable this conclusion, a sentence like “If X is a parent of Y, Y is a child of X” would have to be added to the KB. In contrast to the Closed-World Assumption, missing knowledge in the Open-World Assumption does not lead to the conclusion that a sentence is false. Instead, it is assumed that it is unknown. Hence, this assumption is suited for domains in which complete knowledge is not possible or for domains in which, for example, an agent can observe additional data and create new knowledge.

A further assumption is the Unique Name Assumption [117]. It states that individuals with the same name or identifier in a KB are the same individual. Considering the example above, the individual John that appears in all sentences is the identical individual if the Unique Name Assumption holds. On the other hand, if the Unique Name Assumption does not hold, it has to be explicitly stated that all occurrences of John refer to the same individual. Furthermore, several names can be assigned to John.

Besides the assumptions which are supposed to hold, several logical problems have to be considered during the creation of a KB. One example is the Frame Problem [91]. In general, this problem takes into account how actions and their results can be described without explicitly stating which parts of a KB are (obviously) not affected. For example, let us consider a KB used by a service robot. The robot R is able to perform the actions *PickUp(X,Y)* and *Move(X,Y)*. Besides the robot, there is a cup of coffee C. Furthermore, two sentences or rules describe the results of the actions. These are: "*Carries(X,Y)* holds after *PickUp(X,Y)* has been performed" and "*Position(X,Y)* holds after *Move(X,Y)* has been performed". By applying the action *PickUp(R,C)*, the robot lifts the coffee cup and holds it. Thus, *Carries(R,C)* holds. In a subsequent action, R changes its position to the kitchen by applying the *Move(R,kitchen)* action. An important question after the execution of both actions is: Which results hold? Relying on human common sense, it can be assumed that *Carries(R,C)* and *Position(R,kitchen)* hold. Since this is not stated explicitly, only *Position(R,kitchen)* can be assumed without a doubt. Therefore, Frame Axioms are needed. For example, "*Carries(X,Y)* holds after *Move(X,Z)* if it was true before". Solutions like this work for a small KB since it has to explicitly state what an action changes and what is not influenced. A way to tackle this problem is the Commonsense Law of Inertia [43]. This law states that a situation or result of an action only changes if it is explicitly stated.

In general, logic-based languages address these problems and provide formalisms to model the described assumptions. One of these languages is presented in the next section.

2.4 Answer Set Programming

Answer Set Programming (ASP) [57] is a declarative and non-monotonic logic programming formalism and tailored for NP-hard search problems. Furthermore, ASP combines the results of research in the fields of knowledge representation, logic programming, and constraint satisfaction [22]. This section introduces selected parts of the ASP language. The complete ASP-Core-2 Input Language Format is presented in detail in [24]. Furthermore, this section provides an insight into the solving process to highlight the methods used to generate solutions for an ASP program.

2.4.1 Language

The basic components of the ASP language are *atoms*, which are statements that can be true or false. Furthermore, each atom has the following structure $p(v_0, ..., v_{n-1})$, where p is a predicate denoting the meaning of the atom and $v_0, ...v_{n-1}$ are either constants or variables. Additionally, n marks the arity of the atom. In the case that

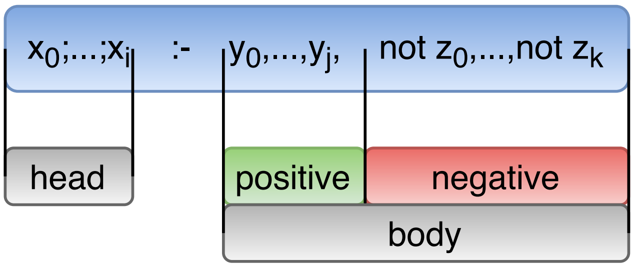


Figure 2.6: Structure of an ASP Rule

the arity of an atom is 0, it is denoted as a constant. By combining atoms, *rules* are formed. The structure of an ASP rule is shown in Figure 2.6.

A rule is divided into two parts, the *head* and the *body*. In general, the head of an ASP rule is derived if all atoms in the positive body are true and no atom in the negative body holds. A detailed description of how an ASP program is solved is presented in Section 2.4.2. The head of the rule shown in Figure 2.6 consists of the $i + 1$ atoms $x_0; ...; x_i$. The body is divided into two different parts, the *positive* $(y_0, ..., y_j)$ and the *negative* body $(not\ z_0, ..., not\ z_k)$. Finally, a rule is terminated by a dot. In contrast to the positive atoms, for example, $y_0, ..., y_j$, negative atoms are annotated with the keyword *not*, which is used to express default negation. In ASP, the default negation is defined as negation-as-failure. This means that a negative atom only holds if its positive version cannot be proven to hold. For example, the atom *not z* only holds if atom *z* cannot be derived. Atoms and their negative version (not) are referred to as literals. Besides default negation, ASP provides classical negation expressed by atoms of the form -*z*. A classically negated atom is complementary to its positive version *z* and, thus, both atoms cannot hold at the same time without rendering the ASP program inconsistent.

ASP provides two special rule types. The first type is a *fact*, which is a rule without a body. Hence, the head of the rule is unconditionally true. On the other hand, rules without a head are denoted as *constraints*. If the literals of a constraint hold, the constant false is derived and, thus, the possible solution will be removed. To provide an overview of the presented features of ASP programs, Listing 2.1 shows a classic example.

This example contains literals with four predicate names or predicates, respectively. The first predicate is *bird* with an arity of 1, for short, *bird/1*. This predicate denotes that the constants *tweety* and *tux* are birds. Besides *bird/1*, the ASP program contains the literals *penguin/1*, *flies/1*, and *-flies/1*. While Lines 1 to 3 are unconditionally true facts, Line 4 and 5 are rules. For example, Line 4 expresses the ability to fly, meaning a bird is able to fly if there is no proof that it is not able to fly (*-flies*). Line 5 is used to model the contrary. A bird is not able to fly if it is a penguin. Finally, Line 6 prohibits that a bird is able to fly and is not able to fly at the same time. Since ASP is a declarative language, a solver (see Section 2.5) is needed to calculate the solution (Answer Set) of the example program. An Answer Set is a minimal set of atoms justified by at least one rule body. The complete process of solving an ASP program is shown in Section 2.4.2. In general, all facts are part of an Answer Set. Based on the facts and derived rules heads, the bodies of the rules and constraints are evaluated. Considering Line 5, *tux* is not able to fly since it is a penguin. Hence, the head of Line 4 is not derived for *tux*. In contrast to this, *-flies* cannot be derived for *tweety* and, therefore, the head of Line 4 is derived. The constraint in Line 6 does not hold for *tweety* and *tux*. Thus, the Answer Set of the example program contains the literals: {*bird(tweety)*, *bird(tux)*, *penguin(tux)*, *flies(tweety)*, *-flies(tux)*}.

```
1 bird(tweety).
2 bird(tux).
3 penguin(tux).
4 flies(X) :- bird(X), not -flies(X).
5 -flies(X) :- bird(X), penguin(X).
6 :- flies(X), -flies(X).
```

Listing 2.1: Example ASP Program

Additionally, ASP supports formulating disjunctions in the rule heads. An example of a disjunctive rule is given in Line 3 in Listing 2.2. This rule states that a bird can either fly or not fly.

```
1 bird(tweety).
2 bird(tux).
3 -flies(X), flies(X) :- bird(X).
```

Listing 2.2: Disjunctive ASP Program

The usage of disjunctions creates multiple Answer Sets, which contain one form of each literal given in the disjunction. Thus, the union of all Answer Sets encompasses all combinations of the literals $-flies(X)$ and $flies(X)$. In this case, four Answer Sets are created. All include the facts $bird(tweety)$ and $bird(tux)$ and, respectively, $-flies(tweety)$ and $flies(tux)$, $-flies(tweety)$ and $-flies(tux)$, $flies(tweety)$ and $flies(tux)$ or $flies(tweety)$ and $-flies(tux)$.

Choice rules are a generalisation of disjunctive rules. They enable the selection of literals in a disjunction by providing a lower and an upper bound. Line 1 of Listing 2.3 is an example of a choice rule with a lower bound of 1 and an upper bound of 2. Hence, at least one bird and at max two birds are selected. Again, each possible choice generates an Answer Set resulting in six solutions for this program. Setting both bounds to 1 will result in the behaviour of a disjunctive rule.

```
1{bird(tweety); bird(tux); bird(eddy)}2.
flies(X) :- bird(X).
```

Listing 2.3: ASP Program Containing a Choice Rules

An essential aspect of the ASP language is its non-monotonic reasoning capabilities. In contrast to monotonic reasoning formalisms, non-monotonic reasoning formalisms can retract knowledge already derived if additional rules are given. This, for example, enables the use of defaults. Providing a way to model defaults is a major advantage of non-monotonic reasoning since defaults provide a way to draw conclusions even if there is no complete information. These conclusions are preliminary and have to be retracted in case contrary information is available. A way to model a default in ASP is given in Listing 2.4.

```
1 bird(tweety).
2 bird(tux).
3 flies(X) :- bird(X), not deviation(defaultFlies(X)).
4 deviation(defaultFlies(X)) :- penguin(X).
```

Listing 2.4: ASP Program With a Default

Line 1 and 2 of this example are again facts stating that *tweety* and *tux* are birds. Line 3 and 4 introduce a default for the literal *flies* called *defaultFlies*. If there is no deviation of the default, it can be assumed that birds are able to fly. A deviation of the default is given, for example, if new information is available stating that tux is a penguin (*penguin(tux)*). By adding this fact, the head of the rule in Line 4 is derived, and the body of the rule in Line 3 no longer holds. Thus, *tux* is no longer able to fly.

2.4.2 Solving an ASP Program

ASP solvers like Clingo [48] typically divide the solving process into two distinct steps. The first step is the *grounding*, which replaces variables with the corresponding part of the Herbrand Universe [37] of the ASP program. This results in a variable free ASP program, which is then given to the second step, the *solving* of the ASP program. It determines the solution of the ASP program denoted as an Answer Set or Stable Model. Generally speaking, the Answer Set consists of all facts as well as all derived rule heads. Both, grounding and solving, will be explained in detail in the following sections. Furthermore, the applied grounding and solving algorithms have a major influence on the design of an ASP program and thus on the modelling of ASP rules used in the main contributions of this thesis. Hence, the following sections highlight these influences.

Grounding

To apply the grounding step, a *safe* ASP program is needed. In a safe ASP program, each variable in each rule head is part of at least one positive atom in the corresponding rule body. If a variable would not be safe, it could be grounded with every part of the Herbrand Universe, which could lead to a potentially infinite number of grounding steps. A simple approach would be to replace all variables with all constants present in the ASP program. Grounding an ASP program with variables will increase the size of the program. In the worst case, it can grow exponentially in size; thus, grounding can be considered as an EXPTIME-hard problem [79]. Furthermore, the simple grounding will generate rules, which cannot be satisfied. For example, it could generate rules that depend positively on atoms not supported by the ASP program. Hence, simple grounding is not applied by state-of-the-art grounders like Gringo [52]. In contrast, grounding algorithms, as presented in [49], typically focus on positive rule bodies and try to find a smaller instantiation preserving the semantic of the ASP program. Algorithm 2.1 shows such an instantiation approach.

Algorithm 2.1: Naive Instantiation Algorithm [49]

```
   Input  : A safe logic program SP
   Output: A ground logic program GP
1  D := $\emptyset$
2  GP := $\emptyset$
3  repeat
4      D' := D
5      foreach $rule \in SP$ do
6          B := positiveBody(rule)
7          foreach $\theta \in \Theta(B,D)$ do
8              D := D $\cup$ {head(rule$\theta$)}
9              GP := GP $\cup$ {rule$\theta$}
10 until $D' == D$
```

Algorithm 2.1 receives a safe ASP program `SP` as input and returns a grounded ASP program `GP`. The algorithm starts by initialising the set `D` that contains the domain predicates, which are variable free literals and constants. Furthermore, a set for ground rules `GP` is created, which will contain the resulting ground ASP program. The operator Θ(B,D) is defined as the minimal match of variables in the positive body `B` of a rule with the domain predicates. For example, a match θ of Line 4 in Listing 2.1 would be *tweety* for the variable X of the literal *bird(X)*. In the case of a fact, this operator will return an empty set. Additionally, the matching operator can detect partial matches. This means, if not all literals in the body can be grounded, a partial grounding is conducted, and the rule has to be revisited. In order to demonstrate the functionality of this algorithm, it is applied to the example ASP program shown in Listing 2.1. Since Lines 1 to 3 are facts, the matching returns an empty set and each fact is added to `D` in a separate iteration. In a subsequent iteration the rule *flies(X) :- bird(X), not -flies(X)* is selected. Since the Naive Grounding only considers the positive body, two matches are found for the variable X, $X := tweety$ and $X := tux$, resulting in two ground instances of this rule. This procedure is applied until every variable has been replaced by constants. The resulting ground ASP program is shown in Listing 2.5 and does not contain rules, which cannot hold.

```
bird(tweety).
bird(tux).
penguin(tux).
flies(tweety) :- bird(tweety), not -flies(tweety).
flies(tux) :- bird(tux), not -flies(tux).
-flies(tux) :- bird(tux), penguin(tux).
:- flies(tweety), -flies(tweety).
:- flies(tux), -flies(tux).
```

Listing 2.5: Grounded ASP Program

To improve the efficiency of grounding, an evaluation order extracted from a dependency graph is proposed by [79]. This dependency graph contains the atoms of the ASP program. In the first step, the dependency graph is divided into strongly connected components [122], which are subgraphs in which every node is reachable by every other node of the subgraph. The resulting strongly connected components represent subprograms of the considered ASP program. Figure 2.7 shows the dependency graph of the ASP program in Listing 2.1. The graph contains three different node types: facts (green), predicate symbols (blue), and rules (grey). Arrows indicate the dependencies and dashed boxes mark strongly connected components. Since rules, including facts and constraints, form the ASP program, each rule is part of the dependency graph. Additionally, predicate symbols are added to model the interdependencies of the rules.

In this example, each node in the dependency graph forms a strongly connected component since there is no subgraph in which every node is reachable from every other node. Following the flow of the dependency graph, the order in which the rules should be grounded is clearly evident. At first, the facts are grounded because they have no dependencies. Subsequently, the only rule of which all dependencies are grounded is `-flies(X) :-bird(X), penguin(X)`. After the grounding of this rule, all dependencies of rule `flies(X) :-bird(X), not -flies(X)` are met, and thus, the last rule can be grounded. All necessary ground literals for each rule are given by applying this order, and no rule needs to be revisited. Thus, the overall complexity of the grounding process is reduced.

Recursive rules are a special case, as shown in Figure 2.8. These rules form a strongly connected component with their recursive predicate symbol (red dashed

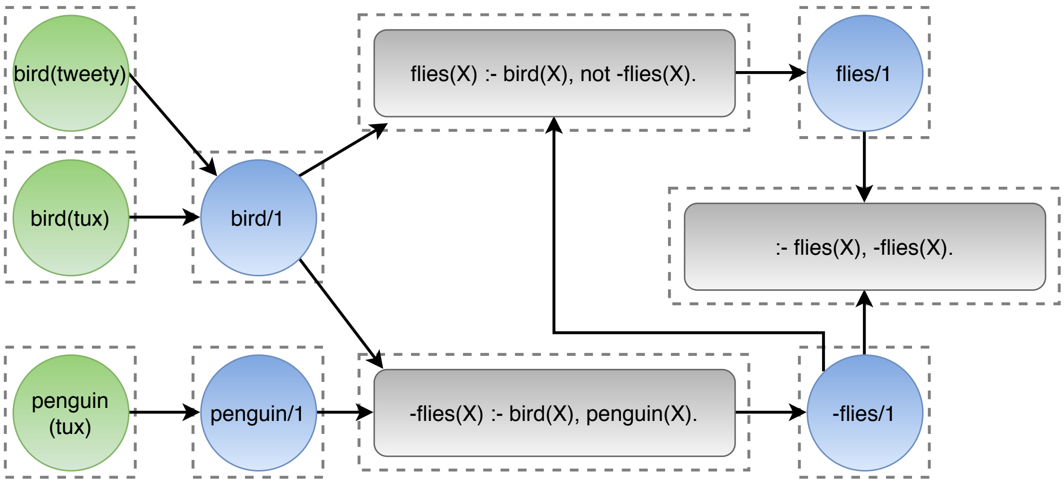


Figure 2.7: Dependency Graph of the ASP Program Shown in Listing 2.1

box). By grounding this type of rule, additional ground literals are added to the domain set, resulting in an infinitely growing domain. Hence, the recursion has to be limited by facts, by the number of iterations, or by a stopping criterion given by the creator of the ASP program.

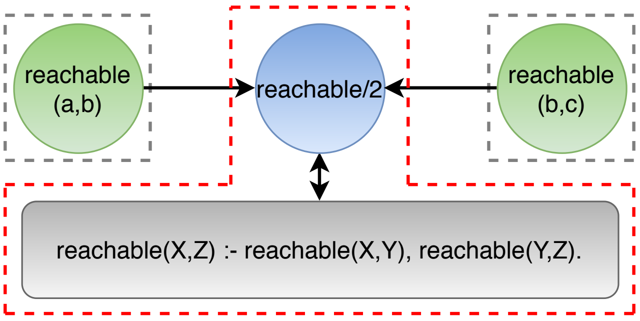


Figure 2.8: Dependency Graph of a Recursive Rule

In this example, both facts are grounded first. Consequently, the rule (grey box) is grounded. Since this grounding step changes the domain, the rule is grounded again. Subsequently, the domain does not change and the grounding is stopped.

To reduce the grounding runtime, state-of-the-art grounders apply additional simplification steps as presented in [49]. A first simplification step would be to initialise the grounding with all facts because they are already grounded, reducing the number of iterations of Algorithm 2.1. A further simplification is to remove rules during grounding if a fact appears in default negation in its body. Since facts are always derivable, the truth value of the default negation of a facts is always false. Hence, the heads or these rules cannot be derived and will not influence the solution of the ASP program. Besides removing complete rules, parts of the rule body can be removed to simplify the grounding process, and for example, default negated atoms in a rule body that do not appear in any rule head. These atoms will never appear in the solution. Therefore, their default negated version is always true and can be removed without altering the semantic of the rule. While this approach is feasible in single-shot solving (no subsequent solving steps after the first one), it limits the multi-shot capabilities (see Section 2.5.2) of ASP since the missing atoms could be added in later iterations. Therefore, Clingo introduces addition predicates (External Statements), which prevent the deletion and enable the incorporation of knowledge that is not initially available. An introduction to External Statements is given in Section 2.5.1.

Solving

After the grounding is finished, solving is applied to determine the solution of the ground ASP program. Since the ground ASP program is variable free, it can be considered as a propositional satisfiability (SAT) problem [27]. A common approach to solve SAT problems is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [31, 32]. Generally speaking, it is a backtracking algorithm, which assigns truth values to a set of literals. Therefore, it receives a set of clauses in Conjunctive Normal Form (CNF) [23] as input and returns a Boolean value stating if the set of clauses is satisfiable. Depending on the structure of the considered clause set, this can already be achieved by the unit propagation step, which sets the truth value of one element clauses depending on the sign of its predicate. This is the case for simple ASP programs since each literal can be considered as a unit clause. An ASP program can have multiple solutions. Thus, ASP solvers should be able to find all possible solutions and have to prove unsatisfiability after the last model has been found. Since this requires an exhaustive search of all literal combinations in the DPLL algorithm and there is no support for loops, modern ASP solvers apply different solving algorithms.

The ASP solver Clingo, for example, applies Conflict-Driven Nogood Learning [54], which is explained in the following paragraphs. A nogood [14] is a set of literals, which are unintended in an assignment. Thus, a nogood is violated if its literals are part of an assignment. In this case, they cannot be part of a solution for an ASP program. To determine all nogoods of an ASP program, the Clark Completion [26] is applied to find local inferences. The Clark Completion consists of the union of two equivalences: One representing the bodies and one representing the atoms of a program. According to [54], the equivalence for all rule bodies $body(\Pi)$ of an ASP program Π can be written as:

$$\begin{aligned}\{p_\beta \Leftrightarrow p_1 \wedge \ldots \wedge p_m, \neg p_{m+1}, \wedge \ldots \wedge \neg p_n \mid \beta \in body(\Pi),\\ \beta = \{p_1, \ldots, p_m, not p_{m+1}, \ldots, not p_n\}\}.\end{aligned}$$

This equivalence can be divided into two implications, which are translated into nogoods. The first one states that the body of a rule is true if all literals of the corresponding body are true. It can be translated into the following set of nogoods [54]:

$$\delta(\beta) = \{\boldsymbol{F}\beta, \boldsymbol{T}p_1, \ldots, \boldsymbol{T}p_m, \boldsymbol{F}p_{m+1}, \ldots, \boldsymbol{F}p_n\}.$$

In these nogoods, $\boldsymbol{T}x$ indicates that x is true, and analogously $\boldsymbol{F}x$ indicates that x is false. Informally speaking, the body of a rule β cannot be false if all positive literals are true and all negative literals are false. The second implication expresses that the body of a rule is false if at least one positive literal is false or at least one negative literal true. Furthermore, if the body β holds, all literals in the body have to be true. Thus, the combinations of a true body and a false literal in this body have to be prevented. This case is expressed by the following nogoods [54]:

$$\Delta(\beta) = \{\{\boldsymbol{T}\beta, \boldsymbol{F}p_1\}, \ldots, \{\boldsymbol{T}\beta, \boldsymbol{F}p_m\}, \{\boldsymbol{T}\beta, \boldsymbol{T}p_{m+1}\}, \ldots, \{\boldsymbol{T}\beta, \boldsymbol{T}p_n\}\}.$$

Additional equivalences and thus sets of nogoods are defined for atoms and loops. The complete set of nogoods is discussed in [54].

In order to provide an insight into the created nogoods, Listing 2.6 shows an excerpt from the nogoods of the grounded example ASP program. It presents ASP rules and the nogood which each rule introduces. The derived nogoods are marked by an arrow $\rightarrow$. For example, Line 1 of Listing 2.6 contains the fact *bird(tweety)* and, thus, is a single atom. This atom is translated into the nogood {$\boldsymbol{F}$bird(tweety), $\boldsymbol{T}\emptyset$} since a fact cannot be false. Analogously, Line 2 and 3 introduce similar nogoods. Considering rule heads, a nogood is introduced for each rule containing the corresponding head. For the rule in Line 4 the nogood {$\boldsymbol{F}$flies(tweety), $\boldsymbol{T}${bird(tweety), not -flies(tweety)}} is added. This nogood states that it is forbidden that *flies(tweety)* is false if the body is true. Corresponding nogoods are added for Lines 6 and 8.

```
1 bird(tweety).   →{Fbird(tweety), T∅}
2 bird(tux).      →{Fbird(tux), T∅}
3 penguin(tux).   →{Fpenguin(tux), T∅}
4 flies(tweety) :- bird(tweety), not -flies(tweety).
5 →{Fflies(tweety), T{bird(tweety), not -flies(tweety)}}
6 flies(tux) :- bird(tux), not -flies(tux).
7 →{Fflies(tux), T{bird(tux), not -flies(tux)}}
8 -flies(tux) :- bird(tux), penguin(tux).
9 →{F-flies(tux), T{bird(tux), penguin(tux)}}
```

Listing 2.6: Atom Nogoods for a Grounded ASP Program

To determine an Answer Set of an ASP program with regard to the presented nogoods, Gebser et al. present in [54] the Conflict-Driven Nogood Learning for ASP (CDNL-ASP), which is based on Conflict-Driven Clause Learning (CDCL) [89].

CDCL is based on the DPLL algorithm and introduces back jumps in case of a conflict and adds constraints to prevent already met conflicts.

CDNL-ASP initialises an empty assignment A representing the Answer Set, an empty set of dynamically created nogoods ∇, and an integer representing the decision level dl that counts literals added during a decision step. The first step is the *NogoodPropagation*, which uses the existing nogoods of the Clark Completion to determine the parts of the final Answer Set, which are added to A. If not all atoms could be assigned during this step, additional dynamic nogoods are added to ∇ and the process is restarted. After the *NogoodPropagation*, three cases can occur, a solution, a conflict, and the decision for a literal. A solution is found if all atoms in A are either true or false. A conflict occurs in the case that a nogood is part of A. If the decision level dl is still zero, the conflict is given by the ASP program itself, and thus it has no solution. If dl is higher than zero, the conflict is analysed, corresponding assignments are removed, and dynamic nogoods that prevent the conflicting assignment are added to ∇. If no conflict occurs or no solution is found, an atom is selected based on a heuristic and dl is incremented. As presented in [58] and [95], heuristics track a score of each literal, which increases if it caused a conflict. Typically, the score decreases over time to prevent the selection of the same literals. Since the highest score is given for the variable causing the most conflicts, it is selected to generate additional dynamic nogoods. After the selection of a literal, its truth value has to be determined. For example, the solver Clasp [55] preferably sets atoms to false and bodies to true, which tries to maximise the number of resulting implications.

Considering the example ASP program and the excerpt from the nogoods show in Listing 2.6, the application of CDNL-ASP results in the following steps. In the beginning, A and ∇ are initialised with an empty set and the dl is set to zero. Afterwards, *NogoodPropagation* is applied. During its unit propagation, the facts *bird(tweety)*, *bird(tux)*, and *penguin(tux)* are added to A. Subsequently, *-flies(tux)* is added to A. In the end, *flies(tweety)* is included in A. Since the unit propagation has reached a fixpoint and no loop is given, the *NogoodPropagation* returns A and the still empty dynamic nogood set ∇. Since all atoms and all bodies have been assigned, the Answer Set { *bird(tweety)*, *bird(tux)*, *penguin(tux)*, *-flies(tux)*, *flies(tweety)*} is returned. A comprehensive example including back jumps is presented in [54].

2.5 Clingo

The Potsdam Answer Set Solving Collection[10] (Potassco) is a set of Answer Set Programming tools developed at the University of Potsdam. This set includes Clingo [50], which is a system for grounding and solving ASP programs. Clingo itself is a sophisticated ASP reasoning system combining the grounder Gringo [47], the solver Clasp [55]. Furthermore, it enhances the plain ASP input language by additional features.

[10]Potassco, The Potsdam Answer Set Solving Collection, https://potassco.org/, Accessed December 29, 2021.

2.5.1 Features

The following sections present several selected features. A complete overview of all features provided by Clingo, including its support for Python [119] and Lua [69], is given in the Potassco Guide[11]. The features presented in the following sections facilitate the modelling of dynamic knowledge and are thus applied in the different contributions of this thesis.

Conditional Literals

Conditional Literals[12] have following form: $L_0 : L_1, \ldots, L_n$. These literals can be seen as nested rules, in which L_0 is considered as the head and $L_1, \ldots, L_n$ as the body. Thus, the head of a conditional literal holds if all body literals are true. An example of a conditional literal is given in Line 4 in Listing 2.7.

```
1 bird(tweety).
2 bird(tux).
3 penguin(tux).
4 flies(X) :- not penguin(X) : bird(X); bird(X).
```

Listing 2.7: ASP Program with a Conditional Literal

In this case, the conditional literal can be seen as a switch for the literal *not penguin(X)*. Thus, for every bird, it is checked if it is not a penguin. Since commas separate the literals in the condition, the end of the condition is marked by a semicolon. Additionally, the literal *bird(X)* has to be added after the semicolon to ensure that the variable X is safe. Furthermore, a conditional literal can be added to the head of a rule. In this case, it can be considered as a rule that creates a disjunction because a literal is added to the rule head every time the literals in the body of the conditional literal hold. Hence, conditional literals influence rules depending on their position.

Aggregates

Aggregates[13] support deriving values from groups or sets of literals. Clingo provides the aggregates #count, #sum, #sum+, #min, and #max. #count returns the number of distinct literals. #sum and #sum+ calculates the sum and positive-sum,

[11] Potassco Guide Version 2.2.0, https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf, Accessed December 29, 2021.
[12] Potassco Guide Version 2.2.0, https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf, p. 28., Accessed December 29, 2021.
[13] Potassco Guide Version 2.2.0, https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf, p. 30., Accessed December 29, 2021.

respectively. #min and #max determine the minimum and maximum values. Furthermore, aggregates act on sets and therefore, duplicates are ignored. Aggregates in the body of a rule have the form $\alpha\{t_1 : L_1; \ldots; t_n : L_n\}$ in which α is a placeholder for the aggregate. The elements of the aggregates are given in pairs consisting of a weight t_i and a literal T_i. Since these aggregate return integer values, they can be compared to thresholds via comparison predicates like $<$ or $\leq$. Furthermore, their results can be assigned to variables. Listing 2.8 shows an example of an aggregate in Line 5, which is used to count the number of different birds that can fly. Since just *tweety* is able to fly, *birds(1)* is part of the Answer Set of this program.

```
1 bird(tweety).
2 bird(tux).
3 penguin(tux).
4 flies(X) :- not penguin(X) : bird(X); bird(X).
5 birds(Z) :- Z = #count{X : flies(X)}.
```

Listing 2.8: ASP Program with the Aggregate #count

Optimisation Statements

In order to find an optimal Answer Set, Clingo provides optimisation statements[14]. Clingo supports three kinds of optimisation statements. #maximize and #minimize to find a maximum and a minimum of weights, respectively. Furthermore, weak constraints (:~) enable weighting of the appearance of literals in an Answer Set. Since optimisation aims at finding an optimal Answer Set with respect to a given set of criteria, an ASP program with multiple Answer Sets is needed. Hence, the example used in this section will differ from the bird example used before. The example in Listing 2.9 is given in the official Clingo guide[15] and describes the selection of a hotel based on stars, costs, and noise level.

```
{ hotel(1..5) } = 1.
stars(1,5). cost(1,170).
stars(2,4). cost(2,140).
stars(3,3). cost(3,90).
stars(4,3). cost(4,75). main_street(4).
stars(5,2). cost(5,60).
noisy :- hotel(X), main_street(X).

:~ noisy. [ 1@3 ]
#minimize { Y/Z@2,X : hotel(X), cost(X,Y), stars(X,Z) }.
#maximize { Y@1,X : hotel(X), stars(X,Y) }.
```

Listing 2.9: ASP Program with Optimisation Statements[15]

[14]Potassco Guide Version 2.2.0, https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf, p. 37., Accessed December 29, 2021.

[15]Potassco Guide Version 2.2.0, https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf, p. 39., Accessed December 29, 2021.

The first line contains two shortcuts. The two dots will create five hotel literals ranging from *hotel(1)* to *hotel(5)*. The second shortcut = 1 expresses that the lower and upper bounds of the choice rule are set to 1. Hence, this choice rule will produce five Answer Sets, one for each hotel. Lines 2 to 6 indicate the properties of each hotel. For example, hotel(1) has 5 *stars* and a *cost* of 170 € per night. Furthermore, hotel(4) is situated on the *main street*. Line 7 defines the notion of *noisy*. A hotel is noisy if it is next to the main street. Line 9 to 11 contain the optimisation criteria, sorted according to their importance, which is denoted by an @ in combination with an integer value. The higher the integer value, the higher is the importance of the optimisation criterion. The optimisation statements in this example are given in decreasing order. Thus, the highest priority is to avoid noisy hotels. Then the costs per star have to be minimised and, finally, the stars should be maximised. Applying the weak constraint will sort out hotel(4). Minimizing the costs per star results in hotel(3) and hotel(5) since they both cost 30 € for each star. Finally, hotel(3) is selected since it has more stars than hotel(5). To sum up, the combination of weak constraints, minimisation, and maximisation enables building of complex multi-criteria optimisation and formulate sophisticated ASP programs.

Program Sections

To provide a better overview and enable reusability, Clingo supports the division of ASP programs into smaller parts called Program Sections[16]. Each Program Section starts with the keyword *#program* and has to be grounded and solved separately. Furthermore, Program Sections can be expanded with variables (lower letters in this case) and can be reused with different variable values. Listing 2.10 is an example of the usage of Program Sections.

```
#program fact(n).
bird(n).

#program rule.
flies(X) :- bird(X).
```

Listing 2.10: ASP Program Consisting of Two Program Sections

This example consists of two Program Sections. The first section is called *fact* and has one variable *n*. During the grounding, this variable can be replaced with literals or constants. For example, this Program Section can be grounded with the constants *tweety* and *tux* to recreate the already existing example used before. Afterwards, the second Program Section *rule* can be grounded and solved. This results in the Answer Set: {*bird(tweety)*, *bird(tux)*, *flies(tweety)*, *flies(tux)*}. A highly important aspect when using Program Sections is the order in which they are grounded since the

[16]Potassco Guide Version 2.2.0, https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf, p. 45., Accessed December 29, 2021.

grounding procedure will simplify the ASP program. For example, if the Program Section *rule* is grounded first, the rule in Line 5 is removed since the literal *bird* does not appear in any rule head. Once the Program Section *facts* has been grounded, the resulting Answer Set only contains the fact since the grounding procedure has removed the rule. Hence, the order in which Program Sections are grounded and solved impacts the solution of the ASP program. Furthermore, the introduction of circular dependencies between Program Sections can cause an unsatisfiable ASP program. A detailed description of this problem and the way to prevent circular dependencies are given in Section 2.5.3.

External Statements

During the grounding procedure, literals that do not appear in any rule head are removed. This removal prevents the modelling of rules containing literals, which could appear in further grounding steps. To tackle this issue, Clingo provides External Statements[17]. These are atoms that are not removed during grounding and marked with the keyword *#external*. Furthermore, they have the following form: *#external* $A : L_1, \dots, L_n$. In this case, A is an atom and the literals $L_1, \dots, L_n$ form a condition, which enables the creation of External Statements based on parts of the ASP program. Furthermore, they have three different states. Two states represent the corresponding truth values: true and false. The last state is *free*. A *free* External Statement can no longer be assigned and, thus, rules that depend on this External Statement are removed during grounding. Once an External Statement has been set to *free*, it can no longer be set to true or false. Additionally, External Statements are initially assumed to be false and can be set to true by using the Clingo API. An example program is given in Listing 2.11.

```
1 bird(tweety).
2 bird(tux).
3 #external penguin(tux).
4 flies(X) :- not penguin(X), bird(X).
```

Listing 2.11: ASP Program Using External Statements

Line 1 and 2 are normal facts stating that *tweety* and *tux* are birds. The knowledge that *tux* is a penguin is given as an External Statement in Line 3 and is initially false. Thus, the application of the rule in Line 4 will derive that both, *tweety* and *tux*, can fly. If the knowledge that *tux* is penguin arises, the External Statement can be set to true. Hence, the usage of External Statements supports influencing ASP programs dynamically and to alter rules on demand.

[17]Potassco Guide Version 2.2.0, https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf, p. 44., Accessed December 29, 2021.

2.5.2 Multi-Shot Solving

Typically, Answer Set solvers are used in a single-shot fashion, which means that each solver instance determines the solution for a given problem, returns the solution, and is discarded in the end. To reuse already derived knowledge, it has to be added to the program, which is given to the new solver instance. To cope with this problem and enable a dynamically changing problem specification, Clingo is capable of multi-shot solving [51]. Furthermore, its multi-shot capabilities enable the use of Clingo as a knowledge base. Especially multi-shot solving relies on the usage of External Statements and Program Sections. While Program Sections enable the structuring and reuse of parts of an ASP program, External Statements support altering the truth value of selected literals dynamically. To provide an introduction to multi-shot solving and the usage of Program Sections and External Statements, Listing G.1 in the appendix provides an example for modelling the Tower of Hanoi game, which is an adapted version of the way the game is modelled in [51]. The ASP program consists of three Program Sections. The first Program Section creates the initial state, including pegs, disks, as well as their initial and goal locations. The second Program Section moves a disk from one peg to another one if it is allowed. The third Program Section checks if the goal is reached. Thus, by subsequently applying the second and third Program Section, the solution of the game is found. The complete interaction with the Program Sections is summarised in Algorithm G.1 in the appendix. Applying the presented algorithm finds a solution in 16 calls of the solve procedure and will return the following Answer Set: {*move(4,b,1)*, *move(3,c,2)*, *move(4,c,3)*, *move(2,b,4)*, *move(4,a,5)*, *move(3,b,6)*, *move(4,b,7)*, *move(1,c,8)*, *move(4,c,9)*, *move(3,a,10)*, *move(4,a,11)*, *move(2,c,12)*, *move(4,b,13)*, *move(3,c,14)*, *move(4,c,15)*}. Each of these atoms indicates a move in the Tower of Hanoi game. For example, *move(4,b,1)* denotes that at iteration 1, disk 4 has to be put on peg *b*. Thus, following the moves given in the Answer Set will lead to a valid solution for the Tower of Hanoi game.

2.5.3 Module Property

The Module Property of ASP programs is based on the Module Theorem presented in [103], which checks if ground ASP programs or, respectively, propositional logic programs can be combined without losing valid Answer Sets or Stable Models. A propositional logic program module is defined as a triple $\mathbb{P} = (P, I, O)$. Furthermore, the following three conditions have to hold. P has to be a finite set of rules. Furthermore, the set of input atoms I and the set of output atoms O have to be disjoint. Finally, no atom given in a rule head is part of I. In order to check if two modules, $\mathbb{P}_1$ and $\mathbb{P}_2$, can be combined, the positive dependency graph (see Section 2.4.2) of their combination is needed. If this dependency graph contains at least one strongly connected component [122], the union $\mathbb{P}_1 \cup \mathbb{P}_2$ forms a positive recursion between the modules. For example, a positive recursion occurs if an atom of O_1 depends positively on an output atom of O_2. Since they are part of a strongly connected component, the atom of O_2 depends positively on the atom in O_1. This positive recursion could cause the loss of valid Answer Sets. Thus, two modules that form strongly connected components between them cannot be combined. Hence, two modules, $\mathbb{P}_1$ and $\mathbb{P}_2$, can only be combined if they have disjoint output sets and there is no positive recursion between them. The join of two modules $\mathbb{P}_1$ and $\mathbb{P}_2$ is defined as $\mathbb{P}_1 \sqcup \mathbb{P}_2$. Again, $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is a triple and is according to [103] defined as $(P_1 \cup P_2,\, (I_1 \setminus O_2) \cup (I_2 \setminus O_1),\, O_1 \cup O_2)$.

In order to demonstrate the join of two modules $\mathbb{P}_1$ and $\mathbb{P}_2$, let us consider the ASP programs $P_1 = a$ *:- not b.* and $P_2 = b$ *:- not a.* [103]. This results in the following modules: $\mathbb{P}_1 = (\{a \text{ :- } not\ b.\}, \{b\}, \{a\})$ and $\mathbb{P}_2 = (\{b \text{ :- } not\ a.\}, \{a\}, \{b\})$. Since their output sets are disjoint and there is no positive recursion between the modules, both modules can be joined resulting in the module $\mathbb{P}_1 \sqcup \mathbb{P}_2 = (\{a \text{ :- } not\ b.\ b \text{ :- } not\ a.\}, \{\emptyset\}, \{a,\ b\})$.

To find the Answer Sets of the combined modules, the compatibility of Answer Sets of a module $SM(\mathbb{P})$ has to be defined. Therefore, the Herbrand Base [37] of both modules is needed. Since already grounded ASP programs are considered, the Herbrand Base of a module $\mathbb{P}$ ($HB(\mathbb{P})$) contains all atoms of P, I, and O. Furthermore, the visible Herbrand Base $HB_v(\mathbb{P})$ contains all input and output atoms. As presented in [103], two Answer Sets M_1 and M_2 of two modules $\mathbb{P}_1$ and $\mathbb{P}_2$ have to share a common subset of their respective visible Herbrand Bases to be compatible. Thus, they are compatible if the following equation holds: $M_1 \cap HB_v(\mathbb{P}_2) = M_2 \cap HB_v(\mathbb{P}_1)$. By combining the join of two modules and the compatibility of their Answer Sets, the Module Theorem is derived. This theorem states that a set M is an Answer Set for $\mathbb{P}_1 \sqcup \mathbb{P}_2$ if their join is defined and the Answer Sets $M_1 = M \cap HB(\mathbb{P}_1) \in SM(\mathbb{P}_1)$ and $M_2 = M \cap HB(\mathbb{P}_2) \in SM(\mathbb{P}_2)$ are compatible.

To show the effects of a violation of the Module Theorem, Oikarinen et al. provide an example in [103]. In this example, the two modules $\mathbb{P}_1 = (\{a \text{ :- } b.\}, \{b\}, \{a\})$ and $\mathbb{P}_2 = (\{b \text{ :- } a.\}, \{a\}, \{b\})$ are used, which share the Answer Sets $\{\emptyset,\ \{a,\ b\}\}$. Both modules form a positive recursion and, therefore, violate the Module Theorem. A join of programs P_1 and P_2 creates the program $\{b \text{ :- } a.\ a \text{ :- } b.\}$ that has only the empty set as an Answer Set, excluding the set $\{a,\ b\}$. Hence, causing a loss of an Answer Set, which is still valid. Finally, the adherence to the Module Theorem enables the safe combination of ASP Programs without the risk of losing valid Answer Sets.

In order to ease the use of multi-shot solving, we propose in [106] an automatic satisfaction of the Module Property, which Opfer expands in [105]. The automatic satisfaction of the Module Property is realised as a wrapper for Clingo. Generally speaking, the wrapper encapsulates queries in unique predicates and thus prevents positive recursions.

2.6 Ontologies

To be able to share knowledge and to cooperate based on this knowledge, it is necessary to find and share the same terminology [64, 135]. Furthermore, the terminology has to provide a way to represent the classes that exist in an environment [65]. A common approach is to rely on an ontology. Generally speaking, an ontology can be seen as a formalisation of the structure of knowledge. It provides a taxonomy of classes and supports modelling the relationships between them, their properties, and value ranges. Besides classes, instances (individuals) can be specified, which are objects that belong to a specific class. Furthermore, the definition of axioms allows to model rules that hold in an ontology; for example, all properties of a parent class are also inherited by the child classes.

Ontologies can be divided into several types depending on their formality and specificity [135]. The simplest informal ontology type is a *Controlled Vocabulary*, which is a set of keywords or terms. A *Glossary* expands a *Controlled Vocabulary* with natural language explanations for each term. The next type is a *Thesaurus*, which combines terms with the same meaning. The most complex informal ontology type is an *Informal Taxonomy*. It provides a hierarchy of keywords or terms, but there is no formal definition for each level of the hierarchy. In contrast to informal ontology types, formal ontologies provide formal semantics and support reasoning. *Formal Taxonomies* provide a mechanism to subsume the relationships among their classes or terms. Additionally, *Properties*, *Value Restrictions*, and *Arbitrary Logic Constraints* can be added to increase the expressiveness of the ontology. Especially, the addition of *Arbitrary Logic Constraints* increases the complexity of the ontology, which can result in undecidability.

An additional hierarchy of ontologies can be defined regarding their reusability [135]. The lowest reusability is given in an *Application Ontology*. It comprises concepts that are used in a single application. Usually, *Application Ontologies* are based on a *Domain Ontology*. This type of ontologies provides concepts that can be reused and is suited for a specific domain, such as medical terms, sports terms, and political terms. These classes and terms are broader and thus can be easier reused. Finally, an *Upper Ontology* provides the most common classes, terms, and concepts. For example, the notion of an object or a thing could be defined in an *Upper Ontology*.

2.6.1 Ontology Development

The creation of ontologies is a complex task since it involves the selection of classes and sub-classes, their properties, constraints, and the level of abstraction. Therefore, Noy et al. describe in [100] an iterative guideline for the creation of ontologies. Three rules form the base of this guideline, which state that there is no best way to model a domain, that ontology development has to be iterative, and that classes in the ontology should be close to the objects they represent. The guideline itself consists of seven steps, which will be discussed in the following paragraphs.

The first step is to decide which domain should be represented and what scope the ontology should have. Thus, this step provides a general idea of which classes will be needed, how detailed they have to be represented, and where the ontology will be applied. Furthermore, the ontology developer should determine what questions the ontology should be able to answer.

After the initial determination of domain and scope, Noy et al. recommend checking if already defined ontologies can be reused. On the one hand, this can reduce the overall effort which is needed to create the ontology. On the other hand, the adaption of existing ontologies enables the developer to interact with the systems that provide the corresponding ontology.

