Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at
The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

Modelling the Human Memorization Process in an Autonomous Agent

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Verena Himmelbauer
Matrikelnummer 0725523

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: em. Prof. Dr. Dietmar Dietrich
Mitwirkung: Dipl. Ing. Alexander Wendt

Wien, 16.07.2015

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Modelling the Human Memorization Process in an Autonomous Agent

MASTER´S THESIS

in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Verena Himmelbauer
Registration Number 0725523

to the Faculty of Informatics
at the the Vienna University of Technology

Advisor: em. Prof. Dr. Dietmar Dietrich
Assistance: Dipl. Ing. Alexander Wendt

Wien, 16.507.2015

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Abstract

With growing capabilities of nowadays technical equipment the amount of data that has to be processed is constantly increasing. Conventional systems that are in use until now will be reaching their limits when it comes to handling the increasing flood of data in a reasonable amount of time. Therefore new approaches have to be explored in order to provide solutions for future systems. State of the art solutions with static programming flows are going to be less and less efficient while exponentially rising in complexity, whereas classical Artificial Intelligence has not yet provided sufficient solutions to the problem of vast amounts of data. Those systems that are currently successfully used are limited to very small application areas dealing with small and separated tasks and information. In contrast to artificial systems the human mind is perfectly able to comprehend complex situations and process vast amounts of data that is associated to them in almost no time. Therefore a new bionic model for human-like perception based on psychoanalytical theories is followed in this work. Like for a human mind it is necessary that the systems of the future adapt to newly arising situations and also to perform better if the same situation arises again. The existing cognitive architecture is therefore supplemented by a new permanent store in order to enable the system to store and use experiences over several simulation runs. Furthermore the existing file-based declarative semantic memory was parsed into the new database. Finally a first episodic memorization process is implemented, which will allow the agent in the future to use prior experiences in its decision process.

Kurzfassung

Mit steigender Leistungsfähigkeit neuer Technologien wachsen gleichzeitig die Anforderungen an die Leistungsfähigkeit der modernen Datenverarbeitung. Konventionelle Systeme, wie sie derzeit in Verwendung sind, werden in Zukunft an ihre Grenzen stoßen, wenn es darum geht, den zunehmenden Strom an Daten in einer vertretbaren Zeit zu verarbeiten. Daher ist es nötig, neue Lösungsansätze zu erforschen, um den Anforderungen der Zukunft gewachsen zu sein. Die Effizienz statischer Programmabläufe nimmt immer mehr ab, während gleichzeitig die Komplexität exponentiell zu steigen droht. Die klassische künstliche Intelligenz war bisher nicht in der Lage, eine effiziente Lösung für die Verarbeitung großer Datenmengen zu finden und jene Systeme, die heutzutage erfolgreich eingesetzt werden, beschränken sich daher auf begrenzte Anwendungsgebiete mit kleinen abgekapselten Aufgabenbereichen und Informationsinputs. Im Gegensatz zu computergesteuerten Systemen ist der menschliche Wahrnehmungsapparat perfekt in der Lage, komplexe Situationen zu analysieren und sich an diese in kürzester Zeit anzupassen.

Aus diesem Grund beschäftigt sich diese Arbeit mit einem bionisch inspirierten Ansatz von menschenähnlicher Wahrnehmung, basierend auf Theorien der Psychoanalyse. Wie der Mensch auch, müssen Systeme der Zukunft in der Lage sein, flexibel auf neu aufkommende Situationen zu reagieren. Gleichzeitig ist es erforderlich, dass bei wiederholtem Auftreten eines bereits gelösten Problems diese Aufgabe schneller und effizienter gelöst würde als zuvor.

Die bereits existierende kognitive Architektur wurde daher im Verlauf dieser Arbeit um einen permanenten Datenspeicher ergänzt, um das System in die Lage zu versetzen, Erinnerungen über mehrere Simulationsläufe zu speichern und zu verwerten. Weiters wurde das existierende datei-basierte deklarative semantische Gedächtnis in die neue Datenbank transferiert. In einem letzten Schritt wurde ein Prozess zum Speichern episodischer Erinnerungen in das System implementiert. Diese Umstellungen werden dem Agenten in der Zukunft ermöglichen seine bereits gemachten Erfahrungen zur Verbesserung seines Entscheidungsprozesses zu benutzen.

Acknowledgements

First of all I want to thank my advisor em. Prof. Dr. Dietmar Dietrich for supervising my work despite already being in his well-deserved retirement. Further, I would like to thank my second advisor Dipl. Ing. Alexander Wendt for sharing his experience and insights on the project with me. He also distinguished himself by spreading optimism and good spirits when things looked pretty grim.

I would also like to thank my friend Jürgen for his constant encouragement and backup during the whole course of this work. For the hours of discussion on programming topics and your advice. Your support was always noticed and appreciated. Thanks for putting up with my moods, old friend!

Further thanks go to my family for believing in me during the course of my studies. To my siblings for pointing out that some of my sentences might be hard to read and even harder to understand. To my father who is to blame for me ending up with computers instead of something I cannot imagine today and to my mother for setting an example of how to manage a family besides being successful in work and in life.

Finally, I want to thank my partner for his patience and support during the hardships of this work. You were not only a mental aid, but a well-respected discussion partner on diverse topics that came up during the course of this work. Thank you for pushing me forward and for sharing my concerns and thoughts during all this time.

Table of contents

1. Introduction 1
   1.1 Background 1
   1.2 Motivation 2
   1.3 Problem Statement 4
   1.4 Task Description 6
   1.5 Methodology 7
2. Related Work and State of the Art 9
   2.1 SiMA - Simulation of the Mental Apparatus & Applications 9
   2.2 Data Storage Technologies 13
      2.2.1 Relational Databases 14
      2.2.2 NoSQL 16
   2.3 Memory Approaches in other Cognitive Architectures 18
      2.3.1 CHREST 20
      2.3.2 ICARUS 23
      2.3.3 ACT-R – Adaptive Control of Thought Rational 27
      2.3.4 SOAR – State Operator Apply Result 30
      2.3.5 BDI – Belief, Desire, Intention 33
3. Database Selection & Integration 37
   3.1 Database Selection 37
   3.2 Resource Description Framework 40
      3.2.1 RDF Data Model and Terminology 40
      3.2.2 RDF Schema 41
      3.2.3 Querying with SparQL 43
      3.2.4 N-Ary Relations 45
   3.3 Implementation of Database Integration 47
      3.3.1 Project Structure 47
      3.3.2 Database Integration 49
4. Data Structure Conversion and Migration 52
   4.1 Automatic Migration to RDF 52
   4.2 Alternative Migration Concept 55
      4.2.1 Primary Process Data Structures 57
      4.2.2 Secondary Process Data Structures 63
      4.2.3 Associations 65
   4.3 Conversion Implementation 67
      4.3.1 Manual Conversion 67

4.3.2 Search Space Migration and Memory Access 70
5. Memorization Process 73
5.1 Concepts of the Memorization Process 73
5.2 Memorizing Experiences 75
5.2.1 Memorization Trigger 76
5.2.2 Filtering the Memory 76
5.2.3 Creating an act 77
5.3 Implementation of the Episodic Memory 79
5.3.1 Memorizing Experiences 79
6. Simulation 82
6.1 Test Environment 82
6.2 Test Case 1 – Declarative Lexical Memory 84
6.2.1 Simulation run with original memory 85
6.2.2 Simulation run with RDF database 87
6.2.3 Results 89
6.3 Test Case 2 – Episodic Memory 89
6.3.1 Simulation run with active memorization process 90
6.3.2 Results 94
7. Conclusion and Future Work 96
7.1 Conclusion 96
7.2 Future Work 98
Literature 104
Internet references 107

Abbreviations

ACT-R	Adaptive Control of Thought-Rational
ARS	Artificial Recognition System
ARS-PA	ARS-Psychoanalysis
ARS-PC	ARS-Perception
DB	Database
DM	Drive Mesh
LIDA	Learning Intelligent Distribution Agent
OWL	Web Ontology Language
RDF	Resource Description Framework
SmaKi	Smart Kitchen
SOAR	State, Operator Apply Result
TI	Template Image
TP	Thing Presentation
TPM	Thing Presentation Mesh
WP	Word Presentation
WPM	Word Presentation Mesh
XML	Extensible Markup Language

1. Introduction

With growing capabilities in sensor systems, the requirements for building automation are constantly increasing. In the future, one can expect such systems to be equipped with hundreds of thousands of sensors in order to analyse complex situations. To successfully accomplish analyses of this kind, huge amounts of data have to be processed and interpreted. Such an interpretation, however, demands more than the current approach of simply following pre-set procedures that depend on certain sensory input data. Computer science has explored various options to address these newly arising challenges. One of these options is the Simulation of the Mental Apparatus & Applications (SiMA) project [SIMAHOME15], which aims to provide a model of the human mind by combining psychoanalysis, neuroscience, and computer science. The idea is that a model as closely resembling a real human mind as possible should be able to accomplish many tasks in building automation in the future that are so far a restricted field of application for humans.

1.1 Background

In recent years, computer science has made many advances in the area of intelligent systems; however, so far these systems are unable to fulfil the expectations that were raised after the first successes in the late fifties. At present, there is an ever increasing amount of application areas for autonomously acting intelligent agents, ranging from public building security systems which identify possibly dangerous objects to private home automation, such as an intelligent kitchen which starts brewing coffee while you brush your teeth or triggers an alarm when a child comes near the hot stove. SiMA has its beginnings in a project that was started with the intention of getting a few steps closer to the latter of these examples. The Smart Kitchen project (SmaKi) [Rus03, p. 19] aims to perceive situations in a kitchen and be able to react preventively, unlike conventional passively reacting systems taking into account the whole environmental situation. This project was based on the concept of Perceptive Awareness, which enables a control system to increase comfort, security, safety and economy [Lan10, p. 1]. This is achieved by perceiving and recognizing situations and setting the appropriate actions or countermeasures. Such a system is equipped with sensors, which partially simulate the human senses and partially exceed human capabilities (e.g. air composition). In order to be able to deal with the huge amount of data arising in this context, it has to be reduced to the most important information (e.g. hot instead of 400 degrees Celsius). One of the project's major findings was that this vast amount of sensorial data needs very elaborate information handling in order to be processed [Deu11, p. 1]. As it is very likely that future scenarios with far more input data and more complex environmental situations

to consider will overload the current control-systems, the Artificial Recognition System (ARS) project was brought to life.

Back then the main goal was to find an approach to manage the immense flood of data that will be produced by the sensor nodes of future control buildings. The approach, which was finally chosen to be implemented in ARS was a bionic one. A bionic approach basically demands to transfer biological principles into technical ones. The assumption behind such an approach is that biological principles have been developed in the long process of evolution and are therefore near to optimal [DFZB09, p.418]. Even though the first model which was originally presented in 2005 had to be abandoned, it provided the team with many findings which resulted in the new ARS model which is used until today [DFZB09, pp. 53–54]. One major problem of the first approach was the usage of different theories from neuropsychoanalysis which were partly incompatible to each other. As the usage of one model alone was impossible due to the fact that none of the existing theories were providing a model covering all areas in equal detail or even leaving some of them out, work on a new model was started. As no unitary psychoanalytical model was available it was decided to develop the new model which builds on the concept of Perceptive Awareness and adds further concepts from neurobiology, psychology and psychoanalytical theories, based on the work of Sigmund Freud.

In the last few decades, many scientists have been stressing the theory that all human thoughts are generated in our sensor and motor systems [Lan10, p. 2]. Due to the tight coupling between psychoanalytical concepts and bodily needs it was necessary to change the intermediate test environment from building automation to an embodied system. As the area of building automation is not a feasible test environment for developing a system that is inspired by the human brain, the Bubble Family Game was introduced as a test environment [DGLV08, p. 1]. This environment is a simulated world populated by embodied autonomous agents, which are equipped with a neuro-psychoanalytical decision unit. Such an environment may contain a number of agents, various types of food, different terrains and some points of interest [Deu11, p. 62]. The agents roaming in this world trying to satisfy their bodily needs, by checking their environmental situation and taking decisions based on the sensory information they get.

In the course of time the basic research questions have changed and the old project name no longer covered the actual research interests of the project. Therefore, since February 2015 ARS is continued under the name Simulation of the Mental Apparatus & Applications (SiMA). A more detailed description of the actual state of SiMA and its concepts is given in chapter 2.1.

1.2 Motivation

Since 2005 when the first ARS model was presented, many people, from different research areas, like human scientists and technical engineers, have been contributing to this project. As an agent cannot act without any knowledge about its environment or tasks, a file-based mock-up memory has been in use for a long time now. In order to reach the ultimate goal of autonomously deciding agents, however, the ability to learn from experiences and consequently to remember already experienced situations is absolutely necessary for an agent. Based on the outcomes of former decisions, one may react faster or better than in a first encounter of a difficult situation [DGLV08, p. 1]. Furthermore the accessibility

of former experiences might enable an agent to discover new solutions to its problems, for example, if the experiences from the past were not satisfying in their outcome.

A long-term memory, providing the ability to store and retrieve memories will not only improve the outcome of the agent's decisions, it also adds the possibility of implementing learning mechanisms to SiMA in a future step. Learning means not only to remember things, but also to reason about the remembered experiences, which enables an agent to gain new knowledge without actually experiencing a certain situation. For example, knowing that birds have wings and a beak, may enable the agent to identify a flamingo as a bird even though it never learned the fact "a flamingo is a bird". Such abilities might be considered to be one of the last steps towards an agent that is up to undertake tasks that required human interference until now.

In the future intelligent robots could go to places that are too dangerous for human beings, like for example areas of nuclear disasters or burning buildings. When the catastrophe of Fukushima occurred robotic systems were used, but they were not able to take the place of humans in full [ROBO11]. It took 5 weeks until the first robots were able to enter the place of the catastrophe, in order to explore the situation. This exploration, however was dependent on a working transfer signal to human controllers as robots nowadays are not able to react flexible to unknown situations. Existing robot systems are able to work on specialized tasks they were intended for, but cannot adapt themselves to anything unexpected. According to [FUKU14] it is to be expected that for the next 40 years humans have to work on the cleanup of the nuclear power plants. After the first emergency works on the severely damaged buildings 2 people were missing, 17 were injured and 2 were signed off sick [FUKU11]. But this were only the immediate effects to those humans that risked their lives in order to save others. It is practically unknown how many victims the work on the contaminated power plant will claim over time. This incident confronted humanity in a very painful way with the insufficiency of the current state of the art. Until robots have some sort of intelligence that enables them to react and learn from new experiences there is no way that they could take over the place for human beings in such life threatening situations as the Fukushima catastrophe. Further – less dramatic, but still much longed-for is the help in everyday jobs that are boring or exhausting for human beings, or perform tasks that cannot even be done by humans. Some people expect AI systems to permanently watch our bodily state and diagnose illnesses [WIRED]. A system that is able to track and remember all symptoms a person experienced may be imagined to be more accurate than a doctor that is out of time or interest to listen to a patient that forgets half of the problems she experienced. Other factors like shame or misjudgement of the severity of one's problems could be excluded by such a system as well. Again a system that is able to diagnose illnesses correctly has to learn about the habits and bodily states of its human owner in order to prevent wrong diagnosis.

In order to gain experience during run-time a system has to store the gained experience in some sort of storage. Even though for short runs an in-memory solution should be sufficient, a long program run will inevitably lead to a lot of memories that have to be kept in the agent´s long-term memory. If too much information has been gathered a pure in-memory solution is going to slow down the system, until further execution is out of the question. The reason for this is that the system will run out of main memory after running the simulation for some time. In order to prevent such a situation counter-strategies have to be found. One apparent solution to the problem of in-memory overload is to

"outsource" the long-term memory into a database and fetch only the currently important parts of the long-term memory to the in-memory store of the simulation environment. Another solution, which may be combined with the first one, is the implementation of a forgetting mechanism. This may be done by simply deleting information that was experienced a long time ago, or an even more elaborate approach to forgetting is the introduction of activation levels, marking the last usage of a certain part of the memory and forgetting those parts that are not used for a long time.

1.3 Problem Statement

Simulation of the Mental Apparatus & Applications (SiMA) is a simulation environment for autonomous agents based on the psychoanalytical model of human minds. At the moment, agents in this system are not capable of collecting and storing memories themselves. As the process of decision making implies accessing some sort of knowledge, a long-term memory is currently simulated by parsing a Protégé Frames file [WTN+, p. 1] representing knowledge of the agent into the in-memory storage, from where it is retrieved by the agent when needed. Even though this temporary solution allows to test the functionality of the system model, it is not a sufficient permanent solution. In order to enable the agent to react flexible to environmental changes and enable it to improve itself without human interference it is necessary to have a long-term memory that can really memorize experiences. The main research question for this work can be therefore formulated as:

Research question 1: "Which features need to be implemented, in order to enable SiMA to collect experiences?"

The aim of this thesis will therefore be to enrich the SiMA project with a long-term memory, allowing the agent to proactively gain experiences at runtime. As this will be the first time for the SiMA agent to save and recall memories, one challenge of this work will be to filter and decide which experiences are to be saved. As in the human experience process, the agent cannot decide which information might be of importance in the future. Therefore, he keeps his experiences in his short-term memory until they are moved to the long-term memory for permanent storage or are forgotten [MEMORY]. Part of this thesis will be to formalize and implement the procedures that decide which memory parts will be saved to the long-term memory. To decide which information is of value, the agent needs to be aware of his needs and goals and conclude from those which of his actions have a connection. Therefore the first sub-research question is:

Research question 2: "How shall the agent decide which memories are to be kept?"

In order to create memories that enable the agent to reconstruct situations, or even infer new knowledge from them, it is necessary for the agent to link several experienced *images* together to create a sequence of events. The term "image" is used for structures representing perceived situations, while a sequence of events is referred to as an *episode*, or like in SiMA an *act*. Basically, such an act consists of several atomic images, each representing one state of the environment, linked by associations marking the temporal connection between the images. In Figure 1.1 an example for an environmental situation can be seen. It shows one of the basic use cases which exist in the SiMA project. This use case requires the agent to remember how to handle the obstacle (the stone) in its way

and get around it, in order to eat the cake. It also depicts very vividly the necessity of further information, in order to turn this image into valuable information.

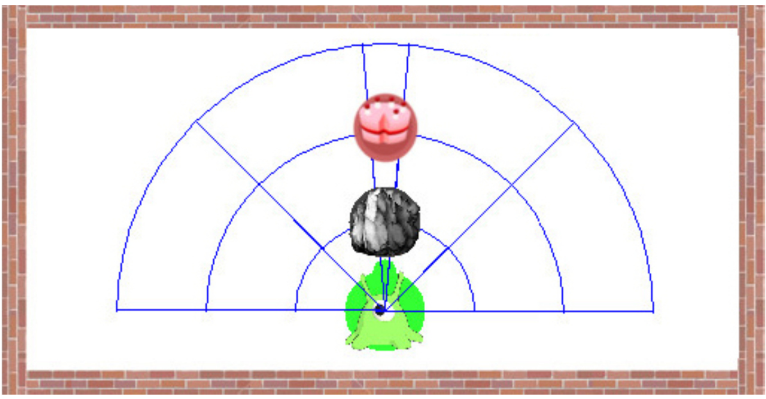


Figure 1.1: Image of SiMA environment

In the further course of its actions the agent is going to eat the cake behind the stone. However, from the agent's point of view, there is currently only a stone to be seen. Only by linking further images, providing the information that it finally found something to eat, just after it looked behind that stone, adds some use to the first picture. This is of course one of many examples of possible connections between separate images. One may imagine that other valuable links between images are possible in the future, like for example similarity links, marking the matching degree between the images. Based on such information links the search and reasoning processes could be further improved. An agent aiming to interpret never experienced situations consequently needs not only to remember experienced situation, but also their connection to other experiences. The next research question can therefore be formulated as:

Research question 3a: "When shall the act generation be triggered?"

Research question 3b: "How shall the act construction be implemented?"

Due to the current file-based storage approach, storage problems may arise in simulations running over a long time. These problems are caused by the circumstance that all information has to be either held in the working memory and the system will run out of allocatable main memory over time. Even though it would be possible to read and write a text-based file, it will not be possible for a large amount of data. The parsing process alone would take too long, let aside the search for the data which is of importance. Therefore the first step towards an agent making experiences and saving them for future use has to be the extension of the existing framework by the implementation of a more flexible data storage solution. As the application area of a simulation environment is not a very common use case, one cannot expect to find best practices for the question what data storage solution may be best. Very clear requirements come from the fact that SiMA is a scientific project which makes an open source solution highly favourable. Also general requirements of long-term projects like stability and long-term support will play a role in the decision process. In the future, an effective query language could take over some of the filtering processes that are currently done in the system itself, always provided their application is not contradicting the psychoanalytical theories behind the filtering process. Besides considering these basic demands the main task will be to decide for a database solution that is fit for

the special requirements that come with a simulation requirement. The research question for this part of the work is therefore:

Research question 4: "Which of the existing database paradigms are most suited for SiMA?"

Furthermore, there is an obvious trade-off between keeping all information at hand in an in-memory store and being able to store a near endless amount of data in a local or server-hosted store. An in-memory store is until a certain amount of data is reached, a very fast solution, as all the data can be accessed immediately. If too much data has to be managed in the in-memory store, however the system will run out of allocatable main memory and no advantage can be gained from this storing method. Local or server storage needs to perform read and write operations and is therefore slower on short-term runs, but gains advantage over an in-memory solution when the amount of data increases. There are many database solutions providing a switch between in-memory and local or server store solution, which might turn out to be an interesting option.

It is important to note that for this work it is out of scope to change any of the current information representation or the systems access to it and that the data structures probably have to be converted between the new database system and the interface access in the SiMA architecture. This might turn out as a serious bottleneck, depending on the chosen database and saving approaches. Nonetheless the chosen approach shall be fit for future use and therefore, if possible, the need of future adoptions should not influence the decision unless the solution should turn out not to work at all with the current set-up. Another research question is therefore:

Research question 5: "How shall the current knowledge base be migrated?"

1.4 Task Description

The aim of this work is to enrich the agents in this system with a basic long-term memory, which enables the agents to store memories for future advantage. In [Zei10, p. 1] the foundation for this work was laid by the definition of the internal representation and an information management system providing an interface between the decision- and information representation layer and handling the stored knowledge. However, in order to realize the final aim of this work, some adjustments to the current set-up have to be done.

The first task of this thesis is therefore to change the currently used file-based knowledge storage to a more advanced database solution, thus enabling the system to outsource memories from the main memory to the data store during runtime. This will enable it to keep track about its experiences for a long time without utilizing the working memory excessively. The conversion of a file-based storage approach into a database system will enable the agent to work with larger amounts of data and enrich the SiMA system with the capability of using database queries as a supplement to the already implemented search processes. The chosen system has to work with the existing models and concepts in SiMA as it is out of scope for this work to redesign the data structures, or the current search process. In order to make an informed decision about the technologies that will be used, some research on state of the art database solutions shall be performed. Furthermore, other well-known agent systems along

with their long-term memory approaches shall be presented, in order to decide for a solution that is fit to fulfil all the requirements that come with a project with the scope of SiMA.

The second task of this thesis is the extension of the current framework to enable the agent to gather experiences and save them into the newly added knowledge storage. This means that the system will be supplemented through appropriate algorithms and methods to extend the SiMA agent's capabilities considerably, it shall give the agent the ability to save important experiences in order to enable it to include them in its decision process. This implies the implementation of algorithms that filter out unnecessary information in order to save only those experiences which are likely to improve the agent's decision process in the future. Even though the agent will be able to save experiences and consequently access them in the further course of its life cycle, or even another program run, it is likely that the new knowledge will not bring any improvement to its behaviour immediately. This is due to the fact that the agent´s search algorithms or decision process will not be adjusted in the course of this work as it is simply out of scope. If the agent has too many equally important and fitting experiences to consider, it is likely to slow the decision process down. Therefore, further improvements like to consider newer experiences as more important or even forget older ones may be necessary before the new long-term memory is actually contributing real benefits to the system.

1.5 Methodology

When developing a project in a scientific environment, it is always a good idea to look around for other approaches and solutions to the topic, in order to prevent "reinventing the wheel" or to avoid or at least anticipate problems that were experienced by others. Therefore the beginning of this work is an extensive research on state of the art cognitive architectures and their approaches to memory and knowledge representation as well as on data storage solutions currently available. Basic questions for this state of the art research are:

- What data storage solutions are available and what are their general advantages and disadvantages?
- What is the intended or preferred application area of the presented data stores?
- What memory aspects are utilized by the cognitive architectures?
- Which information representation was chosen by the cognitive architecture in order to represent the memory contents?
- What approach was chosen to search and find the information the agent needs at runtime?
- How is new information gathered and saved?

One of the expected impacts of this thesis on the SiMA project is the ability to deal with larger amounts of data. This shall be achieved through the conversion of a file-based storage approach into a database system that allows the simulation environment to fetch and hold only the currently relevant data in the working memory. Therefore, the first step is the choice of an existing database solution. Criteria for the selection of this database system are performance, stability, and project specific requirements, like for instance the usage of open source software. In addition to the classical relational database systems which have been dominating the world of data storage for a long time, many specialized data store

paradigms have been presented in the last few years. Many of them were created with the new requirements of web applications, like big amounts of unstructured data, in mind and are therefore highly interesting alternatives to the relational approach. Some RDF databases even provide basic reasoning concepts by using the RDF schema based knowledge language OWL, which is also considered for the selection of an appropriate database system although it will not be used in this thesis.

Currently, SiMA uses a tool called "Protégé" to maintain and create some basic information that is simulating remembered situations for the agent's decision process. This data is loaded and parsed into the in-memory storage at runtime and then parsed into Java data structures. In order to save the existing data to the new database, a reasonable representation of the existing Java data structures has to be found. Depending on the chosen database solution even some sort of conversion might be necessary. For example, many NoSQL databases present their data in structured text or in triple representation which would require the implementation of a conversion routine for the stored data. At present, it is absolutely inevitable to reconstruct this information into the data structures discussed herein. However, in future implementations it should be possible to further reduce the amount of loaded data by fetching not the whole data structure but only information currently needed by the agent. Considering the life span of the SiMA project, one has to keep in mind that new requirements as well as options may arise in the future. Consequently, maintainability is of great concern in this phase of the project and has to be kept in mind when integrating the new database system into SiMA. Through the introduction of interfaces and exchangeable functions, better maintenance and adaptability for the future shall be guaranteed.

Concurrently, first tests for saving data structures will be done in order to create a data structure that is fit for storing memories as well as searching through them. After the basic data structures can be saved the non-trivial topic of defining acts will be considered and implemented. In order to extend the current framework by a memorization process, appropriate algorithms and methods as well as concepts for filtering and storing experiences will be developed and implemented. This will include research on algorithms and mechanisms that will enable the agent to decide which information in his short-term memory should be combined to form an act and afterwards saved to the long-term memory. This will enable the agent to gain experiences and manage them in its short- and long-term memory as well as taking them into account when experiencing similar situations, extending the SiMA agent's capabilities considerably. Finally, the functionality of the whole system is validated by the successful execution of the already existing use cases, using the new database system. The use cases used to validate the system will be defined in the first phase of the project.

2. Related Work and State of the Art

There exist many cognitive architectures, using diverse theories on information representation and memories. While several systems utilize only one permanent memory, others divide theirs into different sub-categories, each specialized for certain tasks. Some approaches provide truly persistent storage that is used and extended over several program executions, while others simulate a long-term memory only in the cognitive sense, do not store their new knowledge after program termination. The first part of the following chapter will discuss some prominent solutions for permanent data storage, along with their strengths and weaknesses, whilst the second part will focus on presenting some well-known cognitive architectures along with their memory approaches and implementations.

2.1 SiMA - Simulation of the Mental Apparatus & Applications

As already mentioned in chapter 1.1 project ARS was originally started in the year 2003 to provide an answer to the growing demands that are put on building automation systems. After it was detected that a kitchen equipped with sensors and actuators, has considerable problems to deal with the large amount of data, when treating the information with common approaches. However the ultimate goals (namely processing incoming sensor data in real time and reacting to unknown situations) are not to be solved by algorithms alone [ZLM09, p. 383]. As a result of this insight the path of simulating the human mind was taken. In order to show intelligent behavior, embodiment and emotional intelligence can be seen as inevitable. For a model of the human mind, which is technically realizable and still correct from our actual knowledge about the human functionality, a close cooperation between human sciences and technical engineers was needed. When first steps were taken to implement a system providing abilities for data processing and decision making, two sub-groups were founded [Vel08, p. 10]. The first group ARS-PC (PerCeption) dealt with perceiving objects, events, scenarios and situations and the resulting model is based on neurological and neuropsychoanalytic theories [ZLM09, p. 383]. ARS-PA (PsychoAnalysis) was employed with the reasoning unit, which is responsible for the process of decision making and based on the psychoanalysis. Due to the fact that a holistic, non-contradicting psychological concept of the human mind was needed, the approach which is in use until today is based on the first and second topographical model defined by Sigmund Freud [Lan10, p. 50]. The second topographical model contains the widely known concepts of Ego, Id and Super-Ego, in contrast to the first topographical model where unconscious and preconscious/conscious were distinguished [DFZB09, p. 16].