In a third step, the ontology developer is supposed to enumerate all terms used in the ontology. This results in a comprehensive list of terms that supports the understanding of the domain and provides the basic modelling blocks that are needed in the following steps.

Based on the terms defined in the third step, the class hierarchy of the ontology and the properties of the classes are defined in the fourth step. According to [100], there are three ways to generate the class hierarchy, which depend on the personal preferences of the ontology developer. The hierarchy can be created in a top-down fashion by selecting the most general term and subsequently adding subclasses. The bottom-up approach starts with the most specific instances and combines them with fitting parent classes. The last way is a combination of both approaches by changing the strategy based on the given classes.

The fifth and sixth steps are the definition of the properties (slots) of each class and value restrictions. Based on the class hierarchy, slots of a class are inherited by its subclasses. Hence, the ontology developer has to decide when to add a slot to a class. In general, a slot should be added to the highest class in the hierarchy that can have the property to prevent redefinitions. Afterwards, value limitations can be given to the properties. This includes cardinalities, ranges, all possible values, and value types.

In the last step, instances are created. Therefore, a fitting class is selected, an instance is created, and values are assigned to the slots.

A further decision that has to be made is the language that is used to create the ontology. A common approach is to use the Web Ontology Language (OWL), which is briefly presented in the following section.

2.6.2 Web Ontology Language

The Web Ontology Language (OWL) is an ontology modelling language that the *World Wide Web Consortium* recommends as a standard for ontology creation [68]. It supports the automatic processing of web documents by applications, by providing semantics of a document by classes, and their properties [92]. OWL allows defining classes, subclass relations, and constraints. A complete description of all language features is given in [68]. Furthermore, OWL provides three sublanguages, which are OWL Lite, OWL DL, and OWL Full. All sublanguages support a different subset of the language constructs, which results in varying expressiveness and complexity. The least complex sublanguage is OWL Lite. It supports the specification of class hierarchies, properties, and limited cardinality restrictions (0 and 1). OWL Lite is decidable and its complexity is ExpTime [92]. OWL DL (description logic) contains OWL Lite and lifts some restrictions. For example, it enables the definition of arbitrary cardinality constraints. Additionally, OWL DL is the decidable subset of OWL Full and has a complexity of NExpTime [92]. Finally, OWL Full supports the complete language specification and contains OWL DL and, thus, OWL Lite. It has the highest expressiveness of the sublanguages. It enables the use of classes as individuals or arbitrary logic constraints. Hence, it is undecidable and not supported by most tools [92]. Additionally, the Unique Name Assumption (see Section 2.3) does not hold. Meaning that different names can be given to a single entity and that it has to be explicitly stated that these names belong to the same entity. Furthermore, the Open-World Assumption is applied in OWL since it is assumed that a knowledge base is always incomplete [92].

A commonly used example to introduce the OWL language is the pizza ontology[18,19]. An excerpt from this ontology is shown in Figure 2.9. The most general class in this figure is *Food*, which is divided into three disjoint subclasses. These are *PizzaTopping*, *Pizza*, and *PizzaBase*. For example, the *PizzaTopping* class has three further disjoint subclasses: *CheeseToppping*, *MeatTopping*, and *SauceTopping*. Explicitly marking the classes as disjoint enables checking the consistency of individuals or classes. For example, the class *Tomatoes* should not be a subclass of *PizzaTopping* and *PizzaBase* simultaneously. The least general class in this excerpt of the pizza ontology is the *MeatyPizza*. It is a subclass of a *NonVegetarianPizza* and, thus, a subclass of *Pizza*. It has a base, which is expressed by the relation *hasBase*, and a *MeatTopping* connected by the *hasTopping* relation. Furthermore, OWL provides a most general class *Thing* [92], which is the superclass of all classes and instances. In contrast to this, *Nothing* is the most specific class. It is the subclass of all OWL classes and has no instances [92]. Both classes, *Thing* and *Nothing*, are omitted in Figure 2.9 to provide a better overview of the classes.

[18]An Ontology About Pizzas and their Toppings, https://protege.stanford.edu/ontologies/pizza/pizza.owl, Accessed December 29, 2021.

[19]Pizzas in 10 Minutes, https://protegewiki.stanford.edu/wiki/Protege4Pizzas10Minutes, Accessed December 29, 2021.

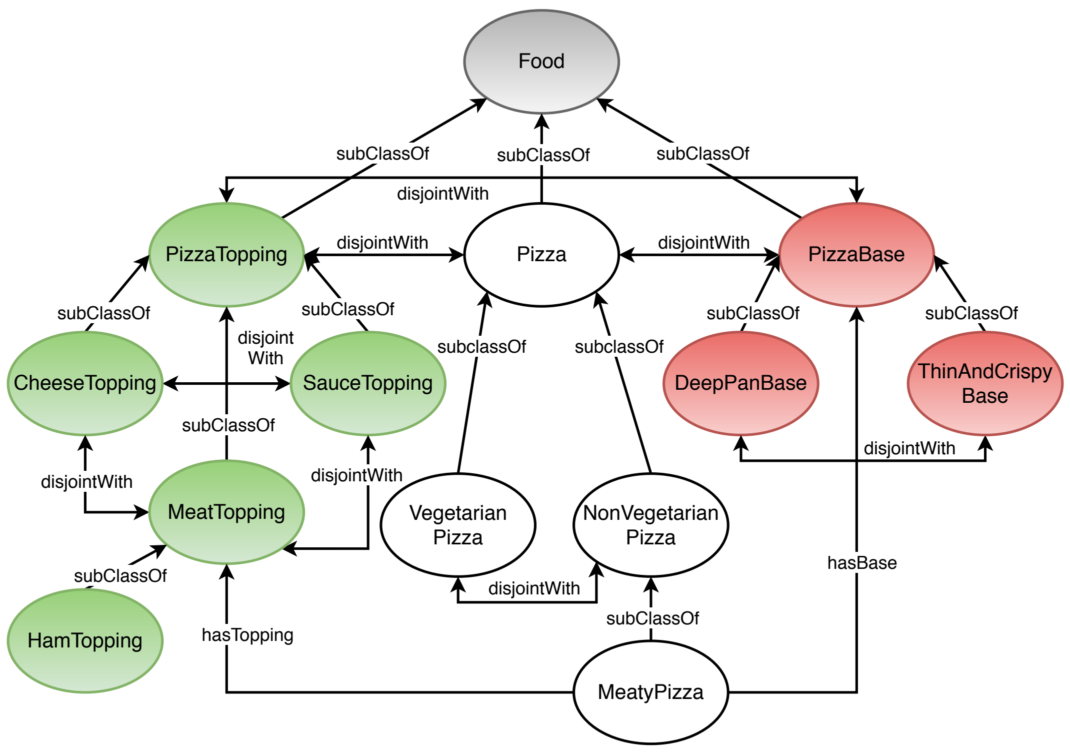


Figure 2.9: Graphical Representation of an Excerpt from the Pizza Ontology[18]

To provide a machine-readable ontology, OWL uses the Resource Description Framework Schema (RDFS)[20]. The definition of the *MeatyPizza* class in RDFS is shown in Listing 2.12.

```
 1 <owl:Class rdf:about="http://www.co-ode.org/ontologies/pizza/
 2 pizza.owl#MeatyPizza">
 3 <owl:equivalentClass>
 4 <owl:Class>
 5 <owl:intersectionOf rdf:parseType="Collection">
 6 <rdf:Description rdf:about="http://www.co-ode.org/
 7 ontologies/pizza/pizza.owl#Pizza"/>
 8 <owl:Restriction>
 9 <owl:onProperty xml:resource="http://www.co-ode.org/
10 ontologies/pizza/pizza.owl#hasTopping"/>
11 <owl:someValuesFrom xml:resource="http://www.co-ode.org/
12 ontologies/pizza/pizza.owl#MeatTopping"/>
13 </owl:Restriction>
14 </owl:intersectionOf>
15 </owl:Class>
16 </owl:equivalentClass>
17 <rdfs:label xml:lang="en">MeatyPizza</rdfs:label>
18 <rdfs:label xml:lang="pt">PizzaDeCarne</rdfs:label>
19 <skos:definition xml:lang="en">Any pizza that has at least one
20 meat topping</skos:definition>
21 <skos:prefLabel xml:lang="en">Meaty Pizza</skos:prefLabel>
22 </owl:Class>
```

Listing 2.12: Excerpt from the RDF Description of the Pizza Ontology[18]

Line 1 of Listing 2.12 defines that the class *MeatyPizza* is part of the pizza ontology, which is given by an URL (http://www.co-ode.org/ontologies/pizza/ pizza.owl). In Lines 3 to 16, an equivalent class is created, which states that a *MeatyPizza* is equal to an anonymous class described in Lines 4 to 15. It has to be part of the *Pizza* class and the *PizzaTopping* has to be a *MeatTopping*, e. g., *HamTopping*. Lines 17 to 21 provide additional information. The English name of the class is *MeatyPizza* and the Portuguese name is *PizzaDeCarne*, which can be used in combination with the ontology URL to address this class. Finally, a natural language description for the *MeatyPizza* is given.

2.7 ConceptNet 5

ConceptNet 5 (CN5) [128] is a multilingual commonsense knowledge base. It aims at supporting machines in understanding the meaning of concepts, which appear in the daily life of humans. The basis of CN5 is a semantic hypergraph. This hypergraph represents commonsense knowledge, which has been extracted from various

[20]RDF Schema 1.1, https://www.w3.org/TR/rdf-schema, Accessed December 29, 2021.

knowledge sources like the Open Mind Common Sense project [125], Wiktionary[21], WordNet [94], DBPedia [5], and Wikipedia[22]. In its current version (5.7), the CN5 hypergraph consists of approximately 34 million edges and supports 304 languages[23]. These include English (1803874 concepts), French (3023144 concepts), and German (825741 concepts). A concept in CN5 is a natural language term, which is annotated with a language tag and a sense label denoting its word class. In order to build a hypergraph, these concepts are connected by edges. The meaning of an edge is given by its relation[24]. An excerpt from the set of relations is shown in Table 2.2.

Table 2.2: 10 from 34 Base Relations of CN5[24]

Relation	Meaning	Example
RelatedTo	A is positively related to B	a cup is related to a glass
HasProperty	A has property B	water has the property
Antonym	A is the opposite of B	day is the opposite of night
Synonym	A is the same as B	feline is a synonym for cat
AtLocation	A can be found at B	a cup can be found at a table
SimilarTo	A is similar to B	a cup is similar to a mug
CapableOf	A is able to B	a boat is able to swim
UsedFor	A is used for B	a pen is used for writing
IsA	A is a sub-class of B	a dog is an animal
FormOf	A is linguistically derived from B	wrote is derived from write

Besides a relation, edges contain their sources, e. g., WordNet or Wikipedia and a weight. This weight is the sum of weights assigned to the sources. A weight greater than or equal to 1.0 denotes that the edge has been extracted from at least one verified source like WordNet. The higher the weight, the more reliable is the knowledge represented by the edge.

To provide an insight into how knowledge is represented in CN5, Figure 2.10 shows an excerpt from the CN5 hypergraph. A *car* usually can be found on a *road* or a *parking lot*. It is used to *drive* or *transport* objects. Furthermore, *drive* is a specific way to *transport* goods. The concept *cars* is linguistically derived from *car*. A *car* is both a *vehicle* and a *machine*. Finally, a *vehicle* is a sub-class of a *machine*, too.

In addition to the hypergraph, CN5 supports the determinations of the relatedness of two concepts [128]. Therefore, CN5 utilises Word Embeddings. Word Embeddings are the mapping of words to a vector of real numbers denoting their distance to other

[21] Wiktionary, The Free Dictionary, https://en.wiktionary.org, Accessed December 29, 2021.
[22] Wikipedia, The Free Encyclopedia, https://en.wikipedia.org, Accessed December 29, 2021.
[23] ConceptNet5 Languages, https://github.com/commonsense/conceptnet5/wiki/Languages, Accessed December 29, 2021.
[24] ConceptNet5 Relations, https://github.com/commonsense/conceptnet5/wiki/Relations, Accessed December 29, 2021.

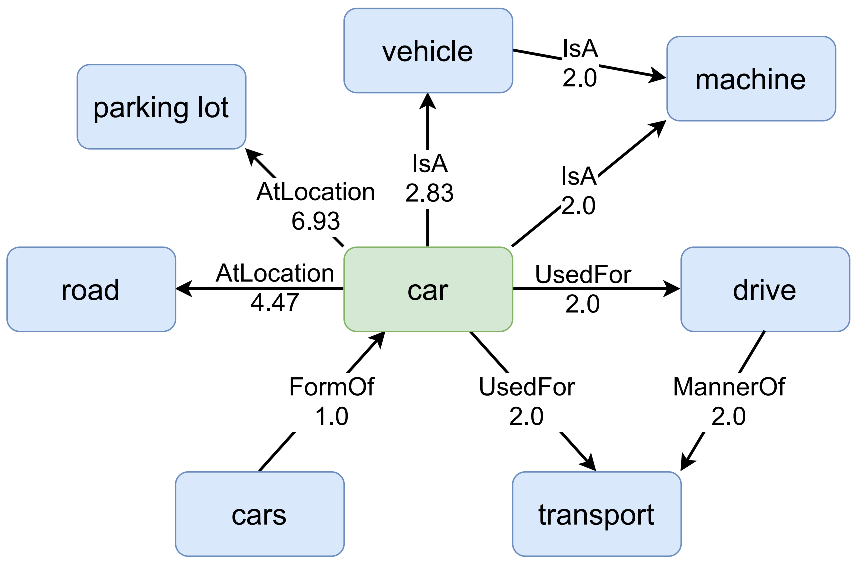


Figure 2.10: Excerpt from the CN5 Hypergraph

words in a text, which results in a description of a word based on words that are close to it in a text [42]. These word embeddings are generated by creating a symmetrical term-term matrix. It includes concepts that appear in at least three edges [128]. The value of each cell is determined by summing up all weights of the edges that connect the corresponding concepts. The relatedness of two concepts is then determined by the similarity of vectors that represent the concepts. Subsequently, the result is normalised, resulting in values ranging from -1 to 1. For example, *machine* and *parking lot* have a low relatedness of 0.077 while *car* and *cars* are strongly related (0.782).

Related Work 3

In this chapter, selected related work is shown. Section 3.1 introduces related work in the field of distributed databases and knowledge management. Subsequently, Section 3.2 presents relevant work focussing on the application of commonsense knowledge in various systems. Hence, both sections form the related work for the distributed knowledge management introduced in Chapter 4. The usage of Answer Set Programming to represent knowledge is discussed in Section 3.3. Section 3.4 discusses publications in the area of ontology generation. Thus, they form the relevant work for the handling of semantic inconsistencies and the generation of commonsense ontologies introduced in Chapter 5. Finally, Section 3.5 introduces relevant work regarding semantic routing. The related work presented in the following sections has been partially discussed and published in [9, 72–78, 105–108].

3.1 Databases and Knowledge Management

In recent years, distributed, non-relational databases and data stores (NoSQL) have emerged [33], which differ in the employed data model, consistency model, and the selected distribution of data. Key-Value Stores like Amazon DynamoDB[25] allow access to data objects by a single key and, thus, enable fast random access to the stored objects. Wide-Column Stores like Apache Cassandra[26] store values using a triple consisting of a row-key, a column-key, and a timestamp. They support batch processing of data and horizontal as well as vertical partitioning of their tables. Document Stores like MongoDB[27], store data similar to Key-Value Stores in key-value pairs. In contrast, values are semi-structured documents like XML or JSON files. By relying on documents, data of any complexity can be stored. Graph Stores like Neo4j[28] store data as a graph consisting of object entities. These entities are considered as the vertices of the graph and the edges between them express their relation. Thus, Graph Stores enable storing object graphs efficiently. Besides the data model, the way consistency is handled differs between the data stores. Strong Consistency provides a single consistent image of the complete data set and thus is hard to achieve in a distributed database since it requires high communication efforts. It can be achieved in parts of Key-Value, Wide Column, and Document Stores [33]. Causal Consistency requires a partial order between causally depending

[25]Amazon DynamoDB, https://aws.amazon.com/dynamodb/, Accessed December 29, 2021.
[26]Apache Cassandra, http://cassandra.apache.org/, Accessed December 29, 2021.
[27]MongoDB, https://www.mongodb.com/ Accessed December 29, 2021.
[28]Neo4j, https://www.neo4j.com/, Accessed December 29, 2021.

operations on the database. It can be achieved in the four presented data models but is not implemented in the presented examples due to its high communication demand [33]. Eventual Consistency has the lowest requirements since replicated data objects will eventually converge to identical values, and thus, no ordering in the data is required. Therefore, this consistency can be achieved with any data model and can be guaranteed in many databases or data stores.

Our proposed Self-Organising Multi-Agent Knowledge Base can be compared with the distributed, non-relational databases introduced in the previous paragraph. The knowledge is distributed to several agents on the network. The agents are able to self-organise, cope with loosely coupled networks, and semantically annotated pieces of knowledge while providing Eventual Consistency. In contrast to Key-Value Stores, the proposed system will not rely on static table schemata. Instead, it will enable the user to define relations at design as well as run time. Since ASP, in combination with the solver Clingo, enables the use of arbitrary strings as symbols, the proposed knowledge base can use file links like Document Stores. Finally, the knowledge base can be compared to Graph Stores. Knowledge is represented by tuples, which are parts of a knowledge graph. In contrast to the presented Graph stores, the suggested knowledge base will be an active part of the system using it since it will automatically distribute and maintain the knowledge.

Bonifacio et al. present a Peer-to-Peer architecture tailored for distributed knowledge management in [17], which is extended in [18]. It is based on JXTA [62], which is a collection of protocols used for communication and cooperation. Each peer can act in two different roles. A Seeker interacts with a user and forwards queries to Providers, which, in turn, try to answer the queries and forward them to neighbouring Providers. The knowledge is stored in files and is annotated using either a schema, a taxonomy or a context. These are created using the Context Description Language, which is an XML-like language supporting the definition of a hierarchical context. Furthermore, Bonifacio et al. define two principles a distributed knowledge management has to adhere to. The first is autonomy, which states that each group has to maintain its knowledge autonomously. The second principle is coordination. Groups share their knowledge and provide mechanisms to translate knowledge.

The system introduced by Bonifacio et al. is similar to our distributed and multi-agent-based knowledge base. Both use groups of distributed nodes to manage the stored knowledge. A significant difference is given in the way the knowledge is stored. While Bonifacio et al. use an XML-like language to model the relations between the stored files, we utilise ASP and the solver Clingo. The combination of ASP and Clingo has, among others, two advantages over XML-like languages. The application of ASP enables reasoning about the stored knowledge and the use of Clingo provides mechanisms to adapt the knowledge storage at run time.

3.2 Application of Commonsense Knowledge

The application of commonsense knowledge to solve everyday tasks is part of human nature. Thus, following the human example, the application of knowledge in general and especially commonsense knowledge has been broadly studied in various fields such as service robotics.

One of these frameworks is *DyKnow*, presented by Heintz et al. in [66]. The central aspect of *DyKnow* is the distributed collection of various kinds of inputs. They include a broad span ranging from raw or processed sensor values up to symbolic representations of objects, which are connected via relations. The *DyKnow* middleware consists of two types of units. The first type of unit is a source, which is a process that provides data samples and is considered as mappings of time points to samples. Computational units conduct the refinement of these samples. They combine inputs provided by sources to higher-level information or knowledge. The declarative *knowledge processing language* [66] is used to specify static networks of the source and computational units. This results in the automatic processing of data and their refinement into information and knowledge.

The general structure of *DyKnow* can be compared to our multi-agent knowledge base. Both distinguish two specific roles, which handle the provided input in different ways. The main difference, however, is given in the way both systems are used. While *DyKnow* can be considered as a distributed source of data, information, and semantically annotated knowledge, our multi-agent knowledge base focusses on the storage of semantically enriched knowledge. Hence, both works can be seen as complementary since the knowledge provided by *DyKnow* could be used as a source for our semantic knowledge handling (see Chapter 5) and, thus, input for the distributed storage of knowledge (see Chapter 4).

A further framework using knowledge to enhance the capabilities of service robots is *KnowRob* [130]. The primary purpose of this framework is the improvement and execution of plans by applying domain-specific or common knowledge. A central aspect of *KnowRob* is its knowledge base. The received information is stored in Prolog predicates. In addition to the classical knowledge base, a virtual knowledge base is created. It is an extension that creates abstract representations of data once it is queried to improve its overall quality. The knowledge base is based on OWL ontologies. These ontologies adhere to the Open-World assumption and are undecidable in the full OWL specification. However, using the decidable subset of OWL is limited in its expressiveness [96]. In the cases that the Closed-World assumption is needed, Prolog and a corresponding reasoner are used. While the OWL ontologies form the first layer of the knowledge base, Prolog predicates in the second layer link the OWL classes to corresponding reasoning methods. A complete specification of the *KnowRob* knowledge base and its knowledge representation is given in [131]. In its second version [13], *KnowRob* is expanded with a simulation.

It enables the grounding of the symbols used in the knowledge base during the simulated execution of the task, which are then used during their actual execution. Besides Prolog predicates, *KnowRob* introduces the concept of computables. Their truth value is initially unknown and later set by external sources. Hence, they can be compared to the External Statements provided by Clingo.

Comparing *KnowRob* to our dynamic knowledge base, the number of involved agents is a major difference. *KnowRob* is designed to be used on a single agent. In contrast, our multi-agent knowledge base distributes its knowledge management on a dynamically changing group of agents. The second difference is given in the dynamic of the application scenarios. Robots equipped with *KnowRob* achieve astonishing results in their application scenarios, for example, preparing breakfast. However, these scenarios are rather static and do not require fast reasoning. In contrast to this, our knowledge base is tailored for dynamically changing environments. Finally, both frameworks rely on a similar kind of symbolic knowledge representation.

In [39], Erdem et al. employ a hybrid planning approach to model the task of tidying a household. It is a combination of ASP, Prolog, ConceptNet 4, and a motion planner. The task of tidying the household as well as the actions of a robot are formulated in ASP. ConceptNet 4, the predecessor of ConceptNet 5, is used as a commonsense knowledge source. Two of its relations are used and translated into further ASP predicates. The *AtLocation* relation provides typical places of objects, while the *HasProperty* relation is used to describe if an object is fragile. The resulting ASP predicates are then used as external input of the Prolog program, which conducts the actual task planning. Finally, the resulting commonsense knowledge enriched task is given to motion planning.

The approach presented by Erdem et al. is similar to the integration of commonsense knowledge into the knowledge base of a robot shown by us in [104, 107]. Besides using ConceptNet 5 instead of ConceptNet 4, the extracted knowledge is directly translated into ASP instead of using Prolog. This reduces the overall complexity of the system since no additional formalism has to be applied. Additionally, the utilisation of Clingo External Statements enables the retraction of commonsense knowledge from the knowledge base in the case of contrary knowledge is received. A further difference is the amount of employed commonsense knowledge. Erdem et al. only use the *AtLocation* and *HasProperty* relation to prevent possible inconsistencies. In contrast, the full set of over 34 base relations is applied in [104, 107]. To prevent semantic inconsistencies in the knowledge base of a robot, we introduce an automatic prevention of inconsistencies in [78], which relies on commonsense knowledge to find contradictions in the properties of an object.

Lemaignan et al. present in [86] a framework for autonomous robots with advanced human-robot interaction skills. They use OWL in combination with the Pellet reasoner [126] as their knowledge representation and reasoning formalism. This limits the addition of further knowledge during runtime since the complete knowledge base has to be classified again to add new classes. The knowledge base of a robot is additionally supplemented with databases such as DBPedia [5] and WordNet [94], which are part of ConceptNet 5. The communication and dialogues between robots are modelled by the Dialogs [85] component. It heavily relies on the knowledge base to resolve the relationships between the perceived words.

The usage of commonsense knowledge shown by Lemaignan et al. is similar to our usage of commonsense knowledge presented in [104, 107] and our detection of inconsistencies introduced in [78]. Both enrich their knowledge bases with commonsense knowledge extracted from external sources. A difference is given in the used reasoning formalism. Lemaignan et al. rely on OWL, which requires the reclassification of the complete knowledge base to introduce additional classes. This disadvantage can be removed using ASP as proposed by us in [78, 104, 107].

Ayari et al. present in [6] a system for ambient assisted living, which applies commonsense knowledge and reasoning. Two ontologies form the basis of their framework. The first ontology provides a hierarchy of classes and individuals using a generalisation / specification relation like *IsA*. Furthermore, it is used to describe static commonsense knowledge. The second ontology is used to represent dynamic terms. Both ontologies can be enriched by commonsense knowledge extracted from external sources like WordNet [94]. Instead of using OWL for the ontologies, the Narrative Knowledge Representation Language (NKRL) [138] is utilised to form the ontologies. In addition to classical ontology features, NKRL supports the definition of events, which serve as templates for actions.

The framework introduced by Ayari et al. is similar to the combination of our modelling of commonsense knowledge [107] and our inconsistency handling [78]. Both approaches use commonsense knowledge to support agents in solving their tasks, for example, in a household scenario. The major difference is given in the selection of the knowledge representation language. Ayari et al. use NKRL, which focuses on the narratives of text. It is not fully declarative and adheres to the Open-World assumption. In contrast, our approaches utilise the fully declarative language ASP, which supports the dynamic adaption of the knowledge and enables the use of the Closed-World assumption.

Chen et al. focus in [25] on the interaction between humans and robots by translating limited segments of natural language, which are restricted to if-then clauses, into ASP. The proposed method is divided into three steps. The first step is the generation of a grammar tree and the relations between the words of the perceived sentence. The second step is a semantic analysis generating logic predicates that define the meaning of the sentence. Finally, a pragmatic analysis forms the ASP predicates.

In contrast to our approach shown in [107] and in [78], no additional forms of commonsense knowledge can be added by Chen et al. [25]. Furthermore, no mechanism to detect or prevent inconsistencies is given. To avoid them, each reasoning step depends on an increasing timestamp.

Böhnstedt et al. present in [16] a tagging concept to enrich resources semantically using individual knowledge networks. In general, a knowledge network models knowledge as a graph with concepts as nodes and relations as edges. In the presented tagging concept, each user builds their own knowledge network by semantically tagging web page fragments. By utilising the semantics of the tags, an efficient search method is created that supports filtering, recommendation, and collaboration between users.

The general idea of Böhnstedt et al. can be compared to our application of commonsense knowledge. We utilise this knowledge to semantically annotate stored pieces of knowledge to enable an efficient knowledge discovery in loosely couple networks. While Böhnstedt et al. rely on users to create individual knowledge networks, we utilise the CN5 hypergraph to extract the necessary commonsense knowledge.

3.3 Knowledge Representation Using ASP

Answer Set Programming (ASP) is a widely used knowledge representation and reasoning formalism. This is the case since ASP provides a rich, declarative, and at the same time easy to use language. ASP enables the definition of defaults, it provides non-monotonic reasoning capabilities, and is supported by efficient solvers. Hence, it is best suited for the dynamic management of knowledge introduced in this thesis. To provide an overview of state-of-the-art applications of ASP, several works utilising ASP are presented in this section.

In [56], Gebser et al. detect inconsistencies in large biological networks using ASP. Vertices of the network model genes, metabolites, proteins, and regulations between them are formed by the edges. The applied ASP program is divided into three parts. The first part contains the problem instance, which consists of facts representing vertices, edges, and observations. The second part is the generation of solution candidates. It consists of disjunctive ASP rules, which create the solution space. The last part is the testing of possible solution candidates by adding further rules and constraints. The resulting ASP program is then applied to experimental measurements to handle incomplete and unreliable data.

The approach shown by Gebser et al. can be compared with our approach [107], which we expanded with an inconsistency handling in [78]. Both aim at locating inconsistencies in a given set of data or knowledge, which is organised in a graph structure. However, Gebser et al. rely on single-shot solving, while we utilise multi-shot solving to create a dynamic commonsense knowledge base.

Gonçalves et al. focus on the process of *intentionally forgetting* in ASP [61]. The focus is set on supporting strong persistence. It requires that all semantic relations between atoms that are not intentionally forgotten have to be preserved. However, they conclude that this is not always possible, especially in two cases, which are either choice rules or atoms that are actively used to determine the truth value of other predicates. Further investigation concerning intentional forgetting has been conducted in [60]. Relying on the notion of modules (see Section 2.5.3), uniform persistence is introduced, which relies on moving input and output atoms to the set of hidden atoms inside each module.

The discouraging results indicated by [61] have influenced our modelling commonsense knowledge presented in [107], our handling of semantic inconsistencies shown in [78], and our semantic routing framework introduced in [75]. To avoid the necessity of intentional forgetting operators, a combination of negation-as-failure, classical negation, as well as External Statements is applied. Informally speaking, an atom in a rule head depends negatively on its negated version in the body. Instead of actively forgetting atoms, the corresponding negative literal is added, stating that the knowledge no longer holds.

Redl et al. focus in [114, 115] on the detection and explanation of inconsistencies in a given ASP program. According to [115], an ASP program is inconsistent concerning a set of input predicates if no Answer Set can be derived. Inconsistency reasons are determined by comparing the input sets with the rules of the ASP program and selecting the pairs of facts and rules which derive the contradiction. This process is refined in [114] to provide consistent program splits.

In contrast to our inconsistency handling [78], Redl et al. do not focus on semantic inconsistencies. Instead, the focus is set on finding the reasons for syntactic inconsistencies. Furthermore, Redl et al. do not provide mechanisms to prevent inconsistencies.

Basu et al. present in [10] a natural language understanding (NLU) framework using ASP and a commonsense knowledge source. To understand English sentences, their algorithm parses the sentences and generates a syntactic tree. Subsequently, the verbs are extracted and their semantic is determined using VerbNet [82], which provides semantic frames for English action verbs. For example, the verb *grab* involves an *agent* that performs the action and a *theme* that is affected by the action. The resulting frames are then translated into s(CASP) [4], which is a grounding free ASP variant. The resulting ASP programs are used in two frameworks: SQuARE, which is a question answering system, and, StaCACK, a chatbot focussed on reservations.

The framework shown by Basu et al. is very similar to the usage of commonsense knowledge we have suggested in [107] and in [75]. Both approaches utilise commonsense knowledge and ASP to solve tasks related to interaction with humans. The main differences are given in the utilised frameworks and the field of application. While Basu et al. focus on verbs, we employ the complete set of concepts provided by ConceptNet. We use the ASP-Core-2 Input Language Format which is described in detail in [24] since it enables the use of prototypic rules. Finally, we focus on the solving of tasks presented to a service robot and prevent possible semantic inconsistencies created during the process.

Evans et al. introduce the *Apperception Engine* in [40], which is an automatic tool for making sense of sensory input. In their work, making sense is defined as the process of *constructing a symbolic theory containing a set of objects that persist over time, with attributes that change over time, according to general laws.* The resulting theory, on the one hand, should explain the sensory input. On the other hand, it has to satisfy the following unity conditions. Objects are united in space. Constraints unite predicates between these objects. Ground atoms form states that have to respect the constraints and static rules. Finally, states form sequences by applying causal rules [40]. To formulate these conditions and to represent the considered sequence of inputs, ASP is used. Therefore, facts are used to model the input, while rules model the unity conditions. In the last step, the most cost-efficient symbolic theory is selected by weak constraints (see Section 2.5.1).

The modelling of sensory input and the application of weak constraints is similar to the knowledge representation of our proposed multi-agent knowledge base discussed in Section 4.2 and in [76]. Input is modelled as External Statements, the interdependencies by rules, and the selection of the best alternative by weak constraints. External Statements enable the dynamic adaption of the facts and, thus, the results of a query. In contrast to this, Evans et al. rely on *holds* predicates, which depend on discrete time steps.

Further related work can be found in the area of Truth Maintenance Systems such as [12, 35]. They support non-monotonic reasoning like ASP. Additionally, a single justification is required to introduce a true proposition. However, syntactic inconsistencies can be caused by rules that add semantic contradictions to a knowledge base, thus, rendering the knowledge base useless. In comparison to Truth Maintenance Systems, Belief Maintenance Systems [41] do not rely on a single justification. Instead, belief support is aggregated. Still, beliefs that cause inconsistencies that have the same belief support cannot be resolved, thus, leading to a contradiction in the knowledge base. Additionally, both types of systems do not have access to external knowledge sources that could be used to deal with the contradictions.

3.4 Ontology Generation

Ontologies are a common approach used to provide a shared terminology, share knowledge, and cooperate utilising the shared knowledge [64, 135]. However, the manual creation of these ontologies is tedious and error-prone. Hence, many approaches to automatically create ontologies have been proposed.

The first kind of approach originates from databases and data warehouses. Both consist of tables, which utilise named columns to store data points in rows. Furthermore, columns can contain references to columns of other tables, which form their relations. Several approaches rely on this structure to automatically generate ontologies. A selection of these works is presented in the following paragraphs. The general idea of these approaches is to employ a fixed set of rules that translates the database or data warehouse scheme into an ontology. Table names are translated into classes, column names serve as properties, and rows are interpreted as individuals.

Following this method, Kiong et al. present the Health Ontology Generator (HOG) in [81], which extracts an ontology from a given SQL database. During this process, bridge tables, references tables, and reference fields are highlighted. To ease the generation for a human user, HOG provides a graphical user interface. Similarly, da Silva et al. [124] extract ontologies from a data warehouse based on a set of fixed rules. A refinement of this method is provided by Zhou et al. in [140]. In contrast to Kiong et al., references are automatically mapped to the corresponding relations instead of introducing special relations for bridge or reference tables. Furthermore, Zhou et al. provide techniques to extract cardinalities of properties. If a column in a relational database is marked as *NOT NULL*, the minimum cardinality of the resulting property is set to *1*. In the case that a column is marked as *UNIQUE*, the maximum cardinality is set to *1*. Further features are included by Al Khuzayem et al. in [2]. Amongst others, these include subclass relations as well as symmetric or reflexive properties. Furthermore, they generate bidirectional mappings between the original database and the resulting ontology. A comprehensive comparison of further approaches is given in [38].

The approaches presented in the previous paragraph achieve good results in automatically generating ontologies. However, the resulting ontologies lack expressive power and flexibility. This is caused by the way databases are usually built. They typically consist of a low number of tables consisting of data points stored in the rows. This results in very few classes and a considerable amount of individuals. Furthermore, the number of relations is restricted by the references between the tables. In contrast to this, we introduce the ARRANGE framework in [77]. Instead of databases, ARRANGE uses hypergraph-based knowledge sources for automatic ontology extraction. This results in a huge amount of classes, which are connected by relations defined by commonsense knowledge. A further difference is given by the selected language used to represent the ontology. In contrast to [2, 81, 124, 140], ARRANGE uses ASP instead of OWL, which results in a dynamically adaptable ontology.

In comparison to fully structured sources like databases, free text sources do not provide fixed names that can be used as classes in an ontology. Mousavi et al. present OntoHarvester in [97], which extracts a domain-specific ontology from a free text. They use an existing ontology as a seed and extend it by applying graph-based search patterns. The resulting classes are added to the ontology if they appear frequently and have high confidence. Additionally, they actively search for aliases and synonyms to increase the quality of the resulting ontology. This process is stopped when no significant amount of new classes can be found. The resulting ontology resembles a typical taxonomy as it does not provide subclasses, properties, and value restrictions.

The approach of OntoHarvester is similar to ARRANGE presented in [77]. OntoHarvester uses a seed class or seed ontology and applies graph-based search patterns to derive subclass relationships. In contrast, ARRANGE uses ASP and, thus, supports the dynamic adaption of the ontology during design and run time and reasoning. Besides OntoHarvester, many approaches exist (see [110]) that utilise knowledge graphs extracted from free texts or semi-structured knowledge. To further refine the resulting ontologies, crowd-based approaches or expert knowledge are used. ARRANGE employs a similar approach since it extracts ontologies from the hypergraph-based knowledge source CN5. The use of ASP in ARRANGE, however, provides axiomatic semantics and reasoning support.

In [118], Ricca et al. describe the OntoDLV system, which is tailored for enterprise ontologies. Instead of modelling the ontologies in OWL, they chose ASP based on two reasons: the application of the Closed-World Assumption (CWA) [116] and the Unique Name Assumption (UNA) [117] in ASP. Both assumptions ease the use of an ontology, especially in enterprise environments. Additionally, CWA and UNA typically hold in the databases, which were used to generate the ontologies. To model the ontologies, OntoDLP is used, which is an extension of the basic ASP syntax. Classes are marked with the keyword *class*. Furthermore, additional keywords for individuals, relations, modules, data types, lists, and sets are available. A taxonomy is defined by adding the additional keyword *isa*. For example, an employee class is defined as follows: ***class*** *employee* ***isa*** *person(salary : integer,company : enterprise)* [118]. This OntoDLP line states, that an *employee* class is a subclass of *person* and has the properties *salary*, which is an *integer*, and *company*, which has the type *enterprise*.

In comparison to OntoDLV, ARRANGE [77] adheres to the ASP-Core-2 standard. By incorporating the External Statements and Program Sections provided by the Clingo solver, dynamically adaptable ontologies are generated.

3.5 Semantic Routing

To locate pieces of knowledge in a distributed knowledge management system, as presented in this thesis, an efficient routing mechanism and knowledge discovery is needed. Many Peer-to-Peer (P2P) systems apply search and routing mechanisms like flooding, random walks, distributed hashing or central repositories. Steinmetz et al. provide in [129] a comprehensive description of these mechanisms and give a detailed insight into P2P systems. Mechanisms like flooding are not suited for unstructured and unstable networks, for example, as present in Search & Rescue scenarios (see Section 1.2.2). Flooding by many users, including authorities, rescue teams, and civilians, can lead to network congestions or, in the worst case, to a complete breakdown of the communication network. Random walks do not assure that the required piece of information or knowledge is found. Central repositories could be unreachable and introduce a bottleneck and a single point of failure to the system. Distributed hash maps like [133] map keys of a document or a piece of knowledge to nodes and, thus, violate the autonomy principles defined by Bonifacio et al. [17]. Furthermore, they are not designed for unstable or dynamic networks since they tend to store knowledge equally distributed on the network. This could result in a mapping of a document to a remote node that could be no longer reachable. Therefore, an efficient content-centric approach to locate knowledge is needed.

The first type of related works is from the area of Semantic Query Routing in P2P systems. Generally speaking, the Semantic Query Routing Problem can be described as discovering the location of semantically annotated files or documents in an unstructured P2P network. A related work belonging to this category is presented by Gómez Santillán et al. in [59]. They use an adapted Ant Colony algorithm to learn paths leading to the requested documents. Therefore, the algorithm subsequently adapts the pheromone levels of routes regarding their hit rates and hops. Thus, the more often a specific query is propagated through the network, the better the route converges to the optimum.

The system proposed by Gómez Santillán et al. is suited for rather static environments with recurring queries containing the same topic since a learning phase is required. Frequent changes in the network structure hamper that the routes converge. In contrast to this, we present in [75] a semantic routing approach that relies on semantic routing tables, which are dynamically adapted to the network structure. These tables are automatically created during the forming process of the used multi-agent system and updated if new knowledge is introduced or the network topology changes. Hence, our system does not require a learning phase.

The second type of related work can be found in the area of Content-Based Routing. A typical approach is to rely on taxonomies to ease the discovery of related knowledge. Michlmayr et al. present in [93] a routing mechanism that utilises taxonomies to determine the similarity of documents. Documents, which share a super concept are considered similar. A related approach is introduced by Pireddu and Nascimento [111]. Nodes in the network use taxonomies to sum up the number of documents of each category and propagate them to their direct neighbours. Additionally, the resulting numbers for upper-level categories are shared between neighbours of neighbours. Finally, if the number of hops required to reach a document is greater than the depth of the taxonomy, the information is removed from the knowledge base.

In comparison to Michlmayr et al., our semantic routing mechanism [75] utilises taxonomies to aggregate routing entries to reduce the size of the routing tables. This approach can be compared to the work of Pireddu and Nascimento. Both apply taxonomies to aggregate knowledge about similar categories. However, the work of Pireddu and Nascimento focuses on the local neighbourhood and is not tailored for routing queries over long distances. Furthermore, our approach shown in [75] is not limited by the depth of the taxonomy and tailored for loosely coupled networks.

Koloniari et al. discuss in [83] a hierarchically organised P2P system managing knowledge stored in XML files. To solve queries, each node applies a local Bloom Filter [34]. Generally speaking, a Bloom Filter is a data structure providing a compact probabilistic representation of a set of objects. To improve the performance of the Bloom Filters, the system proposed by Koloniari et al. supports the merging of filters in a bottom-up way. Parent nodes merge the filters of their child nodes. Thus, a parent node forwards the query to a fitting child node. If no child is able to solve the query, it is forwarded to a higher level of the hierarchy.

The system proposed by Koloniari et al. is suited for static environments since nodes have to stay in the hierarchy to prevent a high number of updates. Furthermore, the system is limited by the resources of the root node since it has to manage most queries and updates. To prevent false positives, Bloom Filters have to be regularly adapted. In contrast to this, our semantic routing relies on ASP to form dynamically adaptable routing tables, which are tailored for dynamic environments. Furthermore, it is not limited to XML files. Instead, it supports the storage of any kind of serialised knowledge.

A further approach used for Content-Based Routing is the application of hierarchical names, which for example, have been used by Jacobson et al. in [70] and Gritter et al. in [63]. In general, hierarchical names consist of a list of concepts starting with the most general one. Furthermore, the concepts are separated by a special character. Since there are no restrictions to the names, consistent names or annotations of the content are not given. For example, names could start with an URL.

While hierarchical names are similar to a path in a taxonomy, the properties of a taxonomy, for example, merging of child concepts, are not applied. Furthermore, identical concepts can appear at arbitrary parts of the hierarchical names and, thus, complicate aggregations. In contrast, our semantic routing method uses concepts organised in a taxonomy to aggregate routing entries to minimise the size of the managed routing tables. Furthermore, the inclusion of an ASP solver enables solving semantic queries.

Manfredi et al. present Scalable Hybrid Adaptive Routing for dynamic multi-hop Environments (SHARE) in [88]. SHARE relies on a combination of techniques to efficiently route queries. This includes gradient-based routing for long-distance queries, local link-state routing, and scoped flooding. To prevent a high load for dense areas in the network, each node only maintains knowledge about its two-hop neighbourhood. Additionally, two kinds of control messages are used. Gradient Establishment Messages for long-distance routing and Heartbeats for local scopes (one hop).

SHARE is similar to our semantic routing mechanism. Both only rely on flooding if no routing information is available. Furthermore, both rely on regular small messages (heartbeat reps. ping messages) to establish routes. However, SHARE is not suited for content-based routing. In contrast to SHARE, our approach utilises semantic information and a taxonomy to aggregate the created routing tables to minimise their size and increase their efficiency.

Orda et al. show an extension to the routing tables used in the Kademlia P2P system [90] in [109]. To improve the routing tables of Kademlia, Orda et al. introduce a multi-dimensional metric and distance calculation. These rely on a complex identifier containing a unique id and coordinates. Furthermore, changes in the network are propagated by adjusting the bucket ranges.

In contrast to our semantic routing approach, Orda et al. do not rely on semantic information and, thus, do not aggregate routing entries leading to similar knowledge or documents. Furthermore, the expansion is suited for mostly static networks to prevent a high number of adaptions to the buckets.

Part II

Solution

Distributed Knowledge Storage and Management 4

The central management of knowledge in a network of heterogeneous participants has a significant advantage. Since all relevant knowledge is stored at a central node of the network, queries can easily be resolved using all relevant available knowledge with as few messages as possible. However, central solutions are not suited for dynamically changing environments since it cannot be guaranteed that centralised management components are reachable by every participant. In a decentralised knowledge management system, knowledge is distributed on several nodes of a network. Since relevant knowledge can be located on several nodes, queries must be disseminated in the system, increasing the message complexity. However, no bottleneck and no single point of failure are introduced to the system. Furthermore, a failed or unreachable node of a distributed knowledge management will not cause a complete system failure. This is especially beneficial in highly dynamic environments.