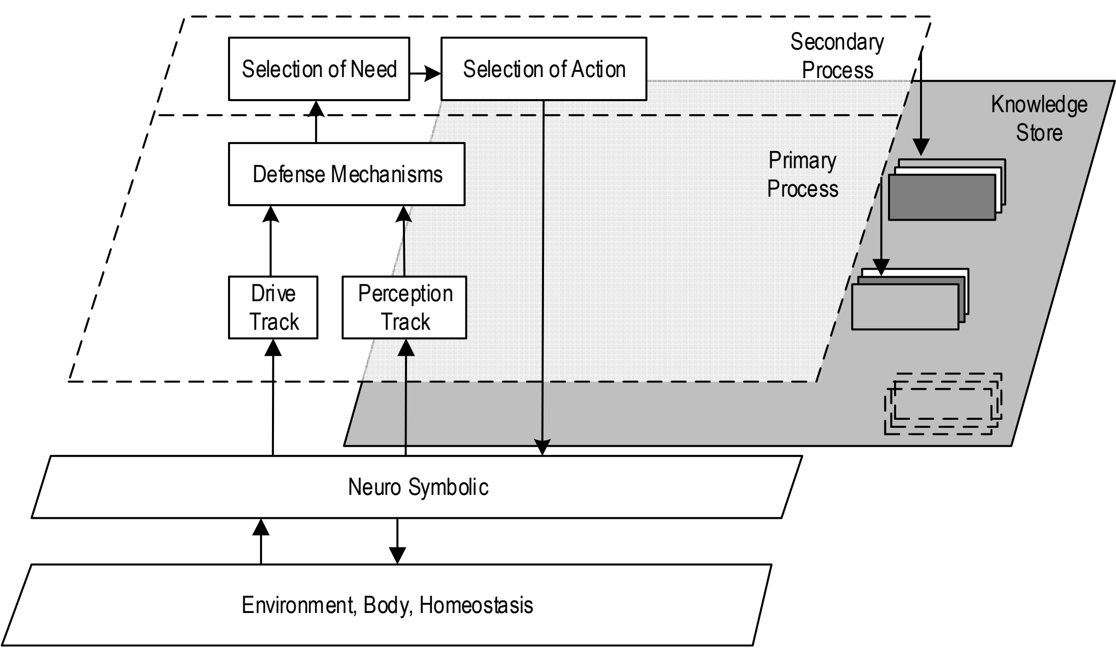


Figure 2.1: SiMA layers plus memory module design based on [SDW+13, p. 6650]

Architecture and Concepts

In Figure 2.1 the basic layers of SiMA with its memory module are presented. *Neuro-symbols* are used to represent perceived images from the environment [VBPG09, p. 1964]. Examples of such neuro-symbols could be a face, a voice or a person. A separation between a *primary process* and a *secondary process* can be seen. According to the first topographical model of Sigmund Freud the primary process performs all unconscious data processing whilst the secondary process performs all pre-conscious and conscious data processing [Deu11, p. 68]. The Id represents the drives and affects of the agent and is part of the primary processing. Its functions are located in the *drive track* and *perception track*. The Super-Ego contains internal rules that are usually gained in childhood and define what is considered to be “socially acceptable” for the agent. It works as a counterpart to the Id, which tries to satisfy its needs immediately. The Super-Ego itself has to be predefined by the developer and contains commandments, bans and information about action handling for predefined scenarios. The Ego mainly works with preconscious or conscious information and is therefore mainly in the secondary process positioned. However, some modules of it (e.g. the *defense mechanisms*) work with unconscious data, which makes them part of the primary process. Its main tasks are mediation of requests from Id and Super-Ego and the reality check of those demands [Deu11, p. 69]. For example demands may be altered or even postponed if something in the environment implies this would be better for the moment (in the figure shown as *selection of need* and *selection of action*). SiMA´s *memory* or knowledge store is connected to the primary process as well as to the secondary process. At the moment it is a file that is parsed into in-memory at runtime and no information can be stored to it. All information in the primary process is represented by thing-presentations, while data which is processed by the secondary process is defined by word-presentations. *Thing-presentations* contain information about sensorial data and may be grouped into visual (like colour, size or shape), taste, audio, olfactory and tactile data. *Affects* represent the drive-demand intensity of a thing-presentation and have to be connected to one

in order to get some meaning. A *word-presentation* is a description of an object by symbolic means (like for example verbal expressions, hand gestures or sound combinations). They summarize thing-presentations and affects to the concept of one object. A very important concept of SiMA are the drives, which represent the motivations for decision making. It combines a *drive source* (a body organ or process), a *drive aim* (an action that will satisfy the bodily demand) and a *drive object* on which the aim may be executed, with a *quota of affect*, stating how important the drive currently is.

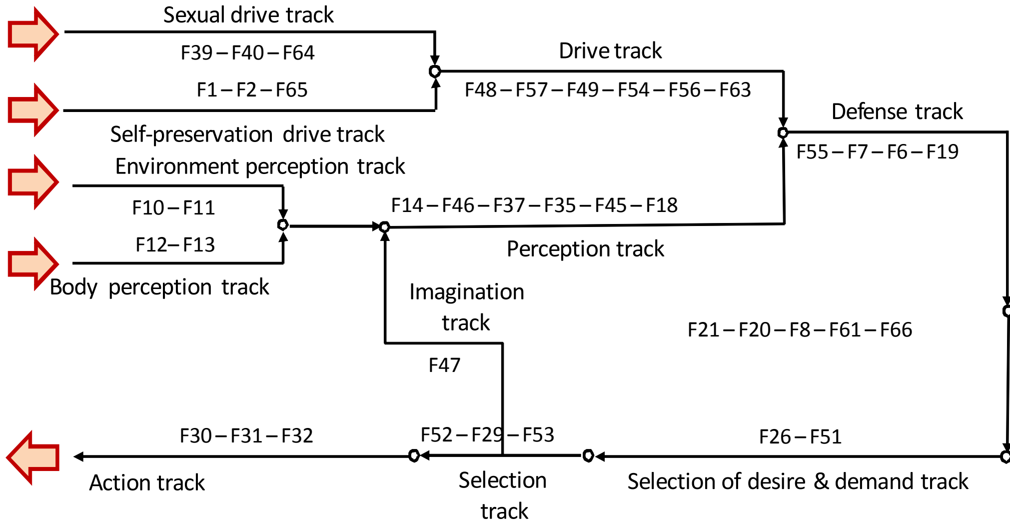


Figure 2.2: Data flow and decision cycle in SiMA [DSBD13, p.6669]

Processing Cycle

A control loop in SiMA starts with the five input sources that can be seen in Figure 2.2. There exist four basic inputs, namely sexual, self-preservation, environment perception and body perception. *Self-preservation drives* represent bodily needs (e.g. blood sugar) while *sexual drives* are internal demands seeking for pleasure [Deu11, p. 70]. Both are input sources from the agent´s body. From the symbolic information that is gained from self-preservation drives and sexual drives, a drive, without quota of affect, is created [Deu11, p. 140]. The next step is to calculate a quota of affect and pass the information on to the Ego. E*nvironment perception* and *body perception* are inputs from the environment. The environment perception provides sensorial data about the environment itself, whilst the body perception informs the system about actuator positioning and sensorial feelings like for example pain [DSBD13, p. 6670]. They are combined with a feedback input from the secondary process, representing fantasies from the preconscious [DSBD13, p. 6671]. For all inputs it holds that their importance has to reach a certain limit in order to be transferred to the Ego [ZLM09, p. 385]. The Super-Ego receives information about the environment from the Ego and returns an operation consistent with the current situation. The Ego has to use its defense mechanisms to resolve conflicts which occur between the drives coming from the Id, the Super-Egos rules and the reality. The incoming demands are first altered by the Ego, in order to increase their chance to be selected. While the information is transferred from the primary to the secondary process, the thing-presentations are completed with word-presentations. From now on they exist in parallel, but only the word-presentations are used for the processing.

In the end the Ego then calculates a “decision” based on the information about the environment and the other modules demands, which is passed on to the actuator control [ZLM09, p. 385].

Memory and Information Representation

Until today SiMA has gone through many changes that were applied after close examination on potential system drawbacks. One of these modifications was the change from an episodic memory to an approach which seemed more suitable for an embodied agent [Deu11, p. 73]. In [ZDI+08, pp. 383–389] a new approach for representing information is introduced to SiMA. The main reason for this change was the widespread opinion that memory is highly dependent on interaction with the environment, which is hard to realize with a “storehouse” approach. As stated before the psychoanalytical approach was chosen for the reasoning units approach, therefore information which is passed between the three modules is not divided between memory sub-groups like episodic or semantic memory, but represented in three basic modules called thing-presentation, word-presentation and affect, which were described in the beginning of this chapter [ZLM09, p. 385].

For search, storage and retrieval of memory the information representation module (see Figure 2.1) is responsible [Deu11, p. 74]. This module is connected to the reasoning unit which may use this connection to access information from memory. In order to save some perceived information to the memory it has to be converted from sensor data to thing-presentations. Thing-presentation-meshes can be used to group temporally associated thing-presentations which represent a situation. For the representation of a temporal context word-presentations have to be used.

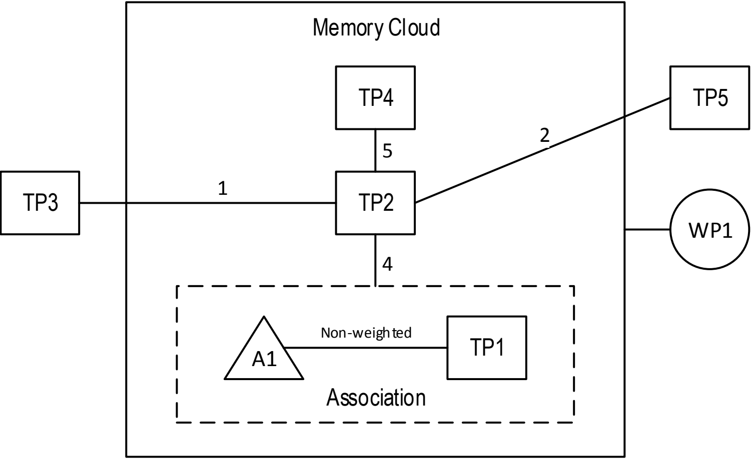


Figure 2.3: Memory Cloud based on [ZLM09, p. 386]

The described information representation modules are used to form so called *memory traces*. A memory trace is a concept stemming from psycho-physiological theory, which represents permanently preserved memories in the mind [ZLM09, p. 386]. In Figure 2.3 the process of creating a memory-cloud from memory-symbols is depicted. The associated memory-symbols affect A1 and thing-presentation TP1 have an association which is weighted with 4 to the thing-presentation TP2. Due to the fact that their association weights are above a certain threshold, together with TP4, which has an association weight of 5 to TP2, they form a memory cloud. The resulting memory cloud is associated with the word-presentation WP1. Memory-symbols in a cloud may be connected to other symbols

outside the cloud, like one can see in Figure 2.3 on the connections TP2-TP3 and TP2-TP5. As the association weight is determined by concurrent appearance, it is possible that these weights increase over time and the symbols are included in the memory-cloud.

Currently in SiMA symbolized sensor data representing situations, so called *template images,* have to be predefined in memory-clouds [ZLM09, p. 386]. There they are compared to incoming perceived images of the environment. If the perceived image does not match any of the existing ones (within a certain range of tolerance) a new template image is stored. If a stored memory-symbol matches a perceived one it is directly activated. If two symbols are directly activated at the same time their connection value increases. However, their connection value may decrease over time again. Besides direct activation there is also indirect activation, if a symbol is activated through a directly activated symbol. Anyway this sort of activation is not increasing the association weight.

Actually there is no learning implemented in SiMA, also the memory (shown as database in Figure 2.1) is a declarative semantic one and no episodic memory has been implemented until today. The knowledge base is created at system start from a Protégé-Frames file, which has to be created by the developer manually. Protégé-Frames is an ontology modelling paradigm, particularly useful for ontologies with a closed-world assumption [WTN+, p. 1]. The information from this file is converted to the already introduced concepts like thing-presentations, word-presentations, thing-presentation-meshes and word-presentation meshes at program start. The converted information is then put into HashMaps (please refer to [HASHMAP] for further information) from where it can be accessed in the agent´s life cycle.

2.2 Data Storage Technologies

Since the 1960s, when the term "database" first came up, many different data storage technologies have been developed. Many of them were created with the aim to become a perfect fit solution for absolutely every area of application. In the times of Web 2.0 applications and cheap hard disk storage have changed the general mind-set. Nowadays one has a variety of technologies to choose from, many of them designed to meet the requirements of a certain application area. If one has to decide for a storage system three main aspects should be taken into account before starting with the storing process itself:

- What is the nature of the data and the requirements of the overall application in which it shall be accessed?
- Which options and constraints are offered by the storage technology itself?
- Where shall the data be located?

The *nature of the data* and the *requirements of the application* in which it shall be accessed are of importance in order to make an informed decision about how to represent the data in a storage system. Such requirements might be access frequency, data change frequency and amount of data added. Other requirements might be consistency of data, stability of the system and security issues. As it is not possible to provide one perfect solution for all requirements, different emphasis for different needs are

provided by the data storage systems. For example, putting an index on the database improves the search speed of retrieval operations, the drawback coming with this improvement is the need of additional storage space and reduced write speed, as every new entry has to be indexed. The data itself may bring up many challenges for the database. For example, in some database applications, there may be the need to store billions of records, in others the data may be multi-format (records are in different formats like text, images, videos etc.). At the same time the *storage technology* itself has to be considered, as different systems provide different representation models, like for instance the table like representation of a relational database, or the subject-predicate-object representation of a triple store. Depending on the area of application one may have to choose where the data will be *located*, in other cases the location is already determined by the application area. Examples of data locations are local stores on the desktop and big server solutions. Some data stores might be restricted to local use, or are at least notably reduced in their performance if used on a server. Besides those possible restrictions the location is only as far connected with the other two aspects as it might impact the performance of the overall system if the chosen approach is not suitable for the size of the stored data or the access frequency. Also for the most technologies the location can be rather easily changed during a project´s life cycle [Law14, p. 285]. For instance, one can change from a local relational database system to an installation located on a server by simply changing the access data in the code and providing the database system of one's choice on the respective location. Having said that much about how to approach the topic in general, the following pages will present some of the most promising approaches in database technologies. Generally those technologies can be distinguished into relational databases and NoSQL databases. According to [Bar10, p. 1] a rivalry between the two camps of NoSQL and relational database followers has been building up, though both approaches have been designed for very different problems, which will become apparent in the next chapters.

2.2.1 Relational Databases

Relational databases have their origins in 1970, when the relational database model was introduced by Edgar Codd. The basic principle is the usage of a table set, where each table represents an entity or object using named columns describing the objects attributes or fields [Lea10, p. 12]. Each row in this table represents one record or tuple. For example, one can have the table food and the columns of the table Food are ID, Name, Type, Colour and Taste. A record for some marble cake has the corresponding field values “1”, “marble cake”, “cake”, “brown-yellow” and “sweet” (see Figure 2.4). The name arises from the relations that exist between the tables which are also represented by columns. For example a table Agent could have a relation “favourite food” to the table Food. This can be represented by adding to the table agent a column favourite food in which the so called “primary key” of the table Food is saved as a “foreign key”.

In Figure 2.4 one can see that the record with the name “Arsin” has a value of “1” for the column FavouriteFood, which basically means that Arsin likes to eat marble cake above everything. Primary keys act as a unique identifier amongst every data set (data row) of each table. Even though a wide-spread practice is to use a continuous numbering as a primary key it is also possible to define a single unique column or combine several columns to a unique identifier to act as primary key. Foreign keys are used to define the relations between database records. Note that the statement “Arsin likes to eat

marble cake above everything" could be also represented by simply writing into the column FavouriteFood "marble cake", not making the column a foreign key column. The statement would still be the same, but the additional information about the food "marble cake" would be lost. However, one could still use the query language SQL (Structured Query Language) to search for a food called "marble cake" and assume that this is the meant food.

Food				
ID	**Name**	**Type**	**Colour**	**Taste**
1	ham	meat	pink	salty
2	marble cake	cake	brown-yellow	sweet
3	grapefruit	fruit	orange	sour

Agent			
ID	**Name**	**Age**	**FavouriteFood**
1	Bodo	1	1
2	Arsin	10	2
3	Anouk	3	2

Figure 2.4: Example tables with relation

As long as only one food with that name exists this would work perfectly well, but of course there is no guarantee that this will always be the case, which is the reason why foreign keys are used for stating relations in a unique way. Due to the fact that there exist standards which are kept by most vendors, relational databases are interchangeable [Law14, p. 285]. Even though there are minor variation of the implemented SQL features, the change from one relational database to another should be comparatively easy. Relational databases are designed to work at high precision and have to meet the ACID (atomicity, consistency, isolation, durability) standard [Lea10, p. 13]. The atomicity property requires the database to perform a transaction completely or not at all, consequently a fail of one part of an operation will terminate the whole operation. Consistency requires the database to move between valid states, considering all the constraints, triggers and so on. Isolation means that only complete transactions are visible to the "outside", which ensures concurrency control. With durability the ability to keep all persisted changes despite system crashes, power loss is addressed. All these properties make relational databases the first choice of many business applications, where precision and integrity are of high importance, but may add unnecessary overhead for a top-tier website system.

Disadvantages and Limitations

Even though there is almost no requirement that cannot be met by relational databases, there are some limitations to them, where other database solutions may overcome them. According to [Lea10, p. 13] relational databases don't work very well in a distributed setting, as joining tables becomes a real challenge there. Also, most relational databases do not support distribution natively. Data partitioning is not part of the relational database concept, which means it has to be implemented specifically for each project. Although this would be some effort, one cannot avoid it when the limits of the powerful and expensive single machines are reached. Another limitation is the tabular representation of data. Every piece of data that shall be saved to a database has to be converted into tables even if the data is not fitting for this form. This can add high complexity to the data manipulation processes and therefore reduce the performance of the system significantly [Lea10, p. 13].