The remainder of this chapter is structured as follows: Section 4.1 presents specific requirements for the distributed storage of knowledge. Section 4.2 introduces the concept of a multi-agent-based and distributed knowledge base. The agents form a tree-like structure to manage knowledge efficiently. Since several distinct knowledge bases can be formed in separate networks, protocols and mechanisms are needed to combine these knowledge bases if new network connections are established. For example, this could happen by introducing additional communication infrastructure during an emergency. The corresponding protocols and mechanisms are introduced in Section 4.3. Section 4.4 summarises this chapter.

4.1 Specific Requirements

Service-oriented architectures (SOAs) are typically deployed in dynamic environments. The incorporation of microservices further increases the complexity of SOAs, since they frequently change. This can either be caused by updates, changes in their context or failures. Even small changes can affect huge parts of a microservice composition and thus requires a broad adaptation of the corresponding microservices. Thus, a distributed knowledge management is needed to handle the frequent changes and provide suitable replacements for microservices. A detailed description of the scenario can be found in Section 1.2.1.

Further examples of highly dynamic environments are future digital cities. Digital cities strongly depend on their IT and communication infrastructure[29] (see Section 1.2.2). In the near future, this dependency will further increase in most areas of a digital city. Besides the already existing infrastructure, heterogeneous participants like autonomous cars, unmanned aerial vehicles (UAVs), and autonomous mobile robots will rely upon the existing communication infrastructure. Hence, future digital cities can be considered as large distributed systems that consist of heterogeneous participants. Furthermore, these participants rely on communication to exchange data, information, and knowledge. Typically, central control instances manage these critical aspects of a city and introduce central failure points or bottlenecks to the distributed system. To overcome this issue, a resilient exchange of data, information, and knowledge is needed. This is especially the case in emergencies or after natural disasters. For example, let us consider an earthquake. During an earthquake, buildings could collapse, water or gas pipes could be damaged, communication infrastructure could be destroyed, and humans could be injured or buried. To manage situations like this, rescuers need an effective and reliable way to access and manage mission-critical knowledge [75]. Due to limited communication capabilities, the reachability of a central knowledge management system cannot be guaranteed. Hence, the knowledge has to be distributed on the remaining nodes of the network to at least guarantee partial access.

Both scenarios shown above indicate that central entities, which manage a network or knowledge are not suited for highly dynamic environments. However, simply splitting a central knowledge base into several parts and distributing them among the network nodes does not solve the presented issues. In addition, a smart method to manage the knowledge is needed. As proposed by us in [78], Multi-Agent Systems (MAS) can manage the knowledge since MAS grant loosely coupled and decentralised organisations. Furthermore, agents are able to act autonomously and react to changes in their environment. They can cope with lost messages or failures of other agents. Thus, the application of agents and the resulting MAS creates a failure-tolerant and robust distributed knowledge base.

Considering the requirements introduced in Section 1.1, this chapter aims to fulfil Requirement **R1 - Handling Dynamic Environments**. Therefore, Section 4.2 presents the fundamental concepts of the multi-agent knowledge base, its distributed knowledge management, and the query resolution. Since several knowledge bases can be deployed in large-scale environments, Section 4.3 discusses protocols that enable the exchange of knowledge and the forwarding of knowledge between several knowledge bases.

[29]Emergency Responsive Digital Cities, https://www.emergencity.de/, Accessed December 29, 2021.

4.2 Agent-based Knowledge Management

As discussed in the previous section, providing a reliable and robust knowledge base in loosely coupled networks consisting of heterogeneous participants is a major challenge and requires a distributed knowledge management [76]. Especially Service-oriented architectures (SOAs) deployed in highly dynamic environments like Fog or Edge computing need a decentralised service registry to be able to cope with service updates or failures. Jahl proposes in [72] NIO-CMS (Networked Intercooperative Objects - Change Management System) to create the knowledge needed to discover a suitable service replacement for a failed service. Figure 4.1 presents the concept of NIO-CMS.

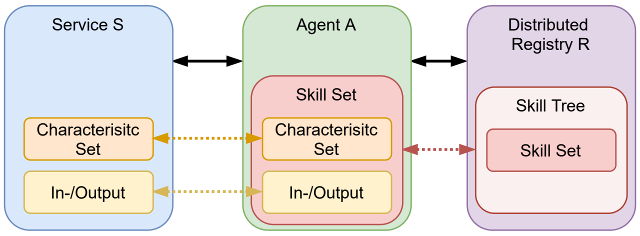


Figure 4.1: Concept of NIO-CMS [72]

NIO-CMS is based on a MAS in which each agent monitors a service and stores a set of *Skills* for the monitored service. On the one hand, this *Skill* set contains *Characteristics*, which include interface descriptions and dynamic capabilities like delay or available bandwidth. On the other hand, the *Skill* set contains a machine-learned behaviour model. Each service is equipped with an agent that monitors its input and output data streams to create this model. By applying hyperplane-based learning techniques, like Support Vector Machines, in combination with Ensemble Learning [137] on the input and output data streams of a service, a comparable model of the behaviour of the service is learned. The resulting *Skill* set is then passed to the distributed registry, which is formed by the distributed knowledge base (*Knowledge Group*) developed in this thesis and presented in Section 4.2.1. In order to separate the concerns of monitoring the services and managing the knowledge, a *Knowledge Group* is a separate MAS.

To ease the search for service replacements, the *Skill* set is enriched with semantic information. Additionally, non-monotonic reasoning is applied to select the best fitting service replacement. ASP is best suited to semantically annotated the *Skills*, which is discussed in detail in Chapter 5. The application of the solver Clingo provides optimisation statements that can be used to select the fitting service replacements. Furthermore, the incorporation of External Statements enables the modelling of dynamically adaptable *Skill* sets.

The following sections introduce the Self-Organising Multi-Agent Knowledge Base. Section 4.2.1 and 4.2.2 present the MAS, its agents, and the representation of the stored knowledge. The knowledge base is discussed using the SOA scenario shown in Chapter 4, but it is not limited to this domain. Afterwards, Section 4.2.3 explains the query resolution.

4.2.1 Multi-Agent System

Multi-Agent Systems (MAS) are ideally suited for dynamic environments like Fog Computing or Search & Rescue scenarios due to their decentralised architecture and ability to execute tasks independently. Therefore, a MAS-based approach is chosen to manage the presented dynamic and decentralised knowledge base (*Knowledge Group*). Each agent of the MAS follows the MAPE-K principle and contains the components shown in Figure 4.2.

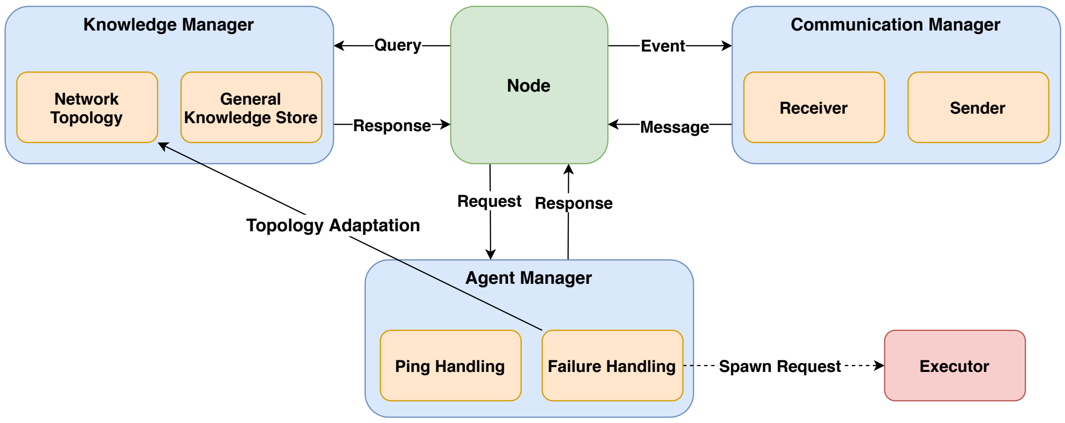


Figure 4.2: Component Diagram of an Agent

The central part of each agent is the *Node* component. It is responsible for the decisions of an agent and controls the remaining components shown in blue and their subcomponents marked in orange. The *Node* component monitors (M) its environment by receiving messages from other agents. Therefore, a chain of responsibility of *Receivers* is provided by the *Communication Manager*. The agent analyses (A) the content of the messages after they have been received by the corresponding *Receivers*. Table 4.1 presents the used message types and briefly describes their usage. A detailed description of each message will be given in the following paragraphs. For example, a *RegistryRequest* is a multicast message used to find a *Registry Node* when a new agent initially enters the system. Furthermore, the Constraint Application Protocol (CoAP) [20] is used to minimise the size of the transmitted messages. For short, CoAP is situated on the Application Layer of the OSI model. It uses UDP to transport messages. To enable confirmable messages, CoAP messages can be equipped with a unique id enabling the detection of duplicates or missing parts of a multi-part message.

Message	Type	Description
RegistryRequest	Multicast	Initial Request for a *Registry Node*
Election	Multicast	Starts the election of the first *Registry Node*
Post	Unicast	Exchange of contact information
PostLevel	Unicast	Response containing the required tree level
Register	Unicast	Registration of a *Registry Leaf/Node* at a *Registry Node*
Acknowledge	Unicast	Acknowledge for a previous message, contains identical ID
Ping	Unicast	Periodic availability check
ASPInform	Unicast	Store knowledge
ASPRequest	Unicast	Forwarding of an ASP query
ASPResponse	Unicast	Answer to an ASP query
SpawnRequest	Unicast	Request to spawn a replacement agents
KnowledgeGroupCheck	Multicast	Checks a network for an existing *Knowledge Group*

Table 4.1: Message Types

After an agent has received a message, it plans (P) suitable actions based on the content of the message. During this process, two primary purposes of messages are distinguished. They are either related to the managed knowledge of an agent or used to coordinate with other agents. In the case of knowledge related messages like *ASPRequests*, the contents are forwarded to the *Knowledge Manager* (K). The knowledge manager incorporates two instances of the ASP solver Clingo to separate knowledge about the *Network Topology* from the *General Knowledge Store*, which contains semantically annotated pieces of information and knowledge (*Knowledge Items*). A detailed description of the mechanism to store knowledge and its representation in ASP is given in Section 4.2.2. Messages related to the maintenance of the connection to neighbouring agents are handled by the *Agent Manager*. For example, *Ping* messages are sent regularly to connected agents to check their availability. Missing *Pings* indicate that an agent is no longer available and will result in an adaption of the *Network Topology*. Since the missing agent could be an essential part of the organisation of the MAS, a replacement for the failed agent can be requested from the *Executor*. It is an additional process used to create further agents. Finally, agents execute (E) their planned actions by sending messages via the *Sender* interface provided by the *Communication Manager*.

To improve the performance of the *Knowledge Group*, agents form a hierarchy. This enables the *Knowledge Group* to effectively forward queries to the corresponding agents, which will be discussed in detail in Section 4.2.3. This hierarchy of agents is shown in the purple part of Figure 4.3. The green part depicts NIO-CMS presented by Jahl in [72].

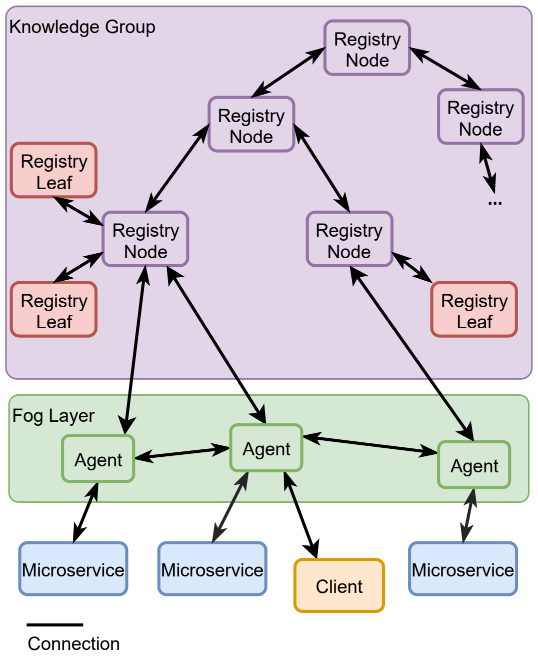


Figure 4.3: Structure of the Distributed Knowledge Base [76]

Inside this hierarchy, agents act in two roles, which are *Registry Leaves* and *Registry Nodes*. *Registry Leaves* are responsible for the storage as well as the maintenance of general knowledge and resolve queries given to the system. To foster the usage of the *Knowledge Group* by heterogeneous participants, ASP is selected as the common knowledge representation language since it has already proven its suitability, as discussed in the related work and will be shown in Chapter 5. Especially the definition of defaults and the non-monotonic reasoning capabilities enable the creation of a dynamic knowledge base. For short, knowledge like the characteristics or skills of a service are formulated as External Statements and enriched with semantics that supports semantic-based queries. A detailed description of the stored knowledge of a *Registry Leaf* is given in Section 4.2.2.

While the *Registry Leaves* concentrate on managing and providing knowledge, *Registry Nodes* focus on maintaining the structure of the *Knowledge Group* and form its backbone. To efficiently manage the *Knowledge Group*, an adapted version of the B*-tree [11] is applied to enable a binary search in $O(log\ n)$, where n is the number of agents. In a B*-tree, each node contains a sorted list of child nodes with a fixed size. To maintain the fixed number of child nodes, new levels are introduced if further nodes are added to the tree. Analogously, branches of a B*-tree are merged if nodes are no longer available. Besides the management of the structure of the *Knowledge Group*, *Registry Nodes* distribute received semantically annotated data, information, and knowledge to the corresponding *Registry Leaves*. Therefore, ASP-based taxonomies are applied. To prevent a translation between different knowledge representation languages, the incorporated taxonomies are formulated in ASP and will be presented in Section 5.2. Generally speaking, *Registry Nodes* aggregate taxonomy branches and forward semantic queries using these branches to the fitting *Registry Leaves*. A detailed description of the semantically enriched query routing is given in Chapter 6.

To initialise a *Knowledge Group*, at least two agents are needed to separate the concerns of administering the new *Knowledge Group* and managing the stored knowledge. Hence, one agent is necessary that acquires the *Registry Leaf* role and one agent that acts as a *Registry Node*. Since these agents can be located on different devices, a mechanism to reach a consensus about the roles is needed. A suitable approach is to rely on the election of a leader, which subsequently will serve as a *Registry Node*. The election procedure is shown in Algorithm 4.1. It is based on the Bully Algorithm [28] and adapts it for asynchronous environments by introducing a request limit and timeout.

Algorithm 4.1: Leader Election

```
   Input  : Request limit $r_l$, Timeout $t$
   Output: Leader $l$
 1 R := $\emptyset$
 2 for int $i := 0; i < r_l; i{+}{+}$ do
 3     Send RegistryRequest
 4     R := R $\cup$ collectRequests($t$)
 5     if $R \neq \emptyset$ then break
 6 if $R == \emptyset$ then return self
 7 $l$ := self
 8 foreach Request $r \in R$ do
 9     if $l.uuid < r.sender.uuid$ then
10         $l$ := r.sender
11 return $l$
```

The initialisation of the Leader Election is shown in the Lines 1 to 5. To start the election, an agent sends a *RegistryRequest* via a predefined multicast address to other agents and waits for the arrival of *RegistryRequests* of other agents until a configurable timeout T is reached. If no requests arrive, the procedure is repeated up to R_l times since new agents could have been introduced to the system during the previous waiting time. In the case that no requests arrive at all, the agent proclaims itself as the winner and serves as the first *Registry Node*. Otherwise, a leader has to be determined. Therefore, agents are equipped with a UUID that supports the definition of a total order. The participating agents evaluate their received *RegistryRequests* and select a leader based on the highest UUID, including their own UUID (Line 7 to 11). After the Leader Election has finished, two results are possible. First, another agent is selected as the leader. In this case, the agent simply waits for a *Post* message containing the contact information of the elected *Registry Node*. Second, the agent itself is elected as the first *Registry Node*. Thus, it sends a *Post* message containing its contact information to the participants of the election to confirm its role as the initial *Registry Node*. In the case that a conflict arises, the Post message of the agent with the lower UUID is overruled by the message of the higher one. The remaining agents will act as *Registry Leaves* and have to register themselves at the newly elected *Registry Node*. Therefore, they send a *Register* message to the *Registry Node*. If the *Registry Node* is still able to manage additional *Registry Nodes*, it adds the received contact information to its *Network Topology* and acknowledges the registration. Upon receiving the acknowledgement, *Registry Leaves* include the *Registry Node* into their *Network Topology*. Nevertheless, the registration of a *Registry Leaf* can be rejected if the *Registry Node* is not able to manage additional *Registry Leaves*. For example, this could be decided based on a configurable number, the size of the managed *Network Topology*, or limitations given by the device the *Registry Node* is running on, like available RAM or CPU usage. Finally, if no agent is available that can act as a *Registry Leaf*, a *Spawn* message is sent by the elected *Registry Nodes* to the *Executor*. Subsequently, it creates a new agent, which will act as a *Registry Leaf*.

To expand the *Knowledge Group*, additional *Registry Nodes* have to be introduced. As already discussed above, a *Knowledge Group* organises its agents based on an adapted B*-tree. Hence, additional levels are introduced to the tree if existing levels are full. The application of Algorithm 4.2 conducts the expansion of a Knowledge Group.

Algorithm 4.2: Knowledge Group Expansion

```
Input  : Knowledge Group $KG$, Agent $ag$, Connection limit $c_l$, Timeout $t$
Output: Expanded Knowledge Group $KG_e$
1 $ag$.sendRegistyRequest()
2 foreach Registry Node $r_n \in KG$ do
3     if $r_n.connections.size < c_l$ then
4         $r_n$.sendPostLevel($r_n$.neededLevel)
5 P := $ag$.collectPostLevel($t$)
6 Registry Node $R_{n_w}$ := $ag$.selectLowestPostLevel(P).sender
7 $ag$.registerAt($R_{n_w}$)
```

This algorithm receives an existing *Knowledge Group* KG, a new agent ag, a connection limit c_l, and a timeout t as input. After its execution, the expanded *Knowledge Group* KG_e is returned. Generally speaking, Algorithm 4.2 is used to find a place for a new agent inside an existing *Knowledge Group*. Furthermore, the following assumptions hold by default but can be adapted by the user:

A1 The connection limit c_l is set to 4 and a timeout of 1 s is assumed.

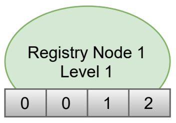
A2 A *Registry Leaf* is only connected to a single *Registry Node*.

A3 A *Registry Node* has 1 connection to a higher level in the B*-tree, 1 on the same level to provide redundancy, and $c_l - 2$ connections to a lower level.

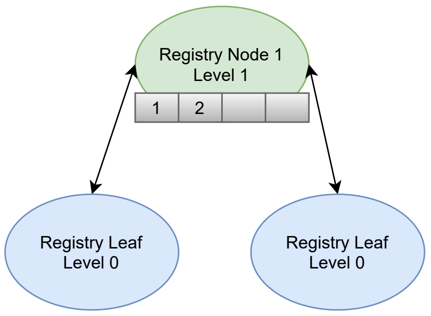
A4 *Registry Nodes* ignore requests by new agents if their connection limit is reached.

As shown in Figure 4.4a, *Registry Node 1* (*RN1*) is not connected to any other agent. Thus, according to the assumptions, *RN1* can establish a connection to two *Leaf Leaves*, which is indicated by zeros, one *Registry Node* on the same level (1), and one *Registry Node* on a higher level (2). In the case that a new agent *ag* is introduced to the system, it sends a *Registry Request* via multicast. Since *RN1* can still register child nodes, it responds with a *PostLevel* message containing the lowest required number of its connection list. This message is received by *ag* and evaluated after timeout t. In this example, the only received level is 0. Hence, *ag* registers itself at *RN1* as a *Registry Leaf*. Analogously, a third agent is introduced to the system, resulting in the constellation shown in Figure 4.4b. After the successful registration of the third agent as a *Registry Leaf*, *RN1* is not allowed to manage additional agents with a lower level according to the assumptions **A1** and **A3**. The next agent which enters the system receives a *PostLevel* message containing the level 1 and, hence, registers itself as *Registry Node 2* (*RN2*) at *RN1*. This is indicated in Figure 4.4c. Since the newly introduced *Registry Node* has two available spots with the level 0, two new *Registry Leaves* can be managed by the system. The introduction of these results in the constellation of agents as shown in Figure 4.4c. Furthermore, *RN1* and *RN2* only have an open connection to an upper level (**A3**). Since there are two *Registry Nodes* that require a further agent at level 2, the next agent receives two *PostLevel* messages with the same level. Thus, it registers itself as a new *Registry Node* at *RN1* and *RN2* and creates a new level in the *Knowledge Group*. The resulting tree is shown in Figure 4.4d. As indicated by a dashed line, the connection between *RN1* and *RN2* is no longer necessary to organise the tree. However, the connection is still maintained in the *Network Topology* of both agents but not used to actively exchange messages to keep the structure of the *Knowledge Group* tree-like. Additionally, it provides a fallback mechanism. In the case *Registry Node 3* (*RN3*) fails, *RN1* and *RN2* can still rely on the dashed connection and thus prevent a division of the network.

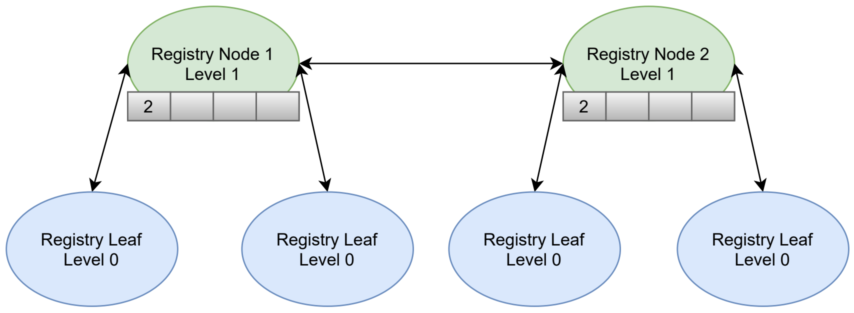
Besides the creation and organisation of the tree-like structure of a *Knowledge Group*, failure handling is an essential part of a *Knowledge Group* to ensure its applicability in highly dynamic domains like SOAs or Search & Rescue scenarios. Therefore, each *Registry Node* and *Registry Leaf* of a *Knowledge Group* actively checks if its connected neighbours are still reachable. Therefore, each agent in a *Knowledge Group* periodically sends *Ping* messages to its connected agents. For example, in Figure 4.4d, *RN3* checks the availability of *RN1* and *RN2*. Since these checks are conducted regularly, the messages have to be small to lower the necessary bandwidth. Therefore, they only contain the UUID of the sender, the UUID of



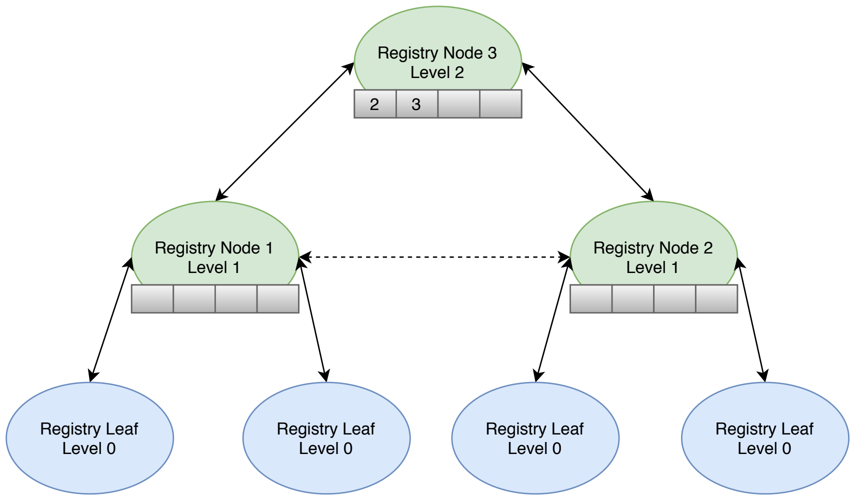
(a) Single *Registry Node*



(b) *Registry Node* with *Registry Leaves*



(c) Introduction of a second *Registry Node*



(d) Introduction of a new level to the *Knowledge Group*

Figure 4.4: Expansion of a *Knowledge Group*

the receiver, a unique message ID used for the acknowledgement, and an optional payload field. The payload can be used to distribute information that has to be updated periodically, for example, to exchange Semantic Routing Entries used for the creation of Semantic Routing Tables as presented in Chapter 6. On the one hand, this slightly increases the message size; on the other hand, the total number of messages is reduced since no additional message is needed for regular updates.

If an agent receives no *Ping* message from a connected agent during a predefined timeout (for example, 5 s) it assumes that the connected agent is no longer reachable. Therefore, the *Ping Handling* shown in Figure 4.2 maintains a count of missing pings for each connected agent. In the case that one of these counts exceeds a predefined threshold (e. g., three pings), the *Failure Handling* is triggered, which determines the required actions to limit the impact of a failing agent. Four major cases are distinguished.

The first case is a failed *Registry Leaf*. Since it is on the lowest level of the *Knowledge Group*, the overall structure of the *Knowledge Group* is not affected. However, a failed *Registry Leaf* will result in the loss of the knowledge stored in its *Knowledge Store*. Therefore, each *Registry Leaves* protocols received *ASPInform* messages and stores them persistently on the device they are running on. In the case of a failing *Registry Leaf*, the Executor tries to create a new *Registry Leaf* on the corresponding device, which then is able to recreate the lost *General Knowledge Store*. Nevertheless, this requires the device of the failed *Registry Leaf* to be still reachable. If this is not the case, the contents of its *General Knowledge Store* are lost until the device is reachable again. To tackle this issue, redundant storage can be introduced. This could be achieved by introducing a second *Registry Leaf* that mirrors the corresponding *General Knowledge Store*. This, however, would double the amount of managed *Registry Leaves* and increase the necessary messages. A second option is to store received *ASPInform* messages redundantly on the level 1 *Registry Nodes* of the network which ensures that each piece of knowledge is at least stored twice. Furthermore, the level 1 *Registry Nodes* do not manage a large number of *Registry Leaves*. The size of the persistently stored history should not limit the applicability of the system. Additionally, a redundancy level can be introduced to the *ASPInform* messages. On the lowest level, the messages are not stored persistently. On the next level, they are stored persistently at the corresponding *Registry Leaf*, and, finally, on the highest level, the messages are stored on the level 1 *Registry Node* of the *Registry Leaf*.

The second case is a failed *Registry Node* with level 1, as shown in Figure 4.4b. Thereby, the network may be divided into several parts, which mostly consist of single *Registry Leaves*. Hence, a new *Registry Node* has to be determined among the remaining *Registry Leaves*, which is achieved by starting the leader election presented in Algorithm 4.1. As a result, a new *Registry Node* is elected, which replaces the failed one. Additionally, a new *Registry Leaf* is requested from the *Executor*. To prevent a loss of knowledge, the newly elected *Registry Note* transfers its *General*

Knowledge Store to the new *Registry Leaf*. Subsequently, this *Registry Leaf* recreates the lost *General Knowledge Store* by applying the stored *ASPInform* messages.

The third case is a failing *Registry Node* at an intermediate level of the *Knowledge Group*, which range from 2 to $maxLevel - 1$. In most cases, a failure of a *Registry Node* on these levels will not result in a separation of the *Knowledge Group* since there are still redundant connections, as indicated in Figure 4.4d. In the case that *Registry Nodes* detect a failed neighbouring agent, they re-enable the corresponding redundant connections to prevent a division of the *Knowledge Group* until a replacement for the failed agent has been introduced to the *Knowledge Group*. To prevent the creation of several replacements, only the *Registry Node* directly above the failed node is allowed to request a replacement from the *Executor*. This can be easily determined by each *Registry Node* that detects the failure by incorporating the knowledge stored in the *Network Topology*. If the failed *Registry Node* had a higher level, the affected agents on the lower level would wait for a replacement. *Registry Nodes* on the same level can check if there is a *Registry Node* available on the next level. If this is the case, they wait for the higher level to request a replacement. If this is not the case, the *Registry Node* on the same level is allowed to request a replacement. After the replacement has been requested, a new agent is spawned by the *Executor*, which then follows Algorithm 4.2 to integrate itself at the place in the tree of the failed agent. Furthermore, no knowledge is lost since the knowledge in the *Network Topology* is restored when the replacement *Registry Node* is integrated into the tree.

The last case is a failure on the highest level of the *Knowledge Group*. In this case, a replacement is not necessary since the *Knowledge Group* can rely on the redundant connections of the second-highest level. This results in a constellation similar to Figure 4.4c. Furthermore, the next agent that enters the *Knowledge Group* will automatically be assigned to the highest level according to Algorithm 4.2.

To summarise, this section has presented the tree-like organisation of a *Knowledge Group*. This includes the incorporation of agents in two distinct roles, which either manage the knowledge or maintain the structure of the *Knowledge Group*. Furthermore, agents are able to integrate themselves into an existing *Knowledge Group* and can handle failures autonomously. Therefore, they rely on their stored knowledge about the *Network Topology*, which is discussed alongside the *General Knowledge Store* in the next section.

4.2.2 Knowledge Management

A central aspect of a *Knowledge Group* is its capability of storing and managing knowledge. Since ASP has already proven its capabilities to represent dynamically changeable knowledge as discussed in the related work, it is employed as knowledge representation and reasoning formalism again. Furthermore, the application of ASP for the stored knowledge of a *Knowledge Group* enables the use of ASP-based ontologies (see Section 5.2) and the detection of semantic inconsistencies (see Section 5.3) without the need to translate between different knowledge representation formalisms. On the one hand, this increases the system performance since no time-consuming translations are needed. On the other hand, it reduces the risk of knowledge loss during the translation process.

To separate concerns, the managed knowledge of a *Knowledge Group* is divided into two scopes: the *Network Topology* that keeps track of the connections between the agent of a *Knowledge Group* and the *General Knowledge Store*, which manages external knowledge. The following paragraphs present ASP templates that are used to model the knowledge of the corresponding scopes.

Network Topology

The *Network Topology* is used to store the knowledge about the connections each agent has inside its *Knowledge Group*, which varies depending on the active connections of an agent. The following listings introduce templates for the different agent roles. Listing 4.1 shows the ASP template for a *Registry Node*.

```
1 registry(uuid("RegistryUUID"), ip("RegistryIP"),
    port("RegistryPort"), location("localhost"))
    :- not -registry(uuid("RegistryUUID"), ip("RegistryIP"),
    port("RegistryPort"), location("localhost")).
2 #external -registry(uuid("RegistryUUID"), ip("RegistryIP"),
    port("RegistryPort"), location("localhost")).
3
4 level(uuid("RegistryUUID"), level("RegistryLevel"),
    timeStamp("TimeStamp")) :- not -registry(uuid("RegistryUUID"),
    ip("RegistryIP"), port("RegistryPort"), location("localhost")),
    not -level(uuid("RegistryUUID"), level("RegistryLevel"),
    timeStamp("TimeStamp")).
5 #external -level(uuid("RegistryUUIDD"), level("RegistryLevel"),
    timeStamp("TimeStamp")).
6 ...
```

Listing 4.1: ASP Representation of a *Registry Node* in the *NetworkTopology*

Line 1 and 2 of Listing 4.1 introduce the definition of a *Registry Node*. It is identified by a universally unique identifier (UUID), is reachable via a specific IP address and port, and is located on the local system (localhost). The body of the rule in Line 1 depends negatively on the External Statement in Line 2. Thus, the head of this rule can be derived as long as the corresponding External Statement is false, which is initially the case. In order to remove the *Registry Node* from the *Network Topology*, the corresponding External Statement has to be set to true. As introduced in Section 4.2.1, each *Registry Node* is annotated with its level in the tree-like organisation of the *Knowledge Group*. The knowledge about the level of a *Registry Node* is modelled like the templates in the Lines 4 and 5. Line 4 states the level of a *Registry Node* by providing its UUID, the actual level, and a timestamp, which is used to determine the current level. Again, Line 4 depends negatively on the External Statement in Line 2. Thus, the knowledge about the level of a *Registry Node* is removed when the External Statement is set to true. Line 5 introduces an additional External Statement, which is analogously used to invalidate the current level.

The representation of a *Registry Leaf* is shown in Listing 4.2 and works analogously to the representation of a *Registry Node*. The only difference in the Lines 1 to 6 is the usage of the *leaf* predicate instead of the *registry* predicate. Besides the use of a different predicate name, *Registry Leaves* have *topics* that are used to categorise and classify managed knowledge. A template for the definition of a *topic* is given in the Lines 7 and 8 of Listing 4.2. Since the *topics* of a *Registry Leaf* are subject to changes, they utilise a combination of a rule that depends negatively on a special External Statement, too. To add further topics for a *Registry Leaf*, additional rules and External Statements have to be added to the *Network Topology*.

```
1 leaf(uuid("LeafUUID"), ip("LeafIP"), port("LeafPort"),
    location("localhost")) :- not -leaf(uuid("LeafUUID"),
    ip("LeafIP"), port("LeafPort"), location("localhost")).
2 #external -leaf(uuid("LeafUUID"), ip("LeafIP"), port("LeafPort"),
    location("localhost")).
3
4 level(uuid("LeafUUID"), level("LeafLevel"), timeStamp("TimeStamp"))
    :- not -leaf(uuid("LeafUUID"), ip("LeafIP"), port("LeafPort"),
    location("localhost")), not -level(uuid("LeafUUID"),
    level("LeafLevel"), timeStamp("TimeStamp")).
5 #external -level(uuid("LeafUUID"), level("LeafLevel"),
    timeStamp("TimeStamp")).
6 ...
7 topic(uuid("LeafUUID"), topic("LeafTopic"))
    :- not -leaf(uuid("LeafUUID"), ip("LeafIP"), port("LeafPort"),
    location("localhost")), not -topic(uuid("LeafUUID"),
    topic("LeafTopic"), timeStamp("TimeStamp")).
8 #external -topic(uuid("LeafUUID"), topic("LeafTopic"),
    timeStamp("TimeStamp")).
9 ...
```

Listing 4.2: ASP Representation of a *Registry Leaf* in the *NetworkTopology*

The last type of rules that are used to represent additional *Registry Nodes* located in remote networks to which a *Knowledge Group* is connected. A detailed description of the handling of multiple *Knowledge Groups* distributed on several networks is given in Section 4.3. A combination of a rule and two External Statements is employed to enable dynamic adaptations of these *Registry Nodes*. In contrast to

Registry Nodes in the local network, they can be seen as representatives of remote *Knowledge Groups*. To emphasis this circumstance, the predicate *networkRegistry* is used instead of the *registry* predicate used in Listing 4.1. Line 1 of Listing 4.3 presents a template for a remote *Registry Node*.

```
1 networkRegistry(uuid("NetworkUUID"), ip("NetworkIP"),
    port("NetworkPort"), location("NetworkAddress"))
    :- not -networkRegistry(uuid("NetworkUUID"), ip("NetworkIP"),
    port("NetworkPort"), location("NetworkAddress")),
    interface(ip("NetworkAddress")).
2 #external interface(ip("NetworkAddress")).
3 #external -networkRegistry(uuid("NetworkUUID"), ip("NetworkIP"),
    port("NetworkPort"), location("NetworkAddress")).
```

Listing 4.3: ASP Representation of a Remote *Registry Node* in the *NetworkTopology*

```
//Registry Node 1
registry(uuid("Registry1"),ip("10.0.0.1"),port("5678"),
   location("localhost"))
level(uuid("Registry1"),level("1"),timeStamp("0"))

leaf(uuid("Leaf1"),ip("10.0.0.1"),port("1112"),location("localhost"))
level(uuid("Leaf1"),level("0"),timeStamp("1"))
topic(uuid("Leaf1"),topic("person"))

leaf(uuid("Leaf2"),ip("10.0.0.1"),port("1314"),location("localhost"))
level(uuid("Leaf2"),level("0"), timeStamp("1"))
topic(uuid("Leaf2"),topic("patient"))

registry(uuid("Registry3"),ip("10.0.0.1"),port("1234"),
   location("localhost"))
level(uuid("Registry3"),level("1"),timeStamp("3"))

//Registry Node 3
registry(uuid("Registry3"),ip("10.0.0.1"),port("1234"),
   location("localhost"))
level(uuid("Registry3"),level("2"),timeStamp("0"))

registry(uuid("Registry1"),ip("10.0.0.1"),port("5678"),
   location("localhost"))
level(uuid("Registry1"),level("1"),timeStamp("1"))

registry(uuid("Registry2"),ip("10.0.0.1"),port("9101"),
   location("localhost"))
level(uuid("Registry2"),level("1"),timeStamp("1"))

networkRegistry(uuid("Network2"),ip("10.11.0.1"),port("9876"),
   location("10.11.0.1"))
interface(ip("10.11.0.1"))
```

Listing 4.4: Answer Sets of Registry Node 1 and 3 of Figure 4.4d

To provide a better understanding of the rules presented in the Listings 4.1 to 4.3, let us consider the Answer Sets that represent their *Network Topology*. The Answer Set for *Registry Node 1* (*RN1*) is shown in Lines 1 to 14 in Listing 4.4. Each segment of lines represents an agent, which is connected to RN1. Line 2 and 3 represent knowledge about *RN1* itself. Its UUID is *Registry1*, it is reachable via the IP address *10.0.0.1* at port *5678*, and is located on the local computer. Furthermore, its current level in the *Knowledge Group* is *1*, which was decided a timestamp *0*. According to the example given in Figure 4.4d, RN1 is connected to two *Registry Leaves*, which are represented by the predicates shown in the Lines 5 to 11. *Leaf1* is located on the local computer with the IP address *10.0.0.1* at port *1112*. It was added at timestamp *1* and had a level *0*. Additionally, it stores *Knowledge Items* (*kItem*) that are annotated with the topic *person*. Further topics are added by introducing additional *topic* predicates. The predicates for *Leaf2* are interpreted analogously.

The Answer Set representing the *Network Topology* of *Registry Node 3* (*RN3*) is shown in the Lines 16 to 27 of Listing 4.4. Again, the Answer Set contains the knowledge about the agent itself and the registries it is connected to (*Registry 1* and *Registry 2*). In contrast to *RN1*, *RN3* is the highest agent in the *Knowledge Group*. Thus, it responsible for maintaining connections to other *Knowledge Groups*. The representation of another *Knowledge Group* is shown in the Lines 26 and 27. They state that *Network2* is located on the computer with the IP address *10.11.0.1* at port *9876*, which is reachable via the interface *10.11.0.1*.

General Knowledge Store

The second part of the *Knowledge Management* is the *General Knowledge Store*. To provide a distinct separation of concerns, it stores all *Knowledge Items* provided by *ASPInform* messages and does not contain any knowledge regarding the structure of the *Knowledge Group*. Generally speaking, the *General Knowledge Store* manages arbitrary knowledge, which is encapsulated in ASP predicates to provide semantic information. Therefore, a suitable representation in ASP is needed. The following paragraphs discuss several alternatives. During this discussion, the *Knowledge Items* shown in Listing 4.5 are used. They contain knowledge about three persons during an emergency, e. g., an earthquake in a Smart City, as mentioned in Section 1.2.2. The knowledge about *Alice* is defined in Lines 1 to 5 of Listing 4.5. She is *36* years old, her location is known, and she was not injured (*healthy*). Bob is *72* years old and his location is known. In contrast to *Alice*, *Bob* has been injured in the earthquake, indicated by the health status *injured*. Furthermore, the kind of his injury is known (*broken leg*). Last but not least, *Charlie* is 20 years old, but there is no current information about her location or health status.

To model knowledge in the *General Knowledge Store*, a suitable ASP representation is needed. The first way to model *Knowledge Items* (*kItems*) in the *General Knowledge Store* is the use of a single External Statement as shown in Listing 4.6.

```
person:
    name: Alice
    age: 36
    location: 51°18’41.2"N 9°28’26.5"E
    healthStatus: "healthy"

person:
    name: Bob
    age: 72
    location: 51°19’03.0"N 9°27’31.3"E
    healthStatus: "injured"
    injury: "broken leg"

person:
    name: Charlie
    age: 20
    location: "unknown"
    healthStatus: "unknown"
```

Listing 4.5: Example Knowledge Items Encoded in YAML

```
#external kItem(topic("human","person"),
  content("person: name: Alice age: 36
  location: 51°18’41.2"N 9°28’26.5"E healthStatus: healthy"),
  format("text/YAML"), timestamp(1610356452)).
```

Listing 4.6: *kItem* Modelled as an External Statement

One of the benefits of modelling *kItems* as single External Statements is that no additional rules are needed, and thus, the knowledge is represented with the least possible overhead. Each *kItem* consists of a semantic annotation (*topic*) based on the ontologies presented in Section 5.2, the stored *content*, its format, and a timestamp that enables determining the most recent *kItems*. Generally, a topic is a sequence of comma-separated concepts that form a branch of an ontology or taxonomy. Additionally, the leftmost concept is the most generic and the rightmost concept the most distinct one. The format is a string, which indicates the type of the *kItem*. It adheres to the Multipurpose Internet Mail Extensions Type (MIME-Type) [45, 46] and is divided into a type (*text* in Listing 4.6) and subtype (*YAML* in Listing 4.6). Furthermore, the type *ASP* expresses that the stored knowledge is represented by an ASP program itself. However, simply using External Statements has a major drawback. While concepts in an ontology are typically short statements, the content can have an arbitrary length, which is only limited by the available memory (see Section 7.1). Since the External Statement has to be set to true in order to add the *kItem* to the *General Knowledge Store*, the content of the knowledge base has to be parsed by Clingo during the adding procedure, and each time the truth value of the External Statement is changed. Especially for large *kItems*, this will result in an increased runtime (see Section 7.1). Hence, this way of modelling *kItems* is unsuited for contents with arbitrary size.

The second way of modelling *kItems* is incorporating a combination of a rule and an External Statement. This kind of modelling preserves the advantages of External Statements since *kItems* can be dynamically removed by changing their truth value. Nevertheless, the overhead of using an External Statement and a rule is higher in comparison to the modelling strategy presented in Listing 4.6. Listings 4.7 and 4.8 show examples of this way of modelling *kItems*. Generally speaking, both methods introduce a UUID to the External Statement, which links it to the corresponding *kItem* via a rule. This prevents that the contents of the *kItem* have to be parsed several times. Furthermore, this supports adding further rules to expand a *kItem*. The major difference between the modelling methods presented in the Listings 4.7 and 4.8 is the sign of the employed External Statement. In Listing 4.7, a negative External Statement is created in Line 1. Informally speaking, it states that there is no *kItem* with the topic *"human","person"* with UUID *123*, which has arrived at timestamp *1610363038*. Line 2 models the actual *kItem*, which is stored in the rule head that depends negatively on the corresponding External Statement. A benefit of using negative External Statements in combination with default negation (see Section 2.4) is that the External Statement does not have to be set to true to derive the rule head. Hence, the overall runtime for adding a *kItem* to the knowledge base is reduced to parsing and a single solving step. However, this kind of modelling is semantically counterintuitive. Informally speaking, the *kItem* holding the knowledge about *Charlie* is derived since it is unknown that the corresponding External Statement is true. Hence, arbitrary pieces of knowledge are derived from the absence of information.