One of the biggest advantages of relational databases also leads to one of their biggest drawbacks – the requirement of ACID transactions. In comparison with other database solutions, performing all

these constraints slows a relational database down significantly. Furthermore this implies that relational databases cannot handle incomplete information. As already mentioned SQL is the main querying language for relational databases, which is an advantage as well as a limitation. For structured data it is really convenient and works nearly without competition, but with other types of data it is much more difficult. Furthermore the big set of features adds high complexity and cost and is not necessarily needed by some applications [Lea10, p. 13]. Adding new data not matching the present database schema is possible, but might require rebuilds [WDM+12, p. 220]. To circumvent problems with unstructured data records solutions for providing a more extendable data model have been researched. One approach to a more extendable data model is the application of the "Universal Data Model". The Universal Data Model is basically a generic data model, which aims to provide a representation in which all sorts of data may be converted to. In [WDM+12, p. 220] a variation of this model was successfully used to integrate data from arbitrary sources into one relational database. While this would enable the database to be completely adaptable for new data sources, it would slow down the execution time. Another disadvantage of this approach would be the limitation to string values and the loss of cardinalities.

2.2.2 NoSQL

With the advent of Web 2.0 the advance of the so called NoSQL (Not Only SQL) databases from the niches of software development to widespread usage has begun. They have been introduced to solve problems which were poorly served by relational database solutions. The biggest advantage these storage approaches have in common is the ability of handling unstructured data, which primarily made them so attractive for the usage in Web 2.0 applications [Law14, p. 285]. Well known examples of such databases are Dynamo or Big Table which were developed by Amazon and Google [Lea10, p. 12, [Sto10, p. 10]. A very interesting advantage which some, but not all NoSQL databases provide, is the option to use them in a distributed setting, meaning that a single database can be distributed over several inexpensive machines. For high traffic websites, depending on massive scalability, low latency and the ability to grow on demand, this is a big argument for the usage of NoSQL databases, but there are several other non-website-scenario scenarios where this might be a better suiting setup than the usage of one expensive high performance machine. Also, most NoSQL storages are open source [Law14, p. 286] and therefore have another argument for smaller budgets or open source developers. According to [Lea10, p. 13] NoSQL databases process data faster than relational ones and give more flexibility to the developers, but to achieve this high performance more adjustment have to be done. This processing power can also be attributed to the work spreading over several machines, which results in lower latency even if a large number of read and write operations are performed [Bar10, p. 1]. NoSQL being only a collective term for non-relational databases, leaves to say that different NoSQL databases use different approaches as to how to handle their data. NoSQL systems are based on the CAP (Consistency, Availability, Partition tolerance) theorem and the BASE (Basically Available, Soft State, Eventual Consistency) paradigm. The BASE paradigm states that NoSQL databases renounce some of their consistency in order to improve their availability and performance [BLS+11, p. 484]. CAP states that a distributed computer system can only provide two of the three parts of CAP simultaneously. Consistency means again that all parts of the system see the same state at the same time, availability demands that every request is answered in some way and partition

tolerance enables a system to operate despite some failure in one part of a system. Many NoSQL systems approach the partition tolerance problem by mirroring database clusters between multiple data centres. However, this means that high consistency is not possible at all times. Changes will be applied at some time, but there is a high possibility that at some point a single node or group of nodes is not up-to-date. Obviously partition tolerance is no problem for relational database systems running on single machines [Bar10, p. 1]. Up to this day there exist four types of NoSQL databases: key-value stores, column oriented databases, document oriented databases and graph oriented databases [BLS+11, p. 484]. The following pages aim to provide an insight into the different NoSQL database concepts.

Document-oriented stores

Document-based stores are intended to manage semi-structured data, which is stored in some sort of "document". In the language of a NoSQL database "document" means a collection of various fields of information [Bar10, p. 1]. The offered functionalities differ widely, but the basic concept is the usage of some sort of internal standard format. The main advantage of this storage approach is the ability to add a different number of fields of varying lengths to the document [Bar10, p. 1, [Lea10, p. 13]. This sort of data store is very useful for semi-structured data and according to [Bar10, p. 1] they are very well suited for object oriented programming models. Popular implementations are CouchDB from Apache Software Foundation, 10gens MongoDB and Basho Technologies´ Risk [Lea10, p. 13].

Key-value stores

A Key-value store is exactly what its name implies – it stores values indexed by keys and works equally well for structured and unstructured data. [Lea10, p. 13] states that key-value stores achieve extremely low latency under high load, when paired with a fast network and a large number of servers. Examples of key-value stores are Amazon´s SimpleDB, Amos II or Scalaris.

Column-oriented databases

Column-oriented databases are designed for closely related data, which is stored in one extendable column. The basic principle is to store column by column rather than row by row. The table "Food" from Figure 2.4 would look like: 1, 2, 3; ham, cake, fruit; pink, brown-yellow, orange; salty, sweet, sour. Examples for column-oriented databases are Facebooks Cassandra and Apache Software Foundations Hbase [Lea10, p. 13].

Graph-oriented databases

Graph-oriented databases have the ability to add new information very easily [WDM+12, p. 221]. Many graph based databases are so called triple stores based on the Resource Description Framework (RDF). Triples are built in a sentence like manner, having a subject, a predicate and an object. A triple in RDF, stating that marble cake is Arsins favourite food, would look like: S: Arsin P: FavouriteFood O: marble cake. The advantage of RDF databases is the fact that in contrast to other NoSQL databases there exist standards and a unified query language, which is called (SPARQL). However [WDM+12, p. 224] state that queries scale very badly and that execution time is proportional to database size, structure of the query and limitation. Their usage in large databases is therefore restricted to queries

with low complexity. One great advantage of graph databases is their ability to integrate new models on the fly without the necessity to change anything on the existing data [WDM+12, p. 223]. They also support federation with the cost of a 30% slow down for each additional repository. Examples for graph-oriented stores are AllegroGraph [ALLEGRO] and Neo4j [NEO4J].

Disadvantages and Limitations

For NoSQL databases goes the same as for relational databases – the advantage leads to some disadvantage as well. When one looks for consistency and reliability, the database system itself will not guarantee it natively – it has to be added specifically for each application. This is the trade-off that is done in order to achieve a very good processing performance. Also, most NoSQL data models have to be simpler to achieve the much desired performance boost. Even though some hybrids exist, most NoSQL databases don't work with SQL and require therefore manual queries which may become very time-consuming for more complex tasks [Law14, p. 285, [Lea10, p. 14]. As NoSQL databases are so different to each other there will never be anything like a unified query language or API [Bar10, p. 2]. As most NoSQL databases are open source products, they offer little or no customer support at all. In contrast to relational databases, the choice of one NoSQL system will possibly limit the further development process to a handful of programming languages and access methods, as each NoSQL database has its own APIs, libraries and preferred programming languages. Other problems, which will decrease within the next years, are the little number of professionals with the knowledge about these new technologies and the lack of maturity in most of the products [Lea10, p. 14]. Predictions for the future of NoSQL databases are that they will be mainly adopted for specific applications where large amounts of data and scaling are involved. At this point Monty Widenius shall be cited: "Non-SQL gives you a sharp knife to solve a selected set of issues. If you find SQL too hard to use, you should not try Non-SQL.", basically meaning that one should know the advantages and disadvantages of relational and NoSQL databases well, in order to decide for the best fitting option [Bar10, p. 1].

2.3 Memory Approaches in other Cognitive Architectures

Until today a great deal of research on *cognitive architectures* has been done, however a long time of research on any topic usually results in many opinions and different definitions. Generally a cognitive architecture should act in a way which can be said to be intelligent, but a general view on how this intelligence should look like has not been established until now. A definition widely accepted is that an agent of such a system should be able to act satisfactory in situations which are partly or totally unknown to it [VMS07, p. 151]. In order to do so, these systems have to provide some sort of functionality to use existent knowledge to "imagine" possible future outcomes of its current situation. Some people working on the topic suggest that a cognitive system needs to learn from experience in order to improve performance over time and some even demand the ability of self-reflection. Some of the architectures resulting from the research on this topic specialise on planning and problem solving, while others try to model human behavior, like emotions or motivation, along with their drawbacks, like a limited area of view. In the following some terms that are essential for understanding the presented architectures are discussed.

As can be seen from the previous examples, there are various interpretations of the topic of cognition. However, two basic approaches have evolved over time [VMS07, p. 152]. The so called *cognitivist system* approach treats cognition rather algorithmic, making use of symbolic information processing representational systems. It adopts many theories from artificial intelligence, like for example the hypotheses of Newell and Simon [VMS07, p. 154]. They stated that a physical symbol system is sufficient for general intelligent actions and that heuristic search should be used by such systems in order to solve tasks. In heuristic search rules are applied to symbol structures until a structure providing a solution is produced. In contrast to this approach, the *emergent system* supporters plead for a dynamical, self-organized approach to cognition, by using connectionist systems. They see cognition as the process that enables an agent to act reasonably in its environment. Therefore, in emergent systems, perception is considered to be a reaction to environmental changes. Also, they are bound to real-time operations within their environment, which is not the case for cognitivist systems. In an emergent approach embodiment is obligatory, while even though many cognitivist architectures have some sort of embodiment, they do not need it by rule.

In order to be able to act in a way that may be considered intelligent, an agent of a cognitive architecture needs to have some information about its environment and a basic understanding of situations that are likely to happen. Based on this knowledge the agent should be able to decide how to react to similar or even totally new events. Depending on the cognitive architecture the agent may then be able to memorize his new experiences, which may be of use for other situations in the future. These experiences are stored in the long-term memory. When it comes to implementing such a permanent memory, again there are many approaches, which are often influenced by theories on the human brain. Work in psychology highly suggests that the human memory is separated into several sub-categories of memory, each having different functional responsibilities [Deu11, p. 73]. The psychological theory of separating the memory in distinct types has been realized in many artificial intelligence projects until today. According to [ZLM09, p. 384] the most common memory approach in autonomous agents is the implementation of an “episodic” and a “semantic memory”, based on the theories of E. Tulving and A. Baddeley. The *episodic memory* represents what one “remembers” in contrast to the *semantic memory* which represents what one “knows” [DGLV08, pp. 7–8]. With an episodic memory, for example, it is possible to remember regularities which were not realized originally and combine them with actually experienced events. According to [NLA07, p. 1560] episodic memory greatly enhances the reasoning and learning capabilities of agents. For long-lived agents however such a powerful memory raises numerous challenges, like providing access to all collected memories in a reasonable amount of time.

Another psychological theory, that is often adopted, is the so called *chunking theory*. It states that the human mind uses discrete collections of features, all having a strong association to each other [GLC+01, p. 236]. The resulting conceptual groups are called *chunks* [SGL09, p. 2]. A popular example of such a group of elements, that gains a meaning when grouped, are words which are constructed through the grouping of single letters. According to [GLC+01, p. 236] the diverse research on chunking has also resulted in different meanings and applications. However, two widespread approaches are the so called *goal-oriented chunking* and the *perceptual chunking*. Goal-oriented chunking assumes that the process of chunking is a conscious one and therefore controllable, whilst

automated and continuous learning through perception is assumed by the latter one. The most common definition of a chunk is that of a declarative unit of knowledge or information.

The “unified theory of cognition” is a very popular theory, which was presented by Allen Newell in the year 1990. He basically stated, that the mind is one single system and that any cognitive model should provide a theory for all of its functions. Consequently a system based on the unified theory should provide a concept for knowledge representation, a learning process along with a way to react to unknown situations and to achieve its objectives. According to Newell some of the advantages that are supposed to result from this approach are the ability to attack real-world problems and that it is possible to integrate the huge knowledge that is provided by cognitive neuroscience methods [ABB+04, p. 1037].

There are several agent-based simulation systems dealing with the problem of storing and retrieving memories. In the following sections some of them will be presented. As the main interest of this work lies in storing and retrieving agent memory data, particular focus will be on answering the following questions:

- What memory aspects are used by the system?
- How is the memorized information represented?
- What approach was chosen to search and find the information the agent needs at runtime?
- How is new information gathered and saved?

However, as the application areas and approaches differ widely, not all the questions can be answered for each of the presented systems.

2.3.1 CHREST

CHREST (Chunk Hierarchy and Retrieval Structures) was developed to be a general-purpose architecture, simulating selected areas of the human mind [SGL09, p. 1]. The favoured test scenario for this architecture is a chess setup where circumstances threatening the king shall be recognized. As the architectures name already indicates, one of CHREST´s basic concepts is the adoption of the chunking theory [SGL09, p. 2]. A remarkable feature that makes CHREST stand out from several other architectures is the fact that it imposes costs for cognitive operations, to make the simulation of the human´s cognitive system even more realistic. The imposed constraints and the modelled processes were obtained from empirical studies on human behavior. Examples for restraints are limitations on the visual short-term memory and the systems learning rate, the time for moving the eye is set to 30ms and also the time for mentally moving a chess piece is simulated by an internal system clock [GL10, p. 2, [SL07, p. 3]. This approach is in high contrast to many other systems which are designed to attach greater importance in succeeding in complex tasks on large amounts of data. Despite those self-imposed constraints CHREST has proven itself a successful model of the human mind in various application areas like physics representation, ageing and perception, language acquisition and chess expertise [SGL09, p. 2].

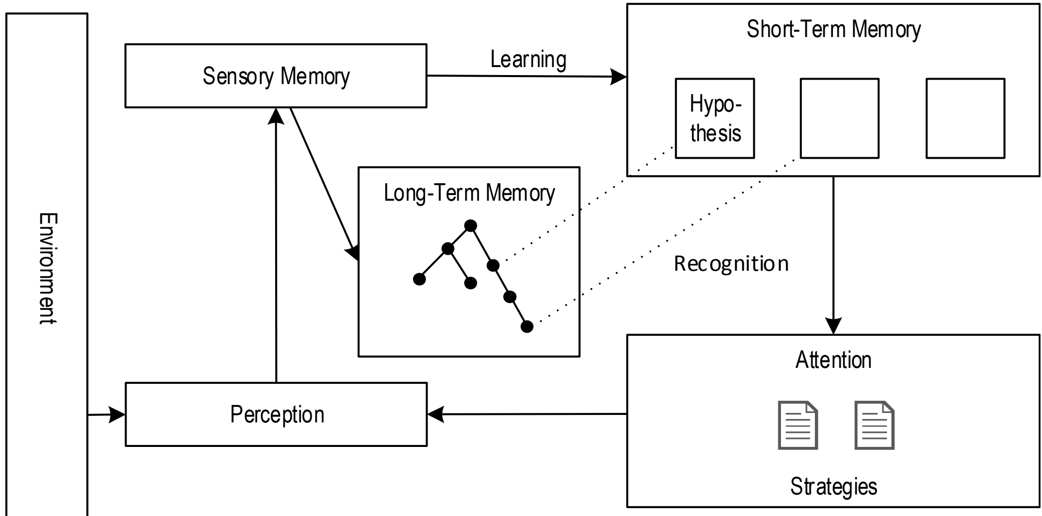


Figure 2.5: CHREST architecture based on [SGL09, p. 2]

Architecture and Concepts

In Figure 2.5 a basic depiction of CHREST´s architecture is shown. CHREST possesses an elaborated attention and *perception system*, simulating eye movements to the actual areas of attention [SGL09, p. 2]. The *sensory memory* is a short term memory responsible for storing the objects currently in view. For example, if the system learned in an earlier game that the king, being positioned on field c6, was in check by a rook on c2, it would probably lead its attention to field c2 after realising the king is on field c6. Newly added information may lead the system to direct its attention to other parts of the scene during the course of the program execution. This is part of CHREST´s approach to decide which information is of relevance and which may be safely ignored [GL10, p. 2]. The thus limited perception and the already mentioned cost for cognitive operations act as an effective filter for separating useful information from needless distraction. The perception system is also responsible for the systems learning process, using the links between chunks in the long-term memory to determine whether it is part of a previously learnt chunk [PDH+07, p. 3]. This leads to a very close interaction between perception, learning and memory. Perceived information directs the attention and consequently perception. Finally, perception decides which information is learned and stored to memory [GL10, p. 2].