To overcome this semantic issue, a positive External Statement is used, as shown in Listing 4.8. The rule in Line 2 of this listing positively depends on the External Statement presented in Line 1. Since the External Statement has to be set to true, the runtime slightly increases. However, this method of modelling a *kItem* is semantically intuitive since the *kItem* is derived as long as the corresponding External Statement holds. Additionally, *kItems* now have a justification instead of being derived from the unknown.

```
1 #external
    -kItem(topic("human","person"),uuid("123"),timestamp(1610363038)).
2 kItem(topic("human","person"), content("person: name: Charlie age:
    20 location: unkown healthStatus: unkown"), format("text/YAML"),
    uuid("123"), timestamp(1610363038))
    :- not -kItem(topic("human","person"), uuid("123"),
    timestamp(1610363038)).
```

Listing 4.7: *kItem* Modelled as a Negative External Statement and a Rule

```
#external kItem(topic("human","person","patient"),uuid("456"),
   timestamp(1610367448)).
kItem(topic("human","person","patient"), content("person: name: Bob
   age: 72 location: 51°19’03.0"N 9°27’31.3"E healthStatus: injured
   injury:   broken leg"), format("text/YAML"), uuid("456"),
   timestamp(1610367448))
   :- kItem(topic("human","person","patient"), uuid("456"),
   timestamp(1610367448)).
```

Listing 4.8: *kItem* Modelled as a Positive External Statement and a Rule

One major drawback shared by the modelling strategies presented above is the representation of the *content* as a string of arbitrary size. As already discussed, the string of arbitrary length has to be parsed by Clingo while adding it to its knowledge base. On the one hand, this has a significant influence on the run time (see Section 7.1). On the other hand, it enforces the storage of all knowledge inside the main memory and thus influencing the performance of the system the *Knowledge Group* is running on. To tackle this problem, the arbitrarily sized string in the *content* is replaced by a fixed size identifier, for example, by a UUID or the hash of the *content* string. Furthermore, a local Key-Value Store is used, which utilises the hash or the UUID as key and the original *content* string as value. This combination has two advantages. Such stores support the serialisation of knowledge, provide persistence, and prevent a loss of knowledge if a node fails. First, the representation of a *kItem* is no longer dominated by the length of the *content*. Second, the load can be shifted from the primary storage to the secondary storage if the system uses a persistently stored Key-Value Store instead of an in-memory one. Hence, this combination is a suitable approach to manage and represent knowledge in the *General Knowledge Store*.

Besides selecting the most suitable modelling strategy, determining the predicate name for *kItems* is an essential step. The straightforward approach is selecting `kItem` as the predicate name, as shown in the listings above. The predicate in the rule represents a *kItem* and, thus, it should be named accordingly. Nevertheless, choosing `kItem` as the predicate name negatively affects the filtering for *kItems* encapsulating specific topics. While the *External Statements* (arity 3) and *kItems* (arity 4) can still be distinguished by their arity, all *kItems* have to be considered when querying for specific topics. Hence, selecting `kItem` as the predicate name can be compared to creating a database solely containing a single table, which stores all available knowledge. Therefore, another name has to be chosen. Since each *kItem* is annotated with a topic, parts of the topic string can be used as the predicate name. Two parts of the topic are especially suited, the most generic concept (leftmost concept) and the most distinct concept (rightmost concept).

Selecting the most generic concept as the predicate name, e. g., `human` (see Listing 4.8), has the advantage that a part of the knowledge base can be easily accessed since the less generic *kItems* are summarised under the same predicate name. Nevertheless, selecting the most generic concept has several drawbacks. For example, let us consider a knowledge base used in a hospital only containing *kItems* dealing with knowledge about humans like patients, nurses, and doctors. Selecting the most generic concept as the predicate name results in the identical behaviour as selecting `kItem` since no filtering is possible. Furthermore, to select all patients, several queries have to be conducted.

In contrast to selecting the most generic concept, choosing the most distinct concept as the predicate name enables efficient filtering. Typically, this kind of query is more common than selecting the most generic concept. For example, rather than determining all humans in a hospital, a doctor is typically interested in all patients, which can easily be queried by filtering the knowledge base for *kItems* with the predicate name patient. Furthermore, selecting the most distinct concept provides the most detailed semantic description. For example, if *Bob* (see Listing 4.5) is categorised as a patient, it can be derived that he is injured in some way, and that he probably needs help. If *Bob* was only classified as a *human*, this derivation of additional commonsense knowledge would not be easily possible. However, a disadvantage is present when querying the most generic concept. Selecting all humans requires several query steps, which include the remaining concepts provided by the topic string and the application of the ontologies that will be presented in Section 5.2.

Comparing the presented strategies for modelling *kItems*, a positive External Statement using a hash or a UUID in the *content* is the best fitting solution. The piece of knowledge that it holds has to be parsed only once. Additionally, it prevents the storage of the complete knowledge inside the primary storage, thus not limiting the system. Selecting the most distinct concept as the predicate name is favoured since they provide the most efficient filtering. Finally, this results in *kItems* as shown in Listing 4.9.

```
#external conceptₙ(topic("concept₁","concept₂",...,"conceptₙ"),
   content("hash/uuid"), format("format"), timestamp(ts)).
```

Listing 4.9: Final Modelling Scheme of a *kItem*

4.2.3 Query Resolution

After the representation of knowledge in the *General Knowledge Store* has been discussed in detail in the previous section, this section introduces the query mechanism needed to extract stored *Knowledge Items* (*kItems*). It consists of the general forwarding of queries from *Registry Nodes* to *Registry Leaves* as described by us in [76].

In this case, a *Knowledge Group* is deployed in a service-oriented scenario to manage knowledge of the characteristics of services. Hence it can be considered as a distributed service registry. For short, a client interacts with a chessboard-like game service. During the interaction, the chessboard-like game service does no longer respond and the client has to find a replacement. Therefore, a service registry that contains possible alternatives is needed. While static environments can rely on central service registries, dynamic environments like Fog Computing scenarios demand a decentralised approach. Therefore, Jahl presents in [72] a multi-agent-based change management system, which monitors the interaction between a client and a service. By using hyperplane-based machine learning algorithms, agents in the change management system learn the behaviour of their corresponding service employing the monitored input and output data. The resulting behaviour models are then stored in a *Knowledge Group* alongside service characteristics like response delay, available bandwidth, and interface descriptions. To provide an overview of the query resolution, the workflow of queries is depicted in Figure 4.5.

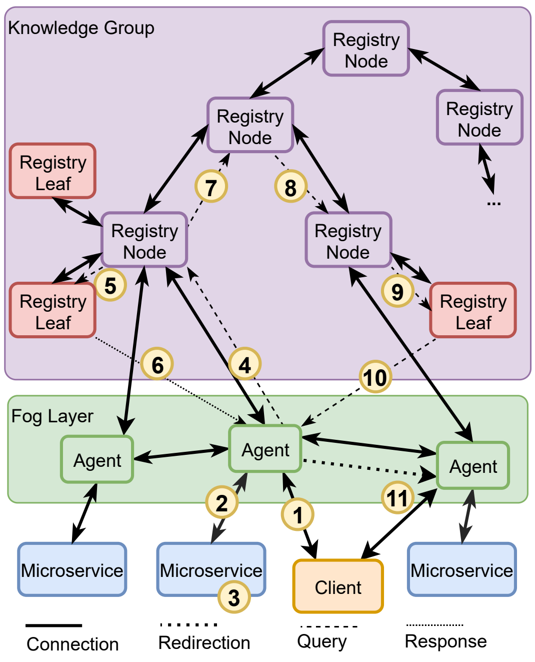


Figure 4.5: Query Workflow for a Service Replacement [76]

During its normal operation, a client in this system interacts with an agent in the Fog Layer (1) that monitors the received input data and forwards them to its managed microservice (2). Subsequently, the received input is processed and the resulting output is returned to the client. In this case, no interaction with the *Knowledge Group* is needed. A failure of the microservice (3) can be either detected if it is no longer reachable or by comparing its output with the learned behaviour model. If the latter one is considered as an outlier, a service failure can be assumed.

In both cases, the corresponding agent queries a *Registry Node* in the *Knowledge Group* for a service replacement ④. This query includes the behaviour model of the failed service, its characteristics, an ASP program to determine the optimal replacement, and a semantic classification of the failed microservice if it is available. For example, let us consider that the failed service was semantically annotated with the classification *{"service","game","chessboard_like"}*. A semantic description like this is not necessary, but it reduces the overall query runtime since it decreases the possible replacement candidates. Once the query has arrived at the corresponding *Registry Node* ⑤, two cases have to be distinguished.

In the first case, the managed *Registry Leaves* of the *Registry Node* are able to answer the query. This can be checked by comparing the stored service descriptions with the received query. If the query contains a semantic description, it is used to filter the stored service descriptions and, thus, reduces the considered replacements to semantically appropriate replacements. For example, by providing the semantic classification *{"service","game","chessboard_like"}* solely chessboard-like game services are considered. If there is no semantic annotation, each stored service has to be compared with the information provided by the query. Therefore, the corresponding *kItems* are selected from the *Registry Leaves*, their content is extracted, and a service replacement is determined. Generally speaking, the contents of the *kItems* are extracted, a new Clingo instance is created, and the query solution is determined based on the provided ASP program. A detailed description of this process is given in a later paragraph. After successfully determining a service replacement, the *Registry Leaf* forwards the solution to the querying agent ⑥. The agent extracts this information and forwards its client to the corresponding replacement ⑪.

In the second case, the managed *Registry Leaves* of the *Registry Node*, which is queried first, are unable to answer the query ⑤. Therefore, the query is forwarded to a higher level of the *Knowledge Group* ⑦, if present. Since the next *Registry Node* is an inner node of the *Knowledge Group*, it has no attached *Registry Leaves* but manages further *Registry Nodes* on the lower level. Therefore, it forwards the received query to them ⑧. Again, providing a semantic classification in the query is beneficial since it reduces the number of *Registry Nodes*, which have to be considered. After a *Registry Node* on the lower level has been selected, the *Registry Leaf* determines a solution if it can answer the query ⑨. Analogously to the case in the previous paragraph, the *Registry Leaf* forwards the results to the querying agent ⑩, which refers the client to the replacement ⑪. Finally, the last *Registry Leaf* will inform the agent if there is no possible replacement.

After a detailed description of the query forwarding process in the previous paragraphs, query resolution is discussed in the following passages. Therefore, let us consider the following scenario: In Figure 4.5, a client interacts with a chessboard-like game service via an agent in the Fog Layer. After some time, the service is no longer reachable (see ③). Subsequently, its corresponding agent queries the *Knowledge Group* for a replacement, which can be provided by one *Registry Leaf* managed by the first *Registry Node* (see ⑤). Listing 4.10 presents fitting *kItems* stored in the *General Knowledge Store* of the *Registry Leaf* mentioned above. They contain knowledge about three different chessboard-like game services. Their location (IP-address, port, ...) are omitted for the sake of clarity.

```
1 chessboard_like(topic("service","game","chessboard_like"),
    content("characteristic("ChineseChess 1.1", bandwidthKB(10000)).
    characteristic("ChineseChess 1.1",delayMS(50))."),
    format("ASP"), timestamp(1610356390)).
2 chessboard_like(topic("service","game","chessboard_like"),
    content("characteristic("StandardChess 1.0.1",
    bandwidthKB(12000)).
    characteristic("StandardChess 1.0.1",delayMS(70))."),
    format("ASP"), timestamp(1609545600)).
3 chessboard_like(topic("service","game","chessboard_like"),
    content("characteristic("Checkers 2.0", delayMS(100))."),
    format("ASP"), timestamp(1608508800)).
```

Listing 4.10: *kItem* Containing Service Characteristics

Line 1 of Listing 4.10 describes the knowledge about the *Chinese Chess* service in version *1.1* as a separate ASP program (*format("ASP")*) encapsulated in the content section of the *kItem*. This ASP program states that the delay of the *Chinese Chess* service is 50 ms and that 10000 kB/s of bandwidth is available. Line 2 defines knowledge about the *Standard Chess* service, which has version *1.0.1* and provides an available bandwidth of 12000 kB/s. Information about the delay of the *Standard Chess* service is not available. Knowledge about the *Checkers* service in version *2.0* is defined in Line 3. It has a delay of 100 ms but does not provide any information on the available delay. After the determination of the service replacements, their contents are extracted. This results in the facts shown in the Lines 1 to 5 of Listing 4.11. Additionally, a choice rule is created, encapsulation the names and versions of each possible replacement (Line 6). Both are added to a new instance of the Clingo solver and form the basis for the optimisation ASP program provided by the received query. The optimisation program is shown in the Lines 8 to 12 of Listing 4.11.

Line 8 and 9 are auxiliary rules, which define the notion of a slow service. A service is considered slow if its delay is higher than 75 ms or if there is no information on the delay. The Lines 10 to 12 define the actual optimisation. The highest priority (@3 in Line 10) is the prevention of slow services. This is the case for *Checkers 2.0* since its delay is higher than 75 ms and, thus, it is considered slow based on Line 8. Subsequently, the delay is minimised by the optimisation statement in Line 9 with the second-highest priority (@2), which removes *Standard Chess 1.0.1* since its delay is higher than the delay of *ChineseChess 1.1*. As the last optimisation (@1), the highest available bandwidth is determined. This results in the following Answer Set, which contains the possible replacement: *{characteristic("ChineseChess 1.1",bandwidthKB(10000)) characteristic("StandardChess 1.0.1",bandwidthKB(12000)) characteristic("ChineseChess 1.1",delayMS(50)) characteristic("StandardChess 1.0.1", delayMS(70)) characteristic("Checkers 2.0", delayMS(100)) characteristic("ChineseChess 1.0.1")}*. The selected replacement is the only characteristic predicate with an arity of *1*. Finally, the *Knowledge Group* returns this result to the agent in the Fog Layer (⑥ in Figure 4.5), which then refers the client to the replacement (⑪ in Figure 4.5).

```
characteristic("ChineseChess 1.1", bandwidthKB(10000)).
characteristic("StandardChess 1.0.1", bandwidthKB(12000)).
characteristic("ChineseChess 1.1", delayMS(50)).
characteristic("StandardChess 1.0.1", delayMS(70)).
characteristic("Checkers 2.0", delayMS(100)).
{characteristic("ChineseChess 1.1"); characteristic("StandardChess
   1.0.1"); characteristic("Checkers 2.0")} = 1.

slow :- characteristic(X), characteristic(X, delayMS(Y)), Y > 75.
slow :- characteristic(X), not characteristic(X, delayMS(_)).
:~ slow. [1@3]
#minimize {Y@2,X : characteristic(X), characteristic(X, delayMS(Y))}.
#maximize {Y@1,X : characteristic(X), characteristic(X,
   bandwidthKB(Y))}.
```

Listing 4.11: ASP Representation of Characteristics and Query Rules [76]

Following the example presented above, arbitrary optimisation queries can be formed. By providing auxiliary rules, unwanted properties can be assigned to objects, which are removed by corresponding weak constraints. Additionally, thresholds can be modelled by utilising the minimize and maximize directives. Thus, the combination of both enables the formalisation of complex queries that are not limited to the service domain.

4.3 Combining Knowledge Groups

So far, we only considered a single *Knowledge Group*. While a single *Knowledge Group* can be used in relatively small Fog Computing or smart home environments, solely relying on a single *Knowledge Group* is impractical in large-scale Service-Oriented Architectures or Search & Rescue missions with unreliable communication. For example, in large-scale Fog Computing scenarios, parts of a single *Knowledge Group* could be distributed on distant parts of the network, which could lead to *Registry Nodes* managing *Registry Leaves* located on different parts of the Fog Computing environment. To tackle this issue, the *Knowledge Group* would have to be reorganised according to changes in the underlying network. However, this introduces a high workload to the *Knowledge Group* and a significant increase in the number of required messages. Furthermore, the increased message load caused by the reorganisation of a large *Knowledge Group* or the distribution of nodes across an unreliable network is especially harmful in Search & Rescue missions. Therefore, creating separate *Knowledge Groups* is beneficial since they can be created independently and located in distant areas of the network. However, this leads to another problem that has to be solved: The exchange of knowledge between *Knowledge Groups* without merging their tree-like organisation scheme.

To enable this exchange, additional protocols are introduced. The first protocol is used to discover further *Knowledge Groups*. A detailed description of this protocol is given in Section 4.3.1. To keep the information about connected *Knowledge Groups* up to date, Section 4.3.2 introduces protocols used to maintain the communication connections. After new *Knowledge Groups* have been discovered, the exchange of knowledge and subsequent queries have to be enabled. The corresponding protocols are presented in Section 4.3.3.

4.3.1 Discovery of *Knowledge Groups*

The general idea of combining *Knowledge Groups* is to gain access to potentially new knowledge. For example, the discovery of new service alternatives in a Fog Computing environment or locations of injured persons in a Search & Rescue scenario. The first step in the combination of several *Knowledge Groups* is the discovery of other *Knowledge Groups*. Therefore, a protocol is needed that enables the *Registry Nodes* with the highest level to scan available network interfaces for other *Knowledge Nodes*. Figure 4.6 presents this protocol. In this case, it is assumed that *Knowledge Group 1* (*KG1*) has sent the first *KnowledgeGroupCheck* message (see Table 4.1 in Section 4.2.1). This message includes its IP address, port, the multicast address used by *KG1*, and a list of available topics.

Before sending a *KnowledgeGroupCheck* in the first step (①), *KG1* checks if its *connection limit* is reached. To prevent the creation of a fully connected mesh network, each *Knowledge Group* has a maximum number of connections to other *Knowledge Groups*, which is configured during its initialisation. Preventing a fully connected mesh network has two main reasons: The first reason is the management overhead since each *Knowledge Group* would have to store the contact information of all other nodes. The second reason is the vast amount of messages needed to maintain the information mentioned above, which is impractical in scenarios with unreliable communication.

In the case that *KG1* is still able to form connections with other *Knowledge Groups*, it sends a *KnowledgeGroupCheck* message for each of its active network interfaces via a predefined multicast address. This message is received by *Knowledge*

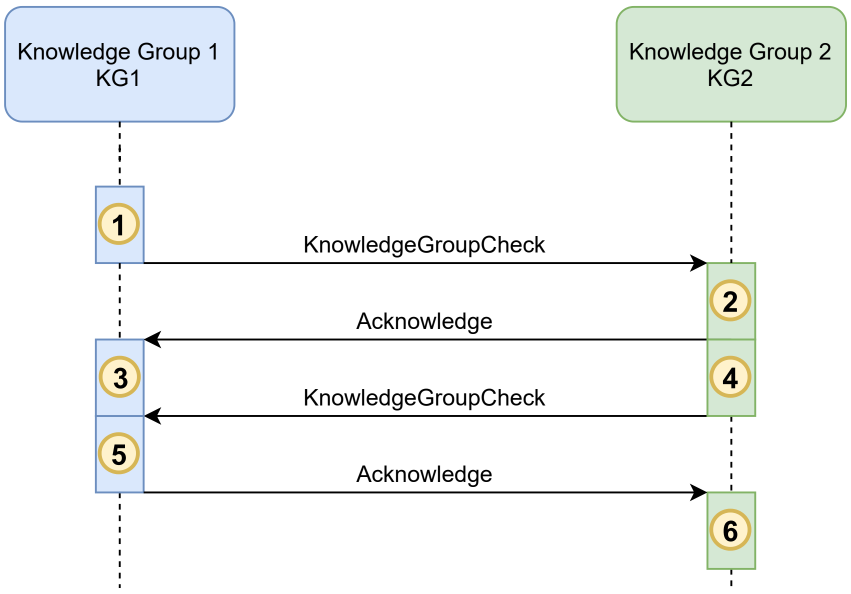


Figure 4.6: *Knowledge Group* Discovery Protocol

Group 2 (*KG2*) in 2. Subsequently, *KG2* checks if its connection limit is reached. If this is not the case, it responds with an *Acknowledge* message. Upon receiving this acknowledgement, *KG1* waits for a *KnowledgeGroupCheck* message from *KG2* (3). Simultaneously, *KG2* generates the ASP representation of *KG1* (see Listing 4.3 in Section 4.2.2), adds the contact information to its *Network Topology*, and sends a *KnowledgeGroupCheck* message containing its own contact information to *KG1* (4). *KG1* acts analogously by adding the necessary information to its *Network Topology*, enabling the exchange of queries with *KG2*, and sending an *Acknowledge* message (5). Finally, *KG2* receives this message and enables the exchange of queries (6).

Summarising the protocol in Figure 4.6, it provides a two-way handshake, exchange of contact information, and enables query forwarding.

4.3.2 Connection Maintenance

In order to keep the information that was exchanged during the discovery protocol up to date, *Knowledge Groups* periodically exchange *Ping* messages. These messages serve two purposes. The first purpose is the validation of the reachability of the connected *Knowledge Groups*. Generally speaking, if *Knowledge Group 1* (*KG1*) regularly receives *Ping* messages from *Knowledge Group 2* (*KG2*), it is able to derive that *KG2* is reachable. The second purpose is the update of relevant information regarding the connected *Knowledge Groups*. Since these messages are exchanged regularly, they are suited to propagate changes in the managed topics of a *Knowledge Group* by including lists of topics that have been included and topics that are no longer provided. A *Knowledge Group* that receives such messages parses these lists and adapts the contact information of connected *Knowledge Groups* accordingly.

4.3.3 Exchange of Knowledge and Queries

After the discovery of *Knowledge Groups* and exchange of *Ping* messages, the last part of the interaction between *Knowledge Groups* is the forwarding and exchange of queries in the form of *ASPRequest* and *ASPResponse* messages. The protocol for query forwarding is shown in Figure 4.7. To demonstrate the protocol of queries, which involve multiple *Knowledge Groups*, let us consider the Search & Rescue scenario presented in Section 1.2.2. In this scenario, a natural disaster has occurred in a Smart City, which damaged buildings, limited communication, and injured people. Subsequently, rescue organisations try to locate the injured people and possible safety hazards like gas leaks. Therefore, they employ UAVs, mobile ground robots, and the smart infrastructure of the city. During this process, two *Knowledge Groups* have been formed. Therefore, three parties are considered. The first is a client application (*CA*) representing the rescuers, who intend to acquire knowledge from a *Knowledge Group*. The second party is *Knowledge Group 1* (*KG1*), which consists of *Registry Node 1* (*RN1*) that manages *Registry Leaf 1* (*RL1*). The third party contains *Knowledge Group 2* (*KG2*). It encompasses *Registry Node 2* (RN2), which manages *Registry Leaf 2* (*RL2*). While *CA* and *KG1* share a network (grey box), *KG2* is situated in another network. Additionally, it is assumed that the *CA* is able to communicate with *KG1*, that *KG1* and *KG2* have established a connection, and that *KG1* and *KG2* have exchanged their managed topics.

In the first step (①), *CA* sends an *ASPRequest* querying all known positions of injured people to *KG1*. In *KG1*, the query is forwarded to the *Registry Node*, which manages the knowledge about injured people. The routing inside a *Knowledge Group* is straightforward. Each *Registry Node* accumulates the topics managed by its child nodes. Hence, queries can be passed down to the *Registry Leaves* containing the corresponding knowledge. In this case, the query is passed to *RN1*, which checks its knowledgebase for the contact information containing the knowledge about injured people (②) and forwards the *ASPRequest* message to *RL1*. *RL1* subsequently extracts fitting *Knowledge Items* from its *General Knowledge Store* (③), encapsulates them in an *ASPResponse* message and sends it to the *CA*, which extracts the received knowledge (④) and distributes it to the rescuers.

The second task of *CA* is to fix gas leaks to prevent further injured people. It sends a second *ASPRequest* in ④ to *KG1*. In this case, *RN1* does not manage any agents that contain corresponding knowledge. Therefore, it checks its *Network Topology* for routing entries (⑤) that indicated the existence of the requested knowledge. The creation and exchange of these routing entries are discussed in detail in Chapter 6. In this example, *KG2* is in possession of the corresponding knowledge. Thus, *KG1* forwards the *ASPRequest* to *KG2* (⑥). Since the forwarding of the query involves multiple *Knowledge Groups* located in different networks or sub-networks, the *ASPRequest* is expanded with a list of visited *Knowledge Groups* to return of the corresponding response successfully. Additionally, a hop count is

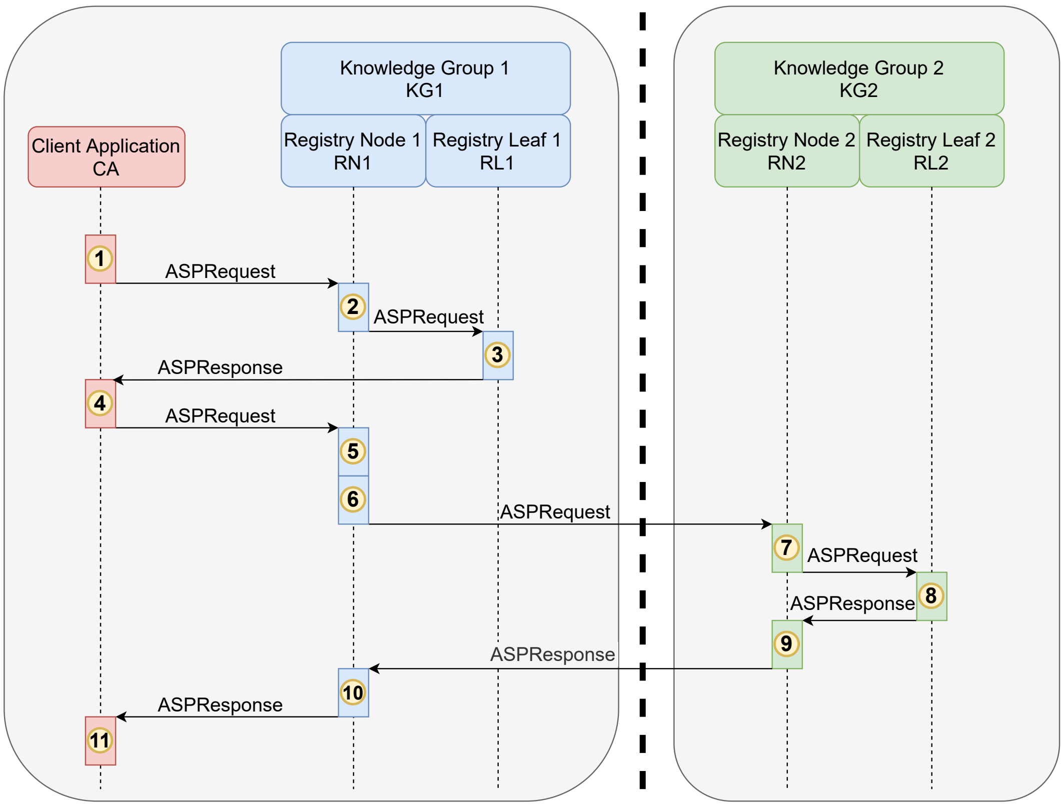


Figure 4.7: ASPRequest Involving Multiple *Knowledge Groups*

included to measure the distance from its origin. Once the *ASPRequest* has arrived at *KG2*, it is forwarded to the corresponding *Registry Leaf* (7) and solved (8). This process is analogous to steps 2 and 3. Since the origin of the *ASPRequest* is located in another *Knowledge Group*, the resulting *ASPResponse* is sent to the highest level *Registry Node*, which doubles the hop count and sends it to the last element of the list of passed *Knowledge Groups*. However, this may not be possible due to a connection loss. In this case, the *Knowledge Group* tries to forward the response to other connected *Knowledge Groups* to find a possible route to the origin. To prevent that the response is exchanged indefinitely between *Knowledge Groups*, the hop count is decremented each time it is forwarded. In the end, the query is deleted if the hop count reached zero. Once the response has reached its destination *Knowledge Group* (10), it is forwarded to the *CA*, which parses the response (11).

4.4 Summary and Discussion

In this chapter, the first main contribution of this thesis has been discussed, which is the distributed management of semantically annotated knowledge in highly dynamic domains with unreliable communication. The foundation of the presented distributed knowledge base is a Multi-Agent System, which is denoted as a *Knowledge Group*. Agents in this system act in two roles, which are *Registry Nodes* and *Registry Leaves*. While *Registry Nodes* manage the tree-like structure of the *Knowledge Group*, general knowledge is maintained by the *Registry Leaves*. Since ASP and Clingo have already proven their suitability as a knowledge representation language, both are chosen to represent the knowledge about the *Network Topology* and the *General Knowledge Store* of the system.

To further improve the scalability of *Knowledge Groups* and their applicability in large scale environments, protocols have been introduced that foster the interaction between different *Knowledge Groups*. This includes the distribution of queries, the exchange of knowledge, and failure mechanisms in the case that *Knowledge Groups* lose their connection to other *Knowledge Groups*.

Comparing the *Knowledge Groups* presented in this chapter with the distributed databases discussed in Section 3.1, the main difference is that the *Knowledge Groups* rely on autonomous entities that manage and store knowledge. Bonifacio et al. present in [17, 18] a Peer-to-Peer architecture that is tailored for distributed knowledge management utilising an XML-like language to store knowledge and to form relations. In contrast to this, *Knowledge Groups* store and manage knowledge using ASP in combination with Clingo. Thus, they provide reasoning support as well as dynamic adaptation of stored knowledge at run-time.

Summing up, this chapter presented a dynamic MAS that fosters the use of semantically annotated knowledge, which is distributed in a loosely coupled network. Thus, it fulfils Requirement **R1 - Handling Dynamic Environments**. Furthermore, it relies on a single formalism to represent the topology of the network and the storage of general knowledge. This enables the use of all relevant knowledge without a need to translate between representations.

Handling Symbolic Commonsense Knowledge 5

A central aspect of human communication is the usage of commonsense knowledge [29]. We rely on it while solving everyday chores or while giving tasks to each other. For example, the simple question "Could you please fetch me a cup?" creates a task that is hardly solvable without commonsense knowledge. To understand this problem, let us consider this task from the point of view of a sophisticated service robot. Since its a multi-purpose service robot, it can be assumed that it has some kind of actuator and behaviour that enables it to fetch objects. The first question that now arises is what kind of object the robot is supposed to fetch. For a human, it is clearly evident what the concept of a cup expresses. It is an object made of ceramic, it has an open side, as well as a closed side, a handle is attached to it, and it is usually used to hold liquids. The second question is, where do you usually find a cup? Typically, a cup could be found on a shelf or in a cupboard which is located in the kitchen or, in the case of an office, on the desks of the employees. To enable the service to solve the task of fetching a cup, it has to have access to such kind of knowledge. It could be provided by its human user or could be extracted from a commonsense knowledge source like ConceptNet 5.

The remainder of this chapter is structured as follows. Section 5.1 elaborates specific requirements for the handling of symbolic commonsense knowledge introduced in this chapter. Section 5.2 presents a technique to model ontologies using ASP and relying on hypergraph-based knowledge sources. The ontologies and taxonomies created in this process are used to annotate the *Knowledge Items* presented in Chapter 4 and enable the aggregation of Semantic Routing Table Entries introduced in Chapter 6. After introducing a commonsense ontology, Section 5.3 introduces a mechanism to prevent semantic inconsistencies in the properties of an object, which will improve the quality of the knowledge base of a robot. Section 5.4 concludes this chapter.

5.1 Specific Requirements

Besides the access to a commonsense knowledge source, a suitable representation of the extracted knowledge is needed. Krötzsch presents in [84] several requirements that are necessary for a suitable knowledge representation. The first one is smart editing, which means that tool support should be present for a knowledge representation language to ease its use. The second requirement is robustness. The knowledge representation language, the resulting knowledge base, and reasoners have to be tolerant to errors and should support the adaptation to the current application field. The third requirement is quantitative computation. Thus, the knowledge representation should provide access to numerical reasoning. The last requirement is modularisation, which means that the knowledge representation language should provide mechanisms to divide programs into reusable subprograms. This leads Krötzsch to the conclusion that a declarative and symbolic knowledge representation is needed, which supports modelling ontologies and provides access to a reasoning formalism.

A declarative and non-monotonic language, which fulfils the requirements defined by Krötzsch, is Answer Set Programming (ASP). It provides negation, numeric constraints, the incorporation of script languages like Python, and methods to model defaults. Further, ASP provides a three-valued logic. Querying an ASP program for a specific predicate can either result in *true* if the predicate is part of the Answer Set, *false* if the strongly negated predicate is part of the Answer Set or *unknown* otherwise. The state-of-the-art ASP solver Clingo introduces, amongst others, two extensions to the ASP syntax, which enable the change of the truth value of a predicate (External Statement) at run-time and the division of ASP programs into smaller and reusable parts (Program Section). The general notion of External Statements is that their truth value is not known and thus can be set manually by the user. ASP and the solver Clingo [48, 50] best meet the proposed requirements, especially in dynamic reasoning [53, 105, 106], modelling of knowledge [39], and the application of commonsense knowledge [8].

To provide a common understanding of the used ASP predicates, an ontology is needed. Typically, this ontology would be developed in OWL [92] or OWL2 [67]. However, OWL in its full specification is undecidable. It is monotonous, which prevents the definition of defaults. ASP, on the other hand, does not have these shortcomings. Hence, it can be used to create an ontology and has already shown its applicability in ontology merging [15]. By formulating ontologies in ASP, the same formalism is used for storing and managing knowledge. Furthermore, it provides a formal representation and naming, categorisation, and relations between the predicates.

In addition to an ontology, a mechanism to handle semantic inconsistencies is needed, especially when commonsense knowledge is incorporated since contradicting properties can be easily introduced to the knowledge base. For example, it is commonsense knowledge that a knife can have the properties sharp and dull. While there is no contradiction between these properties in classical Boolean logic, a semantic inconsistency is introduced to the knowledge base of the robot if a specific knife has both properties at the same time. Therefore, a mechanism is required that prevents such kinds of inconsistencies.

Revisiting Section 1.1, this chapter addresses the Requirements **R2 - Efficient Management of Knowledge** and **R3 - Handling of Semantic Inconsistencies**. Section 5.2 focuses on the fulfilment of Requirement **R2** by presenting a (semi-)automatic and ASP-based ontology generation approach. Section 5.3 aims at the fulfilment of **R3** by introducing methods to formulate ASP rules that prevent semantic inconsistencies in a knowledge base by utilising a commonsense knowledge source.

5.2 ASP-based Commonsense Knowledge Ontology Modelling

Smart environments and smart devices are an increasingly important part of our life. They consist of various entities like services, smart devices, and robots. Currently, these robots are mainly built for a single purpose. These robots, for example, include lawnmowers or vacuum cleaners. However, the research focus in the area of service robotics is set on versatile and multi-purpose robots [30]. To enable the cooperation of these entities, communication based on a shared understanding of knowledge and reasoning about this shared knowledge is needed. In contrast to machine-to-machine communication, the interaction with humans requires the incorporation of commonsense knowledge since humans rely on this kind of knowledge while solving everyday tasks [29]. Thus, robots working with humans should have access to a commonsense knowledge source like ConceptNet 5 (CN5). To incorporate and reason about this commonsense knowledge, a suited formalism is needed.

A typical approach to model such knowledge and to provide a common vocabulary is the use of an ontology. Ontologies provide mechanisms to model sub-class relations, define individuals and their properties, and reason about the modelled knowledge. However, current ontology frameworks do not support a dynamic adaptation of the represented knowledge, are monotonic, do not provide a formalism to express negation, and cannot handle huge amounts of data [96]. These issues demand a new way to model ontologies. Krötzsch suggests in [84] that a declarative and symbolic knowledge representation is needed to overcome these issues. Furthermore, it should not purely be used for representation and additionally should provide a computation paradigm. ASP is particularly suited for this. It provides non-monotonic reasoning capabilities and allows to model default knowledge by applying negation-as-failure and classical negation. It relies on symbolic knowledge representation and supports the application of the closed world assumption.

Besides the selection of a suited representation, several key challenges arise. The first challenge is the vast amount of commonsense knowledge itself. For example, CN5 has several million edges which are almost impossible to handle manually. Hence, an automated approach is needed. Storing a complete commonsense knowledge base could limit its usage to robots with high computational power and would limit its applicability. Therefore, the ontology generation has to be configurable to adapt it to its application scenario. However, an increased configurability can reduce the usability of the ontology generation. The second challenge is the adaptability of the ontology. Since the necessary commonsense knowledge can vary depending on the environment of a robot, parts of the ontology have to be adapted or expanded during run-time.

By relying on the beneficial features of ASP, we propose a novel method for generating a dynamic and non-monotonic ontology in [77]. The resulting ontology is extracted from a hypergraph-based knowledge source. It can be adapted during run-time and does not require a rebuild of the ontology if further knowledge is added. In contrast to the handling of commonsense knowledge presented by Opfer in [104], which focusses on the provisioning and teaching of commonsense knowledge to robots, our focus is set on the creation of dynamic ontologies that enable classification of individuals, the derivation of super- and subclasses, as well as the incorporation of facets.

The remainder of this section introduces the components of this ontology generation method. Therefore, the extraction of a commonsense knowledge ontology from a hypergraph-based database or knowledge source is described in Section 5.2.1. In order to use the extracted knowledge, inference rules are needed. Section 5.2.2 describes the formulation of these rules. Facets enable modelling properties in an ontology. Their definition in ASP is shown in Section 5.2.3. To ease the modelling of the ASP-based ontologies, Section 5.2.4 presents graphical tool that supports a user during the creation of an ontology. Finally, the dynamic interaction with the ontology is shown in Section 5.2.5.

5.2.1 Ontology Generation

The initial step in the generation of a commonsense knowledge ontology is the extraction of edges from a hypergraph-based knowledge source like CN5 and their translation into ASP. Generally speaking, during this process, the initial classes of the ontology and their relations are formed. The general idea of the ontology generation presented by us in [77] is to start from a given root concept and traverse the hypergraph until a set of stopping criteria is met. Algorithm 5.1 shows the automatic process of the hypergraph traversal.

The inputs of Algorithm 5.1 are a root concept c_r, a set of relations that are used for the ontology R_o, a set of relations denoting synonyms R_s, a set of relations expressing properties R_p, and a set of stopping criteria SC. This enables the adaption of the ontology generation to its corresponding application. In Line 1, an adapted breadth-first search (BFS) is applied. The BFS starts at the given root concept c_r and follows edges annotated with at least one relation given in R_o. The BFS stops once no additional edges can be found according to the stopping criteria SC and returns a set of all encountered concepts C and all traversed edges E. In a subsequent step, all synonyms (R_s) for all concepts in C are determined in order

Algorithm 5.1: Ontology Extraction [77]

```
  Input  : Root Concept $c_r$,
           Set of Ontology Relations $R_o$,
           Set of Synonym Relations $R_s$,
           Set of Property Relations $R_p$,
           Set of Stopping Criteria $SC$
  Output: ASP Commonsense Ontology $o_{asp}$
1 $<C, E>$:= adaptedBreadthFirstSearch($c_r$, $R_o$, $SC_o$)
2 $E := E \cup$ getSynonyms($C$, $R_s$, $SC_s$)
3 $E := E \cup$ getProperties($C$, $R_p$, $SCn_p$)
4 $o_{asp}$ := translateEdges($E$)
5 return $o_{asp}$
```

to increase the expressiveness of the resulting commonsense ontology. Again, the stopping criteria are applied to decide if an edge is added to the ontology. If a new concept is reached during this step, it is added to C. After the determination of synonyms, Algorithm 5.1 extracts all properties of all concepts from the commonsense knowledge source. Therefore, edges annotated with relations given in R_p that adhere to the stopping criteria are added to the ontology. For example, to create a simple taxonomy using CN5, the relation `IsA` could be used in R_o, the `Synonym` relation in R_s, the `HasProperty` relation as part of R_p, and an edge weight of `2.0` as the stopping criterion. Selecting such a high edge weight as a stopping criterion will only consider edges with at least two verified sources to be part of the ontology. Thus, a small ontology consisting of trusted and reliable edges is created. In the case of unweighted knowledge sources like WordNet [94], a maximum number of hops can be used as a stopping criterion. Therefore, the edges directly connected to c_r are annotated with the maximum hop number. Subsequently, each layer of edges connected to the previous layer is annotated with a decreased hop count.

After successfully extracting relevant edges in the Lines 1 to 3, the resulting set of edges E is translated into ASP in Line 4 by applying Algorithm 5.2. Its input is a set of edges E and returns an ASP program p_{asp}. In the first step, an empty ASP program p_{asp} is created. Subsequently, each edge e that is part of E is translated into ASP in the Lines 2 to 8. Therefore, a Universally Unique Identifier (UUID) is generated in Line 3 that connects the two ASP rules generated in the following lines. Line 4 is the actual translation of e into an ASP fact. As mentioned in Section 2.5.1, facts annotated with the keyword `#external` are denoted as External Statements and, thus, their truth value can be dynamically altered. By relying on External Statements, the ontology gains flexibility since it is adaptable to its application scenario during run-time. Hence, parts of the ontology can be added or removed depending on the field of application. Additionally, the prefix `cs_` is added to express that the External Statement represents commonsense knowledge.

Algorithm 5.2: translateEdges

```
    Input  : Set of Edges $E$
    Output: ASP Program $p_{asp}$
1  ASP Program $p_{asp}$
2  foreach Edge $e \in E$ do
3      UUID id := getNewID()
4      String edgeExternal := "#external cs_" + $e$.relation
           + "(" + $e$.startConcept + "," + $e$.endConcept + "," + id + ")."
5      String weightRule := "weight(" + id + ","
           + (int)(($e$.weight + $e$.relatedness) * 100) + "," + timeStamp
           + ") : −" + edgeExternal
6      $p_{asp} := p_{asp} \cup$ edgeExternal
7      $p_{asp} := p_{asp} \cup$ weightRule
8  return $p_{asp}$
```

The resulting fact uses the relation of the edge as predicate name, the start concept as the first argument, the end concept as the second argument, and the id as the last argument. Line 5 denotes the weight of the edge. Therefore, a rule is created. This predicate has the name `weight` and uses the generated UUID as the first argument to map the weight to the previously generated External Statement. The second argument is the actual weight. It is the edge weight, and in the case of CN5, the relatedness between both concepts is added. Since ASP can only handle integer values, the weight is multiplied by 100 to include the first two decimal points. In the case of other knowledge sources, any numeric value representing the reliability of an edge can be used. The last argument is a timestamp used to determine the current weight of an edge by selecting the maximum value (see Section 5.2.2). The weight rule in Line 5 uses the External Statement created in Line 4 as a body. Hence, the weight can be removed from the knowledge base if the External Statement is set to false. After generating both rules, they are added to the ASP program. Finally, after all edges in E have been translated, Algorithm 5.2 returns the ASP program.

To demonstrate the functionality of Algorithm 5.1 and Algorithm 5.2, both are applied using the excerpt from CN5 shown in Figure 5.1. The relatedness is omitted in this figure and assumed to be `0` during the translation process, to provide a better overview. The concept `puppy` is selected as the root concept r_c, R_o contains the `IsA` relation, R_s the `Synonym` relation, and R_p the `HasProperty` relation. As a stopping criterion, a minimum weight of `1.5` is assumed for all relations.

By applying Algorithm 5.1, the edges from `puppy` to `immature dog` and from `puppy` to `dog` are selected and added to the set of edges E. Furthermore, the concepts `immature dog` and `dog` are added to the set of concepts C. Since `immature dog` has no outgoing `IsA` edges, the outgoing edges of `dog` are considered next. These edges are all added to E since their weight is higher than `1.5`.

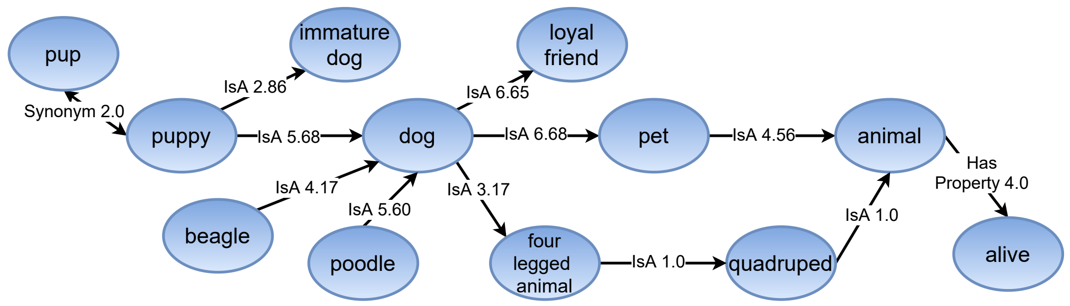


Figure 5.1: Excerpt from the CN5 Hypergraph [77]

Subsequently, the concepts `loyal friend`, `pet`, and `four legged animal` are inserted into C. The remaining outgoing `IsA` edges are `pet` to `animal` and `four legged animal` to `quadruped`. While the first is added to E, the second edge is omitted since its weight is too low. Since no further outgoing `IsA` edges remain, Algorithm 5.1 stops the first step. In the following step, it determines synonyms for all concepts in C. The edge from `puppy` to `pup` is added to E since it has the `synonym` relation and satisfies the minimum weight. Additionally, `pup` is added to C. After the determination of synonyms, properties are collected for all concepts in C. During this process, the edge from `animal` to `alive` is added to E. After the collection of all edges adhering to the defined criteria, they are translated into ASP by applying Algorithm 5.2. Listing 5.1 shows an excerpt from the resulting ASP program.

```
 1 ...
 2 #external cs_hasProperty("animal", "alive", 8).
 3 weight(8, 400, 0) :- cs_hasProperty("animal", "alive", 8).
 4 ...
 5 #external cs_isA("dog", "pet", 136).
 6 weight(136, 668, 0) :- cs_isA("dog", "pet", 136).
 7 ...
 8 #external cs_isA("puppy", "dog", 139).
 9 weight(139, 568, 0) :- cs_isA("puppy", "dog", 139).
10 ...
11 #external cs_synonym("pup", "puppy", 357).
12 weight(357, 288, 0) :- cs_synonym("pup", "puppy", 357).
13 ...
```

Listing 5.1: Excerpt from the Resulting Ontology [77]

Line 2 and 3 of this listing are the translation of the edge between the concepts `animal` and `alive`. Line 2 can be read as follows: an `animal` has the property `alive`. The weight rule of this edge (Line 3) is mapped by the UUID `8` to the External Statement in Line 2. Furthermore, the head of the weight rule expresses that the corresponding edge has a weight of `400` at timestamp `0` and is derived as long as the External Statement in the body is set to true. By setting the External

Statement to false, both the External Statement and the weight are removed from the knowledge base.

After the generation of the ASP program, ASP rules are needed to apply commonsense knowledge to individuals. The rules are presented in the next section.

5.2.2 Ontology Inference Rules

While the External Statements and weight rules presented in the previous section model the commonsense knowledge, a second ASP program with additional rules is needed to reason about this knowledge and apply it to individuals. These inference rules mainly serve two purposes. On the one hand, they are used to reason about commonsense knowledge internally. On the other hand, they are used to create human-readable output predicates. Furthermore, the inference rules are divided into six categories: weight handling, commonsense knowledge propagation, subset determination, classification, facet handling, and Answer Set size reduction. Listing 5.2 shows an excerpt from these inference rules, which contains examples for each category. The complete set of example inference rules is available at Bitbucket[30].

Weight Handling The weight handling category consists of one rule, which is shown in Line 1 of Listing 5.2. As mentioned in Section 5.2.1, each External Statement representing an edge of the hypergraph is connected to a weight rule using a UUID and is annotated with a timestamp. By using this timestamp, the weight handling rule determines the current weight for each UUID. If the ontology has just been generated, the maximum timestamp for all UUIDs is `0`, and the extracted weight is used. However, additional weight rules with the same UUID can be later added to adjust the weight, for example, to achieve broader classification results. In this case, the `#max` operator of Clingo selects the newest timestamp and stores its value in the variable *MaxTimeStamp*. The resulting dynamic weight uses the predicate name `currentWeight`, the UUID, the weight given at *MaxTimeStamp*, and *MaxTimeStamp* as parameters. The weight handling is used internally and thus is not included in the final Answer Set.

Commonsense Knowledge Propagation The second category of rules is used to propagate commonsense knowledge. Lines 3 and 4 of Listing 5.2 show an excerpt from these rules. Line 3 marks the initial propagation of commonsense knowledge. Therefore, the predicate `is` is used to represent an initial classification of an individual, which, for example, can be given by a human user or an image classification algorithm. If the variable `FromConcept` of the initial classification fits to a commonsense knowledge predicate (`cs_` prefix), additional classifications are derived.

[30]Inference Rules, `https://bitbucket.org/sjakob872/arrange/src/master/resources/etc/ontology_rules.lp` Accessed December 29, 2021.

```
 1 currentWeight(UUID, Weight, MaxTimeStamp)
     :- MaxTimeStamp = #max{TimeStamp : weight(UUID, _, TimeStamp)},
     weight(UUID, Weight, MaxTimeStamp).
 2 ...
 3 isA(Individual, ToConcept, Weight) :- is(Individual, FromConcept),
     cs_isA(FromConcept, ToConcept, UUID), currentWeight(UUID, Weight,
     MaxTimeStamp).
 4 isA(FromConcept, ToConcept, WeightB) :- isA(FromConcept,
     InterConcept, WeightA), cs_isA(InterConcept, ToConcept, UUID),
     currentWeight(UUID, WeightB, MaxTimeStamp), WeightA < WeightB.
 5 ...
 6 subSetOfInternal(FromConcept, ToConcept, isA, Weight))
     :- isA(InterConcept, ToConcept, _), isA(InterConcept,
     FromConcept, _), cs_isA(FromConcept, ToConcept, UUID),
     currentWeight(UUID, Weight), MaxTimeStamp).
 7 subSetOf(FromConcept, ToConcept, Relation, MaxWeight) :- MaxWeight =
     #max{ Weight : subSetOfInternal(FromConcept, ToConcept, Relation,
     Weight)}, subSetOfInternal(FromConcept, ToConcept, Relation, _).
 8 ...
 9 classifiedAsInternal(Individual, ToConcept, MaxWeight)
     :- MaxWeight = #max{ Weight : isA(Individual, ToConcept,
     Weight)}, isA(Individual, ToConcept, _), is(Individual, _).
10 classifiedAs(FromConcept, ToConcept, MaxWeight)
     :- MaxWeight = #max{ Weight : classifiedAsInternal(FromConcept,
     ToConcept, Weight)}, classifiedAsInternal(FromConcept,
     ToConcept, _).
11 ...
12 hasFacet(Concept, Facet) :- facetOf(Facet, Property),
     hasProperty(Concept, Property, Weight).
13 facetInheritedFrom(Facet, Concept, ParentConcept) :- facetOf(Facet,
     Property), propertyInheritedFrom(Concept, Property,
     ParentConcept, Weight).
14 ...
15 #show subSetOf/4.
16 #show classifiedAs/3.
17 ...
```

Listing 5.2: Excerpt from the Inference Rules [77]

To provide a better understanding, let us consider the following example. An image classification algorithm receives an image of `rex` and classifies `rex` as a puppy, which results in the predicate `is("rex","puppy")`. The excerpt from the hypergraph in Figure 5.1 contains the commonsense knowledge that a `puppy` is a `dog`, which results in the commonsense predicate `cs_isA("puppy", "dog", 139)`. Given these predicates, the rule in Line 3 derives that `rex` is a `dog`, too. Line 4 uses already derived commonsense knowledge to apply further classifications. This process is stopped if no edge with a higher weight than the current one is found. Including equal or lower weights would result in an exhaustive use of further commonsense knowledge predicates resulting in cyclic or impractical classifications. Again, the resulting predicates are used internally and are not included in the Answer Set.

Subset Determination Lines 6 and 7 show an excerpt from the determination of subsets, which utilises the derived rule heads of the commonsense knowledge propagation. For example, Line 6 determines subset relationships between `isA` predicates and checks if this subset is part of the commonsense knowledge. Considering the example given in Figure 5.1, the commonsense knowledge propagation derives the predicates `isA("rex", "puppy", 1)` and `isA("rex", "dog", 568)`. Since it is commonsense knowledge that a `puppy` is a `dog`, the head of the rule in Line 6 is derived. The resulting `subSetOfInternal( "puppy", "dog", isA, 568)` predicate states that `puppy` is a subset of `dog` and was extracted from an edge with the `isA` relation. Rules like Line 6 are used internally and provide predicates used in Line 7. This rule selects the internal subset predicates between two concepts with the highest weight and represents the subsets in a human- and machine-readable format. Furthermore, the `subSetOf` predicates derived by Line 7 are part of the resulting Answer Set and indicate a chain of subset relations starting from the initial classification.

Classification Lines 9 and 10 present an excerpt from the classification. While the subset rules in the previous paragraph form a chain of subset predicates, the classification rules directly link the individuals with the relevant concepts. Rules, like shown in Line 9, are used to create an internal representation of the classification. Therefore, these rules use the predicates derived by the commonsense knowledge propagation and select the concepts for classification based on their weight. Line 10 summarises the internal classification. Therefore, this rule selects the highest internal classification predicates for each concept and marks them as the final results, which are part of the returned Answer Set.

Facet Handling An excerpt from the rules that handle facets is shown in the Lines 12 and 13. The resulting predicates of this rule category are part of the resulting Answer Set. These rules attach facets to concepts by using their properties (Line 12). Therefore, a facet is added to a property via a `facetOf` predicate. Furthermore, rules like Line 13 enable the inheritance of facets based on the inheritance of properties. In general, a property is inherited to a subclass if its parent class has the corresponding property. A detailed description of facets is given in Section 5.2.3.

Answer Set Size Reduction The last category of inference rules is used to reduce the number of predicates that are returned in the Answer Set. This limits the Answer Set to the most important predicates and prevents that the complete ontology is returned to the user. Lines 15 and 16 are examples of the size reduction. They use the `#show` directive of Clingo, which prompt Clingo to return only atoms with the fitting predicate name and arity. The complete set of `#show` directives includes subsets, classification, facets, properties, and the initial classification.

The application of the presented inference rules on the ontology extracted from Figure 5.1 results in the following Answer Set:

$$\begin{aligned}
M = \{&is(\text{"}rex\text{"}, \text{"}puppy\text{"}),\\
&subSetOf(\text{"}rex\text{"}, \text{"}puppy\text{"}, is, 1),\\
&subSetOf(\text{"}puppy\text{"}, \text{"}immature\_dog\text{"}, isA, 286),\\
&subSetOf(\text{"}puppy\text{"}, \text{"}dog\text{"}, isA, 568),\\
&subSetOf(\text{"}dog\text{"}, \text{"}loyal\_friend\text{"}, isA, 665),\\
&subSetOf(\text{"}dog\text{"}, \text{"}pet\text{"}, isA, 668),\\
&classifiedAs(\text{"}rex\text{"}, \text{"}puppy\text{"}, 568),\\
&classifiedAs(\text{"}rex\text{"}, \text{"}immature\_dog\text{"}, 286),\\
&classifiedAs(\text{"}rex\text{"}, \text{"}dog\text{"}, 568),\\
&classifiedAs(\text{"}rex\text{"}, \text{"}loyal\_friend\text{"}, 665),\\
&classifiedAs(\text{"}rex\text{"}, \text{"}pet\text{"}, 668)\}.
\end{aligned}$$

The Answer Set M can be interpreted as follows. `Rex` has been initially classified as a `puppy`. Thus, it is a subset of this concept. Subsequently, a `puppy` is a subset of both `immature_dog` and `dog`. Finally, a `dog` is a subset of the concepts `loyal_friend` and `pet`. Following these subset predicates, several classifications with different weights are given. For example, `rex` is classified as a `dog`. By selecting the highest weight, the most generic concept is used, which is `pet` in this example.

Following the pattern of the inference rules discussed in the previous sections, a further set of inference rules can be manually created enabling the inclusion of custom relations and custom inheritance structures. After the definition of inference rules, the last step in the generation of a commonsense knowledge ontology is the definition of facets, which provide several rules for a detailed specification of properties. The facet specification is shown in the next section.

5.2.3 Facets

The final step in the generation of a commonsense knowledge ontology is the creation of facets. These include the definition of values, their types and ranges, sub-properties, and domains. This enables the detailed description of concepts and their properties. To demonstrate the definition of facets, an additional edge is introduced to the excerpt of the CN5 hypergraph shown in Figure 5.1. This edge models the commonsense knowledge that a `dog` has the property `coat_colour`. Listing 5.3 demonstrates the construction of a facet for the property `coat_colour` called `colour`. This includes the definition of the facet itself, its type, a value range, and the actual value. Domain and sub-property facets are defined analogously.

```
1 #external facetOf("colour", "coat_colour").
2 #external typeOf("colour", "coat_colour", "string").
3 #external valueRangeOf("colour", "coat_colour",
    "1{black;grey;brown;white;brindle}1").
4 propertyViolation(Individual, "colour", "coat_colour", "Too many
    Values") :- X = #count{Value : hasValue(Individual, "colour",
    "coat_colour", Value, _)}, hasValue(Individual, "colour",
    "coat_colour", _, _), X > 1.
5 propertyViolation(Individual, "colour", "coat_colour", "Too few
    Values") :- X = #count{Value : hasValue(Individual, "colour",
    "coat_colour", Value, _)}, hasValue(Individual, "colour",
    "coat_colour", _, _), X < 1.
6 propertyViolation(Individual, "colour", "coat_colour", "No Value")
    :- not hasValue(Individual, "colour", "coat_colour", _, _),
    is(Individual, _).
7 #external hasValue("rex", "colour", "coat_colour", "brown", 0).
```

Listing 5.3: Facets Defining the Coat Colour of a Dog [77]

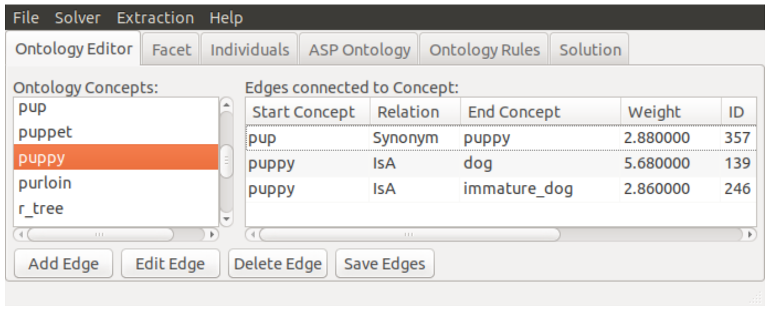
Line 1 of Listing 5.3 is the definition of the facet. It is modelled as an External Statement and, thus, can be activated dynamically. Furthermore, it states that the property `coat_colour` has a facet defining its `colour`. The second line of this listing defines the type of the `colour` facet, which is `string`. The range of the created facet is limited by the External Statement in Line 3. It states that the `colour` has to have at least and at most one value included in the set `{black;grey;brown;white;brindle}`. These restrictions are enforced in two ways. While the restriction of the actual values is conducted by the graphical user interface ARRANGE presented in Section 5.2.4, the minimum and maximum cardinalities are asserted by additional ASP rules. Lines 4 to 6 present these rules. Line 4 checks if the number of defined values for the `colour` for a specific individual is higher than the given maximum cardinality. If this is the case, the comparison $X > 1$ is evaluated to true, and the head of the rule is derived. The resulting predicate then indicates that the given individual has too many values for the facet `colour` of the property `coat_colour`. Line 5 performs the analogous check for the minimum cardinality and signals too few values if the number is lower than the corresponding cardinality. Additionally, the rule in Line 6 checks if at least one value is specified in the case that the minimum cardinality is higher than `0`. Finally, a value is specified in Line 7. It is modelled as an External Statement, and therefore its truth value can be changed dynamically. Informally speaking, this External Statement expresses that the `coat_colour` of `rex` is `brown`.

5.2.4 Graphical User Interface

To support the generation, definition, and adaption of ASP-based ontologies, we provide a graphical user interface denoted as ARRANGE (Answer set pRogRAmming oNtoloGy gEneration) [77] that is publicly available at Bitbucket[31]. ARRANGE supports the automatic extraction of a commonsense ontology from a hypergraph-based knowledge source as presented in Section 5.2.1, shows the ontology inference rules, which are defined in Section 5.2.2, supports the developer while creating facets, and provides access to the resulting Answer Set and solving statistics. Figure 5.2 depicts the graphical user interface ARRANGE.

To manually create, alter, or generate an ASP ontology, the ontology tab shown in Figure 5.2a is used. It is divided into three areas. On the left side, a list of all concepts is given that are part of the ontology. By selecting a concept, all connected edges are shown on the right side. Edges can be added, edited, deleted, and saved using the corresponding buttons. Figure 5.2b shows the facet tab, which is used to model facets. Additional rules like property violation are not shown since they should not be removed or edited manually to ensure the correct indication of errors. Furthermore, individuals can be created in an additional tab. To provide a detailed overview of the ASP ontology, separate tabs present the basic ontology program and the inference rules. The last tab is the solution tab shown in Figure 5.2c. It gives an overview of the resulting Answer Set and provides statistics of the ontology and the solving process. Furthermore, the results are returned as a graph to ease the understanding of the returned Answer Set. The graph representation of the Answer Set shown in Figure 5.2c is presented in Figure 5.3. Blue arrows mark facets. For example, `rex` has the facet `colour`. Black arrows indicate property relations like `hasProperty`. Green arrows denote subsets, e. g., `puppy` is a subset of `dog`. Finally, red arrows present the classification results.

[31]ARRANGE, https://bitbucket.org/sjakob872/arrange/src/master/ Accessed December 29, 2021.

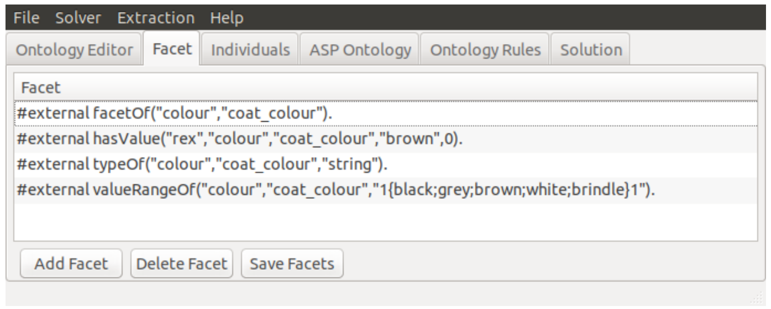


Ontology Concepts:
pup
puppet
puppy
purloin
r_tree

Edges connected to Concept:

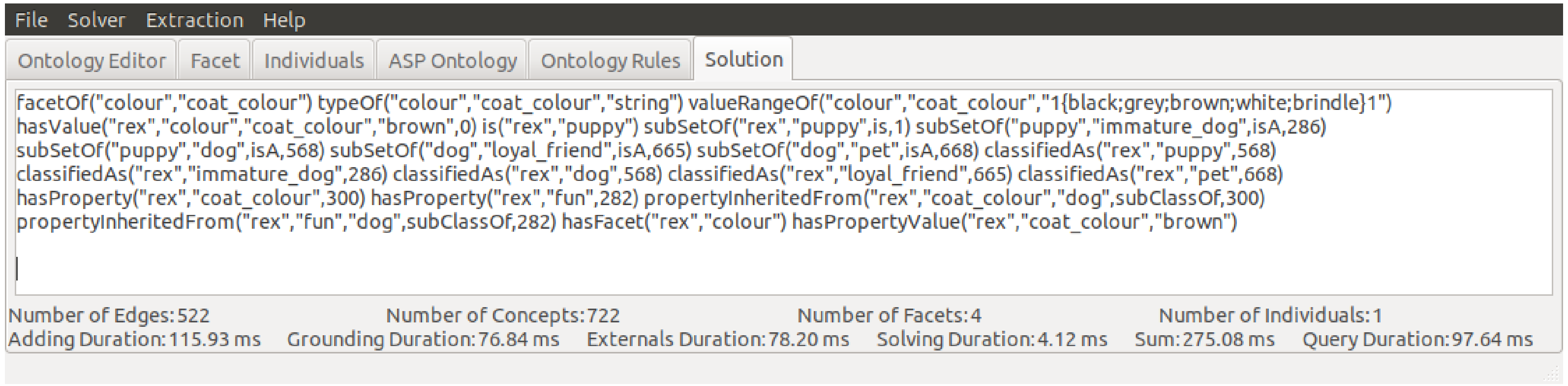
Start Concept	Relation	End Concept	Weight	ID
pup	Synonym	puppy	2.880000	357
puppy	IsA	dog	5.680000	139
puppy	IsA	immature_dog	2.860000	246

(a) Ontology Tab [77]



Facet
```
#external facetOf("colour","coat_colour").
#external hasValue("rex","colour","coat_colour","brown",0).
#external typeOf("colour","coat_colour","string").
#external valueRangeOf("colour","coat_colour","1{black;grey;brown;white;brindle}1").
```

(b) Facet Tab



```
facetOf("colour","coat_colour") typeOf("colour","coat_colour","string") valueRangeOf("colour","coat_colour","1{black;grey;brown;white;brindle}1")
hasValue("rex","colour","coat_colour","brown",0) is("rex","puppy") subSetOf("rex","puppy",is,1) subSetOf("puppy","immature_dog",isA,286)
subSetOf("puppy","dog",isA,568) subSetOf("dog","loyal_friend",isA,665) subSetOf("dog","pet",isA,668) classifiedAs("rex","puppy",568)
classifiedAs("rex","immature_dog",286) classifiedAs("rex","dog",568) classifiedAs("rex","loyal_friend",665) classifiedAs("rex","pet",668)
hasProperty("rex","coat_colour",300) hasProperty("rex","fun",282) propertyInheritedFrom("rex","coat_colour","dog",subClassOf,300)
propertyInheritedFrom("rex","fun","dog",subClassOf,282) hasFacet("rex","colour") hasPropertyValue("rex","coat_colour","brown")
```

```
Number of Edges:522           Number of Concepts:722           Number of Facets:4           Number of Individuals:1
Adding Duration:115.93 ms   Grounding Duration:76.84 ms   Externals Duration:78.20 ms   Solving Duration:4.12 ms   Sum:275.08 ms   Query Duration:97.64 ms
```

(c) Solution and Statistics Tab

Figure 5.2: Graphical User Interface ARRANGE

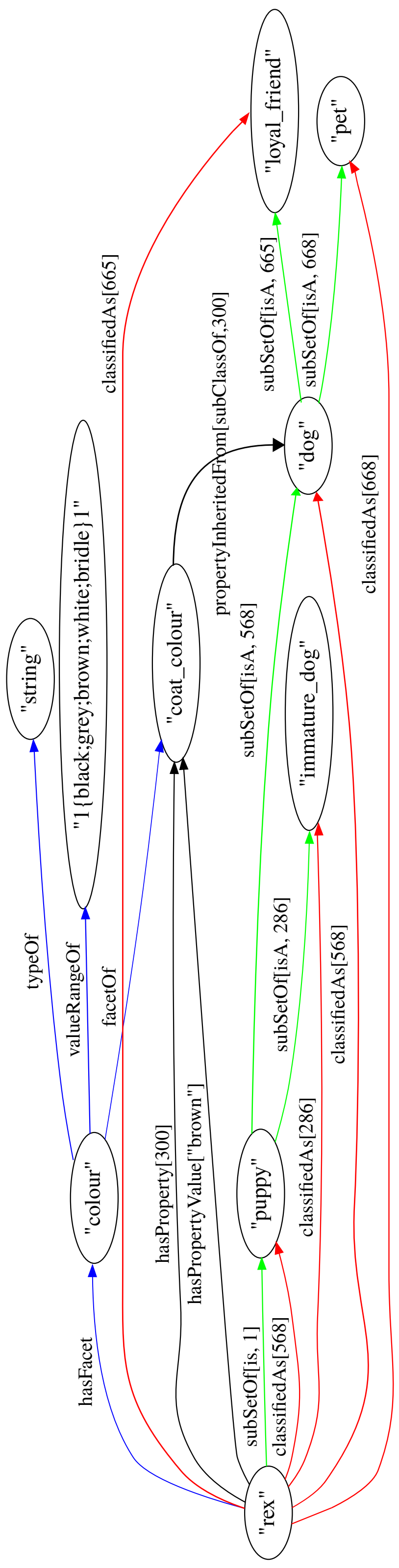


Figure 5.3: Graph Representation of the Resulting Classification [77]

5.2.5 Dynamic Ontology Interaction

The classification provided by the automatically generated ontology already achieves satisfying results without any manual interference. Nevertheless, the classification can miss some classes. Figure 5.4 shows an example. Using the initial classification that `rex` is a `dog`, the application of the ontology extracted from the presented example classifies `rex` as a `pet` and a `four legged animal`. Further classes are not derived. This is caused by the inference rules since they only use edges with a higher weight. By relying on increasing weights, an exhaustive traversal of the hypergraph or loops are prevented since both would lead to impractical or no classification. To add further classes without causing a risk of an exhaustive search, two mechanisms are provided.

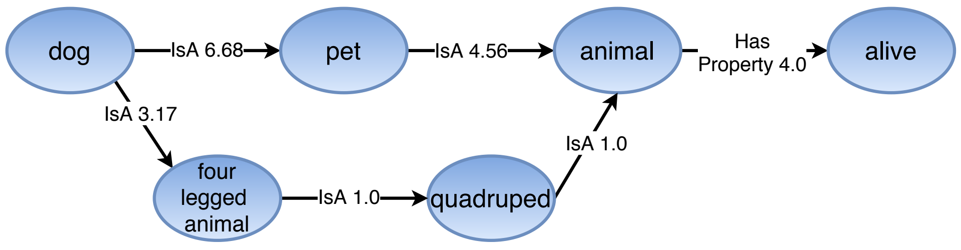


Figure 5.4: Subgraph of CN5

The first mechanism is the adaption of the weights during the design-time of the ontology. After the generation of the ontology, ARRANGE supports the adaption of weights. Thus, the weights of the corresponding edges can be adapted using the ontology tap. For example, by increasing the edge weight of the edge between `pet` and `animal` in Figure 5.4 to `7.0`, the concept `animal` is included in the classification.

The second mechanism is to integrate further weight rules, which can be added during design-time and run-time. An example of the weight adaption of the edge between `pet` and `animal` is shown in Listing 5.4.

```
#external cs_isA("pet", "animal", 19).
weight(19, 456, 0) :- cs_isA("pet", "animal", 19).
weight(19, 700, 1) :- cs_isA("pet", "animal", 19).
...
weight(19, 306, 10) :- cs_isA("pet", "animal", 19).
```

Listing 5.4: Dynamic Weight Adaption

The first two lines of this listing are the translation of an edge with its current weight. In order to adapt the weight of this edge, Line 3 adds an additional weight rule. The rule head contains the UUID of the edge, the adapted weight, and an increased timestamp. Thus, the weight handling shown in Section 5.2.2 selects the weight of Line 3 for the current weight. Considering the example presented above, `rex` is now classified as an `animal` and inherits the property `alive`. In the case the classification is no longer suited, the weight can be lowered by introducing further weight rules (Line 5).

5.3 Handling Semantic Inconsistencies

Commonsense knowledge is used in our everyday life [29]. We utilise this knowledge to solve our daily chores and rely on it during communication. Hence, this kind of knowledge is needed to improve the communication between service robots and humans. Besides the classification of objects by an ontology as presented in Section 5.2, the service robots have to have access to a source of commonsense knowledge, need a suited representation of it, and have to be able to adapt this knowledge according to their environment [78, 104, 107]. Therefore, let us consider the following example: a human states that it is cold. To solve this implicitly formulated task, a service robot could fetch a blanket or turn on the heater. Commonsense knowledge is needed to understand the implicitly formulated task and the connection between the room temperature and the heating.

Depending on the operating place, the service robots have to adapt their knowledge and either extract commonsense knowledge from a knowledge source like CN5 or have to be taught by a human [104]. This process can cause inconsistencies in the knowledge base, which can be divided into syntactic and semantic inconsistencies. Syntactic inconsistencies are literals that are considered to be true and false simultaneously. For example, the literals $\neg X$ and X could be part of the same knowledge base and, thus, only the trivial solution false could be derived. While these syntactic inconsistencies are handled by the used reasoning formalism, e. g., Boolean algebra, semantic inconsistencies are not considered by these formalisms. For example, the predicates `temperature(livingRoom, cold)` and `temperature(livingRoom, hot)`, stating the temperature of a room is `cold` and `hot` at the same time, are syntactically correct but introduce a semantic inconsistency. There is an apparent contradiction for humans. However, robots without any commonsense knowledge are not able to understand this contradiction. Therefore, the knowledge base of the robot has to be able to cope with semantic inconsistencies and needs to prevent them in case further knowledge is introduced [78].

The remainder of this section is structured as follows: Section 5.3.1 presents a method to formalise commonsense knowledge in ASP. Since inconsistencies can arise during the incorporation of commonsense knowledge, a semantic inconsistency detection is shown in Section 5.3.2. Finally, a way to prevent these inconsistencies is presented in Section 5.3.3.

5.3.1 Modelling of Commonsense Knowledge in ASP

In order to incorporate commonsense knowledge, a suitable representation and knowledge source are needed. ASP, with its non-monotonic reasoning capabilities and mechanisms to model defaults, is particularly suited. Furthermore, CN5 is suited as a commonsense knowledge source since its hypergraph based knowledge representation allows fast and easy access to the stored knowledge. Hence, an application of ASP to model commonsense knowledge based on CN5 was introduced by Opfer et al. in [107], which we adapt in [78]. To model commonsense knowledge and to apply it to sensor inputs, Opfer et al. suggest three distinct Program Sections. The first one is the `commonsenseKnowledge` Program Section. It contains facts that model commonsense knowledge extracted from hypergraph-based knowledge sources like CN5 or taught by human users. Furthermore, this kind of knowledge is assumed to hold and rarely changes. Hence, it has been modelled as facts. An excerpt of this Program Section modelling commonsense knowledge about a `cup` is shown in Listing 5.5.

```
1 #program commonsenseKnowledge.
2 cs_AtLocation("coffee", "cup", 200).
3 cs_AtLocation("cup", "shelf", 282).
4 cs_AtLocation("cup", "sink", 100).
5 cs_AtLocation("cup", "table", 400).
6 cs_CapableOf("cup", "hold_liquids", 692).
7 cs_IsA("cup", "drinking_vessel", 100).
8 cs_FormOf("cups", "cup", 200).
```

Listing 5.5: Excerpt of the Commonsesne Knowledge on Cups [107]

Line 1 of Listing 5.5 marks the `commonsenseKnowledge` Program Section. The following lines model the commonsense knowledge about a `cup`. For example, `"coffee"` can be found in a `cup`. A `cup` can be located on a `shelf`, in a `sink`, and on a `table`. Additionally, a `cup` is able to `hold liquids`.

The second Program Section is denoted as `sensorInput`. It contains external input about objects in the environment of the robot. This input can be given by sources like YOLO (You Only Look Once)[32], by the Google Vision API[33], by the classification based on an ontology, or simply by statements of a human user. In contrast to commonsense knowledge, knowledge given in this Program Section often changes, for example, if the information is outdated. Thus, the statements given in this Program Section are marked as External Statements. Listing 5.6 shows an example of this Program Section.

[32]YOLO: Real-Time Object Detection, https://pjreddie.com/darknet/yolo/ Accessed December 29, 2021.
[33]Vision AI, https://cloud.google.com/vision Accessed December 29, 2021.

```
1 #program sensorInput.
2 #external is("blueCup", "cup").
3 #external is("kitchenTable", "table").
4 #external is("kitchenShelf", "shelf").
```

Listing 5.6: Example of the `sensorInput` Program Section [107]

Line 1 of Listing 5.6 marks the beginning of the `sensorInput` Program Section. The following lines are External Statements representing objects detected via image classification. The object `blueCup` is classified as a `cup`, the `kitchenTable` as a `table`, and the `kitchenShelf` as a `shelf`.

To combine the commonsense knowledge with the sensor input, the Program Section `situationalKnowledge` is defined [107], which provides corresponding rules. Listing 5.7 depicts an excerpt from this Program Section.

```
1 #program situationalKnowledge(n, m).
2 #external -atLocation(n, m).
3 atLocation(n, m, W) :- not -atLocation(n, m), is(n, "cup"),
     is(m, "table"), cs_atLocation("cup", "table", W).
4 atLocation(n, m, W) :- not -atLocation(n, m), is(n, "cofee"),
     is(m, "cup"), cs_atLocation("cofee", "cup", W).
5 atLocation(n, m, W) :- not -atLocation(n, m), is(n, "cup"),
     is(m, "shelf"), cs_atLocation("cup", "shelf", W).
6 ...
7 #external -formOf(n, m).
8 formOf(n, m, W) :- not -formOf(n, m), is(n, "cups"), is(m, "cup"),
     cs_formOf("cups", "cup", W).
```

Listing 5.7: Excerpt of the `situationalKnowledge` Program Section [107]

Line 1 of Listing 5.7 is the beginning of the `situationalKnowledge`. In contrast to the previous Program Sections, two variables, `n` and `m`, are added to its definition. During the grounding of the ASP program, these variables are replaced by concrete values, which enable reusing this Program Section for different pairs of objects. By using these objects, External Statements (see Line 2 and 7) are created, which can be used to negate the predicates derived by the corresponding rules. For example, grounding this Program Section with the objects `blueCup` and `kitchenTable`, creates, among others, the External Statement `-atLocation("blueCup", "kitchenTable")`. Setting this External Statement to true expresses that the `blueCup` is not on the `kitchenTable`.

Besides the External Statements, additional rules are part of the `situational-Knowledge` Program Section, which apply commonsense knowledge onto the sensor input. Lines 3 to 5 and Line 8 show such rules. Informally speaking, the rule in Line 3 can be interpreted as follows: if a robot has detected an object `n` which is classified as a `cup`, an object `m` with the class `table`, it is commonsense that a `cup` can be located on a `table`, and it is unknown that `n` is not located at `m`, it can be derived that `n` is possibly located at `m`. Furthermore, the variable `W` indicates the reliability of the derived rule head. For example, the edge weight of CN5 can be used to define the reliability of the derived knowledge.

Each rule in the `situationalKnowledge` Program Section depends on the truth value of three External Statements, two of them representing the sensor input and one that states contrary knowledge. Hence, the predicates derived from the rules can be removed from the knowledge base in two ways. The first one is to change the truth value of the External Statements representing the sensor input if the information is outdated. The second one is that the robot is able to derive contrary knowledge. For example, the robot could check if the `blueCup` is on the `kitchenTable` and if that is not the case, the truth value of the corresponding External Statement is set to true.

After presenting a general way to model commonsense knowledge introduced by Opfer et al. in [104, 107], we aim at detecting and preventing semantic inconsistencies in this knowledge to keep the resulting knowledge base consistent. Both aspects are discussed in the following sections.

5.3.2 Semantic Inconsistency Detection

The first step in the handling of semantic inconsistencies is their detection. In order to apply the presented inconsistency detection method, the used commonsense knowledge source has to be hypergraph-based and should be able to represent properties of a concept. Furthermore, a relation to determine antonyms is needed. A relation defining synonyms can improve the detection of inconsistencies. CN5, for example, fulfils these requirements since it provides corresponding relations. Properties are supported by the `HasProperty` relation, antonyms by the `Antonym` relation, and synonyms by relations like `Synonym` or `SimilarTo`. Hence, the semantic inconsistency detection presented in Algorithm 5.3 can be applied using CN5.

The semantic inconsistency detection receives a concept c as input and returns a set E of edges connected to c. In the case that edges would introduce semantic inconsistencies, they are marked for the semantic inconsistency prevention shown in Section 5.3.3. During the first step, the algorithm extracts all edges connected to c in the set of edges E (Line 1). Subsequently, all properties of c are gathered in set P, which are concepts connected by a property edge to c (Line 2). Lines 3 to 9 show the detection of inconsistencies. To detect inconsistencies, antonym edges are used. Therefore, Algorithm 5.3 checks if any property $p \in P$ is connected to any other property in P via an antonym edge. If this is the case, the properties contradict each other and the corresponding pair of edges can introduce an inconsistency to the knowledge base. Thus, both edges are marked as `inconsistent`.

Algorithm 5.3: Semantic Inconsistency Detection [78]

```
    Input   : Concept $c$
    Output: Set of edges $E$ connected to $c$
1  $E$ := Extract edges connected to $c$
2  $P$ := Gather properties from $E$
3  foreach Property $p \in P$ do
4      if $p$ is an Antonym to any property in $P$
5       then Mark corresponding edge in $E$ as inconsistent
6      $S$ := Collect all Synonyms for $p$
7      foreach Synonym $s \in S$ do
8          if $s$ is an Antonym to any Property in $P$
9           then Mark corresponding edge in $E$ as inconsistent
10 return $E$
```

Only using this step could leave some inconsistencies unmarked because synonyms of properties could cause further semantic inconsistencies. Therefore, synonyms (S) of the considered property are extracted from the utilised knowledge sources. In the case of CN5, the edges with the relations `SimilarTo`, `Synonym`, and `InstanceOf` could be used. Subsequently, the algorithm checks if any of the gathered synonyms is connected via an antonym edge to a property and marks the corresponding edges as possible inconsistencies. The step of collecting synonyms is only applied once. Considering synonyms of synonyms would lead to a traversal of an exponentially growing number of synonyms and would lower the quality of the result of the inconsistency detection. Finally, the set E containing all edges extracted from the knowledge source is returned, which are then translated into ASP. If an edge has been marked as inconsistent, it is translated as described in Section 5.3.3. If it is not marked as an inconsistency, it is translated according to Section 5.3.1.

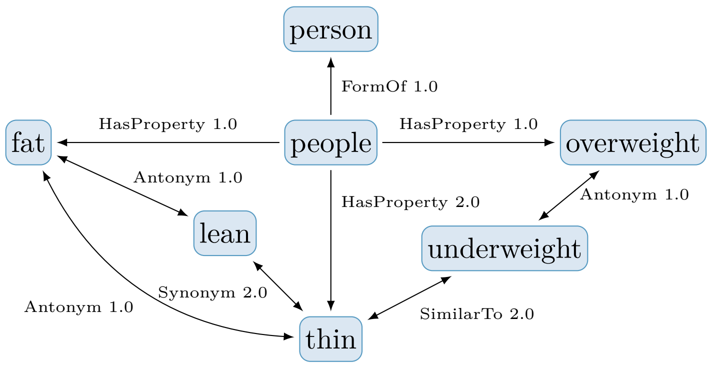


Figure 5.5: Inconsistencies in the Properties of People [78]

In order to understand the steps of Algorithm 5.3, Figure 5.5 presents an example based on CN5, which contains the concept `people` and an excerpt of its properties. The first step of Algorithm 5.3 extracts four edges, which connect people to person, fat, overweight, and thin. Since person is not connected via a HasProperty edge, it is not considered in the following steps. The three remaining edges have the relation HasProperty, and thus the concepts fat, overweight, and thin are considered as properties. In the initial detection of inconsistencies (Lines 4 and 5), the edges between people and fat, as well as people and thin, are marked as inconsistent since an antonym edge connects thin and fat. However, there is no such edge between thin and overweight. In order to detect such inconsistencies, synonyms are extracted. In this example, lean is a synonym for thin and underweight is similar to thin. Using these synonyms, the inconsistency between thin and overweight is found since its an antonym to underweight. The returned set of edges contains four edges. While the edge between person and people does not introduce an inconsistency, the remaining three can cause an inconsistency.

5.3.3 Semantic Inconsistency Prevention

After possible inconsistencies have been marked by Algorithm 5.3 shown in the previous section, these edges have to be translated in a different way than unmarked edges. Therefore, we introduce a variant of the situationalKnowledge Program Section. Instead of applying commonsense knowledge to a pair of objects provided by two variables, n and m, the variant presented in this section is used to represent the properties of a single object n. In general, this Program Section is used to derive properties based on the sensor input presented in Section 5.3.1. Inconsistencies in this input cannot be prevented. Therefore, the rules presented in this section enable deriving consistent properties by introducing the additional predicate hasProperty, which will model consistent properties. Listing 5.8 shows an example of rules that are used to prevent inconsistencies.

```
#program situationalKnowledge(n).
hasProperty(n, "thin", W) :- not is(n, "-thin"),
   cs_HasProperty("people", "thin", W), is(n, "thin"),
   is(n, "people"), not hasProperty(n, "fat", _) :
   cs_Antonym("thin", "fat", _).
-hasProperty(n, "thin", W) :- not is(n, "thin"),
   cs_HasProperty("people", "thin", W), is(n, "-thin"),
   is(n, "people").
hasProperty(n, "fat", W) :- not is(n, "-fat"),
   cs_HasProperty("people", "fat", W), is(n, "fat"),
   is(n, "people"), not hasProperty(n, "thin", _) :
   cs_Antonym("thin", "fat", _).
-hasProperty(n, "fat", W) :- not is(n, "fat"),
   cs_HasProperty("people", "fat", W), is(n, "-fat"),
   is(n, "people").
is(X, "people") :- is(X, "person").
```

Listing 5.8: Modelling of the Inconsisteny Prevention [78]

Line 1 marks the beginning of the adapted `situationalKnowledge` Program Section. The following four rules (Lines 2 to 5) model a pair of edges that has been marked as inconsistent by Algorithm 5.3. In this example, the translation of the edges from `people` to `thin` and from `people` to `fat` are shown. Line 2 derives that the object `n` has the property `thin`. This knowledge can only be derived if the following knowledge is present in the knowledge base:

1. There is no proof that `n` is not `thin`;
2. it is commonsense knowledge that `people` can be `thin`;
3. the sensor input indicates that `n` is `thin`;
4. the sensor input states that `n` is classified as `people`;
5. `n` does not have the property `fat` if it is commonsense knowledge that `thin` and `fat` are antonyms.

Line 3 represents the logical opposite of Line 2 since it states that `n` does not have the property `thin`. To prevent that both syntactically contradicting rule heads can be derived simultaneously, both rules rely on contrary input predicates in the body. For example, the rule in Line 2 negatively depends on `is(n, "-thin")` while it is positively used in Line 3. Hence, it prevents that both rule heads can be derived at the same time. The rules for the property `fat` shown in Line 4 and 5 are used analogously to Line 2 and 3.

To provide a better understanding of the interaction of the rules shown in Listing 5.8, let us consider the following example. The author of this thesis has finally lost some weight. His service robot sees him as a thin person now and adds the sensor input `is("stefan","thin")` and `is("stefan","person")` to its knowledge base. By applying the auxiliary rule in Line 6, the service robot derives the predicate `is("stefan","people")`. Hence, the inconsistency prevention rules in Listing 5.8 are applied. Since there is no proof that `stefan` is not `thin`, the corresponding sensor input is given, it is commonsense knowledge that `people` can have the property `thin`, and no contrary information is given, the robot derives that `stefan` has the property `thin`. At a later time, the service robot receives the input `is("stefan","fat")`. This causes a semantic inconsistency in the input. However, the way the rules in Listing 5.8 are modelled prevents inconsistencies in the `hasProperty` predicates. Instead of creating a single semantically inconsistent Answer Set, two consistent Answer Sets are created that both contain one of the contradicting properties. For this example, the Answer Sets are:

$$M_1 = \{cs\_Antonym(\text{"}thin\text{"}, \text{"}fat\text{"}, 1), is(\text{"}stefan\text{"}, \text{"}thin\text{"}), is(\text{"}stefan\text{"}, \text{"}fat\text{"}), is(\text{"}stefan\text{"}, \text{"}person\text{"}), is(\text{"}stefan\text{"}, \text{"}people\text{"}), cs\_HasProperty(\text{"}people\text{"}, \text{"}thin\text{"}, 2), cs\_HasProperty(\text{"}people\text{"}, \text{"}fat\text{"}, 1), hasProperty(\text{"}stefan\text{"}, \text{"}fat\text{"}, 1)\}$$

$$M_2 = \{cs\_Antonym("thin", "fat", 1), is("stefan", "thin"), is("stefan", "fat"), is("stefan", "person"), is("stefan", "people"), cs\_HasProperty("people", "thin", 2), cs\_HasProperty("people", "fat", 1), hasProperty("stefan", "thin", 2)\}$$

The differences between M_1 and M_2 are highlighted in blue. Since the number of Answer Sets is increased, the robot can detect the possible inconsistency and has three ways to solve it. The first way is to rely on the weights attached to the `hasProperty` predicates. In this example, the property `thin` has a higher weight and the robot could thus remove the input that caused the inconsistency. The second option is to rely on the most current information. In this case, the robot could keep the property `fat` and invalidate the old input `is("stefan","thin")`. The last option is to request the assistance of a human user, who can state which input predicate does not hold.

By combining the modelling of commonsense knowledge shown in Section 5.3.1, the inconsistency detection presented in Section 5.3.2, and the prevention of inconsistencies by creating several consistent Answer Sets in parallel discussed in Section 5.3.3, a dynamically adaptable and consistent commonsense knowledge base can be created.

5.4 Summary and Discussion

In this chapter, one of the main contributions of this thesis has been presented, which is the semantic handling of symbolically represented commonsense knowledge. The introduced handling of symbolic commonsense knowledge is divided into two parts.