Processing Cycle

A typical program cycle in CHREST starts with perceiving an image, which is limited to the area of view. The perception is also done in a cyclic process, where at first the last used heuristics are cleared and then the actual centre of view is determined [LGS09, p. 187]. Then, until the presentation time or the number of allowed fixations is reached, all fixated items are stored in a list and put into the sensory memory. The perceived information is then used to search for a familiar pattern in the long-term memory. Patterns that were found are then transferred to the short-term memory. In order to decide to where the attention should be paid, previously learnt information from the *sensory memory* and heuristics are combined. This behavior is based on the fact that what has led to gaining important information in the past, may as well be of importance in the current situation [GL10, p. 2]. This retrieved

knowledge is then combined with domain specific knowledge and information from the peripheral view to determine the next point of attention and determine whether the king is in a check [LGS09, p. 2].

Memory and Information Representation

CHREST´s long-term memory represents its data, connected by semantic links, in a graph like network. This "chunking network" is a discrimination network which is responsible for retrieving and sorting chunks [GL10, p. 2]. Chunks are familiar patterns, which are held in nodes by the network. Beginning from the root node test links are used to go through the network, tracing those links where a match could be found [LGS09, p. 185]. New information is added by two learning mechanisms which make use of pattern matching: if a new pattern is sent to a node and the pattern mismatches the chunk stored at the node as well as the test links succeeding the node, a new test link is created from the mismatching part of the pattern and added to the long-term memory. If only the test link is a mismatch, additional information from the partially matching pattern is added to the already stored chunk, adding more detail to the already existing one, by this means extending the tree structure by branches and depth. The process of enriching already existing information is called *familiarisation*, whereas adding a totally new chunk is referred to as *discrimination* [SGL09, p. 2]. Again CHREST simulates measured human behavior, by using learning mechanisms that slow down the learning at the beginning, but gain speed after the system has gotten more acquainted with the knowledge domain [LGS09, p. 186]. Efficiency in a large data store is a typical problem of cognitive architectures, which is not shared by CHREST. Even if it aims to simulate a human chess master who has an average of 250ms when retrieving a pattern from his or her memories, CHREST can easily compete even within a network of 300,000 chunks [LGS09, p. 186]. A typical chunk in CHREST may represent the position of a chess piece on a field, like "white king on square g1" [SGL09, p. 2]. The short-term memory, which is divided in a visual and a verbal part, can hold up to four chunks, which is imitating the average short-term memory capacity of a young human adult [SL07, p. 2]. Chunks in the short-term memory are references to those stored in the long-term memory. Due to the familiarisation process a system with a lot of prior knowledge may be in possession of chunks with a lot of detail added and consequently hold much more information in the short-term memory than a system with only basic knowledge [LGS09, p. 2]. However, larger patterns must prove to be of real consequence for the actual situation in order to be held in the short-term memory.

A very essential part of CHREST is the concept of *templates,* which is an attempt to explain human expert recall abilities [LGS09, p. 186]. For example, it would explain how a chess master manages to recall chess positions after only seeing them for 1 or 2 seconds [GL10, p. 2]. Templates are schema-like structures which are created from often reoccurring patterns [SL07, p. 2]. A template combines a constant core information together with variable information which is held in so called slots, which allow rapid encoding of values. To mark two nodes, which are in the possession of similar features, s*imilarity links* are used. A similarity link is created if two chunks were retrieved into the short-term memory and the two nodes match. If the chunks in the nodes are of different type but have similar descendant test links, a so called *generative link* is created instead. When a node has built up a satisfactory number of these links and the so connected nodes match an overlap criterion, the information of these nodes is combined to form a template.

In Figure 2.6 an example for a visual training setup along with a possible memory outcome is presented. On the left side a chess-like board with field numberings and game figures (represented by the letters on the fields) are shown. This setup is used to fill and test the agent´s memory by presenting it for some seconds. After a certain amount of time has passed the agent has to recall as much as possible of the shown setup. Due to the theory about human expert recall abilities, which was explained earlier in this chapter, after a period of training the agent should be able to recall much more of the shown setup.

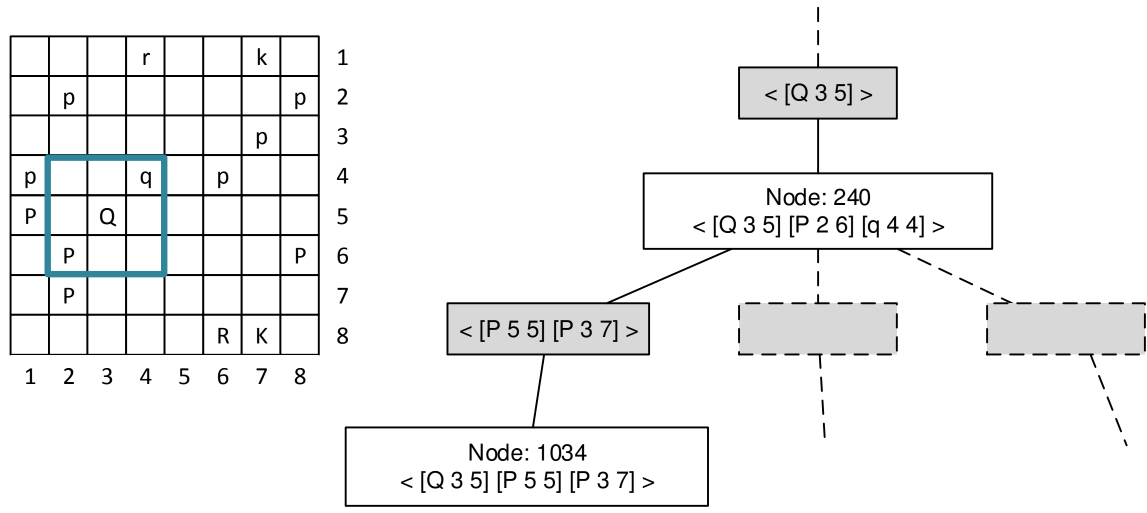


Figure 2.6: Example situation with discrimination network

On the right side of Figure 2.6 a part of an agent’s discrimination network is shown. The grey rectangles represent test links, while the white rectangles represent “real” nodes, each having a unique number and containing a chunk of information. In node 240 one can recognise the scene that is marked by the rectangle containing the fields from line 4 to 6 between columns 2 to 4. As this part of the setup was already encountered before, the agent is also likely to remember this part of the setup, even if only shown shortly. In node 1034 one can see a node which was created through discrimination after encountering a pattern which contained [Q 3 5], but mismatched the rest of the pattern which was stored in node 240. Therefore a new test link, containing the mismatching information, leads to node 1034 containing the new pattern.

The usage of discrimination networks might be an interesting option for the SiMA long-term memory implementation. It should be possible to save the SiMA data structures in such a structure, however performance might become an issue, due to the complexity of the SiMA data structures.

2.3.2 ICARUS

Being originally designed an architecture for reactive execution [LCT11, p. 4], ICARUS considers interaction with the physical world as much more important than many other cognitive architectures. Its primary aim is to provide a computational theory of intelligent behavior and a unified theory. According to [CTN09, p. 109] the basic principles of this cognitive architecture are: cognitive reality of

physical objects, cognitive separation of categories and skills, primacy of categorization and skill execution, hierarchical organization of long-term memory, correspondence of long-term or short-term structure and modulation of symbolic structures with utility functions. Even though the system modules all have their own tasks and purpose, they depend on each other, which is due to the declared aim to provide a unified theory [LCT11, p. 1].

Architecture and Concepts

In Figure 2.7 ICARUS´s basic architecture and its four main modules are depicted [CTN09, pp. 109–111]. In order to describe its environmental situation for the actual cycle ICARUS utilizes a short-term memory called *perceptual buffer* [LCT11, p. 5]. Elements in this memory are called *percept* and specify one particular object in the environment by its type, unique name and a set of attribute-value pairs, describing the object's actual state. The *motor buffer* is responsible for transferring skill signals from the short-term skill memory to execution in the environment. The *conceptual memory* has a short-term aspect which is also known as the belief memory and a long-term aspect representing the known conceptual structures. *Beliefs* are inferences that are created by the agent on observation of the environment, representing the current problem state [LCT11, p. 20]. G*oals* are the agent´s objectives or desires, while *intentions* represent the activities which are to be carried out in order to change the actual state of the environment to reach the current goal. The *skill memory* contains the knowledge about executable skills that can be utilized to reach the respective goal. While the long-term aspect of this memory contains all executable skills, is the short-term aspect responsible for transferring the skill that was chosen to be executed to the motor buffer. In order to be able to represent situations (for example in skills or goals) in a more general way, special atoms known as *pattern-match variables* are used. For example the specific pattern (on blockA blockB) can be expressed in a more general way like (on ?x ?y), to state the fact that one thing is on top of another, instead of restricting it to “blockA” and “blockB”.

Processing Cycle

Like in other architecture´s the actions that are executed are determined in processing cycles. In each cycle ICARUS first updates its beliefs about the environment, by adding them to the perceptual buffer, from where they are matched against long-term concepts. This is done by the *inference module*, which uses a bottom-up approach to match percepts until it has added all beliefs that are deductively implied by the actual perceptions and concept definitions, to the belief memory [LCT11, p. 11]. As a result of this process the inference module finds all logically implied beliefs about the environment for each cycle and the beliefs about the environment are constantly updated. After the conceptual inference is done the system determines which skill clause (also called action) can be applied to the actual situation. This is done by simply checking which skill clause conditions are met and whether the clauses effect would change the actual situation (if the effect matches the actual state, there is no use to apply it) [LCT11, p. 17]. If several skill clauses would match, one of them is selected randomly and an intention, containing all details about sub-skills, bindings and conditions, is created and afterwards carried out. In order to reach a certain goal ICARUS may use sub-skills which are chained one after each other, containing a pointer to their parent skill. In several cycles skills in this chain are executed

until the top-level intention was executed. After task completion, ICARUS halts and awaits new commands. If the system, during execution of a skill chain, realizes that the next skill cannot be carried out, it is considered to be an error. Possible error sources may be too general or too specific memories or influences from the environment like other agents [LCT11, p. 19]. If such problems arise the system simply drops all intentions along with their ancestor intentions and starts again.

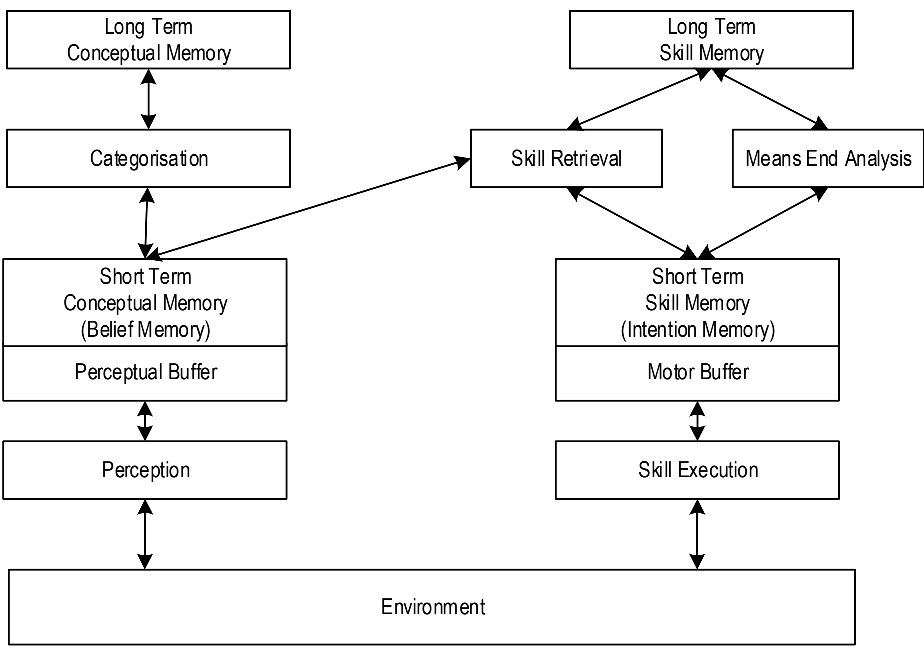


Figure 2.7: Schematic diagram of ICARUS based on [CTN09, p. 110]

Memory and Information Representation

As can be seen in Figure 2.7 ICARUS utilizes several memory aspects in order to manage its knowledge. One of these memories is the perceptual buffer, which is a short-term memory containing the perceived information about the current environment. Based on this information some higher-level inferences are created and stored in a belief memory, which is also referred to as the short-term conceptual memory. Usually knowledge that is held in this memory contains information about the physical relations among objects. Belief memory elements combine a predicate and a set of symbolic arguments to form a list. Arguments in this list have to be connected to the long-term memory, by referring to one of its objects. From this it follows that no short-term knowledge can exist without a corresponding long-term counterpart, leading to a strong relation between those two memories [LCT11, p. 7]. However, for programming convenience static beliefs, that will not change over time and therefore need no link to the long-term memory, may be defined for the belief memory [LCT11, p. 7]. Concepts which can be instantiated by beliefs in this memory are held in the long-term conceptual memory [LCT11, p. 7]. They are symbolic and relational structures, describing classes of situations in the environment. Clauses in this memory have a body stating the matching conditions for the short-term memory and a head specifying their name and arguments. In the body it is defined which objects have to be present and which other concepts, like numeric relations or other Boolean constraints have to be

true. The long-term as well as the short-term aspect of the conceptual memory contain modular elements that are composed dynamically during performance or learning [LCT11, p. 3]. Those memory elements are represented by list structures, on which pattern matching is performed in order to match long-term and short-term memory information.

In order to take action in its environment, ICARUS utilizes further memories which are linked by performance mechanisms. One of these memories is the long-term skill memory which stores information about actions that can be used in the environment. The information structure of a skill has a high resemblance to the elements of the conceptual memory [LCT11, p. 12]. As before the head contains name and arguments, the body defines which entities have to be present and the conditions that have to be matched along with the effects that trigger in the case of performing this action. When it comes to taking action the system utilizes a short-term *intention memory* where a skill clause from the long-term skill memory is instantiated in order to execute an action in a particular way [LCT11, p. 16]. This memory keeps instances of skill clause elements, describing in which way the agent executes a particular skill. ICARUS intentions may have sub-intentions, but the system can only focus on one intention at a time.

As ICARUS is written in the programming language Lisp, it also expects all its input parameters, like conceptual clauses, skill clauses, goal stacks and beliefs, to be defined by this language [LCT11, pp. 28–29]. The definition of conceptual memory elements shows therefore a similarity to Horn clauses which are adopted by the programming language PROLOG. A concept in ICARUS´s memory might look like:

```
((on ?block1 ?block2)
:percepts ((block ?block1 xpos ?xpos1 ypos ?ypos1)
           (block ?block2 xpos ?xpos2 ypos ?ypos2 height ?height2))
:tests ((equal ?xpos1 ?xpos2) (>= ?ypos1 ?ypos2) (<= ?ypos1 (+ ?ypos2
?height2))))
```

The first line already contains two pattern match variables, which are indicated by question marks. In this case the line states that `?block1` can be put on `?block2`. However the first line gives no information about what `?block1` and `?block2` might be, as this is defined in the the following lines after the label `:percepts`. The percepts section defines, what the agent must perceive in order to take action. The first percept block, which is marked by parenthesis states that `?block1` is a block with some undefined `xpos` and `ypos`. For the variable `?block2` the same is known, plus it has a height. After the percept block a test block follows, stating that the `xpos` of `?block1` and `?block2` `(?xpos1 ?xpos2)` have to be equal, it also states that he `ypos` of `?block1` has to be greater or equal the ypos of `?block2` . At last it states, that the height and `ypos` of `?block2` have to be greater or equal than the `ypos` of `?block1`. In order to be able to apply this concept all percepts and tests must hold.