The first part is the automatic generation of ASP-based ontologies, which are used to represent commonsense knowledge. ASP is ideal for the representation of ontologies since it adheres to the requirements for a knowledge representation language defined by Krötzsch [84]. ASP and the solver Clingo provide failure robustness since rules can be formulated that catch errors. Furthermore, its three-valued logic provides methods to express that something is unknown, which eases the indication of errors. By providing External Statements and Program Sections, ASP programs can be dynamically adapted fulfilling the modularisation requirement defined by Krötzsch. Additionally, ASP provides mechanisms to apply numeric computation. The last requirement defined by Krötzsch is tool support. While ASP has sophisticated solvers like Clingo, there are hardly any graphical tools that support the modelling of ASP programs. To tackle this issue, we developed an ontology modelling and generation framework that provides a graphical modelling tool supporting the user in the generation, creation, and manual adaption of ontologies. To ensure the reusability of parts of the ontology, as demanded by Krötzsch, the presented ontologies consist of three distinct parts. The first part is the generated ontology, which is modelled using External Statements and weight rules that enable the dynamic adaptation of the ontology at run-time. The second part contains inference rules that enable the classification of individuals. The last part manages facets, which enable the user to provide values, restrictions, domains, and sub-properties for extracted properties. By combining these three parts, a dynamically adaptable ontology has been created, which is supported by a graphical modelling tool and satisfies the requirements defined by Krötzsch. Furthermore, the presented generation of ASP-based ontologies fulfils Requirement **R2** defined in Section 1.1 since the generated ontologies provide an efficient method to manage and access the stored knowledge. Additionally, they provide a common semantic which supports a decentralised application.

Related work in the field of ontology generation aims at the (semi-)automatic extraction of ontologies from various sources. For example, frameworks like the Health Ontology Generator [81] utilise databases and approaches similar to Onto-Harvester [97] focus on free text. Furthermore, these frameworks typically use OWL to generate the ontologies. In contrast, ARRANGE generates ASP-based ontologies, which are dynamically adaptable and expandable at run-time. Frameworks like OntoDLV [118] model ontologies using ASP, too. OntoDLV expands ASP with additional keywords that enable the modelling of classes as well as sub- and superclass relations. In contrast, ARRANGE adheres to the ASP-Core-2 standard [24] and utilises External Statements and Program Sections provided by Clingo.

The second part of the handling of symbolic commonsense knowledge is a mechanism to manage semantic inconsistencies. Commonsense knowledge is a significant part of our everyday life [29]. Thus, the incorporation of commonsense knowledge into the knowledge base of a service robot fosters the interaction of service robots with its human users because both can rely on similar knowledge. However, the incorporation of commonsense knowledge into the knowledge base of a robot can easily introduce semantic inconsistencies, which can cause faulty executions of tasks or misunderstanding in the communication between robots and humans. To prevent this, commonsense knowledge is extracted from a hypergraph-based knowledge source. During this process, extracted properties are checked for possible inconsistencies and ASP rules that prevent the derivation of semantic inconsistencies are generated. Instead, the rules create several consistent solutions that can be selected by the robot or the human interacting with the robot. Thus, in the second part of the handling of symbolic commonsense knowledge, a dynamic formalisation of commonsense knowledge is presented that provides access to a vast amount of knowledge and prevents possible semantic inconsistencies, which fulfils Requirement **R3**.

In related works, ASP has been broadly used for knowledge representation in various fields (see Section 3.3). However, semantic inconsistencies inside the created ASP programs are seldom addressed automatically. One of the works addressing inconsistencies is presented by Gebser et al. in [56]. The focus is set on detecting inconsistencies in large scale biological networks. The approach presented by Gebser et al. can be compared to the semantic inconsistency handling introduced in Section 5.3, which incorporates a commonsense knowledge source to solve inconsistencies automatically. In contrast, Gebser et al. manually formulate rules to detect inconsistencies based on specifically formulated rules.

6 Adaptive Semantic Routing in Dynamic Environments

By introducing several distinct *Knowledge Groups* as presented in Section 4.3, a loosely coupled and mesh-like network is formed. Typically, locating contents in such unstructured networks is achieved by flooding, which simply forwards messages to all neighbouring nodes. Especially in Search & Rescue scenarios, this could introduce unnecessary load to a potentially unstable network and, thus, cause a breakdown of the network. Additionally, entirely relying on flooding in a Fog Computing environment would create additional stress on the Fog Nodes and could cause network congestion. Thus, an efficient and intelligent way of routing based on the content is needed, which only relies on flooding as a last resort. A possible solution is the introduction of a structured peer-to-peer network, e. g., Distributed Hash Tables (DHT) [139]. While DHTs provide fast access to the stored data, they have a major drawback. They distribute content equally on the network. This can potentially lead to a broad distribution of the content, which then could not be reachable in highly dynamic networks. To tackle this problem of content-based routing in such networks, we present in [75] a routing mechanism based on routing tables since they can be easily updated by adding, changing or deleting entries. In their typical use case, routing tables provide information about routes to destinations on the network. However, this is not suitable for scenarios that require content-based queries instead of location-based ones. Therefore, the routing mechanism we present in [75] shifts the focus of routing tables from location-based to semantic-based.

The remainder of this chapter is structured as follows. In Section 6.1, specific requirements for the semantic routing are discussed. The base of the adaptive semantic routing is formed by Semantic Routing Tables, which are presented in Section 6.2. To reduce the overall size of the tables and to utilise the hierarchy provided by the used taxonomies, table entries are semantically aggregated. Section 6.3 discusses this procedure. Updates of the tables are shown in Section 6.4. The propagation of newly discovered individuals is presented in Section 6.5. Subsequently, Section 6.6 provides an example application. Last but not least, Section 6.7 summarises this chapter.

6.1 Specific Requirements

The first challenge in creating Semantic Routing Tables and solving semantic queries is to provide access to knowledge sources that grant semantic information. ConceptNet5 (CN5) [128] is such a knowledge base, which we utilise for the creation of

ASP-based ontologies in Chapter 5. By relying on a subset of the base relations of CN5, such as *IsA* and *FormOf*, a taxonomy can be generated. While ontologies are a typical approach for providing a common understanding of concepts, taxonomies provide a hierarchical representation of subclass relationships and, thus, enable an aggregation of concepts based on their branches. Figure 6.1 shows a comparison between an ontology focussing on the concept of a *human* and a corresponding taxonomy. For example, a branch in the presented taxonomy consists of the concept *rescuer*, which is a *person* and, thus, a *human*.

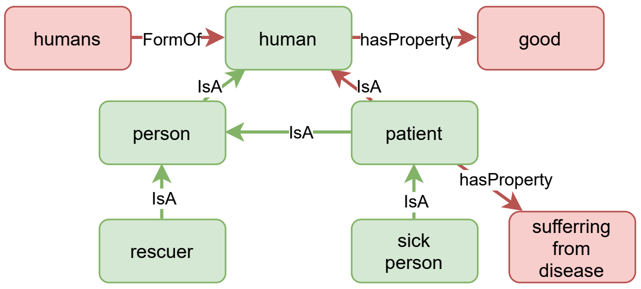


Figure 6.1: Taxonomy Extracted from a Hypergraph [75]

The concepts forming the taxonomy are displayed in green and further concepts introduced by an ontology are highlighted in red. While the ontology forms a mesh-like structure, a tree-like structure is formed by the taxonomy. A downside of mesh-like structures are circles, which could lead to misleading aggregations. Since the tree-like structure eases the aggregation of knowledge based on taxonomy branches and prevents cyclic aggregations they have been chosen as the basis for the Semantic Routing Tables presented by us in [75]. The example scenario, shown in Figure 6.2[34], is used to describe the process of the generation of the Semantic Routing Entries and their maintenance.

The main goal of the rescuers in this example is to locate the four patients A, B, C, and D after a natural disaster, e. g., an earthquake. Each of these patients is associated with their smartphone, which enables communication with the other entities present in the example. Furthermore, the smartphones run a *Knowledge Group* on their own and, thus, can share their knowledge. In order to support the Search & Rescue mission, rescuers have deployed unmanned aerial vehicles (UAVs) and autonomous robots. While UAVs provide an aerial view, robots support the rescuers in locating and rescuing injured people. Additionally, both serve as communication relays to support the potentially damaged communication infrastructure of the city. Since the UAVs and the robots are mobile, the communication connections can break due to their movement. Finally, the smart infrastructure of the city serves as static communication relays. Together, the employed entities form a loosely coupled network of *Knowledge Groups* that can share their knowledge. In

[34]Created with https://app.diagrams.net/ Accessed December 29, 2021.

Figure 6.2, dashed arrows represent the communication between participants. Solid arrows denote a query. Knowledge is indicated as a string consisting of several concepts separated by a slash. A capital letter in this string marks an individual. For example, *"human/person/D"* expresses that individual *D* is a *person* and subsequently a *human*.

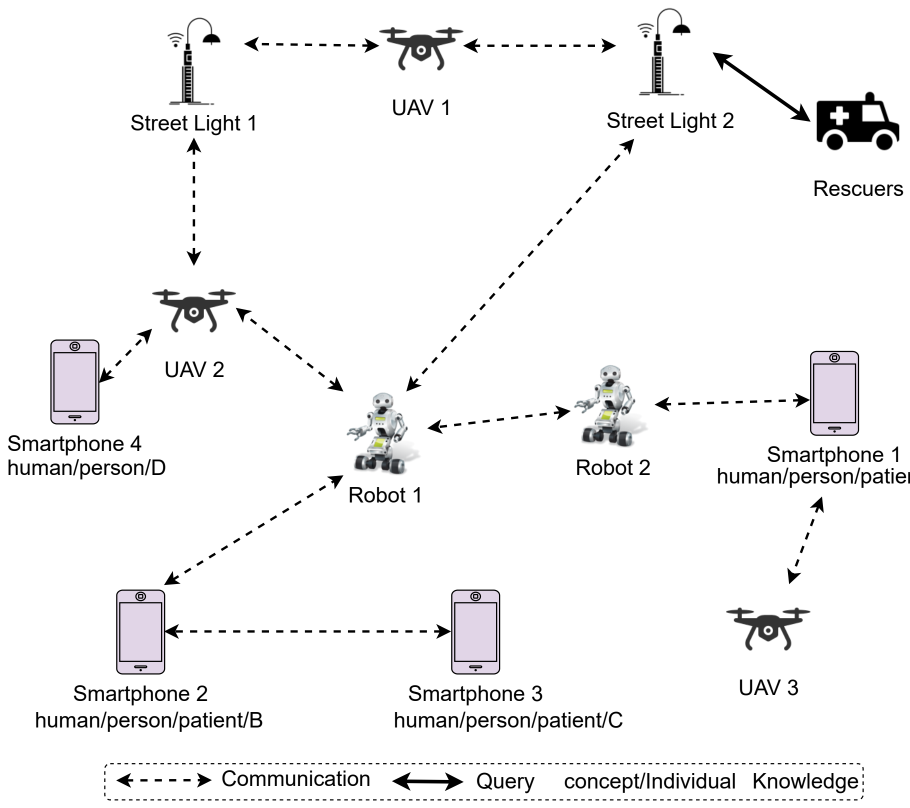


Figure 6.2: Example Scenario[34] [75]

To save patients in this scenario, rescuers send semantic queries to this network, which aim at locating the injured patients. Since these queries focus on the contents of the network, we introduce in [75] the Adaptive Semantic Routing, which is described in detail in the following sections.

This chapter focuses on the fulfilment of Requirement **R4 - Efficient Knowledge Discovery**, which is defined in Section 1.1. Therefore, the following sections present a routing method based on dynamically created routing tables, which rely on the semantics of the stored knowledge. Furthermore, the entries of these tables are represented by ASP rules enabling their adjustment at run-time.

6.2 Semantic Routing Tables

Semantic Routing Tables form the core component of the Adaptive Semantic Routing. Generally speaking, the routing entries indicate which connected *Knowledge Group* has to be addressed for a specific piece of knowledge or which *Knowledge Group* is closer to the requested knowledge. Each routing entry consists of a combination of two External Statements and an ASP rule. The ASP rule is used to model the actual route, while the External Statements represent the availability of the connected *Knowledge Groups* and the validity of the table entry. Templates for the External Statements and the rules are shown in Listing 6.1.

```
1 route(path("concept1", "concept2",...,"conceptn"),
    uuid("location_uuid"), dist(dist), uuid("next_uuid"))
    :- node(uuid("next_uuid")),
    not -route(path("concept1", "concept2",...,"conceptn"),
    uuid("location_uuid"), dist(dist), uuid("next_uuid")).
2 #external node(uuid("next_uuid")).
3 #external -route(path("concept1", "concept2",...,"conceptn"),
    uuid("location_uuid"), dist(dist), uuid("next_uuid")).
```

Listing 6.1: Semantic Routing Table Entry Templates [75]

Line 1 of Listing 6.1 is a template for ASP rules that are used to model the routing table entries. Its head consists of a predicate with the name *route* and arity of 4. The components of this predicate contain the necessary information to locate pieces of knowledge based on their semantics, which is expressed by the *path* predicate. It contains the concepts denoting the semantics of the routing entry, which has been created either during the Semantic Aggregation (see Section 6.3) or due to updates (see Section 6.4) or are given by an initial classification of an individual (see Section 6.5). Since a branch of a taxonomy provides these concepts, they form a chain of subclass relations. For example, $concept_2$ is a subclass of $concept_1$. The second predicate of each routing entry is a *uuid* that indicates the actual location of the knowledge. This knowledge is reachable via the number of hops indicated by the *dist* predicate. Last but not least, the second *uuid* indicates to which *Knowledge Group* the query has to be forwarded to reach the requested knowledge. In the case that the knowledge indicated by the path is stored in the current *Knowledge Group*, both *uuids* are set correspondingly and a *dist* of 0 is selected. To activate a rule, the corresponding instance of the External Statement in Line 2 of Listing 6.1 has to be set to true. Since it represents the available communication to the corresponding *Knowledge Group*, all routes leading to it can be added or removed according to the truth value of the External Statement. While the first External Statement acts as a switch for all routing entries provided by a *Knowledge Group*, the second External Statement acts analogously for a single Routing Entry. It is modelled as the negative variant of the rule head. Thus, it provides contradicting information.

To remove the corresponding routing entry, the External Statement has to be set to true. Subsequently, the negative version of the rule head replaces the positive one, which expresses that the knowledge is no longer reachable via the route.

In contrast to routes, individuals are represented by a single External Statement. Since the *Knowledge Group* manages the individual on its own, it can invalidate the knowledge, for example, if it is outdated, by setting the External Statement to false. The template for the creation of routing entries that represent individuals is shown in Listing 6.2.

```
#external route(path("concept1", "concept2",...,"conceptn",
  "Individual"), uuid("own_uuid"), dist(0), uuid("own_uuid")).
```

Listing 6.2: Semantic Routing Table Entry Template for Individuals

Analogously to the rules presented in Listing 6.1, the *path* predicate represents the semantic classification of the individual, which is the last argument of the *path* predicate. Since the corresponding knowledge is managed by the *Knowledge Group* itself, its UUID is selected for the *uuid* predicates and the distance is set to 0.

By introducing new individuals to the system and by propagating this knowledge, the size of each table would constantly be growing. To prevent this and to keep the tables tractable, an aggregation mechanism is needed, which is introduced in the next section.

6.3 Semantic Aggregation

A central aspect in the creation of Semantic Routing Tables is the Semantic Aggregation. Generally speaking, it reduces the size of the routing table by summarising routing entries based on the provided taxonomies. This has two major impacts on the table. First, the size reduction keeps the tables tractable. Second, the aggregation can stop the propagation of individuals if already aggregated rules exist and, thus, can reduce the number of required messages. The process of the Semantic Aggregation is shown in Algorithm 6.1.

The algorithm receives a routing table rt and a new routing entry re_n as input and returns an aggregated routing entry re_a as well as a list of routing entries re_r that are replaced by re_a and thus have to be removed from rt. The general idea of this algorithm is to compare re_n with each entry of rt and check if one of three possible aggregation steps can be conducted.

The first type of aggregate is possible if an existing routing entry re_e and re_n share the same path and the same origin but differ in the distance entry (see Lines 4

Algorithm 6.1: Semantic Aggregation

```
Input  : Routing Table $rt$, Routing Entry $re_n$
Output: Routing Entry $re_a$, List<Routing Entry> $re_r$
1  Routing Entry $re_a$ = NULL
2  List<Routing Entry> $re_r$
3  foreach Routing Entry $re_e \in rt$ do
4      if $re_e.path$ == $re_n.path$ then
5          if $re_e.origin$ == $re_n.origin$ && $re_n.dist$ >= $re_e.dist$ then
6              return {$SKIP$, $re_r$}
7          else
8              $re_a$ := createAggregatedRoutingEntry($re_e.path$, $re_e.origin$,
                 $re_n.dist$, $self.uuid$)
9              $re_r$.add($re_e$)
10             continue
11     if $re_e.isIndiviual$ && $re_n.isIndividual$ then
12         if $re_e.path.baseConcepts$ == $re_e.path.baseConcepts$ then
13             $re_a$ := createAggregatedRoutingEntry($re_e.path.baseConcepts$,
                 $self.uuid$, 0, $self.uuid$)
14             return {$re_a$, $re_r$}
15     if $re_e.path$ starts with $re_n.path$ then
16         $re_a$ := createAggregatedRoutingEntry($re_e.path \cap re_n.path$,
             $self.uuid$, 0, $self.uuid$)
17         $re_r$.add($re_e$)
18         return {$re_a$, $re_r$}
19 return {$re_a$, $re_r$}
```

to 10). This indicates that both routing entries represent the same knowledge but have reached the *Knowledge Group* separately. Therefore, a new aggregate re_a is created if the new route (re_n) has a lower distance than the existing one (re_e). Routing entries with a higher or equal distance are ignored ($SKIP$). The resulting aggregate re_a comprises the existing path and origin. For the distance, the value of re_n is used since it is lower than the current one. The UUID of the aggregating *Knowledge Group* is set as the contact. As the last step, re_e is added to re_r to replace it in the Semantic Routing Table.

The second type of aggregation can occur if two routing entries, re_n and re_e, represent individuals that share the same base concepts (see Lines 11 to 14). Thus, a routing entry that represents the shared base concepts has to be created. In this process, an aggregated routing entry re_a is created, which contains the path consisting of the shared base concepts, a distance of *0* and the UUID of the aggregating

Knowledge Group, as both the origin and the contact of this entry. This is the case since re_a points at the knowledge that is available at the aggregating *Knowledge Group*. In contrast to the first type of aggregation, re_e is not added to re_r since it could remove the reference to an individual, which is maintained by the *Knowledge Group* itself.

The third type of aggregates is given if the path section of an existing routing entry re_e starts with the path from the newly received entry re_n (see Lines 15 to 18). On the one hand, this summarises routing table entries and, thus, reduces the size of the table. On the other hand, it creates more general entries that support the content-based search. The aggregated routing entry re_a consists of the intersection of both paths, a distance of *0* and the UUID of the aggregating *Knowledge Group* as the origin and the contact of this entry. Subsequently, re_e is added to re_r, which indicates that this routing entry is replaced by re_a.

As the last step (Line 19), a pair containing the aggregate re_a and the list of routing entries that have to be removed (re_r) is returned. If none of the above aggregation types is possible, a pair consisting of a *NULL* value and an empty list is returned. The newly created aggregation re_a and the routing entries that have to be removed re_r are then handled by the update mechanism presented in the next section.

To provide a better understanding of the Semantic Aggregation, the example in Listing 6.3 demonstrates the second type of aggregation. Therefore, this example uses the individuals *B* and *C* introduced in Figure 6.2. Additionally, this example assumes that *Smartphone 2* (*sp2*) conducts the aggregation.

```
1 route(path("human", "person", "patient", "B"), uuid("sp2"), dist(0),
     uuid("sp2")).
2 route(path("human", "person", "patient", "C"), uuid("sp3"), dist(1),
     uuid("sp3")).
3
4 route(path("human", "person", "patient"), uuid("sp2"), dist(0),
     uuid("sp2")).
```

Listing 6.3: Example Aggregation

Line 1 shows the Semantic Routing Entry for individual *B* located on *sp2*. Line 2 contains the Semantic Routing Entry for individual *C*, which was received from *Smartphone 3*. Since both Semantic Routing Entries share the same base concepts, Algorithm 6.1 creates the Semantic Routing Entry shown in Line 4, which states that knowledge about patients can be found at *sp2*.

6.4 Table Updates

The update mechanism is used to adapt the Semantic Routing Tables based on newly arrived information. In general, the update mechanism applies Algorithm 6.2 to enable or disable the representation of connected *Knowledge Groups* and to introduce new routing entries to the routing table.

The algorithm to update the routing table has three inputs. The first one is the routing table that should be updated (rt). The second is a newly received routing entry re_n. The last input is a UUID ($uuid$) used to activate or deactivate the External Statement representing the corresponding *Knowledge Group* if its availability has changed. While the first input is mandatory, the second and third can be replaced by $NULL$ values. The first line defines the boolean flag *aggregated*, which indicates that re_n was aggregated during the execution of the algorithm. The following segment of the algorithm (Lines 2 to 4) relies on the value of the *uuid* input. If it is not a $NULL$ value, the truth value of the corresponding External Statement is switched, which means it is set to false if it was true before or vice versa. This either enables or disables all existing routes to the corresponding *Knowledge Group*. This segment is executed first since adding and removing routes influences the Semantic Aggregation conducted in the next segment by either adding or removing routing entries.

In the case that a new routing entry re_n is given (Line 5), the second and third segments are executed. The second segment of the Table Update algorithm (Lines 6 to 17) handles the new routing entry re_n. If re_n is given (not $NULL$), it is checked if the introduction of re_n results in any aggregation, which is determined by Algorithm 6.1. The pair returned by Algorithm 6.1 consists of a rule re_a indicating a possible aggregation and re_r, a list of rules that are replaced by re_a. Subsequently, Algorithm 6.2 updates the routing table rt by removing all routing entries in re_r by setting the corresponding negative External Statements and the *aggregated* flag to true. Furthermore, re_a is added to rt. Additionally, the created aggregate is propagated to the neighbouring *Knowledge Groups*, excluding the *Knowledge Group* that originally sent re_n. On the receiving side, the distance is increased by one and the update algorithm is called. Furthermore, the ping mechanism of a *Knowledge Group* can be utilised to reduce the overall message complexity since these messages are exchanged regularly. In the last step of this segment, Algorithm 6.2 stops if re_n is not an individual since they are managed by the third segment separately or if Algorithm 6.1 suggested skipping ($SKIP$) the received routing entry.

The third segment (Lines 18 to 25) of Algorithm 6.2 is used if a new routing entry re_n is not part of rt. In this case, re_n is added to rt and propagated to the neighbouring *Knowledge Groups* if it is no individual. Again, the *Knowledge Group* that provided re_n is excluded from the propagation. Subsequently, Line 23 checks that the value of the *aggregated* flag is false. In this case, re_n is a newly received individual containing a new semantic annotation. Therefore, it has to be propagated to the neighbouring nodes. In contrast to the previous propagation steps, this step has to adjust the representation of the individual. Locally, a single External Statement represents an individual (see Listing 6.2). Thus, it has to be transformed into a corresponding rule (see Listing 6.1) that is propagated to the neighbouring nodes.

Algorithm 6.2: Table Update

```
Input: Routing Table $rt$, Routing Entry $re_n$, UUID $uuid$
1  boolean aggregated = false
2  if $uuid \neq NULL$ then
3      rt.switchNodeExternal($uuid$)
4      return
5  if $re_n \neq NULL$ then
6      $< re_a, re_r >$ = semanticAggregation($rt$, $re_n$)
7      if $re_a \neq NULL$ then
8          for $r \in re_r$ do
9              $rt$.set($r$, $true$)
10         if $re_a \notin rt$ then
11             $rt$.add($re_a$)
12             propagate($re_a$)
13         aggregated = true
14         if $re_n$ is no individual then
15             return
16     if $re_a == SKIP$ then
17         return
18     if $re_n \notin rt$ then
19         $rt$.add($re_n$)
20         if $re_n$ is no individual then
21             propagate($re_n$)
22             return
23         if !aggregated then
24             propagateIndividual($re_n$)
25             return
```

6.5 Propagation of Individuals

A central point in the management of Semantic Routing Tables is the propagation of individuals since they are used to answer queries given to the system. Thus, it is essential that their existence is published. The handling of newly created individuals is shown in Algorithm 6.3.

Algorithm 6.3: Individual Propagation

```
    Input   : Routing Table $rt$, Individual $i$
    Output: Void
1 Routing Entry $re_i$ := createRoutingEntry($i$)
2 tableUpdate($rt$, $re_i$, NULL)
3 return
```

In Line 1 of Algorithm 6.3, a new routing entry re_i is created following the rule template presented in Listing 6.2. For example, the introduction of individual *A* at *Smartphone 1* in Figure 6.2 results in the routing entry `#external route(path("human","person","patient","C"), uuid("sp1"), dist(0), uuid("sp1"))`. Simply forwarding the created individuals could result in constantly increasing Semantic Routing Tables. Therefore, the newly created routing entry is given to the update mechanism in Line 2, which checks if the new individual causes any aggregates or updates and forwards it to the neighbouring *Knowledge Groups* if necessary.

6.6 Example Scenario

The combination of the algorithms presented in the previous sections results in dynamically adaptable Semantic Routing Tables. To provide a better understanding of the presented algorithms, their applications are discussed in this section based on the provided example scenario. This scenario encompasses rescuers that want to locate missing persons after a natural disaster. Therefore, they deploy autonomous mobile robots and unmanned aerial vehicles (UAVs) that form a loosely coupled communication network, which includes the smart infrastructure of the city. Each missing person is associated with a smartphone that enables communication with the robots and the UAVs. Additionally, it is assumed that each device (UAVs, robots, etc.) has its own *Knowledge Group*. Figure 6.3 presents the results of Semantic Aggregation, the Table Updates, and the Propagation of Individuals.

In this figure, blue points indicate aggregations. One of these points is located on Smartphone 2 (*sp2*), which stores knowledge about individual *B* that is classified as *human*, *person*, and *patient*. As expressed by the dashed arrow reaching from *sp2* to Smartphone 3 (*sp3*), knowledge about individual *C* is available if *sp2* contacts *sp3*. Since individual *C* is the only piece of knowledge present at *sp3*, it is propagated using Algorithm 6.3. Once the message arrives at *sp2*, the Semantic Aggregation is started. In this case, both individuals are aggregated since they share the base concepts *human*, *person*, and *patient*. The resulting entry is then propagated to

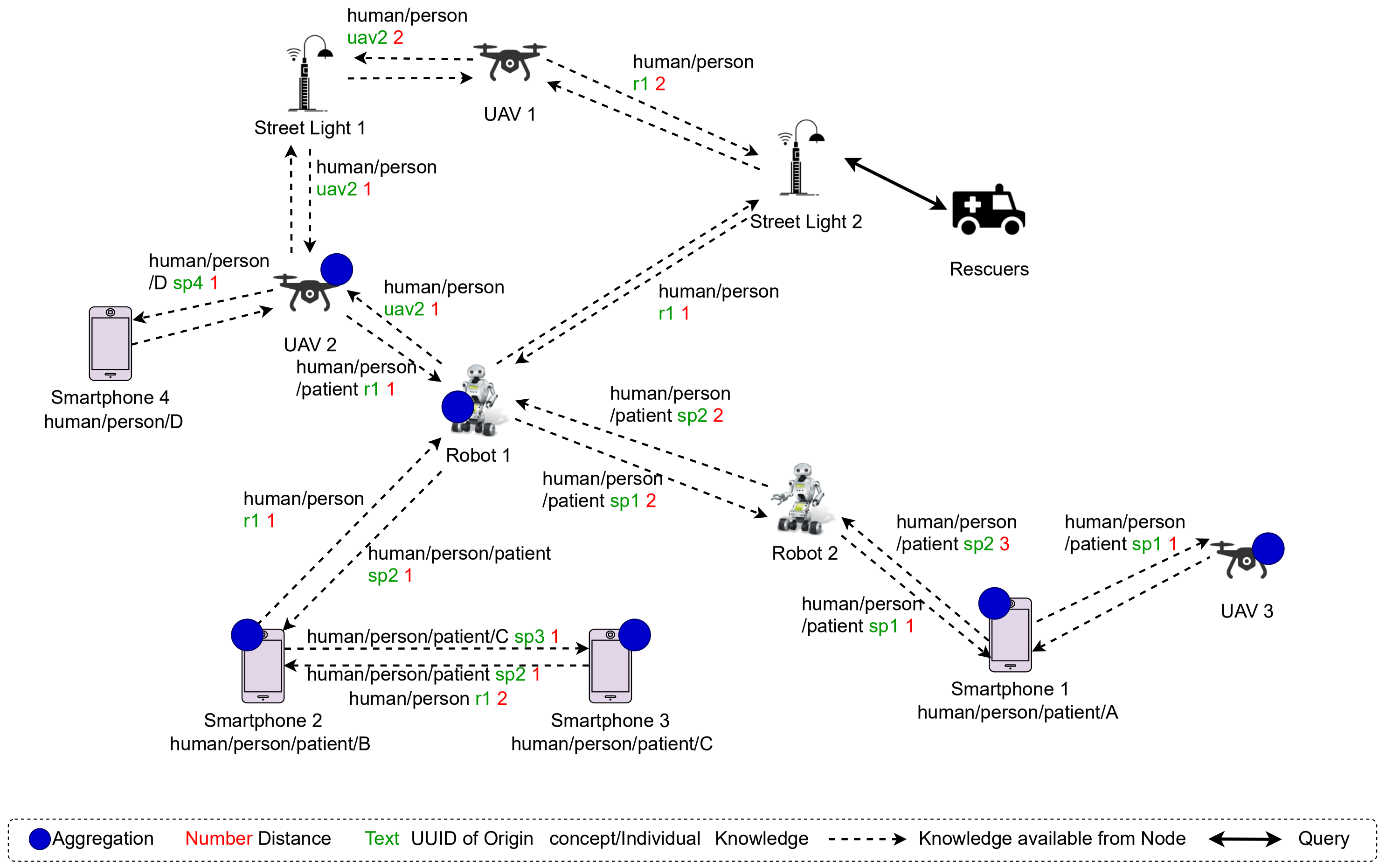


Figure 6.3: Example Scenario[34] [75]

the neighbouring *Knowledge Groups*, which update their routing tables accordingly. This process is conducted analogously for each of the remaining individuals, resulting in the constellation shown in Figure 6.3.

During this process, routing tables are created at each *Knowledge Group*. After the four individuals have been introduced, a fixpoint is reached and the routing entries are not adapted until a new individual is introduced or a *Knowledge Group* is no longer reachable. Listing 6.4 shows the complete ASP program used to create the routing table of *Robot 1* (*r1*). To provide a complete overview, the rules are shown instead of the resulting Answer Set.

```
1 route(path("human", "person", "patient"), uuid("sp1"), dist(2),
    uuid("r2")) :- node(uuid("r2")), not -route(path("human",
    "person", "patient"), uuid("sp1"), dist(2), uuid("r2")).
2 route(path("human", "person", "patient"), uuid("sp2"), dist(1),
    uuid("sp2")) :- node(uuid("sp2")), not -route(path("human",
    "person", "patient"), uuid("sp2"), dist(1), uuid("sp2")).
3 route(path("human", "person"), uuid("uav2"), dist(1), uuid("uav2"))
    :- node(uuid("uav2")), not -route(path("human", "person"),
    uuid("uav2"), dist(1), uuid("uav2")).
4 #external node(uuid("r2")).
5 #external node(uuid("sp2")).
6 #external node(uuid("uav2")).
7 #external -route(path("human", "person", "patient"), uuid("sp1"),
    dist(2), uuid("r2")).
8 #external -route(path("human", "person", "patient"), uuid("sp2"),
    dist(1), uuid("sp2")).
9 #external -route(path("human", "person"), uuid("uav2"), dist(1),
    uuid("uav2")).
```

Listing 6.4: Semantic Routing Table of Robot 1 [75]

The Lines 1 to 3 of this listing model the Semantic Routing Entries. The first route indicates that knowledge about *patients* is available at Smartphone 1 (*sp1*). To reach this knowledge, two hops are required and queries have to be forwarded to Robot 2 (*r2*) to reach the desired knowledge. The existence of this route is linked to the truth value of the External Statements used in the body of the rule. The route represented by the rule head is derived as long as the External Statement representing *r2* (Line 4) is set to true and the External Statement that expresses the opposite of the route (Line 7) is set to false. Since several sources provide knowledge about *patients* in different parts of the network, a second route leading to another *Knowledge Group* is given in Line 2. Its combination with Lines 5 and 8 works analogously to the route presented before. The last set of rules (Lines 3, 6, and 9) indicate that knowledge about *persons* can be found at *uav2*.

6.7 Summary and Discussion

This chapter presents the third main contribution, which is the Adaptive Semantic Routing tailored for dynamic environments. The core of the semantic routing method is formed by Semantic Routing Tables, which consist of routing entries represented by ASP rules. The creation of these entries and the maintenance of the table are divided into three steps. One of these steps is the Semantic Aggregation, which summarises semantically close routing entries by applying a taxonomy and a distance metric. Another step is the update of the routing tables. In the case that new routing entries arrive at a node of the network, the tables are checked for possible updates. These can either be caused by aggregation, the removal of a node from the network, or the introduction of a new individual. The last step is the propagation of individuals, which is required to introduce new individuals to the system.

Reconsidering the related work discussed in Section 3.5, the main difference to the presented Adaptive Semantic Routing in Dynamic Environments is given in the way the routing entries are created and represented. Approaches like [59] utilise Ant Colony Optimization to learn efficient routes. Thus, they are suited for rather static environments since a training phase is required. Pireddu and Nascimento present in [111] an approach, which employs taxonomies to categorise files. However, the routing is limited to a local neighbourhood. Approaches like SHARE [88] rely on multiple techniques like gradient-based routing, local link-state routing, and scoped flooding. In contrast to these approaches, our Semantic Routing Tables rely on semantic information to aggregate routing entries, which minimises the size of the tables and increases their efficiency. Furthermore, by utilising the reasoning capabilities of ASP and Clingo, the routing entries can be dynamically adapted, and thus, our approach is suited for dynamic environments.

Summing up, this chapter presented a dynamic method to discover knowledge in a dynamic and loosely coupled network. Thus, it fulfils Requirement **R4 - Efficient Knowledge Discovery**. Furthermore, it relies on a single formalism to represent the routing entries and the managed individuals. This supports the use of all relevant knowledge without a need to translate between representations, which could influence the overall response time.

Part III

Assessment

7 Evaluation

This chapter presents the evaluation of this thesis. The remainder of this chapter is structured as follows. In Section 7.1, we evaluate the *General Knowledge Store* introduced in Chapter 4. Section 7.2 presents the evaluation of the handling of symbolic commonsense knowledge. Finally, Section 7.3 evaluates the adaptive semantic routing.

The experiments discussed in this section have been conducted on a Lenovo T570 workstation specified in Table 7.1. Additional software, hardware or changes on this setup are highlighted in the corresponding section.

Property	Description
CPU	Intel® Core™ i7-7500U @ 2.70 GHz Dual-Core
RAM	16 GB DDR4-2133
Operating System	Ubuntu 18.04.4
Kernel	4.15.0-112-generic
ConceptNet	5.7
Clingo	5.3.1 with Gringo 5.3.1 and Clasp 3.3.4
Gringo	5.3.1 with Python 2.7.15rc1, without Lua
Clasp	3.3.4 Configuration: WITH_THREADS=1

Table 7.1: Evaluation Setup

7.1 Distributed Knowledge Storage and Management

In order to efficiently manage knowledge stored in the *General Knowledge Store*, a suitable representation of the *Knowledge Items* is needed. Therefore, Section 4.2.2 discusses four modelling schemes:

S1 - External Statements using arbitrary content
S2 - Rule negatively depending on External Statement with arbitrary content
S3 - Rule positively depending on External Statement with arbitrary content
S4 - External Statement using hashed content

The first modelling scheme **S1** uses a single External Statement that holds the content. **S2** encapsulates the content in the rule head, which depends negatively (*not*, see Section 2.4) on an External Statement. **S3** follows this way of modelling. In contrast to **S2**, it depends positively on an External Statement. While **S1**, **S2**, and **S3** hold content of arbitrary size, **S4** uses hashes to represent the content. In general, hashes are created by one-way functions and represent strings of arbitrary length by fixed-size strings while preventing collisions. As discussed in Section 4.2.2, the usage of hashed content should mitigate the influence of the content on the run time, which is evaluated in the following paragraphs.

Figure 7.1 compares the average run time of the first three modelling approaches. To evaluate the influence of content size, different sized strings consisting of randomly selected characters are used. The considered character numbers increase up to 100.000 with a step size of 100 characters. The run time for each number of characters was measured 100 times.

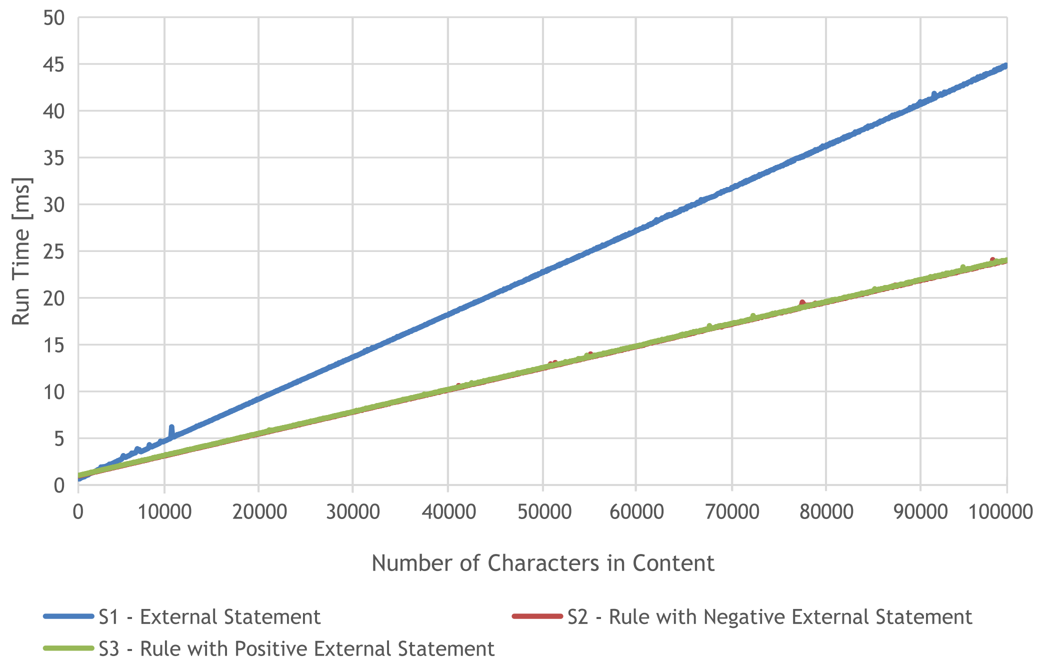


Figure 7.1: Comparison of Different Modelling Schemes

The blue graph shows the average run times of **S1**, the red graph indicates the results of **S2**, and the green graph presents the results of **S3**. As it is clearly evident, all schemes scale linearly with the number of characters used in the content of each *Knowledge Item*. The average run times of **S2** and **S3** are almost identical. However, **S2** achieves the lowest average run times, as indicated in Figure 7.2 which presents an excerpt from the measurements show in Figure 7.1. Since it depends negatively on an External Statement and Clingo assumes External Statements to be false by default, no further steps besides adding, grounding, and solving are required to establish a *Knowledge Item*. **S3** has a slightly higher average run time. This is caused by the positive dependence of the rule head on an External Statement, which has to be set to true before the solving step. This marginally impacts the average run time because a fixed size UUID is used in the External Statement. In total, the UUID consists of 36 characters, 32 characters for the UUID and 4 separators (-) in its string representations. **S1** has the highest average run time. Since it models *Knowledge Items* as a single External Statement, the content has to be parsed twice, once during adding and once during the change of its truth value. Thus, the size of the content has a high impact on the average run time. The measured run times have a standard deviation ranging from 13 % for **S1** up to 22 % for **S2** and **S3**. Considering the low run times, they can be easily influenced by operating system events, which increases the standard deviation.

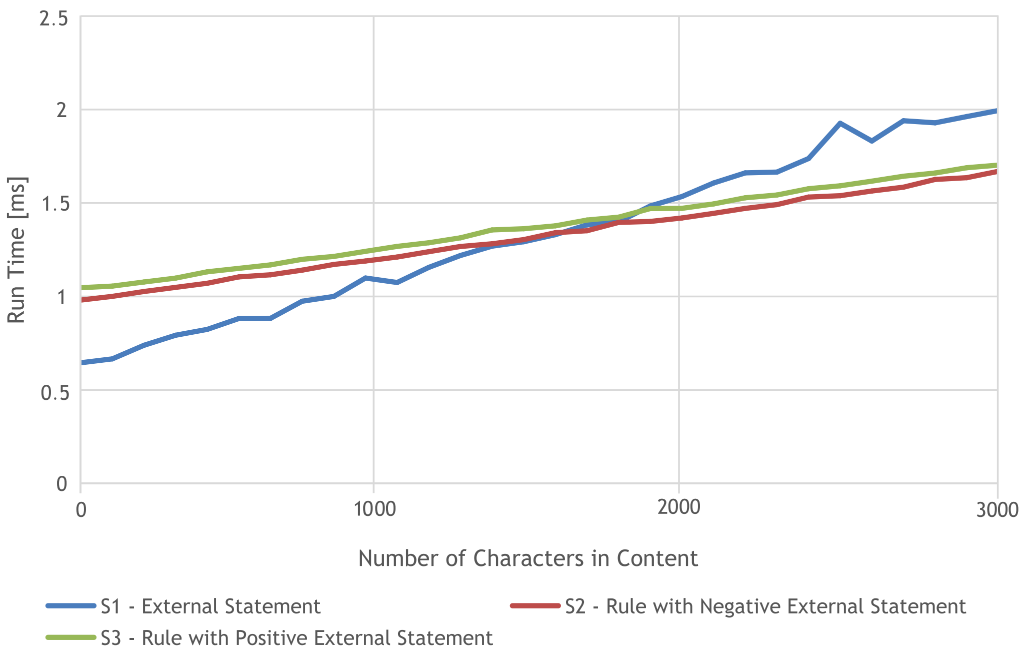


Figure 7.2: Intersection of Run Times

S1 has an advantage when considering a lower number of characters in the content. Figure 7.2 shows the corresponding excerpt from Figure 7.1. In this case, the run time when using **S1** is lower than the run times of **S2** and **S3**. Especially between 0 and 1800 characters, **S1** is faster. Hence, content consisting of a low number of characters is suited for modelling schemes that solely utilise External Statements. If the content contains more than 1800 characters, rule-based schemes achieve lower run times. This result is exploited by **S4**, which uses an External Statement and a fixed-size hash of the content. Thus, it has the lowest modelling overhead and the lowest run time. One drawback is that the content hash still has to be parsed twice. Nevertheless, this only causes a marginal increase in the run time, as shown by the run times of **S2** and **S3**, which differ in the parsing of a fixed-size content. According to the results of **S1**, when considering hash sizes up to 1800 characters, utilising **S4** is best suited to be used in the *General Knowledge Store* since hash algorithms typically produce hashes with lower sizes.

After the selection of a suitable modelling scheme, this part of the evaluation focuses on the influence of different hash sizes on the run time of adding *Knowledge Items* to the *General Knowledge Store*. Therefore, this evaluation considers five hash sizes. These include 64, 128, 256, 512, and 1024 characters. While 64 characters (256 bit) is used by SHA3-256 [36] and 128 characters (512 bit) is used by SHA3-512 [36], the remaining sizes provide measurements for hash functions with arbitrary hash sizes like BLAKE3 [102]. Table 7.2 presents a comparison of the average run times of 100 measurements for each hash size. Furthermore, the number of *Knowledge Items* is steadily increased until 1000 *Knowledge Items* are part of the *General Knowledge Store*.