For SiMA little information can be drawn from the very specific approach of ICARUS. As Lisp is a programming language of its own, ICARUS´s particular approach of defining and persisting its memory, may be considered as of low consequence for SiMA´s long-term memory implementation.

One may notice however that a file-based approach, with preloading into in-memory storage of the program was chosen. The chosen information representation differs greatly from the concepts in SiMA and are possibly not suitable for an approach based on psychoanalytical theories. Reason for this assumption is that ICARUS uses a rule-based system, which is much more related to programming and machine “thinking” than the current concept which is used in SiMA. A collection of if-then statements cannot be used to represent the SiMA data structures, like for example a thing-presentation-mesh, in a reasonable manner.

2.3.3 ACT-R – Adaptive Control of Thought Rational

The project Adaptive Control of Thought Rational (ACT-R) aims to provide an integrated agent system, able to handle real-world problems and capable of handling masses of data [ABB+04, p. 1037]. Its modules are conceptually based on the presumed functions of the cortical regions (special areas of the brain) and their functionality is combined by a central production system. It belongs to the section of production systems, making use of production rules in order to initiate state changes to the system [Zei10, p. 28]. As this architecture is strongly focused on the cognitive process of memory storage, it does not consider other motivational factors, like for example emotions, for its decision process [Lan10, p. 10].

Architecture and Concepts

ACT-R´s basic architecture (see Figure 2.8) contains several modules, each devoted to process a different kind of information for the whole system. Each module progresses independently of the others, but makes its computations available in its own buffer. According to [ABB+04, p. 1038] the perceptual and motor processing has a high influence on the nature of cognition. Furthermore the integration of information from the external world can be of high value for the cognitive processes. Hence, even though ACT-R originally focused on higher level cognition and not perception or action, it implements nowadays a perceptual-motor system. It is based on a *manual buffer*, which is responsible for hand movement, and a *visual module* keeping track of locations and the existing visual objects [ABB+04, p. 1038]. The visual module is divided in a visual-location module, responsible for determining the location of objects and a visual-object module, which has the task to recognize objects [ABB+04, p. 108]. The central *production system*, combines the information from the separate modules` *buffers* and decides for a *production rule* to be executed. Production rules represent ACT-Rs procedural memory and contain knowledge about actions that can be executed under special circumstances. An example for such a production would be: IF declarative memory precondition is met THEN execute this action. The production system is not sensible about the computations and processes which are running inside the separate modules, but is restricted to the selective information that is provided by the modules in their buffers, when deciding for a production rule to execute [ABB+04, p. 1037]. The *declarative module* is responsible for retrieving and providing information from the system memory, while the *goal module* keeps track of the actual intentions ACT-R supports parallelisation for most parts of the architecture, however, some actions have to be processed in a serial manner as they are dependent on the communication between the modules [ABB+04, p. 1038].

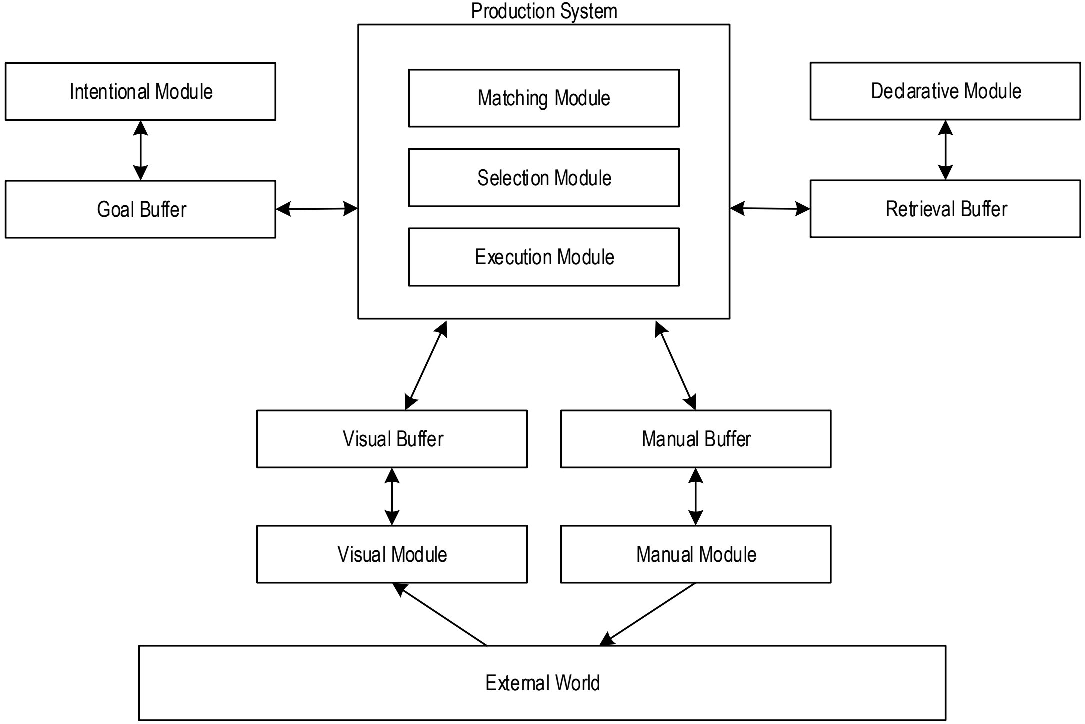


Figure 2.8: ACT-R basic architecture based on [ABB+04, p. 1037]

Processing Cycle

ACT-R´s processing cycle starts with the visual module identifying objects and providing its information in the visual buffer, so that the production system may fetch it from there. The central production system, combines the information from the visual buffer with information from the other modules in order to decide for a single production rule to be executed. The procedural memory, which is in ACT-R represented by its production rules, is responsible for detecting patterns in this information and decides what to do next [ABB+04, p. 1044]. In order to match situations in the environment, with the current goals and information from the memory a pattern matching process is required. Therefore the production system contains a *matching module*, which is performing a pattern-recognition algorithm on the information that is provided by the buffers [VMS07, p. 164]. If the production system finds, that several production rules could be applied in the current situation, the thus acquired information is passed to the *selection module*, which in turn performs a conflict-resolution. In principle several production rules could apply to the actual situation, but due to the seriality of the system only the rule with the highest utility will be selected for application [ABB+04, p. 1044]. The utility value is calculated by taking into account the probability of reaching the goal when the rule is executed, the goal's value and the cost of reaching the goal. The cost and probability values are learned by experience with the production rule. The thus selected production rule is then used to initiate a state change in the system. This is done by updating all buffers for a new processing cycle, by putting chunks into the encapsulated modules` buffers. In order to set an action in the external environment a chunk with an action request is put in the manual module´s buffer. The manual module is then used to control the hand actuators to interact with the environment. To accomplish the final objectives production rules

may create sub goals. New sub goals are pushed on top of their parent goals in a goal stack, which is used to manage the relation of all goals. One drawback of ACT-Rs processing is the fact that it is nearly impossible to interrupt an ongoing goal [TCM03, p. 4]. The reason for this is that in order to change a goal, the system's attention has to be directed to something which is unrelated to the current objective.

Memory and Information Representation

ACT-R provides two sorts of long-term memory, namely a declarative memory, which represents the long-term semantic memory and a procedural memory which is represented by production rules [ABB+04, pp. 1042-1044]. The declarative memory contains information about personal and cultural context, while the production rules represent knowledge about the performance of actions [ABB+04, p. 1042]. All information in ACT-R is represented by chunks. They are used to pass information between the modules and the production system and represent knowledge from the memory. A chunk combines attributes (also called slots) and values, in addition to the usual structure of a chunk, ACT-R chunks also have a name for convenience. Each slot of a chunk is constrained to a single value. A crucial part of ACT-R is the activation process, which basically decides the probability that some chunk of information is retrieved from the memory. The activation of a chunk is decided by its base-level activation, reflecting how useful the respective chunk has been in the past and its association strength to the elements contained in the current goal. In absence of respective stimuli, an agent should still be able to plan its moves in a beneficial way in order to reach his goal. Therefore, some sort of goal tracking is inevitable for any cognitive architecture [ABB+04, p. 1041]. Again chunks are used to model the system's final objective. In this case a chunk representing an end-state of the declarative memory is put into the intentional module at system start [TCM03, p. 3]. This makes the intentional module to a short-term memory for goals, while other short-term information is tracked by the buffers.

An example for chunks, which are used as an information representation for all information, except for declarative knowledge can be seen in Figure 2.9. Each of the two chunks consists of 3 slots each having an attribute an exactly one value [ACTR]. The first chunk is called Action023 and represents the fact that "the dog chased the cat". The second chunk has the name Fact3+4 and states that 3+4=7.

```
Action023
    verb chase
    agent dog
    object cat
```

```
Fact3+4
    addend1 three
    addend1 four
    sum seven
```

Figure 2.9: Action and fact representation in ACT-R

Just like ICARUS ACT-R is a Lisp program and the input files, containing all program information, like the long-term memory, are written in Lisp as well. For the scope of this work ACT-R´s approach is therefore inapplicable. The chosen information structure seems to be highly efficient, however a comparison to the complex data structures which are used in SiMA (described in chapter 2.1), is not appropriate due to the differing natures of the structures.

2.3.4 SOAR – State Operator Apply Result

SOAR (State, Operator, Apply, Result) is a general cognitive architecture, which was started in 1983, aiming to define a model of the human cognitive apparatus [Lai08, p. 1]. SOAR´s declared goal is to support all abilities expected of an intelligent agent, like employing the full range of problem solving methods, learning about all aspects of a task and its performance and representing and using appropriate forms of knowledge [LC14, p. 1]. The hypothesis SOAR builds on is that all goal-oriented behavior can be modelled as selecting and applying operators to states. A state represents the current situation and in order to change this situation operators are used to initiate the transformation of a state. All decisions are made at run-time, by interpreting the current environmental situation from the short-term memory and combining it with relevant knowledge from the long-term memory.

Architecture

In Figure 2.10 the basic components of SOAR´s architecture are presented. A perception module passes information from the environment to the working memory. The working memory is responsible to keep track of the current situation and goals. The decision procedure´s tasks are to detect impasses and select operators. An *operator* is SOAR´s concept to initiate state changes solely in the working memory or in the environment [Lai08, p. 3]. An *impasse* is a situation in which the current information is considered to be not sufficient to decide for an action. The long-term memory is split up in a procedural memory, a semantic memory and an episodic memory. The *procedural memory* contains the production rules, which contain knowledge about which action may be executed if a certain precondition is fulfilled [LC14, p. 7]. The *semantic memory* stores general knowledge about the world, whilst the *episodic memory* contains memories about the agent experiences [LC14, p. 95].

Processing Cycle

SOARs program procedure builds on a processing cycle, which is repeated until a halt action is issued or the user interrupts the processing [LC14, p. 21]. At first the latest *perception* (input from the body sensors) is passed into the working memory, which holds all information about the current environmental situations and the actual goals. Based on the perception and the information that is held in the working memory the proposal is started. In this step productions are fired and retracted to propose operators that meet the actual requirements. If no further proposals are possible, a new state is created either by selecting a new operator or detecting an impasse. In this case a sub-state, in which SOAR tries to solve the impasse, is created. If the impasse was resolved SOAR can use its learning mechanisms, to store the new problem solving approach to its long-term memory [LC14, p. 30]. In the application step productions are fired to apply the operator. Finally the output commands are sent to the external environment and the program returns to the beginning of the cycle.

Memory and Information Representation

SOAR originally distinguished only between permanent knowledge (procedural memory) and temporal knowledge (short-term memory or working memory). Today it supports 2 additional long-term memories (episodic and semantic, see Figure 2.10). The reason for this extension was the finding that productions alone were insufficient for more complex environments [LC14, p. 1]. In order to represent

the agents' permanent knowledge SOAR uses productions, which can be compared to conventional "if-then" statements, known from common programming languages [LC14, p. 7]. *Production rules* define the actions which shall be fired if all conditions are met and are therefore the main drive of the processing cycle. They are stored in the long-term *procedural memory*.

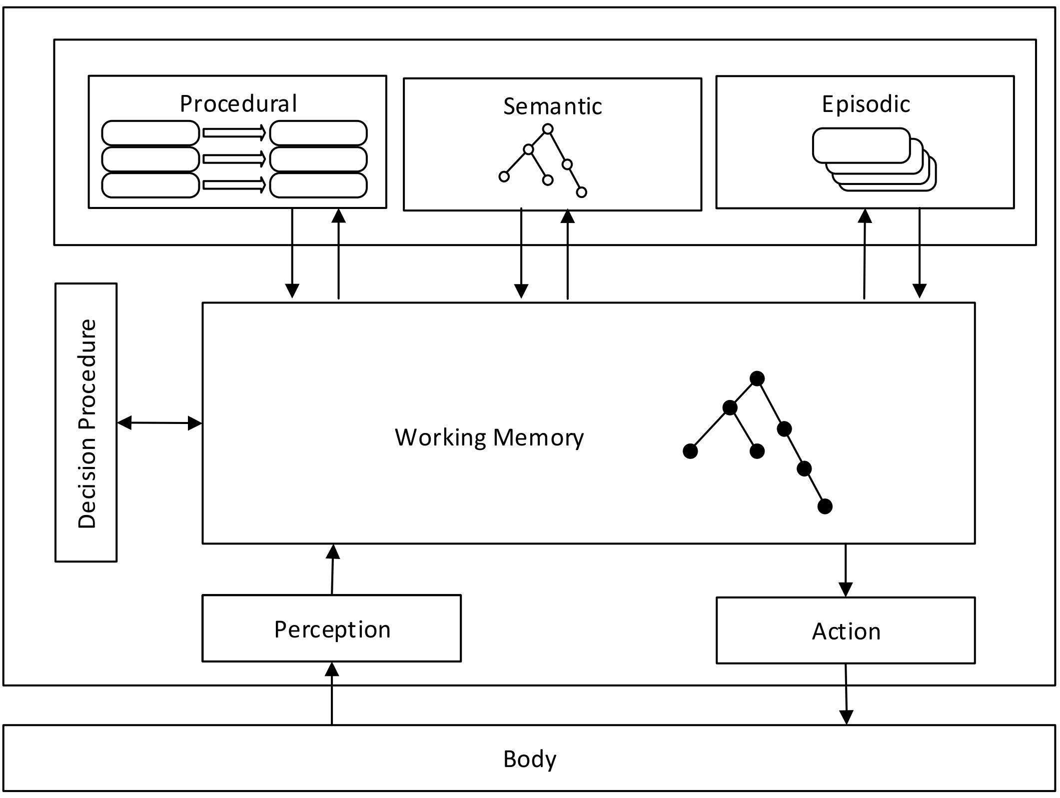


Figure 2.10: The SOAR Architecture based on [Lai08, p. 5]

SOARs short-term memory, the so called *working memory,* tracks information from the sensors, intermediate inferences, goals and operators. In short, it reflects the current knowledge of the world [LC14, pp. 13–15]. It encloses so called working memory elements (WME´s), each containing one separate information about the world or its objects. Every WME has the form of an identifier-attribute-value triple. Identifiers are used to connect several WMEs, each describing one attribute, into one object of the working memory. While every triple can exist only once at the same time, there is the possibility of multi-valued attributes, meaning that the same identifier and attribute are existent with different values [LC14, p. 14].

Declarative facts about the world, such as water is wet, or that cats are animals, are stored in the semantic memory [Lai08, p. 7]. In SOAR the *semantic memory* is created from structures that are held in the working memory. SOAR´s *episodic memory* stores information about when and where certain events have been experienced. It includes and connects specific instances of the working memory structures. Through those connections temporal relationships and the context of past experiences can be reconstructed. Again cues are used to search for the best match, save that the definition of a best match is influenced by recency and working memory activation. The activation level represents the

last usage for any sort of knowledge. Episodic memory is task-independent and enables many sophisticated cognitive abilities, like internal simulation and learning.

SOAR´s memorization process can be divided in three phases: encoding the agent’s state, storing the information as episodic knowledge and take measures to support future retrieval. SOAR uses connected digraphs to store its states. By constructing a cue, which is a directed connected acyclic graph containing task-relevant relations and features, the memory can be queried for memories [DALL12, p. 1]. A cue-matching process searches for the episodes best match, which is defined as the most recent and most similar episode that is stored in the memory. If an episode that has at least one feature in common is found, the cue-matching process returns it. A working memory activation system increases the activation level of a WME every time it is used by a rule. In the further course of the agent’s life time this activation level automatically decreases over time, so that recently used information is considered more important than older one [NLA07, p. 1561]. In order to avoid memory blow up SOAR uses the fact that changes between agent states are very small and many structures are going to repeat themselves over the time the agent knowledge builds up and therefore can be reused [DALL12, p. 1].This leads to a worst case of linear scaling for episodic memory processes in comparison to the state changes.

As a storage solution for its episodic and semantic memory SOAR uses the relational database system SQLite, in order to facilitate standardized storage and querying of knowledge [LC14, pp. 95–114]. This enables users to use any standardized SQLite program or component to access the information gained. Although a disk-based storage is possible, it is recommended to use the in-memory mode for most runs. To improve performance several configuration options are provided by the system. First, a *lazy-commit* parameter may be set to true, which will cause the system to make changes to the memories permanent only after the SOAR kernel has shut down. Fewer disk writes greatly improve the performance of the system [LC14, p. 102]. Secondly, A *thresh* parameter is used to set the upper bound of augmentations after which an elements activation is stored. As every WME augmentation is sorted by activation, this enables the user to keep a balance between updating activations and the number of long-term identifiers that have to be resorted. Furthermore, there are two parameters that are connected to the SQLite cache, which is a storage structure that keeps in memory information from the database, in order to speed up some operations like querying [LC14, p. 103]. There is the *page-size* parameter to set the page size of the cache in bytes and a *cache-size* parameter to set the general cache size, in terms of pages. Finally, it is possible to speed up the system by the *performance* parameter, which reduces data consistency by no longer waiting for writes to complete, before continuing the execution.

A textual representation of an episode be seen in Figure 2.11. The chosen example represents a PAC-MAN-like game, which is introduced in the SOAR Tutorial [Lai14, p. 46]. There exists one blue agent in the environment, information about which was saved to line 11 in the episodic memory representation of that scene. The line states that the thing with id13 has the name “blue”, an actual game score of “0”, an x-position of 6 and a y-position of 16. This is also an example of a multi-valued attribute, which could also be read as the triples "<id13> ^name blue", "<id13> ^score 0", "<id13> ^x 5" and " <id13> ^y 9".

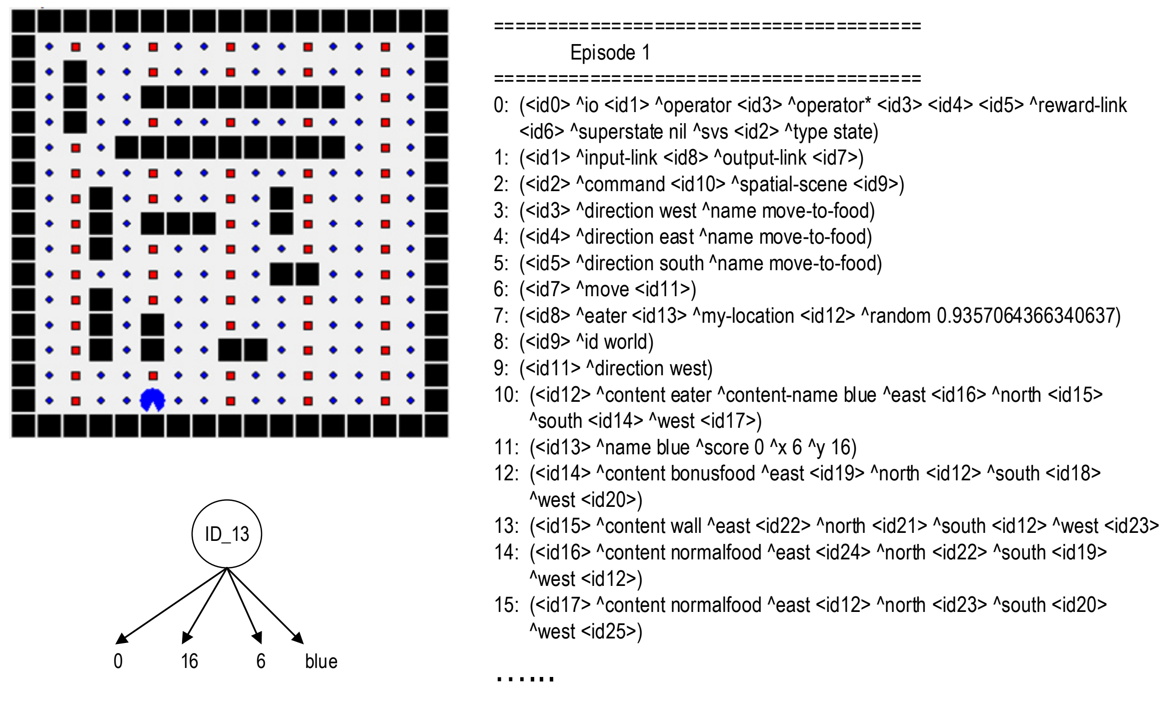


Figure 2.11: Eaters game with episodic memory content

As SOAR uses a triple representation of its memory contents, it is possible to represent the information in a graph view as well as in a textual representation. SOAR therefore offers the possibility print contents of the memory to a file, which can be opened by any Graphviz renderer, for further inspections [Lai14, pp. 213–214]. A graphical representation of the blue agent is as well presented in Figure 2.11. Identifiers are always presented in circles, attributes are indicated by a labelled arrow, with a value belonging to that attribute at the end of it.

There are several aspects of SOAR´s long-term memory approach, which are of interest for the long-term memory implementation of SiMA. First of all it is of importance to acknowledge the cost of a "real" permanent storage in the form of a disk-based approach, which was experienced by the SOAR developers. Also the possible countermeasures to performance loss, if such an approach is still chosen, may be of interest in the further progress of this work. Secondly, the triple representation which was chosen for the representation of the working memory elements, seems to be an interesting approach to information representation. Furthermore the usage of a relational database could be adopted for SiMA, as it is written in the programming language Java and is therefore absolutely compatible with a remarkable number of relational database implementations.

2.3.5 BDI – Belief, Desire, Intention

Meeting real time requirements is a very difficult task for any cognitive architecture, as it puts very hard time constraints on processing and decision making. The architecture of BDI (Belief, Desire, Intention) was developed to meet these constraints, by reducing time consumption in its planning and

reasoning processes [CTN09, p. 111]. Due to this approach it is able to react to the environment, without changing the agent´s original intentions. Notable at this point is that BDI is no specific implementation, but was originally created as a planning theory and has many implementations until today.

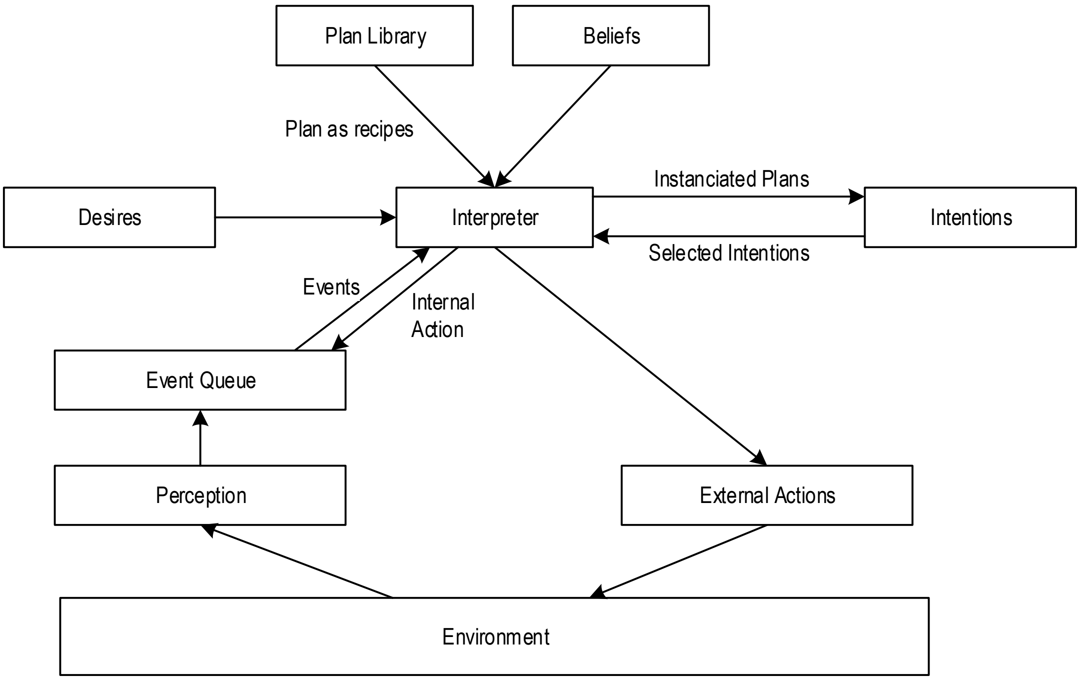


Figure 2.12: BDI architecture based on [CTN09, p. 112]

Architecture and Concepts

In Figure 2.12 a schematic depiction of a BDI´s architecture is presented. Together with the name giving components belief, desire and intention, the interpreter forms the main structure of BDI. *Beliefs* represent facts about the environment and inference rules, which in turn can produce new belief through inference on the existing knowledge [CTN09, p. 111]. Through execution of intentions and perception of the environment the agent´s beliefs are updated regularly. Knowledge about how to achieve intentions is stored in sequences of action which are referred to as *plans* and are stored in a *plan library*. A plan always states the conditions under which it can be executed and may also contain subgoals that have to be achieved. Execution of a plan containing subgoals may lead to the invocation of new plans. The term *intention* is used for plans that were picked to be executed by the agent in order to reach its goals. An intention takes the form of a stack containing plan instances. Goals, in BDI referred to as *desires*, describe the state which the agent strives to reach [CTN09, p. 111]. An *interpreter* is responsible for the coordination of the different components in BDI. In order to control the components it manipulates their contents. An *event queue* keeps track of historical information, which can be the acquisition or the removal of a belief, the reception of a message or the acquisition of a desire [HFsP+04, p. 2].

Processing Cycle

First newly perceived events are updated in the event queue. Afterwards the plan library is checked for new desires that may arise from the recently perceived events. From all the potentially fitting plans one is selected and transferred into an instance plan and pushed onto an intention stack. Depending on whether it was triggered by an internal event, for example by a plan, or an external event, it is pushed onto the existing intention stack or respectively a new stack is created. After executing an intention the cycle starts again. BDI tries to cut down decision time, by sticking to a committed strategy without considering any other solutions, even though they may be better than the one that was picked. The only two ways for the agent to drop a goal are the accomplishment of it or to realize that all applicable plans have failed and the goal therefore cannot be accomplished [CTN09, p. 112].

Memory and Information Representation

The original BDI structure was not presented with any concept of a long-term memory or a learning approach. The basic concept depends on a *plan library* where predefined plans are stored for retrieval by the interpreter at runtime. However the BDI concept does not exclude the implementation of a long-term memory and learning by principle. One example of a learning BDI based architecture is dMars where different learning levels were implemented [GHES05, p. 193]. A plan typically consists of a trigger or invocation condition, a context or precondition, a maintenance condition and a body [DLG+04, p. 9]. The *trigger* states a condition under which a plan may be taken into consideration. The context states which conditions have to be true in order to start a plan execution. A maintenance condition may define conditions which have to be true over the whole plan execution. Finally the body contains actions, goals and subgoals, that are needed to fulfill the final goal.

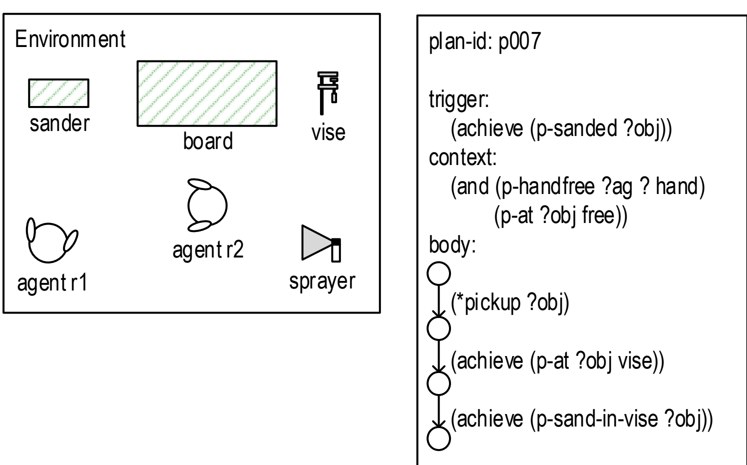


Figure 2.13: Simplified BDI plan based on [GHES05, p. 187]

In Figure 2.13 a simplified BDI plan along with a representational environment is shown. The environment contains two agents and several objects which may be used by them. The plan contains a trigger, a context and a body. The trigger states that if the goal is to sand an arbitrary object, the plan may be taken into consideration. In this case the variable `?obj` is used as a placeholder for anything that is contained in the environment. The context states that the agent's hand as well as the object have

to be free for use, in order to start the execution of the body. To mark external actions * are used, like in the first line of the body, which states that the object which shall be sanded has to be picked up. The external action is followed by two goals which are to be sent to the event queue if the plan is executed, from there other plans may be picked up in order to achieve the final goal. In this case the goals are to put the object into the vise and to sand it there.

In dMars two levels of learning were successfully introduced: learning from the environment and learning from direct interaction with other agents [GHES05, p. 193]. In the first case the agent may try to learn why an intention has failed, in the second case it may choose to communicate with another agent in order to get more learning examples or a reason why his intention failed, when the agent is not able to explain it without help. For the learning process the "learning from interpretation system" ACE was embedded into the architecture. ACE operates on logical decision trees, therefore dMars generates an ILP (Inductive Logic Programming) learning set containing training examples, background theories, ACE configuration, desired output, etc. This learning set splits up in several files with different extensions (like for example .kb, .s and .bg), which are then transformed into a decision tree for the agent.

The dMars approach to learning and memory is, just like the ones ICARUS and ACT-R (see chapter 2.3.2 and 2.3.3), one that adopts the power of logic programming languages to gain long-term memory information. Logic programming has proofed itself to be an efficient tool for many agent systems, but is rather incompatible with the current structure that is used in SiMA (see chapter 2.1).

3. Database Selection & Integration

As discussed in chapter 2.2 there are several database strategies, with different strengths and drawbacks available. Out of the presented data stores, three approaches seem to be of particular interest for the implementation in SiMA. First the classical relational approach due to its widespread usage, community and the fact that it is very flexible in that sense that any static concept can be modelled with it. As long as the structures are well-known and seldom changing this approach is a very stable one. Another promising approach is the usage of a graph-oriented one. The usage of triples and graph structures has already found its application in other simulation environments like for example CHREST or SOAR (see chapter 2.3.1 and 2.3.4). Additionally, there are several advanced implementations of Resource Description Frameworks (RDF) like OWL (Web Ontology Language) [OWL] that offers even further functionalities, like reasoning. Finally, there is the document-oriented approach, which is of interest due to its flexibility in the structure of its stored data.