	Number of *Knowledge Items*			
Characters	250	500	750	1000
64	9.1	17.8	26.8	35.6
128	9.1	17.8	26.8	35.5
256	9.1	17.9	26.9	35.6
512	9.3	18.1	27.1	35.8
1024	9.4	18.3	27.3	36.1

Table 7.2: Influence of *Knowledge Items* on the Run Time in [ms]

The average run time of introducing a new *Knowledge Item* to the *General Knowledge Store* increases linearly with the number of added *Knowledge Items*, which indicates the scalability of the selected modelling scheme. Furthermore, the standard deviation is roughly 2 % for all measurements, which shows their stability. Each additional *Knowledge Item* increases the run time on average by 0,035 ms, supporting the scalability and applicability of the selected modelling scheme. The selection of the hash size has only a low impact on the average run time. As expected, the lowest average run times were measured for 64 characters and the highest for 1024 characters. Additionally, the run time increases linearly with the size of the selected hash. Hashes with 64 and 128 characters achieve almost identical results. Setting the size to 256 characters increases run time by roughly 0.1 ms. Doubling the hash size from 256 characters to 521 requires, on average, additional 0.2 ms and increasing from 512 to 1024 characters requires 0.2 ms. Thus, selecting hashes with sizes up to 1024 characters has no significant impact on the scalability of the selected modelling scheme.

7.2 Handling of Symbolic Commonsense Knowledge

This section presents the evaluation of the Symbolic Commonsense Knowledge Handling. The selected stopping criteria significantly influence the size of extracted commonsense ontologies and the run time of the extraction. Thus, Section 7.2.1 evaluates their impact on the size of the resulting ontologies. Section 7.2.2 presents the corresponding run times. A comparison between the Web Ontology Language (OWL) [3] is given in Section 7.2.3. The results shown in this section have partially been published by us in [77]. Subsequently, Section 7.2.4 evaluates the inconsistency detection and prevention discussed in Section 5.3. The presented results have been published in [78].

7.2.1 Ontology Size

Two parameters mainly influence the automatic extraction of commonsense ontologies. The first is the selected root concept, which determines the starting position of the extraction in the selected hypergraph. The second parameter is the set of stopping criteria, which influence the decision to add further edges to the ontology. The evaluation discussed in this section utilises CN5 as the commonsense knowledge source. The considered root concepts are *Animal*, *Car*, *Person*, and *Thing*. Furthermore, minimum edge weights act as stopping criteria since they express the reliability of the extracted edges. For the creation of the ontology, edges labelled with the relations `IsA`, `FormOf`, `Synonym`, and `HasProperty` are used. Since the highest number of edges is given for the `IsA` relation, the adaption of the minimum weight will have the highest impact on the size of the resulting ontology. Therefore, this evaluation uses the minimum weights 2.5, 2.0, and 1.0 for the `IsA` relation. The minimum weights for the remaining relations have a fixed value of 2.0. Table 7.3 summarises the resulting ontology sizes.

	Root Concept			
Minimum Weight	Animal	Car	Person	Thing
2.5	522	34	17	196
2.0	95353	95353	95353	95353
1.0	201148	201148	201148	201148

Table 7.3: Ontology Sizes Based on Minimum Weight and Root Concept [77]

The minimum edge weight of 2.5 is the most restrictive criterion. Thus, the resulting ontologies have the lowest number of edges. The ontology extracted for the root concept *Animal* contains 522 edges, 34 edges for the root concept *Car*, 17 for *Person*, and 196 for *Thing*. Lowering the minimum weight increases the size of the ontology. In the case of a minimum weight of 2.0, the influence of the root concept is no longer present and a big part of the CN5 hypergraph is explored, resulting in ontologies with 95353 edges. Lowering the minimum weight to 1.0 roughly doubles the ontology size. Considering these results, higher minimum edge weights are suited to create small domain-specific ontologies. In contrast to that, lower minimum weights create more general commonsense ontologies.

7.2.2 Ontology Generation and Query Run Time

In addition to the ontology size, the selection of stopping criteria and the root concept has a major influence on the run time of ontology generation. Again, CN5 is used as the commonsense knowledge source. The ontologies generation considers edges annotated with the relations `IsA`, `FormOf`, `Synonym`, and `HasProperty`. Furthermore, the minimum edge weight is used as the stopping criterion. The evaluated root concepts are *Animal*, *Car*, *Person*, and *Thing*. The minimum weights are 2.5, 2.0, and 1.0 for the `IsA` relation. The remaining relations have a fixed minimum weight of 2.0. The run time of the ontology generation has been measured 20 times for each root concept and minimum weight. Table 7.4 shows the mean values of the measured run times. The corresponding standard deviations are given in Table 7.5.

	Root Concept			
Minimum Weight	Animal	Car	Person	Thing
2.5	142.05	5.01	3.65	50.75
2.0	8097.45	8139.04	8199.49	8142.40
1.0	13505.68	13402.36	13341.25	13134.26

Table 7.4: Run Time of the Ontology Generation in [s] [77]

	Root Concept			
Minimum Weight	Animal	Car	Person	Thing
2.5	0.61	0.22	0.20	0.43
2.0	51.54	20.95	66.44	34.64
1.0	103.87	83.11	54.40	84.25

Table 7.5: Standard Deviation in [s] [77]

The lowest run times have been measured for the ontologies generated using the minimum weight of 2.5. In this case, the selection of the root concept has a significant influence on the run time. The generation takes 3.65 s for the root concept `Person`, 5.01 s for `Car`, 50.75 s for `Thing`, and 142.05 s for `Animal`. Considering the ontology sizes presented in Table 7.3, it can be seen that the generation run time scales linearly with the ontology size. For lower minimum weights, similar run times were measured for all root concepts. Again, the run time scales linearly to the ontology sizes. If the size of the ontology is doubled, the run time of the generation is roughly increased by a factor of 1.5. The standard deviation shown in Table 7.5 is roughly 1 % for all measurements, which indicates that there is no high deviation or outliers in the measurements.

Another essential aspect besides the ontology generation is reasoning. This process consists of seven steps. The first four steps ensure that Clingo can use the ontology during subsequent queries. They include the adding and grounding of the edges of the ontology, which are modelled by External Statements. Clingo assumes newly added External Statements as false. Thus, they have to be set to true in the third step, including them in the reasoning process. After the solving process is finished, the ontology can be used to classify individuals via queries. Queries are separate ASP programs. Hence, they require three steps, which are adding, grounding, and solving. The run times have been measured 20 times. As a query, the ASP predicate *is("rex","puppy")* is used (see Section 5.2.2), which results in a classification of *rex* based on the extracted ontologies. In order to remove this predicate, it depends negatively on a unique External Statement that is set to true after the query has been answered. Figure 7.3 depicts the results for a minimum weight of 2.5. The complete evaluation results are shown in Table H.1 and Table H.2 in the appendix.

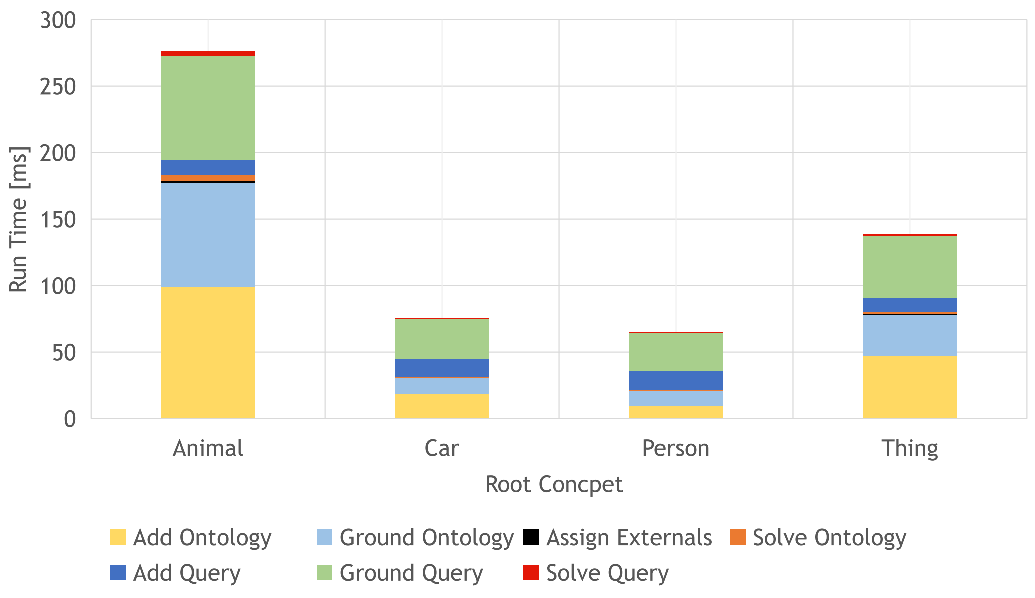


Figure 7.3: Classification and Query Run Times

The measured run times show similar behaviour to the run times of the ontology generation. A minimum weight of 2.5 achieves the overall lowest run times. In the case the minimum weight is lowered, the run times increase linearly with the ontology size. The highest impact on the total run time is given by the introduction of the ontology to Clingo. Especially adding (yellow) and grounding (light blue) of the ontology have a major impact on the run time, which is caused by the high number of rules. In comparison to the adding and grounding, the run times of assigning External Statements (black) and solving (orange) are negligible since no complex rules, e. g. disjunctions are involved. The run time of a query is generally lower than the complete ontology reasoning process. The run time of adding a query (dark blue) is similar for all minimum weights and root concepts since they contain a fixed number of rules. However, the grounding of a query (green) requires the inclusion of the ontology rules, which increases the run time linearly based on the size of the ontology. Finally, the solving step (red) is only slightly affected by the inclusion of the ontology rules. The measured run times scale linearly with the ontology size. Thus, *Person* achieves the lowest total run time (64.69 ms) followed by *Car* (75.55 ms) and *Thing* (138.49 ms). The root concept *Animal* has the highest total run time (276.29 ms).

Table H.2 presents the standard deviation of the measurements. The standard deviation for larger ontologies is lower than 1 %, which indicates the stability of the measurements. The standard deviation slightly increases to roughly 2.5 % for ontologies created with a minimum weight of 2.5 since the low run times can easily be influenced by concurrently running processes. The highest standard deviation is given for the query solving using the ontology with `Person` as the root concept and a minimum weight of 2.5, which is roughly 7 %. This is caused by the very low run time of 0.29 ms. Such low measurements can be influenced by any concurrent process on the system and, thus, causing the high standard deviation.

7.2.3 Comparison to OWL

The Web Ontology Language (OWL) [68] is the de facto standard for representing knowledge in the Semantic Web and for the creation of ontologies. Thus, this section compares our ASP-based ontology generation and modelling with OWL-based ontology modelling. Therefore, we translate the Pizza Ontology[35,36] using the ASP-based modelling scheme and ARRANGE presented in Section 5.2. The OWL reasoning uses HermiT [123] version 1.4.3.456 and Protégé 5.5.0 [99]. In order to compare both formalisms, the evaluation presented in this section creates an individual with the base class *margherita*. Subsequently, it applies a classification query. Figure 7.4 shows the average run time of 20 measurements. For each measurement, a new solver instance was used.

The presented measurements include the classification of the individual based on the ontology and the determination of all super- and subclasses. As shown in Figure 7.4, the ASP-based formalism takes 82.15 ms for the preparation of the ontology, which includes adding, grounding, and solving. Solving the query takes

[35]An Ontology About Pizzas and their Toppings, https://protege.stanford.edu/ontologies/pizza/pizza.owl, Accessed December 29, 2021.

[36]Pizzas in 10 Minutes, https://protegewiki.stanford.edu/wiki/Protege4Pizzas10Minutes, Accessed December 29, 2021.

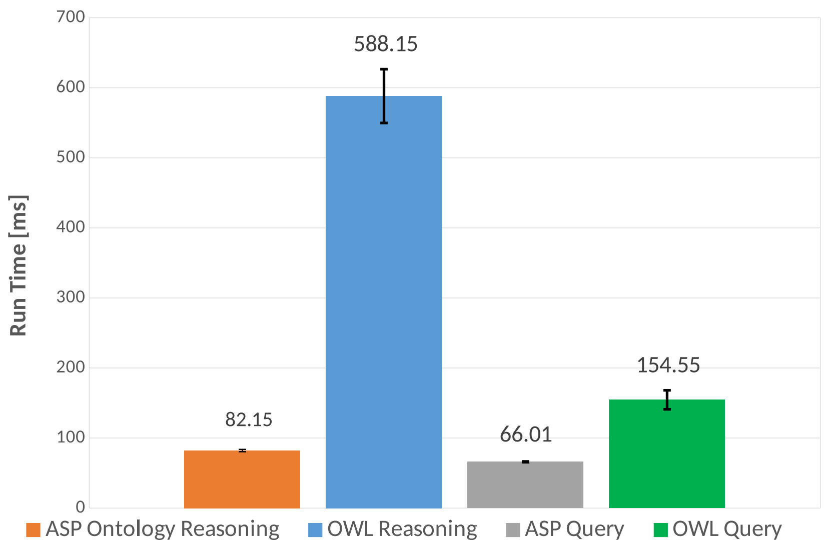


Figure 7.4: Comparison of ASP and OWL Run Times [77]

66.01 ms. Furthermore, the measurements have a low standard deviation of 1.63 ms and 0.55 ms, indicated by the error bars. In comparison to this, HermiT takes 588.15 ms for the reasoning process, which includes the determination of the super- and subclass as well as the classification. The query resolution has a mean run time of 154.55 ms. The standard deviation is higher in comparison to the ASP-based approach. The measurement for the ontology reasoning has a standard deviation of 38.36 ms and 13.57 ms for the query resolution. Comparing these measurements, the ASP ontology reasoning is roughly seven times faster than the OWL reasoning process. One of the reasons for the differences in the measured run times is the underlying reasoning mechanism. At the base of the ASP reasoning process, a SAT-solver solves the grounded ASP program. On the one hand, they provide fast results. On the other hand, they do not provide any explanation for the results, which complicates debugging. In contrast to ASP, OWL uses a tableau algorithm [7], which is typically slower than SAT-based approaches. However, tableau algorithms support explanations of the results and thus enable efficient debugging.

Besides the measured run times, language properties like decidability and assumptions are important aspects. The full specification of OWL is based on first-order logic and, hence, undecidable. To mitigate this drawback, many applications rely on decidable subsets, such as OWL Lite or OWL DL (see Section 2.6.2). All specifications adhere to the same assumptions. First, there is no Unique Name Assumption, which supports providing distinct names for a single individual. Second, OWL ontologies are monotonic. Thus, already derived knowledge cannot be removed if novel or contradicting knowledge arises. To incorporate such knowledge, the OWL reasoning process has to be restarted. Third, the Open-World Assumption holds, which limits the reasoning process to the concepts and relations explicitly modelled in the ontology. Thus, an OWL reasoner is not able to derive that a statement is false solely on its absence in the ontology. Based on these assumptions, defaults that may be overwritten by additional knowledge cannot be defined.

In comparison to OWL, ASP complies with the Unique Name Assumption. Thus, an individual can only be represented by a single name. On the one hand, it reduces the overall modelling effort since there is no need to state that names refer to the same individual explicitly. On the other hand, it prevents the modelling of distinct individuals with the same name or identifier. In contrast to monotonic reasoning, non-monotonic reasoning supports the retraction of already derived knowledge if contrary knowledge is given. Furthermore, ASP adheres to the Closed-World Assumption, which enables the definition of defaults since knowledge that is not explicitly stated is assumed to be false. Last but not least, the incorporation of the ASP solver Clingo supports the creation of dynamically adaptable ontologies. Thus, ASP, in combination with Clingo, is suited to model commonsense knowledge ontologies, which can be dynamically adapted to their current application field. In contrast to that, OWL ontologies are suited for relatively static environments in which new concepts or relations appear seldomly.

7.2.4 Inconsistency Detection and Prevention Run Time

Incorporating commonsense knowledge into a knowledge base can introduce semantic inconsistencies, for example, by adding contradicting properties of an object. To provide an insight into the inconsistent properties that can occur, we tested in [78] 200 concepts of CN5 that typically appear in households. Among these, 12 concepts had at least one pair of contradicting properties. In total, 18 pairs of contradicting properties were found. Table 7.6 shows selected concepts alongside their contradicting properties.

Concept	Contradicting Properties
glass	opaque $\Longleftrightarrow$ clear, see through
human	good $\Longleftrightarrow$ evil
knife	sharp $\Longleftrightarrow$ dull
matter	solid $\Longleftrightarrow$ liquid
people	thin $\Longleftrightarrow$ overweight, fat

Table 7.6: Examples of Contradicting Properties in CN5 [78]

One of these examples is the concept of a *knife*. While it is commonsense knowledge that the concept *knife* can have the properties *sharp* and *dull*, assigning both properties to a single instance of a *knife* would cause a contradiction and thus introduce a semantic inconsistency to the knowledge base.

To handle semantic inconsistencies, we propose in [78] a method to detect them and provide rules preventing them. Since these rules are generated during the extraction of the commonsense knowledge from a graph-based source, this section evaluates the influence of the semantic inconsistency detection and prevention on the run time. The overall process consists of three steps: the extraction of edges from CN5, the detection of possible inconsistencies, and subsequent translation into ASP. The run time is measured for four base concepts. The concept *bed* is connected to 831 other concepts, which includes six properties. The concept of *fire* is linked to 955 concepts, including 28 properties. *Water* is connected to 2468 concepts containing 69 properties. Finally, the concept of *people* has connections to 2635 concepts that contain 271 properties. Figure 7.5 presents the average of 1000 run time measurements. The run time of extraction of the connected concepts is presented by the green bars. Blue bars indicate the run time of the inconsistency detection and red bars for the translation into ASP. After each measurement, the local CN5 instance is restarted to prevent influences from previous interactions.

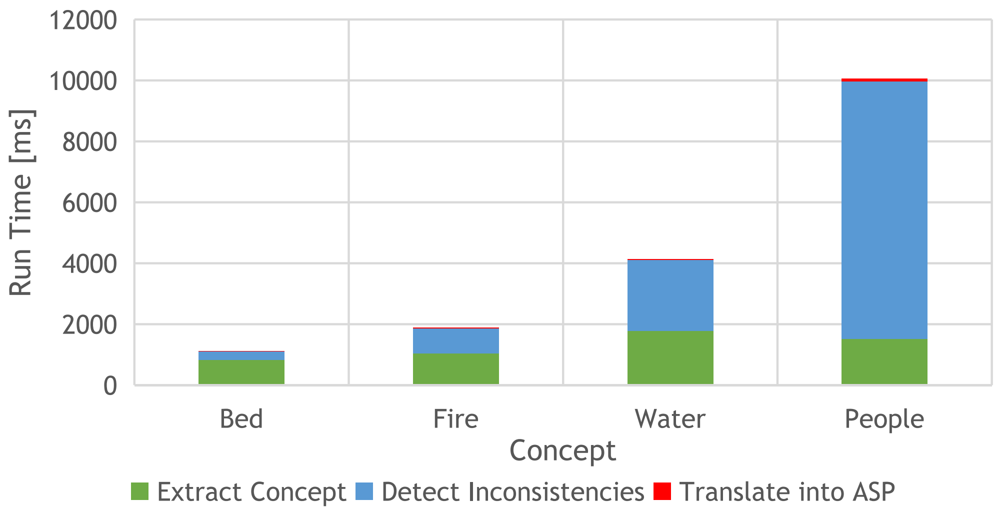


Figure 7.5: Run Time of Inconsistency Handling for Different Concepts [78]

The run time of the concept extraction scales linearly with the number of connected concepts. Thus, the measurements for *bed* (838.16 ms) and *fire* (1043.24 ms) as well as for *water* (1782.53 ms) and *people* (1513.44 ms) are similar. The standard deviation is roughly 10 % for *bed* and *fire*. This can be caused by the interaction with CN5. During this process, an HTTP-GET is sent to a local instance of CN5, which consists of several interacting Docker containers. The concept limit of the request is set to 1000, and thus only a single query is required. After the request is answered, the results are returned. Thus, the measured run time depends on the communication between several processes and containers. The standard deviation of the measurements of *water* and *people* is roughly 2 %. In this case, several HTTP-GET requests are required, which mitigates the influence of outliers.

In contrast to the extraction of concepts, the detection of inconsistencies focuses on the properties and scales linearly with their number. The concept *bed* is connected to six properties and the detection of possible inconsistencies takes 267.32 ms on average. Checking the properties of *fire* takes 826.95 ms, 2318.79 ms for *water*, and 8450.49 ms for *people*. The standard deviation is roughly 7 % for *bed*, 4 % for *fire*, 2.3 % for *water*, and 2 % for *people*. Again, the standard deviation decreases when the number of interactions with CN5 increases. In comparison to the extraction of concepts from CN5 (roughly 1 ms per concept), the inconsistency detection takes roughly 35 ms per concept. This increase is caused by the number of interactions with CN5. Instead of extracting concepts in a batch, the inconsistency detection requires several small requests to CN5 regarding properties, their synonyms, and possible antonymic relations between them, resulting in a higher run time.

The translation into ASP has the lowest impact on the overall run time since no interaction with CN5 is required. It scales linearly with the number of edges connecting the extracted concepts. The translation of the concept *bed* takes 4.2 ms on average, 10.12 ms for *fire*, 21.38 ms for *water*, and 92.25 ms for *people*. The measurements of the translation of *bed* and *fire* have a standard deviation of roughly 30 %, which reflects that events of the operating system easily influence the low run time. These events have a lower impact on higher run times resulting in a standard deviation of 14 % for *water* and 6.5 % for *people*.

7.3 Adaptive Semantic Routing in Dynamic Environments

This section presents the evaluation of the Adaptive Semantic Routing mechanism illustrated in Chapter 6. It includes the analysis of the message complexity, which is discussed in Section 7.3.1. Furthermore, Section 7.3.2 evaluates the run time of the creation of the Semantic Routing Tables, their updates, and the propagation of new individuals.

7.3.1 Message Complexity

A central aspect of the creation of the Semantic Routing Tables is the number of exchanged messages during their creation. Especially in highly dynamic environments as well as loosely coupled networks, a low message complexity is preferred to avoid additional stress on the network. Furthermore, the focus of the presented routing mechanism is set on the content. Hence, no explicit addressing, e. g. IP-based addressing is given. Typically, systems apply flooding in such environments, which introduces a high message load to the system. The following paragraphs summarise the analysis of the message complexity, which we have already published in [75].

In general, there are two types of flooding algorithms. The first type is Naive Flooding, which forwards messages to all neighbours except its original sender. If a node receives an already known message, it is ignored. Thus, in a fully connected network of n nodes, a message complexity of $O(n\cdot(n-1))$ is given, which holds for the best and the worst case. Maintaining a fully connected network in highly dynamic environments is a challenging task and impractical in general. Thus, we will focus on not fully connected mesh networks. This reduces the message complexity of Naive Flooding to $O(n\cdot m)$, where m is the number of neighbours without the origin of the message. The second type is Selective Flooding. To reduce the message complexity of updates and queries to $O(n)$, a tree structure is created, which is responsible for message forwarding. Thus, the best-case complexity for queries is defined by the depth of the corresponding branch of the tree. However, additional messages are required to create and maintain the tree structure.

Our Adaptive Semantic Routing mechanism relies on flooding in two cases. The first case is the introduction of the first individual on a node. The second case is the introduction of a new taxonomy branch. In both situations, the necessary routing entries are disseminated through the network resulting in a worst-case message complexity of $O(n \cdot m)$. However, this complexity is reduced by the creation of aggregation points since each aggregation point acts as a filter for already existing aggregations. Thus, in the best case, the message complexity to introduce a new individual can be reduced to $O(1)$ if it is located at an aggregation point. Additionally, aggregation points reduce the message complexity to $O(m)$ if the neighbouring nodes already contain corresponding routing entries. Besides the creation of the routing tables, the message complexity for queries is an important metric. The worst-case complexity is $O(n)$ if all nodes contain fitting knowledge, which can occur if a general concept in the taxonomy is used in the semantic query. In contrast to this, the best case of $O(1)$ can be achieved if the query is given to a node managing the queried individual. Finally, adding a new node has a message complexity of $O(1)$ since the routing table of a neighbouring node can be copied.

To provide more context, Table 7.7 provides a comparison of the message complexities for the related work discussed in Section 3.5. In this table, n denotes the total number of nodes in the network and m denotes the number of neighbours. $O(1)$ indicates a constant number of messages. A central aspect is the message complexity of updating the routing information. For updates, three different approach types are considered. Learning approaches like Michlmayr et al. [93] and Gómez Santillán et al. [59] need to learn new routes during updates. Thus, additional messages are required with results in a worst- and best-case complexity of $O(n\cdot m)$. The same holds for systems that do not apply any additional management structures like Jacobson et al. [70] or our Adaptive Semantic Routing. Hence, they have a worst-case message complexity of $O(n \cdot m)$. Nevertheless, once the routing tables are built, a constant best-case message complexity can be achieved. For example, the Adaptive Semantic Routing does not need to propagate routing entries if corresponding aggregates are

	Update		Query	
Algorithm	Worst Case	Best Case	Worst Case	Best Case
Naive Flooding	$O(n \cdot m)$	$O(n \cdot m)$	$O(n \cdot m)$	$O(n \cdot m)$
Selective Flooding	$O(n)$	$O(n)$	$O(n)$	$O(height(tree))$
Koloniari et al. [83]	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Jacobson et at. [70]	$O(n \cdot m)$	$O(n \cdot m)$	$O(n \cdot m)$	$O(1)$
Pireddu et al. [111]	$O(m^{height(taxonomy)})$	$O(m^{height(taxonomy)})$	$O(n \cdot m)$	$O(1)$
Michlmayr et al. [93]	$O(n \cdot m)$	$O(n \cdot m)$	$O(n)$	$O(1)$
Gómez Santillán et al. [59]	$O(n \cdot m)$	$O(n \cdot m)$	$O(n)$	$O(1)$
Manfredi et al. [88]	$O(n \cdot m)$	$O(n \cdot m)$	$O(n \cdot m)$	$O(1)$
Gritter et al. [63]	$O(n \cdot m)$	$O(1)$	$O(n)$	$O(1)$
Orda et al. [109]	$O(1)$	$O(1)$	$O(log(n))$	$O(1)$
Adapt. Sem. Routing	$O(n \cdot m)$	$O(1)$	$O(n)$	$O(1)$

Table 7.7: Comparison of Message Complexities

present. The worst-case message complexity can be mitigated by introducing additional management structures, for example, tree-like structures (Selective Flooding or Koloniari et al. [83]). This reduces the worst-case message complexity for updates to $O(n)$. A further reduction can be achieved by utilising more complex structures like Distributed Hash Tables (DHTs), which can lead to constant update message complexity (see Orda et al. [109]). However, the creation of management structures requires additional messages. Furthermore, these structures are typically suited for networks of stationary nodes since a reallocation of a node would require a complex update of the management structure. Furthermore, the application of DHTs like Kademlia [90], as suggested by Orda et al. [109], organises the nodes based on their location and does not provide any support for semantic information. Thus, semantically close knowledge could be stored on distant nodes. Learning approaches like Michlmayr et al. [93] and Gómez Santillán et al. [59] require a learning phase to establish routes. Hence, they are not suited for environments where nodes are relocated or frequent changes of the network structure occur. Comparing our Adaptive Semantic Routing with the algorithms shown in Table 7.7, it achieves a competitive message complexity for updates while supporting the relocation of nodes and data.

Besides the updates of the routing information, the message complexity of queries is an important aspect. In this complexity analysis, it is assumed that all updates have been finished and that the stored routing information is stable. Furthermore, the worst case indicates that the queried information or knowledge is distributed on the network, while the best case assumes that the queried information or knowledge is located on one node or some relatively close nodes. Considering the worst-case scenario, several approaches, like Pireddu et al. [111] or Manfredi et al. [88], rely on flooding, which results in a message complexity of $O(n \cdot m)$. Again, DHT-based approaches require fewer messages since they simply have to calculate hashes and address the corresponding nodes. The remaining approaches achieve a message complexity of $O(n)$ since they have to contact all nodes to acquire all queried information or knowledge. The message complexity for the best case is constant for most of the presented algorithms. Again, our Adaptive Semantic Routing achieves a competitive message complexity.

Summarising the message complexity analysis presented in the previous paragraphs, it can be said that the Adaptive Semantic Routing introduced in this thesis achieves competitive results. Updates require $O(n \cdot m)$ messages in the worst case, which is caused by initial flooding. In the best case, a constant message complexity is achieved. Queries have a message complexity of $O(n)$ in the worst case since all nodes of the network could contain the required information and knowledge. Queries have a constant message complexity if the answer is located on a single node. Finally, the introduction of multiple aggregation points tends to steer the message complexity of updates and queries to the best case since they filter messages which contain already met taxonomy concepts.

7.3.2 Run Time Analysis

After a comprehensive discussion of the message complexity of the Adaptive Semantic Routing mechanism, this section focuses on the run time analysis of its components as well as the evaluation of different test setups. It is divided into two parts. In the first part, the focus is set on the run time analysis of update and aggregation mechanisms. The second part provides an analysis of different scenarios.

The centrepiece of the Adaptive Semantic Routing is the update mechanism (see Section 6.4) and the aggregation (see Section 6.3) since they manage the routing tables, handle new routes, summarise routes according to their semantics, and propagate routes to neighbouring nodes. Hence, both mechanisms have a major impact on the performance of the Semantic Routing Tables. To focus on the run time of the mechanisms, a single node is considered during this part of the evaluation. During the measurements, 50 individuals with the semantic type *path("human","person", "patient")*, 50 individuals with the semantic type *path("human","person")*, and 50 individuals with the semantic type *path("human")* are subsequently added to the node. Figure 7.6 presents the average run time of 100 runs of this experiment. The blue graph indicates the total measured run time, including updates and aggregation.

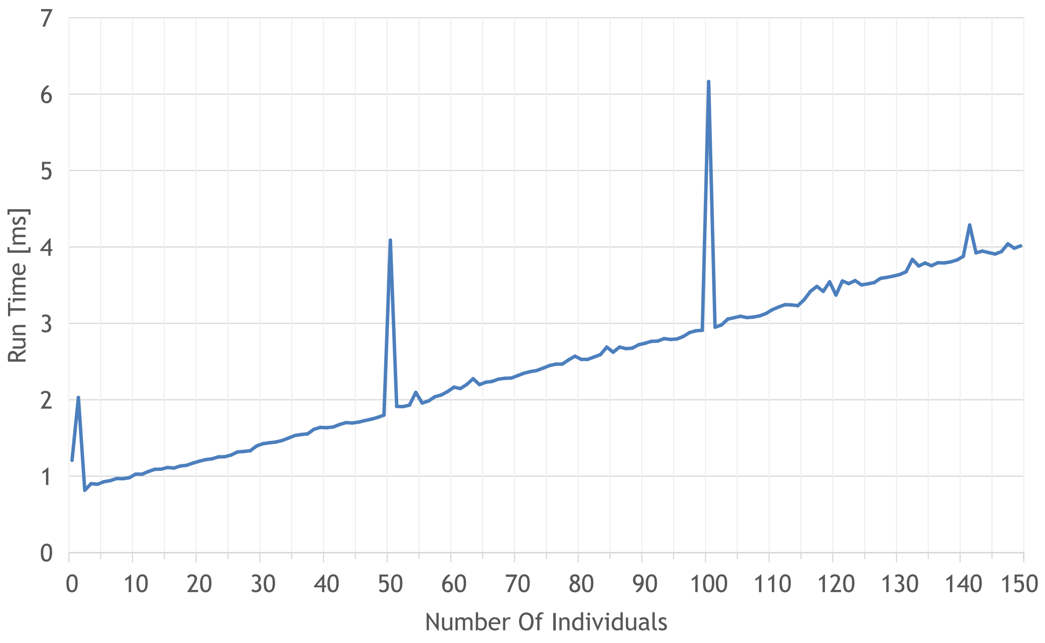


Figure 7.6: Average Run Time of Aggregation and Update Mechanisms

The aggregation has a low impact of 0.5 % on the overall run time, which is achieved by two factors. The first factor is the usage of taxonomy branches to define the semantics of a routing entry. In general, these branches consist of a low number of concepts, which need to be compared. The second factor is the introduction of aggregated routes to the Semantic Routing Tables. Once an aggregated rule representing a corresponding superclass (e. g. $path("human", "person")$ for $path("human", "person", "patient")$) is part of a Semantic Routing Table, routing entries representing individuals of a subclass do not have to be aggregated again and are skipped. On average, the run time of the aggregation is 0.0099 ms with a standard deviation of 0.002 ms, which is roughly 20 %. The overall run time is mainly impacted by the update mechanism (99.5 %). In this case, a route has to be added to the table. Whether it is a new or an aggregated one, the update mechanism has to interact with Clingo to derive the Semantic Routing Table. Since this requires adding, grounding, and solving a Program Section, the run time is higher in comparison to the aggregation mechanism. In general, the run time of the update mechanism scales linearly with the number of routing entries. However, the measurements show three spikes, which are located at 2, 51, and 101 individuals. In these cases, an individual is added to the table that causes an aggregation. Thus, two routes have to be added to the routing table, one that represents the individual and one that models the aggregation, which increases the run time during this step. The measured run times range from roughly 0.9 ms for the first individuals up to 4 ms for the last individuals. The introduction of the last aggregate has the highest run time of 6.16 ms. The standard deviation is 0.4 ms which is roughly 16.5 % of the run time.

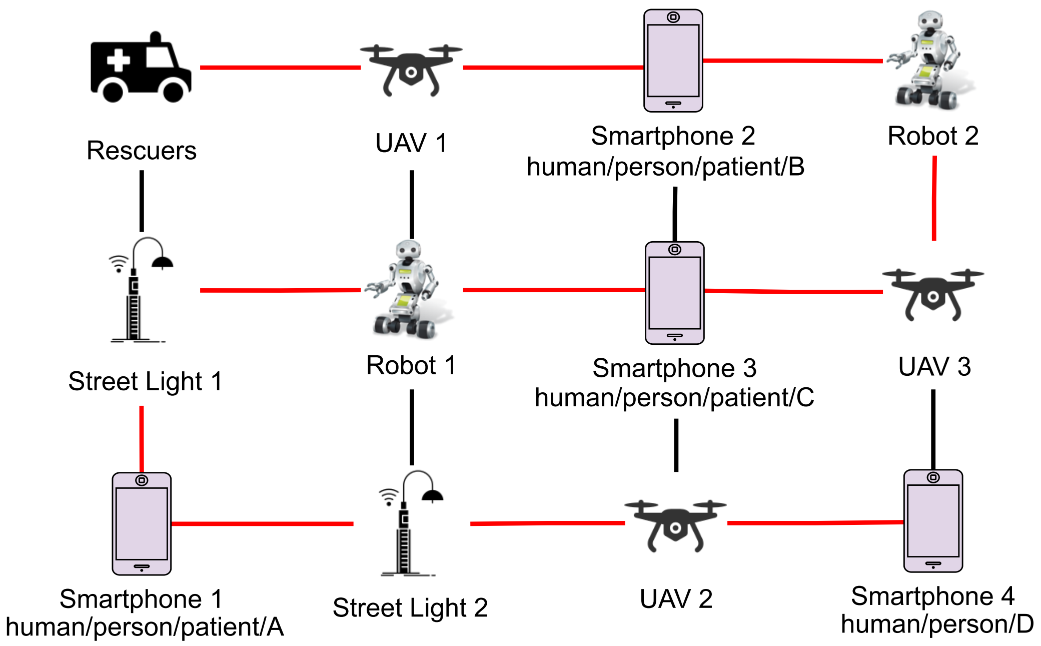


Figure 7.7: Line and Grid Scenario[37]

Another important aspect besides the performance of the Adaptive Semantic Routing on a single node is its overall performance in application scenarios like

Search & Rescue considered by emergenCITY. Our scenarios encompass twelve nodes and four individuals (A - D). Figure 7.7[37] indicates the distribution of nodes and individuals for three organisational schemes. The first scheme is a Line using the red connections. A Circle is created by using the Line scenario and establishing a connection between *Smartphone 4* and the *Rescuers*. The Grid scheme utilises both the red and black connections. The last scenario is the example used in Chapter 6 to describe the components of the Adaptive Semantic Routing. Figure 6.2 shows the scenario, which is basically a combination of lines and a circle. To focus on the performance of the created algorithms and to prevent the influence of network communication, the scenarios are implemented in a single thread application. Furthermore, this evaluation considers eight categories, including the time needed to generate the Semantic Routing Tables on all nodes, a query for a specific individual (C), a query for all patients (A, B, C), a query for all humans (A - D), and the total amount of sent messages for the generation and each query. In all scenarios, *Rescuers* formulate the queries and do not create a routing table themselves. Thus, they simulate a party that interacts with the network of nodes. Table 7.8 presents the average results of 100 test runs. Table 7.9 shows the corresponding standard deviation. The number of messages is stable during the measurements. Thus the corresponding standard deviations are zero and left out in Table 7.9.

	Line	Circle	Test Scenario	Grid
Table Generation [ms]	69.574	70.700	76.737	148.951
Messages [#]	40	40	51	115
Query Individual C [ms]	0.108	0.219	0.150	0.158
Messages [#]	5	11	8	7
Query Patient [ms]	0.247	0.246	0.166	0.284
Messages [#]	11	11	8	12
Query Human [ms]	0.213	0.209	0.143	0.241
Messages [#]	11	11	8	12

Table 7.8: Average Run Time and Number of Messages

	Line	Circle	Test Scenario	Grid
Table Generation [ms]	1.021	2.796	1.074	10.811
Query Individual C [ms]	0.005	0.016	0.009	0.003
Query Patient [ms]	0.013	0.028	0.012	0.004
Query Human [ms]	0.020	0.021	0.010	0.003

Table 7.9: Standard Deviation

The Line and the Circle scenario achieve almost identical results. This is mainly caused by their similar connectivity. On average, each node in the Line is connected

[37]Created with https://app.diagrams.net/ Accessed December 29, 2021.

to 1.83 other nodes and 2.0 nodes in the Circle. Thus, new individuals, new routes, and aggregated routes are only published to a small number of neighbours during the generation of the Semantic Routing Tables. In total, the generation requires 40 messages to create the Semantic Routing Tables on all nodes, which adheres to the worst-case message complexity presented in Table 7.7. Each introduction of an individual is the first of this type on the corresponding node. Thus, an individual or an aggregated route has to be propagated to the remaining nodes. A difference is given in the number of query messages. While the *Rescuers* are connected to a single node in the Line scenario, they are connected to two nodes in the Circle scenario. Thus, they send two queries that traverse both halves of the Circle resulting in eleven messages(five, when contacting *UAV1*; six, when contacting *Smartphone4*). The Test Scenario has a higher run time and requires additional messages for the table generation. While it has similar average connectivity of 2.0, the nodes *Robot1* and *UAV2* are connected to four and three nodes, respectively. Thus, they slightly increase the required messages. However, these nodes accumulate routing entries, which has a beneficial effect on the messages required during queries. The Grid scenario has the highest run times and required messages. On average, each node is connected to 3.1 other nodes. Hence, each new individual or aggregation is propagated to additional nodes resulting in 115 messages. Table 7.9 summarises the standard deviations of the measurements. The standard deviation of the run times is below 2 %, which further underlines the stability of the Adaptive Semantic Routing.

8 Conclusion

In this thesis, we have presented the conceptual foundations and the development of a self-organising and multi-agent-based knowledge base. The research goal has been to provide a distributed knowledge base capable of managing semantically annotated knowledge in loosely coupled networks consisting of heterogeneous participants. Furthermore, it is able to detect and prevent semantic inconsistencies by applying commonsense knowledge extracted from a hypergraph-based knowledge source. Since the managed knowledge is distributed on a network of agents, the presented solution offers an efficient discovery of knowledge based on its semantics.

Section 8.1 summarises the results of this thesis. Subsequently, Section 8.2 revisits the requirements defined in Section 1.1 and discusses their fulfilment. Finally, Section 8.3 concludes this thesis and provides an outlook on future research.

8.1 Summary

The first contribution is the organisation of the distributed knowledge base (*Knowledge Group*) tailored for highly dynamic and loosely coupled domains. The backbone is formed by a Multi-Agent System consisting of agents that act in two roles, *Registry Nodes* and *Registry Leaves*. *Registry Nodes* manage the tree-like structure of the *Knowledge Group* and compensate failures. In contrast, *Registry Leaves* store semantically annotated knowledge and answer queries. Each *Registry Leaf* manages knowledge in its *General Knowledge Store*, which utilises ASP and Clingo to provide an efficient representation of the knowledge. Furthermore, we introduce mechanisms and protocols that replace failed agents, repair the structure of the knowledge base, and enable the exchange of knowledge between several *Knowledge Groups*.

The second contribution of this thesis is the handling of symbolic commonsense knowledge, which is employed by the *Knowledge Groups* to annotate *Knowledge Items* and by the *Adaptive Semantic Routing* to aggregate *Routing Entries*. This contribution is divided into two parts. First, a generation of commonsense ontologies utilising ASP, Clingo, and a hypergraph-based commonsense knowledge source. By incorporating External Statements and Program Sections, we generate ontologies and taxonomies that can be dynamically adapted during run-time. Furthermore, we provide the graphical user interface ARRANGE, which supports a user during the generation and adaption of ontologies as well as the definition of facets. The second part is a mechanism to handle semantic inconsistencies. While commonsense knowledge is essential in our everyday life, its incorporation into a knowledge base can introduce semantic inconsistencies. To prevent these, we evaluate the properties of a concept and generate ASP rules to prevent contradictions. By incorporating the resulting rules into a knowledge base, multiple consistent solutions are created, which can be selected by the user.

The third contribution provides an *Adaptive Semantic Routing* tailored for loosely coupled environments. It is based on *Semantic Routing Tables* that utilise taxonomy branches to aggregate semantically close routing entries. This shifts the focus of the routing from the location (e. g., IP-based) of the managed knowledge to its semantics. By only propagating changes, the *Adaptive Semantic Routing* reduces the number of messages and thus additional stress on loosely coupled networks.

8.2 Requirements Revisited

The handling of knowledge in dynamic environments has several requirements on the used knowledge base. Section 1.1 outlines four main requirements derived from highly dynamic application environment like Search & Rescue scenarios. The presented Self-Organising Multi-Agent Knowledge Base addresses and fulfils these requirements, which is discussed in the following paragraphs.

R1 - Handling Dynamic Environments The distributed knowledge base presented in this thesis utilises a Multi-Agent System to manage its knowledge. Depending on their roles, agents either focus on maintaining the tree-like structure or store semantically annotated knowledge. Furthermore, it relies on a single formalism (ASP) to represent the topology of the network (Network Topology), the storage of general knowledge (General Knowledge Store), and routing entries (Adaptive Semantic Routing). This enables the use of all relevant knowledge without a need to translate between representations. Thus, the Self-Organising Multi-Agent Knowledge Base fulfils this requirement.

R2 - Efficient Management of Knowledge The presented knowledge base has to manage and store semantically annotated knowledge efficiently and decentrally. To fulfil this requirement, we presented the generation of ASP-based ontologies in Section 5.2. They rely on the efficient knowledge representation of ASP (see Section 2.4) and provide a common vocabulary to annotate decentrally stored knowledge. Furthermore, the use of ASP and the multi-shot capabilities of Clingo enable the dynamic adaption of the ontologies during run-time. For example, ontology edges can be removed by changing the truth value of an External Statement without restarting the complete ontology reasoning process. The evaluation results underline the efficiency of the presented approach (see Section 7.2.2). Even when incorporating huge ontologies (over 200.000 edges), both ontology reasoning and classification queries achieve low run times.

R3 - Handling Semantic Inconsistencies The usage of commonsense knowledge is a central aspect of human communication [29]. However, simply providing access to a commonsense knowledge source can introduce semantic inconsistencies to a knowledge base. Thus, our solution has to be able to detect and prevent this kind of inconsistencies. This requirement is fulfilled by the mechanism presented in Section 5.3. It extracts commonsense knowledge from a hypergraph-based source and generates special rules which create several consistent solutions instead of a single inconsistent one. Section 7.2.4 evaluates the run time of the presented approach. The detection of inconsistencies has a high impact on the extraction of commonsense knowledge. Nevertheless, it scales linearly with the number of considered properties and enables the creation of a semantically consistent knowledge base.

R4 - Efficient Knowledge Discovery The last requirement is the efficient discovery of knowledge in a loosely coupled network. To fulfil this requirement, we developed the Adaptive Semantic Routing in Dynamic Environments. It relies on taxonomy branches to annotate *Knowledge Items* or individuals, respectively. Furthermore, it utilises the taxonomy branches to aggregate routing entries and only forwards routing entries to neighbours if *Knowledge Items* with new semantics arrive. The evaluation discussed in Section 7.3 underlines the efficiency of the Adaptive Semantic Routing in Dynamic Environments by comparing its message complexity to similar approaches. Especially the aggregation based on semantics reduces the required messages. Furthermore, the low average run time of the table generation and query resolution underlines its efficiency.

8.3 Future Work

We discovered several challenging research aspects and extensions of the system during the creation of this thesis that deserve further attention but are out of the scope of this thesis. The following paragraphs present a selection of these aspects.

Distributed Access Control The first aspect is the introduction of a distributed access control. So far, the Self-Organising Multi-Agent Knowledge Base does not consider the identity of the agents. Thus, potentially malicious agents could gain access to the knowledge base, harm its structure, influence the routing entries, and alter *Knowledge Items*. An access control mechanism could prevent this. Typically such mechanisms require a central instance either on the network or in the Cloud to manage the access rights. Hence, they are not suited for loosely coupled environments. A possible solution would be a distributed access control. A highly promising distributed access control and team management platform is developed by Jahl in [71]. It utilises a Blockchain to decentrally store access rights to groups, which are granted by the group members themselves by applying voting mechanisms. Thus, the application of this access control mechanism would provide additional security measures without introducing a central component.

Application of ALICA Agents The ALICA framework has been applied in several highly dynamic domains like robotic soccer and autonomous driving. Thus, ALICA could be suited to implement the agents of the Self-Organising Multi-Agent Knowledge Base. However, some aspects of the current ALICA implementation have to be adapted to enable its application. While we have already introduced a discovery module in [108] that is capable of adding unknown agents to an ALICA team, further features have to be added. For example, in its current implementation, the role assignment of agents is fixed and predefined by a configuration file, preventing agents from switching roles and the inclusion of completely unknown agents. Furthermore, ALICA assumes that all agents work in a single team. Thus, agents of a *Knowledge Group* could be assigned for tasks in another *Knowledge Group*. In the case of a *Registry Leaf*, its knowledge would be transferred to the other *Knowledge Group* requiring the creation of new routing tables and the replacement of the missing *Registry Leaf* in its original *Knowledge Group*. Both points, the static role assignment and the single team, are addressed by Jahl in [71]. Thus, the resulting ALICA implementation would provide a sophisticated method to remodel the agents of the Self-Organising Multi-Agent Knowledge Base.

Consistency Mechanisms and Knowledge Priority Classes In its current implementation, the Self-Organising Multi-Agent Knowledge Base only provides eventual consistency of the stored knowledge if it is distributed among several *Knowledge Groups*. This means that duplicates of a *Knowledge Item* are updated over time and that there is no explicit mechanism that guarantees the consistency of all *Knowledge Items* at any time. However, application domains may demand stronger consistency models. Thus, the optional provision of stronger consistency models would increase the applicability of the Self-Organising Multi-Agent Knowledge Base. However, this would increase the overall complexity of the used algorithms and would introduce further messages to achieve consensus. Furthermore, knowledge priority classes could foster the duplicated storage of critical knowledge. A class with lower priority could be recoverable network knowledge, e. g., the *Network Topology* presented in Section 4.2.2 and a class with high priority could, for example, be the location and health status of injured people in Search & Rescue scenarios.

Part IV

Appendices

Publications as (Co-)Author A

[I] Stefan Jakob. 'Where is a Cup and What is it Good for? Crafting an ASP-based Commonsense Knowledgebase for Robotic Agents'. Master Thesis. Distributed Systems Research Group, July 2017. URL: http://das-lab.vs.eecs.uni-kassel.de/publications/Jakob2017-Master-CommonsenseKnowledge.pdf.

[II] Stephan Opfer, Stefan Jakob and Kurt Geihs. 'Reasoning for Autonomous Agents in Dynamic Domains'. In: *9th International Conference on Agents and Artificial Intelligence (ICAART)*. Ed. by Jaap van de Herik, Ana Paula Rocha and Joaquim Filipe. 2017, pp. 340–351. ISBN: 9789897582202.

[III] Stephan Opfer, Stefan Jakob and Kurt Geihs. 'Reasoning for Autonomous Agents in Dynamic Domains: Towards Automatic Satisfaction of the Module Property'. In: *International Conference on Agents and Artificial Intelligence*. 2017, pp. 22–47.

[IV] Harun Baraki, Corvin Schwarzbach, Stefan Jakob, Alexander Jahl and Kurt Geihs. 'SAM: A Semantic-Aware Middleware for Mobile Cloud Computing'. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 375–382.

[V] Harun Baraki, Alexander Jahl, Stefan Jakob, Corvin Schwarzbach, Malte Fax and Kurt Geihs. 'Optimizing Applications for Mobile Cloud Computing through MOCCAA'. In: *Journal of Grid Computing* 17.4 (2019), pp. 651–676.

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

[VII] Stephan Opfer, Stefan Jakob, Alexander Jahl and Kurt Geihs. 'ALICA 2.0-Domain-Independent Teamwork'. In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Ed. by Christoph Benzmüller and Heiner Stuckenschmidt. Springer. Springer International Publishing, 2019, pp. 264–272.

[VIII] Stefan Jakob, Alexander Jahl, Harun Baraki and Kurt Geihs. 'A Self-Organising Multi-Agent Knowledge Base'. In: *2020 IEEE International Conference on Web Services (ICWS)*. IEEE. 2020, pp. 327–329.

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

[X] Alexander Jahl, Stefan Jakob, Harun Baraki, Yasin Alhamwy and Kurt Geihs. 'Blockchain-based Task-centric Team Building'. In: *Proceedings of the ICAART 2021*. Vol. 1. SCITEPRESS, 4th Feb. 2021, pp. 250–257.

[XI] Stefan Jakob, Harun Baraki, Alexander Jahl, Eric Douglas Nyakam Chiadjeu, Yasin Alhamwy and Kurt Geihs. 'Adaptive Semantic Routing in Dynamic Environments'. In: *Proceedings of the ICAART 2021*. Vol. 2. ICAART2021. SCITEPRESS, 4th Feb. 2021, pp. 997–1004.

[XII] Stefan Jakob, Alexander Jahl, Harun Baraki and Kurt Geihs. 'Generating Commonsense Ontologies with Answer Set Programming'. In: *Proceedings of the ICAART 2021*. Vol. 2. ICAART2021. SCITEPRESS, 2021, pp. 538–545.

[XIII] Alexander Jahl, Harun Baraki, Stefan Jakob, Malte Fax and Kurt Geihs. *Machine-learned Behaviour Models for a Distributed Behaviour Discovery.* Manuscript accepted for publication in the Proceedings of the ICAART 2022. 2022.

B Bibliography

[1] Russell L. Ackhoff. ‘From Data to Wisdom’. In: *Journal of Applied Systems Analysis* 16.1 (1989), pp. 3–9 (cit. on pp. 5, 22).

[2] Lama Al Khuzayem and Peter McBrien. ‘OWLRel: Learning Rich Ontologies from Relational Databases’. In: *Baltic Journal of Modern Computing* 4.3 (2016), p. 466 (cit. on p. 55).

[3] Grigoris Antoniou and Frank van Harmelen. ‘Web Ontology Language: OWL’. In: *Handbook on Ontologies.* Springer Berlin Heidelberg, 2004, pp. 67–92 (cit. on pp. 10, 139).

[4] Joaquin Arias, Manuel Carro, Elmer Salazar, Kyle Marple and Gopal Gupta. ‘Constraint Answer Set Programming without Grounding’. In: *Theory and Practice of Logic Programming* 18.3-4 (2018), pp. 337–354 (cit. on p. 53).

[5] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak and Zachary Ives. ‘Dbpedia: A Nucleus for a Web of Open Data’. In: *The Semantic Web.* Springer, 2007, pp. 722–735 (cit. on pp. 45, 51).

[6] Naouel Ayari, Abdelghani Chibani, Yacine Amirat and Eric T. Matson. ‘A Novel Approach Based on Commonsense Knowledge Representation and Reasoning in Open World for Intelligent Ambient Assisted Living Services’. In: *IROS 2015 - IEEE/RSJ International Conference on Intelligent Robots and Systems.* 2015, pp. 6007–6013 (cit. on p. 51).

[7] Franz Baader and Ulrike Sattler. ‘An Overview of Tableau Algorithms for Description Logics’. In: *Studia Logica* 69.1 (2001), pp. 5–40 (cit. on p. 143).

[8] Marcello Balduccini. ‘How Flexible Is Answer Set Programming? An Experiment in Formalizing Commonsense in ASP’. In: *International Conference on Logic Programming and Nonmonotonic Reasoning.* Springer. 2009, pp. 4–16 (cit. on p. 94).

[9] Harun Baraki, Corvin Schwarzbach, Stefan Jakob, Alexander Jahl and Kurt Geihs. ‘SAM: A Semantic-Aware Middleware for Mobile Cloud Computing’. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD).* IEEE. 2018, pp. 375–382 (cit. on p. 47).

[10] Kinjal Basu, Sarat Varanasi, Farhad Shakerin, Joaquin Arias and Gopal Gupta. 'Knowledge-driven Natural Language Understanding of English Text and its Applications'. In: *arXiv Preprint arXiv:2101.11707* (2021) (cit. on p. 53).

[11] Rudolf Bayer and Edward McCreight. 'Organization and Maintenance of Large Ordered Indexes'. In: *Software Pioneers*. Springer, 2002, pp. 245–262 (cit. on p. 68).

[12] Harald Beck. *Reviewing Justification-based Truth Maintenance Systems from a Logic Programming Perspective*. Tech. rep. INFSYS RR-1843-17-02, Institute of Information Systems, TU Vienna, 2017 (cit. on p. 54).

[13] Michael Beetz, Daniel Beßler, Andrei Haidu, Mihai Pomarlan, Asil Kaan Bozcuoğlu and Georg Bartels. 'Know Rob 2.0 - A 2nd Generation Knowledge Processing Framework for Cognition-Enabled Robotic Agents'. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 512–519 (cit. on p. 49).

[14] Armin Biere, Marijn Heule and Hans van Maaren. *Handbook of Satisfiability*. Vol. 185. IOS Press, 2009 (cit. on p. 31).

[15] Jürgen Bock, Rodney Topor and Raphael Volz. 'Ontology Merging Using Answer Set Programming and Linguistic Knowledge'. In: *Proceedings of the 2nd International Conference on Ontology Matching-Volume 304*. CEUR-WS. org. 2007, pp. 321–325 (cit. on p. 94).

[16] Doreen Böhnstedt, Philipp Scholl, Christoph Rensing and Ralf Steinmetz. 'Collaborative Semantic Tagging of Web Resources on the Basis of Individual Knowledge Networks'. In: *International Conference on User Modeling, Adaptation, and Personalization*. Springer. 2009, pp. 379–384 (cit. on p. 52).

[17] Matteo Bonifacio, Paolo Bouquet and Paolo Traverso. 'Enabling Distributed Knowledge Management: Managerial and Technological Implications'. In: *UPGRADE* (Mar. 2002) (cit. on pp. 48, 57, 91).

[18] Matteo Bonifacio, Roberta Cuel, Gianluca Mameli and Michele Nori. *A Peer-to-Peer Architecture for Distributed Knowledge Management*. Tech. rep. Oct. 2002 (cit. on pp. 48, 91).

[19] Taylor L. Booth. *Sequential Machines and Automata Theory*. Wiley, 1967. ISBN: 978-0471088486 (cit. on p. 18).

[20] Carsten Bormann, Angelo P. Castellani and Zach Shelby. 'CoAP: An Application Protocol for Billions of Tiny Internet Nodes'. In: *IEEE Internet Computing* 16.2 (2012), pp. 62–67 (cit. on p. 66).

[21] 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. 22, 23).

[22] 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. 8, 13, 24).

[23] Hans Kleine Büning and Theodor Lettmann.
Propositional Logic: Deduction and Algorithms. Vol. 48.
Cambridge University Press, 1999 (cit. on p. 31).

[24] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca and Torsten Schaub.
'ASP-Core-2 Input Language Format'.
In: *Theory and Practice of Logic Programming* 20.2 (2020), pp. 294–309
(cit. on pp. 24, 54, 117).

[25] Xiaoping Chen, Jianmin Ji, Jiehui Jiang, Guoqiang Jin, Feng Wang and Jiongkun Xie.
'Developing High-level Cognitive Functions for Service Robots'.
In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1.* AAMAS '10. Toronto, Canada, 2010, pp. 989–996. ISBN: 978-0-9826571-1-9 (cit. on p. 51).

[26] Keith L. Clark. 'Negation as Failure'. In: *Logic and Data Bases.*
Springer, 1978, pp. 293–322 (cit. on p. 31).

[27] Stephen A. Cook. 'The Complexity of Theorem-proving Procedures'. In:
Proceedings of the Third Annual ACM Symposium on Theory of Computing.
1971, pp. 151–158 (cit. on p. 31).

[28] George F. Coulouris, Jean Dollimore and Tim Kindberg.
Distributed Systems: Concepts and Design. Pearson Education, 2005
(cit. on p. 69).

[29] Ernest Davis. *Representations of Commonsense Knowledge.*
Morgan Kaufmann, 2014 (cit. on pp. 93, 95, 109, 117, 157).

[30] Ernest Davis and Gary Marcus. 'Commonsense Reasoning and Commonsense Knowledge in Artificial Intelligence'.
In: *Communications of the ACM* 58.9 (2015), pp. 92–103 (cit. on p. 95).

[31] 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 p. 31).

[32] Martin Davis and Hilary Putnam.
'A Computing Procedure for Quantification Theory'.
In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215 (cit. on p. 31).

[33] Ali Davoudian, Liu Chen and Mengchi Liu. 'A Survey on NoSQL Stores'.
In: *ACM Computing Surveys (CSUR)* 51.2 (2018), p. 40 (cit. on pp. 47, 48).

[34] Biplob Debnath, Sudipta Sengupta, Jin Li, David J. Lilja and
David H. C. Du. 'BloomFlash: Bloom Filter on Flash-based Storage'.
In: *2011 31st International Conference on Distributed Computing Systems.*
IEEE. 2011, pp. 635–644 (cit. on p. 58).

[35] Jon Doyle. 'A Truth Maintenance System'.
In: *Artificial Intelligence* 12.3 (1979), pp. 231–272 (cit. on p. 54).

[36] Morris J. Dworkin. 'SHA-3 Standard: Permutation-Based Hash and
Extendable-Output Functions'.
In: *Federal Inf. Process. Stds. (NIST FIPS)* (2015) (cit. on p. 138).

[37] H.-D. Ebbinghaus, Jörg Flum and Wolfgang Thomas. *Mathematical logic.*
Springer Science & Business Media, 2013 (cit. on pp. 27, 39).

[38] Bouchra El Idrissi, Salah Baïna and Karim Baïna.
'Automatic Generation of Ontology from Data Models: A Practical
Evaluation of Existing Approaches'. In: *IEEE 7th International Conference
on Research Challenges in Information Science (RCIS).* IEEE. 2013,
pp. 1–12 (cit. on p. 55).

[39] Esra Erdem, Erdi Aker and Volkan Patoglu.
'Answer Set Programming for Collaborative Housekeeping Robotics:
Representation, Reasoning, and Execution'.
In: *Intelligent Service Robotics* 5.4 (2012), pp. 275–291 (cit. on pp. 50, 94).

[40] Richard Evans, José Hernández-Orallo, Johannes Welbl, Pushmeet Kohli
and Marek Sergot. 'Making Sense of Sensory Input'.
In: *Artificial Intelligence* 293 (2021), p. 103438. ISSN: 0004-3702.
DOI: https://doi.org/10.1016/j.artint.2020.103438.
URL: http://www.sciencedirect.com/science/article/pii/
S0004370220301855 (cit. on p. 54).

[41] Brian Falkenhainer.
'Towards a General-purpose Belief Maintenance System'.
In: *Machine Intelligence and Pattern Recognition.* Vol. 5. Elsevier, 1988,
pp. 125–131 (cit. on p. 54).

[42] John R. Firth. 'A Synopsis of Linguistic Theory, 1930-1955'.
In: *Studies in Linguistic Analysis* (1957) (cit. on p. 46).

[43] J. A. Fodor.
'Modules, Frames, Fridgeons, Sleeping Dogs and the Music of the Spheres'.
In: *The Robot's Dilemma: The Frame Problem in Artificial Intelligence.*
Ed. by Zenon W. Pylyshyn. Ablex, 1987, pp. 139–149 (cit. on p. 24).

[44] Stan Franklin and Art Graesser.
'Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents'.
In: *International Workshop on Agent Theories, Architectures, and Languages*. Springer. 1996, pp. 21–35 (cit. on pp. 13, 14).

[45] Ned Freed and Nathaniel S. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* RFC 2045. http://www.rfc-editor.org/rfc/rfc2045.txt.
RFC Editor, Nov. 1996.
URL: http://www.rfc-editor.org/rfc/rfc2045.txt
(cit. on p. 79).

[46] Ned Freed and Nathaniel S. Borenstein.
Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.
RFC 2046. http://www.rfc-editor.org/rfc/rfc2046.txt.
RFC Editor, Nov. 1996.
URL: http://www.rfc-editor.org/rfc/rfc2046.txt
(cit. on p. 79).

[47] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz and Torsten Schaub. 'Abstract Gringo'.
In: *Theory and Practice of Logic Programming* 15.4-5 (2015), pp. 449–463 (cit. on p. 33).

[48] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub and Sven Thiele. 'Engineering an Incremental ASP Solver'.
In: *International Conference on Logic Programming*. 2008, pp. 190–205 (cit. on pp. 27, 94).

[49] Martin Gebser, Roland Kaminski, Benjamin Kaufmann and Torsten Schaub. *Answer Set Solving in Practice*. Vol. 6.
Morgan & Claypool Publishers, 2012 (cit. on pp. 28, 30, 183).

[50] Martin Gebser, Roland Kaminski, Benjamin Kaufmann and Torsten Schaub. *Clingo = ASP + Control: Extended Report.*
University of Potsdam, 2014 (cit. on pp. 9, 33, 94).

[51] Martin Gebser, Roland Kaminski, Benjamin Kaufmann and Torsten Schaub. 'Multi-shot ASP Solving with Clingo'.
In: *Computing Research Repository* abs/1705.09811 (2017)
(cit. on pp. 38, 182, 183, 185).

[52] Martin Gebser, Roland Kaminski, Arne König and Torsten Schaub.
'Advances in Gringo Series 3'. In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer. 2011, pp. 345–351 (cit. on p. 28).

[53] Martin Gebser, Roland Kaminski and Torsten Schaub. 'Complex Optimization in Answer Set Programming'. In: *27th International Conference on Logic Programming.* Ed. by John Gallagher and Michael Gelfond. Vol. 11. 2011, pp. 821–839 (cit. on p. 94).

[54] Martin Gebser, Benjamin Kaufmann and Torsten Schaub. 'Conflict-driven Answer Set Solving: From Theory to Practice'. In: *Artificial Intelligence* 187 (2012), pp. 52–89 (cit. on pp. 31–33).

[55] Martin Gebser, Benjamin Kaufmann and Torsten Schaub. 'Multi-threaded ASP Solving with Clasp'. In: *Theory and Practice of Logic Programming* 12.4-5 (2012), pp. 525–545 (cit. on p. 33).

[56] Martin Gebser, Torsten Schaub, Sven Thiele and Philippe Veber. 'Detecting Inconsistencies in Large Biological Networks With Answer Set Programming'. In: *Theory and Practice of Logic Programming* 11.2-3 (2011), pp. 323–360 (cit. on pp. 52, 118).

[57] 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. 10, 24).

[58] Eugene Goldberg and Yakov Novikov. 'BerkMin: A Fast and Robust SAT-solver'. In: *Discrete Applied Mathematics* 155.12 (2007), pp. 1549–1561 (cit. on p. 33).

[59] Claudia Gómez Santillán, Laura Cruz Reyes, Eustorgio Meza Conde, Elisa Schaeffer and Guadalupe Castilla Valdez. 'A Self-Adaptive Ant Colony System for Semantic Query Routing Problem in P2P Networks'. In: *Computación y Sistemas* 13.4 (2010), pp. 433–448 (cit. on pp. 57, 131, 147–149).

[60] Ricardo Gonçalves, Tomi Janhunen, Matthias Knorr, Joao Leite and Stefan Woltran. 'Forgetting in Modular Answer Set Programming'. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 33. 01. 2019, pp. 2843–2850 (cit. on p. 53).

[61] Ricardo Gonçalves, Matthias Knorr and Joao Leite. 'You Can't Always Forget What You Want: On the Limits of Forgetting in Answer Set Programming.' In: *European Conference on Artificial Intelligence (ECAI).* Frontiers in Artificial Intelligence and Applications. 2016, pp. 957–965 (cit. on pp. 52, 53).

[62] Li Gong. ‘JXTA: A Network Programming Environment’. In: *IEEE Internet Computing* 5.3 (2001), pp. 88–95 (cit. on p. 48).

[63] Mark Gritter and David R. Cheriton. ‘An Architecture for Content Routing Support in the Internet.’ In: *Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems*. Vol. 1. 2001, pp. 4–4 (cit. on pp. 58, 148).

[64] Thomas R. Gruber. ‘A Translation Approach to Portable Ontology Specifications’. In: *Knowledge Acquisition* 5.2 (1993), pp. 199–221 (cit. on pp. 40, 54).

[65] Thomas R. Gruber. ‘Toward Principles for the Design of Ontologies used for Knowledge Sharing?’ In: *International Journal of Human-computer Studies* 43.5-6 (1995), pp. 907–928 (cit. on p. 40).

[66] Fredrik Heintz, Jonas Kvarnström and Patrick Doherty. ‘Bridging the Sense-Reasoning Gap: DyKnow – Stream-based Middleware for Knowledge Processing’. In: *Advanced Engineering Informatics* 24.1 (2010), pp. 14–26. ISSN: 1474-0346 (cit. on p. 49).

[67] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, Sebastian Rudolph et al. ‘OWL 2 Web Ontology Language Primer’. In: *W3C Recommendation* 27.1 (2009), p. 123 (cit. on p. 94).

[68] Pascal Hitzler, Markus Krotzsch and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. CRC Press, 2009 (cit. on pp. 42, 142).

[69] Roberto Ierusalimschy, Luiz Henrique De Figueiredo and Waldemar Celes Filho. ‘Lua - An Extensible Extension Language’. In: *Software: Practice and Experience* 26.6 (1996), pp. 635–652 (cit. on p. 34).

[70] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs and Rebecca L. Braynard. ‘Networking Named Content’. In: *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*. 2009, pp. 1–12 (cit. on pp. 58, 147, 148).

[71] Alexander Jahl. *Situative Teams in Heterogeneous Multi-Agent Systems.* Ongoing PhD Thesis (cit. on pp. 22, 157, 158).

[72] Alexander Jahl, Harun Baraki, Stefan Jakob, Malte Fax and Kurt Geihs. *Machine-learned Behaviour Models for a Distributed Behaviour Discovery.* Manuscript accepted for publication in the Proceedings of the ICAART 2022. 2022 (cit. on pp. 47, 65, 67, 83).

[73] Alexander Jahl, Stefan Jakob, Harun Baraki, Yasin Alhamwy and Kurt Geihs. 'Blockchain-based Task-centric Team Building'. In: *Proceedings of the ICAART 2021*. Vol. 1. SCITEPRESS, 4th Feb. 2021, pp. 250–257 (cit. on p. 47).

[74] Stefan Jakob. 'Where is a Cup and What is it Good for? Crafting an ASP-based Commonsense Knowledgebase for Robotic Agents'. Master Thesis. Distributed Systems Research Group, July 2017. URL: `http://das-lab.vs.eecs.uni-kassel.de/publications/Jakob2017-Master-CommonsenseKnowledge.pdf` (cit. on p. 47).

[75] Stefan Jakob, Harun Baraki, Alexander Jahl, Eric Douglas Nyakam Chiadjeu, Yasin Alhamwy and Kurt Geihs. 'Adaptive Semantic Routing in Dynamic Environments'. In: *Proceedings of the ICAART 2021*. Vol. 2. ICAART2021. SCITEPRESS, 4th Feb. 2021, pp. 997–1004 (cit. on pp. 47, 53, 57, 58, 64, 119–122, 129, 130, 146, 181).

[76] Stefan Jakob, Alexander Jahl, Harun Baraki and Kurt Geihs. 'A Self-Organising Multi-Agent Knowledge Base'. In: *2020 IEEE International Conference on Web Services (ICWS)*. IEEE. 2020, pp. 327–329 (cit. on pp. 47, 54, 65, 68, 82, 83, 86, 181).

[77] Stefan Jakob, Alexander Jahl, Harun Baraki and Kurt Geihs. 'Generating Commonsense Ontologies with Answer Set Programming'. In: *Proceedings of the ICAART 2021*. Vol. 2. ICAART2021. SCITEPRESS, 2021, pp. 538–545 (cit. on pp. 47, 55, 56, 96, 97, 99, 101, 104–107, 139, 140, 143, 181, 183, 187).

[78] Stefan Jakob, Stephan Opfer, Alexander Jahl, Harun Baraki and Kurt Geihs. 'Handling Semantic Inconsistencies in Commonsense Knowledge for Autonomous Service Robots'. In: *2020 IEEE 14th International Conference on Semantic Computing (ICSC)*. IEEE. 2020, pp. 136–140 (cit. on pp. 47, 50–53, 64, 109, 110, 113, 114, 139, 144, 145, 181, 183).

[79] Benjamin Kaufmann, Nicola Leone, Simona Perri and Torsten Schaub. 'Grounding and Solving in Answer Set Programming'. In: *AI Magazine* 37.3 (2016), pp. 25–32 (cit. on pp. 28, 29).

[80] Jeffrey O. Kephart and David M. Chess. 'The Vision of Autonomic Computing'. In: *Computer* 36.1 (2003), pp. 41–50 (cit. on pp. 14, 15).

[81] Yip Chi Kiong, Sellappan Palaniappan and Nor Adnan Yahaya. 'Health Ontology Generator: Design And Implementation'. In: *International Journal of Computer Science and Network Security* 9.2 (2009), p. 104 (cit. on pp. 55, 117).

[82] Karin Kipper, Anna Korhonen, Neville Ryant and Martha Palmer. 'A Large-Scale Classification of English Verbs'. In: *Language Resources and Evaluation* 42.1 (2008), pp. 21–40 (cit. on p. 53).

[83] Georgia Koloniari and Evaggelia Pitoura. 'Content-based Routing of Path Queries in Peer-to-Peer Systems'. In: *International Conference on Extending Database Technology.* Springer. 2004, pp. 29–47 (cit. on pp. 58, 148, 149).

[84] Markus Krötzsch. 'Ontologies for Knowledge Graphs?' In: *30th Int. Workshop on Description Logics.* Vol. 1879. CEUR-WS.org, 2017 (cit. on pp. 8, 93, 95, 116).

[85] 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 Robotics* 4.2 (2012), pp. 181–199 (cit. on p. 51).

[86] 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 (cit. on p. 50).

[87] Vladimir Lifschitz. *Answer Set Programming.* Springer International Publishing, 2019 (cit. on p. 8).

[88] V. Manfredi, R. Ramanathan, W. Tetteh, R. Hain and D. Ryder. 'SHARE: Scalable Hybrid Adaptive Routing for Dynamic Multi-hop Environments'. In: *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI).* 2017, pp. 1–8. DOI: 10.1109/UIC-ATC.2017.8397491 (cit. on pp. 58, 131, 148, 149).

[89] Joao Marques-Silva, Inês Lynce and Sharad Malik. 'Conflict-driven Clause Learning SAT Solvers'. In: *Handbook of Satisfiability.* ios Press, 2009, pp. 131–153 (cit. on p. 32).

[90] Petar Maymounkov and David Mazieres. 'Kademlia: A Peer-to-Peer Information System Based on the XOR Metric'. In: *International Workshop on Peer-to-Peer Systems.* Springer. 2002, pp. 53–65 (cit. on pp. 59, 149).

[91] John McCarthy and Patrick J. Hayes. 'Some Philosophical Problems from the Standpoint of Artificial Intelligence'. In: *Readings in Artificial Intelligence.* Elsevier, 1981, pp. 431–450 (cit. on p. 24).

[92] Deborah L. McGuinness, Frank Van Harmelen et al. ‘OWL Web Ontology Language Overview’. In: *W3C Recommendation* 10.10 (2004), p. 2004 (cit. on pp. 42, 94).

[93] Elke Michlmayr, Arno Pany and Gerti Kappel. ‘Using Taxonomies for Content-based Routing with Ants’. In: *Computer Networks* 51.16 (2007), pp. 4514–4528 (cit. on pp. 57, 147–149).

[94] George A. Miller. ‘WordNet: A Lexical Database for English’. In: *Communications of the ACM* 38.11 (1995), pp. 39–41 (cit. on pp. 45, 51, 97).

[95] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang and Sharad Malik. ‘Chaff: Engineering an Efficient SAT Solver’. In: *Proceedings of the 38th Annual Design Automation Conference*. 2001, pp. 530–535 (cit. on p. 33).

[96] 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. 2006, pp. 501–514. ISBN: 978-3-540-49029-6 (cit. on pp. 49, 95).

[97] Hamid Mousavi, Deirdre Kerr, Markus Iseli and Carlo Zaniolo. *Ontoharvester: An Unsupervised Ontology Generator from Free Text.* Tech. rep. InCSD TechnicalReport 130003, UCLA, 2013 (cit. on pp. 55, 117).

[98] Mithun Mukherjee, Lei Shu and Di Wang. ‘Survey of Fog Computing: Fundamental, Network Applications, and Research Challenges’. In: *IEEE Communications Surveys & Tutorials* 20.3 (2018), pp. 1826–1857 (cit. on p. 7).

[99] Mark A. Musen. ‘The Protégé Project: A Look Back and a Look Forward’. In: *AI Matters* 1.4 (2015), pp. 4–12 (cit. on p. 142).

[100] N. Noy and Deborah Mcguinness. ‘Ontology Development 101: A Guide to Creating Your First Ontology’. In: *Knowledge Systems Laboratory* 32 (Jan. 2001) (cit. on pp. 40, 41).

[101] 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 (cit. on p. 17).

[102] Jack O’Connor, Jean-Philippe Aumasson, Samuel Neves and Zooko Wilcox-O’Hearn. *BLAKE3 - One Function, Fast Everywhere.* 2020. URL: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf (cit. on p. 138).

[103] Emilia Oikarinen and Tomi Janhunen.
'Modular Equivalence for Normal Logic Programs'. In: vol. 141. Jan. 2006, pp. 412–416 (cit. on pp. 38, 39).

[104] Stephan Opfer.
'Symbolic Representation of Dynamic Knowledge for Robotic Teams'. PhD thesis. Kassel, University of Kassel, Faculty of Electrical Engineering and Computer Science, Nov. 2020
(cit. on pp. 17, 18, 20, 21, 50, 51, 96, 109, 112).

[105] Stephan Opfer, Stefan Jakob and Kurt Geihs.
'Reasoning for Autonomous Agents in Dynamic Domains'. In: *9th International Conference on Agents and Artificial Intelligence (ICAART).* Ed. by Jaap van de Herik, Ana Paula Rocha and Joaquim Filipe. 2017, pp. 340–351. ISBN: 9789897582202 (cit. on pp. 39, 47, 94).

[106] Stephan Opfer, Stefan Jakob and Kurt Geihs.
'Reasoning for Autonomous Agents in Dynamic Domains: Towards Automatic Satisfaction of the Module Property'. In: *International Conference on Agents and Artificial Intelligence (ICAART).* 2017, pp. 22–47 (cit. on pp. 39, 47, 94).

[107] Stephan Opfer, Stefan Jakob and Kurt Geihs.
'Teaching Commonsense and Dynamic Knowledge to Service Robots'. In: *International Conference on Social Robotics.* Springer. 2019, pp. 645–654 (cit. on pp. 47, 50–53, 109–112, 181).

[108] Stephan Opfer, Stefan Jakob, Alexander Jahl and Kurt Geihs.
'ALICA 2.0-Domain-Independent Teamwork'. In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz).* Ed. by Christoph Benzmüller and Heiner Stuckenschmidt. Springer. Springer International Publishing, 2019, pp. 264–272 (cit. on pp. 17, 18, 20, 21, 47, 158).

[109] Lasse Orda, Tue Jensen, Oliver Gehrke and Henrik Bindner.
'Efficient Routing for Overlay Networks in a Smart Grid Context'. In: *Proceedings of the 8th International Conference on Smart Cities and Green ICT Systems - SMARTGREENS,* INSTICC. SciTePress, 2019, pp. 251–258. ISBN: 978-989-758-373-5. DOI: 10.5220/0007717702510258 (cit. on pp. 59, 148, 149).

[110] Heiko Paulheim. 'Knowledge Graph Refinement: A Survey of Approaches and Evaluation Methods'. In: *Semantic Web* 8.3 (2017), pp. 489–508 (cit. on p. 56).

[111] Luca Pireddu and Mario A. Nascimento.
'Taxonomy-Based Routing Indices for Peer-to-Peer Networks'. In: *Workshop on Peer-to-Peer Information Retrieval.* 2004 (cit. on pp. 57, 131, 148, 149).

[112] Jiahu Qin, Qichao Ma, Yang Shi and Long Wang.
'Recent Advances in Consensus of Multi-agent Systems: A Brief Survey'.
In: *IEEE Transactions on Industrial Electronics* 64.6 (2016), pp. 4972–4983 (cit. on p. 17).

[113] Anand S. Rao, Michael P. Georgeff et al.
'BDI Agents: From Theory to Practice'.
In: *Proceedings of the First International Conference on Multiagent Systems.* Vol. 95. 1995, pp. 312–319 (cit. on pp. 15, 16).

[114] Christoph Redl.
'Conflict-driven ASP Solving with External Sources and Program Splits'.
In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17.* 2017, pp. 1239–1246.
DOI: 10.24963/ijcai.2017/172.
URL: https://doi.org/10.24963/ijcai.2017/172 (cit. on p. 53).

[115] Christoph Redl.
'Explaining Inconsistency in Answer Set Programs and Extensions'.
In: *International Conference on Logic Programming and Nonmonotonic Reasoning.* Springer. 2017, pp. 176–190 (cit. on p. 53).

[116] Raymond Reiter. 'On Closed World Data Bases'.
In: *Readings in Artificial Intelligence.* Elsevier, 1981, pp. 119–140 (cit. on pp. 23, 56).

[117] Raymond Reiter.
'Towards a Logical Reconstruction of Relational Database Theory'.
In: *On Conceptual Modelling.* Springer, 1984, pp. 191–238 (cit. on pp. 23, 56).

[118] Francesco Ricca, Lorenzo Gallucci, Roman Schindlauer, Tina Dell'Armi, Giovanni Grasso and Nicola Leone.
'OntoDLV: An ASP-based System for Enterprise Ontologies'.
In: *Journal of Logic and Computation* 19.4 (2009), pp. 643–670 (cit. on pp. 56, 117).

[119] Guido Rossum. *Python Reference Manual.*
CWI (Centre for Mathematics and Computer Science), 1995 (cit. on p. 34).

[120] Jennifer Rowley.
'The Wisdom Hierarchy: Representations of the DIKW Hierarchy'.
In: *Journal of Information Science* 33.2 (2007), pp. 163–180 (cit. on p. 22).

[121] Stuart Russell and Peter Norvig.
Artificial Intelligence: A Modern Approach. Pearson, 2002 (cit. on pp. 13, 14).

[122] 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 pp. 29, 38).

[123] Rob Shearer, Boris Motik and Ian Horrocks. ‘HermiT: A Highly-Efficient OWL Reasoner’. In: *Owled*. Vol. 432. 2008, p. 91 (cit. on p. 142).

[124] Tiago Outerelo da Silva, Fernanda Araujo Baião and Kate Revoredo. ‘OntoDW: An Approach for Extraction of Conceptualizations from Data Warehouses’. In: *ONTOBRAS*. 2016, pp. 83–94 (cit. on p. 55).

[125] Push Singh, Thomas Lin, Erik T. Mueller, Grace Lim, Travell Perkins and Wan Li Zhu. ‘Open Mind Common Sense: Knowledge Acquisition from the General Public’. In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2002, pp. 1223–1237 (cit. on p. 45).

[126] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur and Yarden Katz. ‘Pellet: A Practical OWL-DL Reasoner’. In: *Journal of Web Semantics* 5.2 (2007), pp. 51–53 (cit. on p. 50).

[127] Hendrik Skubch. ‘Modelling and Controlling Behaviour of Cooperative Autonomous Mobile Robots’. Dissertation. Kassel: University of Kassel, 2012 (cit. on pp. 17, 18, 21).

[128] 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-17)*. 2017, pp. 4444–4451 (cit. on pp. 9, 44–46, 119).

[129] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications*. Vol. 3485. Springer, 2005 (cit. on p. 57).

[130] Moritz Tenorth and Michael Beetz. ‘KnowRob: A Knowledge Processing Infrastructure for Cognition-Enabled Robots’. In: *International Journal of Robotics Research* 32.5 (2013), pp. 566–590. ISSN: 0278-3649 (cit. on p. 49).

[131] Moritz Tenorth and Michael Beetz. ‘Representations for Robot Knowledge in the KnowRob Framework’. In: *Artificial Intelligence* 247 (2015), pp. 151–169 (cit. on p. 49).

[132] Luis M. Vaquero and Luis Rodero-Merino. ‘Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing’. In: *ACM SIGCOMM Computer Communication Review* 44.5 (2014), pp. 27–32 (cit. on p. 7).

[133] Kari Visala, Dmitrij Lagutin and Sasu Tarkoma.
'LANES: An Inter-Domain Data-Oriented Routing Architecture'.
In: *Proceedings of the 2009 Workshop on Re-architecting the Internet*. 2009, pp. 55–60 (cit. on p. 57).

[134] Jun Jie Woo.
'Technology and Governance in Singapore's Smart Nation Initiative'.
In: *Ash Center Policy Briefs Series* (2018) (cit. on p. 3).

[135] Michael Wooldridge. *An Introduction to Multiagent Systems*.
John Wiley & Sons, 2009 (cit. on pp. 13, 14, 17, 40, 54).

[136] Michael Wooldridge and Nicholas R. Jennings.
'Intelligent Agents: Theory and Practice'.
In: *The Knowledge Engineering Review* 10.2 (1995), pp. 115–152 (cit. on p. 14).

[137] Hong-Jie Xing and Wei-Tao Liu.
'Robust AdaBoost Based Ensemble of One-class Support Vector Machines'.
In: *Information Fusion* 55 (2020), pp. 45–58 (cit. on p. 65).

[138] Gian Piero Zarri. 'NKRL, A Kowledge Representation Tool for Encoding the Meaning of Complex Narrative Texts'.
In: *Natural Language Engineering* 3.2 (1997), pp. 231–253 (cit. on p. 51).

[139] Hao Zhang, Yonggang Wen, Haiyong Xie and Nenghai Yu.
Distributed Hash Table: Theory, Platforms and Applications. Springer, 2013 (cit. on p. 119).

[140] Shufeng Zhou, Haiyun Ling, Mei Han and Huaiwei Zhang.
'Ontology Generator from Relational Database Based on Jena'.
In: *Computer and Information Science* 3.2 (2010), pp. 263–267 (cit. on p. 55).

List of Figures

C

D

List of Tables

E

List of Listings

F List of Algorithms

G Multi-Shot Solving

```
#program instance.
peg(a;b;c).
disk(1..4).
init_on(1..4,a).
goal_on(1..4,c).
on(D,P,0) :- init_on(D,P).
#show move/3.

#program step(t).
1 { move(D,P,t) : disk(D), peg(P) } 1.
move(D,t) :- move(D,P,t).
on(D,P,t) :- move(D,P,t).
on(D,P,t) :- on(D,P,t-1), not move(D,t).
blocked(D-1,P,t) :- on(D,P,t-1).
blocked(D-1,P,t) :- blocked(D,P,t), disk(D).
:- move(D,P,t), blocked(D-1,P,t).
:- move(D,t), on(D,P,t-1), blocked(D,P,t).
:- disk(D), not 1 { on(D,P,t) } 1.

#program check(t).
#external query(t).
:- goal_on(D,P), not on(D,P,t), query(t).
```

Listing G.1: Multi-Shot Encoding for the Towers of Hanoi [51]

Algorithm G.1: Iteratively Solving the Tower of Hanoi Game [51]

Input : ASP Program of Listing G.1
Output: Sequence of Moves to Solve the Tower of Hanoi Game

```
1  iteration = 0
2  ground(instance)
3  ground(check(iteration))
4  assignExternal(query(iteration), true)
5  solution = solve()
6  if $solution \neq \emptyset$ then return solution
7  while $solution == \emptyset$ do
8      releaseExternal(iteration)
9      iteration++
10     ground(step(iteration))
11     ground(check(iteration))
12     assignExternal(query(iteration), true)
13     solution = solve()
14     if $solution \neq \emptyset$ then return solution
```

H Ontology Solving Run Time

Concept	Animal			Car			Person			Thing		
Min. Weight	2.5	2.0	1.0	2.5	2.0	1.0	2.5	2.0	1.0	2.5	2.0	1.0
Add Ont.	98.69	15730.43	36299.99	18.44	15694.34	36939.30	9.34	15616.37	37003.04	47.57	15716.67	37020.90
Ground Ont.	78.86	14800.16	31524.86	12.02	14826.59	31604.30	11.42	14679.48	31721.09	30.51	14769.02	31823.20
Assign Ext.	1.39	256.51	523.85	0.12	247.62	524.86	0.15	247.63	520.87	0.58	262.93	521.74
Solve Ont.	3.86	1010.83	2257.52	0.53	1012.63	2270.44	0.54	1000.27	2272.28	1.47	1005.21	2271.59
Add Query	11.59	11.76	12.67	13.58	11.74	12.96	14.49	11.71	13.07	10.96	11.75	13.05
Ground Query	78.45	9299.59	19320.29	30.47	9318.18	19331.80	28.49	9331.43	19566.84	46.33	9350.11	19525.80
Solve Query	3.45	743.13	1635.60	0.38	744.79	1645.42	0.29	735.81	1650.13	1.06	740.22	1637.25

Table H.1: Run Time of Ontology Solving and Classification Query in [ms] [77]

Concept	Animal			Car			Person			Thing		
Min. Weight	2.5	2.0	1.0	2.5	2.0	1.0	2.5	2.0	1.0	2.5	2.0	1.0
Add Ont.	2.67	213.29	333.37	4.90	200.48	514.48	3.57	267.64	524.11	5.45	26.11	511.37
Ground Ont.	0.42	67.05	301.63	1.62	70.88	262.49	1.57	69.14	266.10	0.18	58.27	402.30
Assign Ext.	0.078	2.07	9.02	0.01	1.69	3.55	0.03	1.88	2.95	0.01	1.04	4.27
Solve Ont.	0.057	8.84	33.39	0.05	11.30	41.25	0.09	8.25	35.21	0.01	7.33	42.45
Add Query	0.15	0.02	0.03	0.67	0.02	0.03	1.03	0.09	0.48	1.02	0.07	0.48
Ground Query	0.44	23.84	188.46	0.41	19.68	193.47	1.18	18.63	320.37	0.34	16.20	187.17
Solve Query	0.10	7.18	40.33	0.08	8.69	35.67	0.02	7.73	40.42	0.04	4.77	41.77

Table H.2: Standard Deviation in [ms] [77]