3.1 Database Selection

In this section the decision process for a fitting database solution for SiMA is covered. In [WDM+12, p. 1] some preliminary research on all three approaches, with special focus on their practicability for dynamic data models, was done. The three approaches were compared by examining their capabilities in model extension, data integration, data access, querying and their distributions abilities. In the following a short summary of the most important findings, with respect to this work, is presented. The findings of the preliminary work will be supplemented by in-depth research if needed.

Model Extension

Model extension is of high importance to a complex data structure model like it is used in SiMA. Even though the data structures like thing-presentation, thing-presentation-mesh etc. are fixed, a flexible model can be of use for SiMA. Reason for this is the broad definition of the data structures, which leads to many empty and unused attribute. In a flexible model it would be possible to leave unused attributes out and by that save space and writing time. According to [WDM+12, p. 220] a relational database is not really flexible even though it is possible to change the data model later on. However, this may require a rebuild as the concept of this database technology is to define the data model before importing the data. In order to meet the problem of the unused attributes in SiMA the *Universal Data Model* pattern might be implemented (see chapter 2.2.1). However, this would be a workaround and a native flexibility support in a database system should be preferred to this approach. Graph-oriented

databases support flexible data models on the fly [WDM+12, p. 223]. It is as simple as adding a new node to add a further attribute to the model. The same may be said about document-oriented databases [WDM+12, p. 223], just that in this case it is not a node, but some sort of String that has to be added to the document.

```
1: (defclass CAKE
2:                          (is-a TPM%3AENTITY)
3:                          (role concrete)
4:                          (single-slot value
5:                            (type STRING)
6: ;+                         (value "CAKE")
7: ;+                         (cardinality 1 1)
8:                            (create-accessor read-write))
9:                          (multislot class_association
10:                           (type INSTANCE)
11: ;+                        (allowed-classes)
12: ;+                        (value           [TP%3ASEN-
SOR_EXT%3AVISION%3ASHAPE%3
           ACIRCLE] [TP%3AINTENSITY%3AMEDIUM])
13:                           (create-accessor        read-
write)))
```

Figure 3.1: Cake object representation in Frames

Data Integration

Data integration plays a very prominent role in this work, as the data, which is until now mocking the agent's memory, is needed for the agent to interact with the simulation environment. As in many other agent systems, the agent needs some initial knowledge to be able to take action in its world. Therefore a transformation of the existing data to the new system is obligatory. According to [WDM+12, p. 223] for all three database technologies a special integration toolset is needed or recommended to perform data integration.

In Figure 3.1 a snippet of the Frames file, which is currently used to represent the agents memory (see chapter 1.3), is presented. The shown file snippet is used to describe the class “CAKE”. Line 1 defines that the class is called cake, while line 2 define that it is a subclass or representative of the classes “TPM” and “ENTITY”. It is a concrete class (in contrast to an abstract one that cannot be instantiated), which is stated in line 3. It owns a String value “CAKE” as an attribute (line 5-8) as well as some associations to TP classes (line 9-12).They define that the cake has the shape of a “CIRCLE” (located in the TP subclasses SENSOR_EXT ->VISION->SHAPE) and an “MEDIUM” “INTENSITY”, which

is a subclass of TP as well. Line 13 is a mere program instruction and of no interest for the conversion process.

The underlying Frames file is managed in a tool called Protégé, which also offers some export abilities for RDF. This option highly speaks for the use of RDF as the manual conversion of the existing data might turn out to be very time consuming. If a document-oriented database is used the Frames file might be processed by a text parser, as the structure of the Frames file is already very close to a document-based approach. If a relational database is used, first the data structure schema has to be defined and then the information has to be written into the database. In this case it would be preferable not to use the Frames file, but to link into the Java program and use the Java data structures that are already parsed for the usage in the simulation.

Data Access and Querying

For all three database technologies it is possible to use automatic query generators for simple query generation [WDM+12, p. 224], but in SiMA simple "select all" statements will not suffice, more complex queries are the ones of interest. Here the relational database is clearly the best performer as it works for large amount of data as well as for complex queries [WDM+12, p. 224]. Graph databases offer a very flexible querying, but may suffer from performance problems depending on the situation. Document-oriented databases perform well in suitable queries, but generally the missing JOIN statement reduces the query flexibility.

Distributed Systems

For SiMA the usage of distributed systems is not planned in the near future, nonetheless it may as well be taken into account as an additional (less important factor) for the decision process. As it was already covered in chapter 2.2.1 relational databases are not made for the use in distributed systems and therefore perform not too well. Both other approaches allow to use federated repositories, however, in the graph-oriented approach every additional repository slows the querying down by another 30% [WDM+12, p. 224].

Functionality	Relational database	Graph-oriented	Document-oriented
Model extension	-	++	++
Data integration	+	++	+
Querying	++	+	-
Distributed Systems	NA	++	+

Figure 3.2: Performance of database support for selected functionalities based on [WDM+12, p. 224] and additional research

Summary

In the course of this section three data store approaches were compared with respect to their suitability for SiMA. In Figure 3.2 a summary of the checked functionalities is given. As can be seen the graph-

oriented approach seems to be the most promising one, especially because of the flexible model and the option to convert the existing data model to RDF. With respect to the mentioned performance problems it has to be tested whether they affect SiMA too much and if they can be diminished in some way. The next chapter will cover the chosen database solution and the features it is providing for this work.

3.2 Resource Description Framework

The Resource Description Framework (RDF) is originally a W3C data standard for data interchange on the web [W3CRDF]. In order to represent RDF data one has several options to choose from [BHS03, p. 1]. First of all there is the possibility to use an XML representation. Then there is the option to use a *triple representation*, which was already mentioned in this work and finally there is the possibility to present the data in a *graph structure* which has the advantage of semantic interpretation for human readers. According to [BHS03, p. 1] all existing RDF system use object relational databases as a base to store RDF data. Although, originally designed for representing metadata about web resources, RDF is nowadays more often used as a generic data model for structured data management and reasoning [KDA11, p. 240].

3.2.1 RDF Data Model and Terminology

Due to the fact that RDF is merely a standard and not a technology developed by one company or person, there exist a lot of terminologies, which are used synonymously. In the following section the concept of the RDF *data model* and the most common terms will be explained. The whole RDF data model consists of only three data types, but many different names and notations are used:

- **Resource:** *Resources* are uniquely identified objects [BHS03, p. 2]. They are represented by unique identifiers called *Uniform Resource Identifier* (URI) [AH08, p. 33]. There is also an internationalized form of the URI in use, which is called *Internationalized Resource Identifier* (IRI). The IRI simply extends the character set of the URI to nearly all characters of the Universal Character Set [IRI]. URIs look exactly like URLs, which are known from the World Wide Web, which is due to the origins of RDF. An example URI referencing a cake in SiMA could be `http://ars.org/cake`.
- **Property:** Relations or attributes of resources are defined by *properties [BHS03, p.* 2]. They may be used to define the relation between two resources or to attach a *value* by some meaning to a resource. Properties are like resources defined by URIs. Examples for the usage of a property would be `http://ars.org/cake http://ars.org/hasColor "PINK"` or `http://ars.org/cake http://ars.org/hasAssociation http://ars.org/mothersbreast`. The first example links the resource cake to the value `"PINK"` by the property `http://ars.org/hasColor`. The second example associates the two resources `http://ars.org/cake` and `http://ars.org/mothersbreast` with each other.

- **Statement:** A *statement* is the combination of a resource, its property and value [BHS03, p. 2] *[BHS03, p.* 2]. A statement can be written like `P(R,V)` (for example `http://ars.org/hasColor(http://ars.org/cake, PINK) `) or simply in sentence form like `R P V` (like already used in the property example. This sentence like form is borrowed from elementary grammar and it is very common to refer to the elements of a statement as *subject*, *predicate* and *object* [AH08, p. 31]. In combination with the sentence analogy the term *triple* is often used instead of statement. It is noteworthy that a value or object of a triple can be a resource as well. If the object is not a resource it is often referred to as a *literal*. The main difference between URIs and literals is that the latter one´s content is not unique. Figure 3.3 shows the difference between the two concepts. If the triple `ars:BASICCAKE rdfs:label "CAKE"` is inserted 2 times the subject `ars:BASICCAKE` gets 2 new literals with the value `"CAKE"` attached. On the other hand, if the `triple ars:BASICCAKE rdfs:subClassOf ars:CAKE` is inserted twice the resulting graph and rdf triple store will contain the triple only once. Another difference between URIs and literals is that only URIs are allowed in subject and predicate position of a triple. Reason for that is that the literal represents a primitive data value and there is no use to attach a description to such a value [KDA11, p. 241].
- **Notation:** When writing about triples it is also very common to leave the first part of the URI out, in order to make the sentence more readable. For example http://ars.org/cake `http://ars.org/hasColor "PINK"` may also be written as `cake hasColor PINK`. Note that this is only for communication purposes and not valid RDF syntax. In order to keep a valid RDF syntax and provide readability it is possible to define namespaces, like they are known from XML. For the further course of this chapter for `http://ars.org/` the namespace "`ars`" shall be used. For the RDF schema the namespaces `rdf` and `rdfs` are used instead of `http://www.w3.org/1999/02/22-rdf-syntax-ns#` and `http://www.w3.org/2000/01/rdf-schema#`. If a namespace is defined in an RDF document one can use the namespace followed by a colon instead of writing the whole prefix down. For example `http://ars.org/cake` can be written as `ars:cake`.
- **Graph representation:** In Figure 3.4 both already introduced statements are transferred into an RDF graph. As can be seen all resources are surrounded by circles, even if they are attached to another resource representing the object of the triple, while literals like the value "PINK" are shown in a rectangular shape. The connecting property is used as a label for the edge between them.

3.2.2 RDF Schema

As the RDF model makes no assumption about the application area of the data, there exist no concepts for modelling any specific data information, like for example types or classes [BHS03, p. 2]. In order to overcome this shortcoming schemas may be used. For general use a very generic schema called "RDF schema" was proposed by the W3C, but for different application areas it is possible to define different schemas. The RDF schema provides a number of classes, properties and attributes to structure RDF data in a standardized manner. In Figure 3.5 a selection of the RDF schema elements is

presented. The RDF schema allows to define a fine grained structure, like it may be known from several programming languages. There are elements for the definition of classes and subclasses (rdfs:Class and rdfs:subClassOf) as well as for marking the definition of a datatype (rdfs:Datatype and rdf:type).

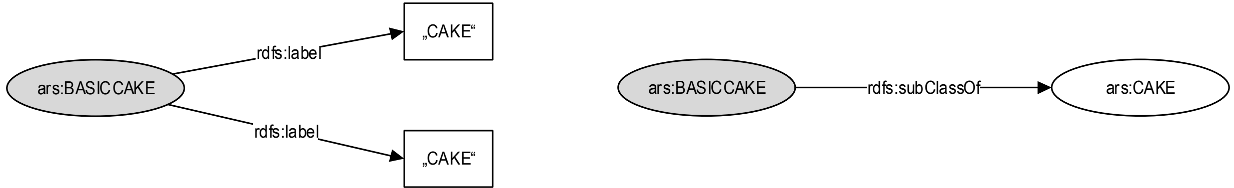


Figure 3.3: Concept Example URI versus Literal

It is also possible to explicitly mark resources and literals or to define a subproperty (rdfs:Resource, rdfs:Literal and rdfs:subPropertyOf). In addition to data type and class definition it is also possible to put some sort of "constraints" on the data by using the elements rdfs:domain and rdf:range.

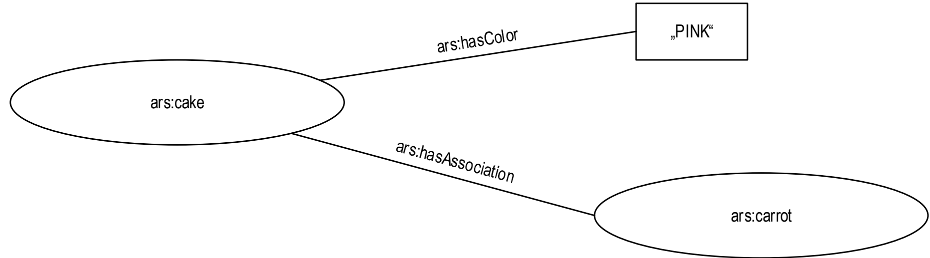


Figure 3.4: Graph representation of RDF data

The range property may be applied to properties in order to state that all objects which are used in combination with that property are of a specific class type. For example the triples `ars:isCake rdf:range ars:sweets` and `ars:mothersgift ars:isCake ars:marblecake` states that `ars:marblecake` has to be of class `ars:sweets`. The domain property does the same thing for the subject of a triple. In Figure 3.6 a possible application of the RDF schema to the example of the frames data which was presented in Figure 3.1 is shown.

Element	Type
rdfs:Class	Class
rdfs:Datatype	Class
rdfs:Resource	Class
rdfs:Literal	Class
rdf:Property	Class

Element	Type
rdfs:subClassOf	Property
rdfs:subPropertyOf	Property
rdf:type	Property
rdfs:domain	Property
rdf:range	Property

Figure 3.5: Selection of RDF schema elements based on [W3SCHOOL]

This example is reduced to the information that was provided in Figure 3.1, in a more wholesome example ars:ENTITY, ars:TPM etc. would have more information attached to them. For example there would be an edge with the label rdf:type to rdfs:Class too. One can see from that example, that this sort of information representation puts nearly no restriction on to the information that can be stored. Therefore the difficulty lies foremost in finding a representation model which is suitable for the application area to which the triple store is applied. When it comes to information representation, SiMA has different requirements than a common web application or wiki store, for which RDF stores are mostly used. The next chapter therefore covers in detail how the existing data model is mapped to a triple store representation and the decisions that were taken during the design process.

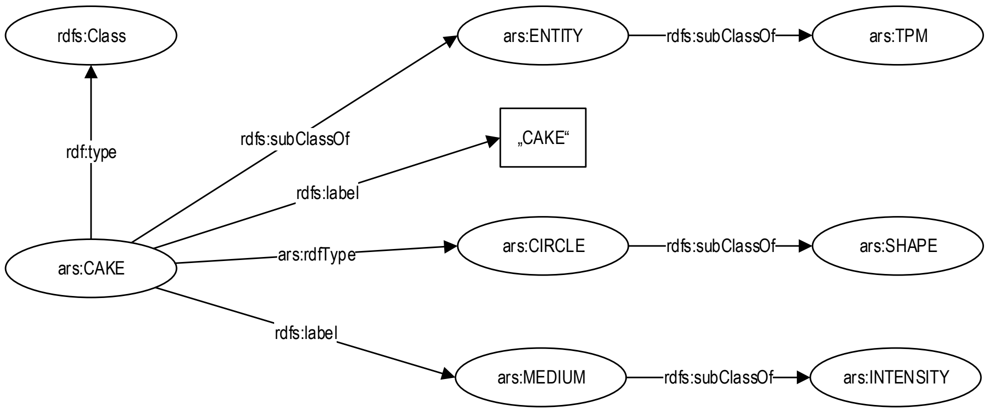


Figure 3.6: RDF schema application for a cake

3.2.3 Querying with SparQL

RDF being a W3C standard it suggests itself that the mainly used query language is a W3C recommendation as well [KDA11, p. 241]. Even though for querying RDF one may choose between some query languages, which are partly SQL-based and partly XML-based [RDFQUERY], the most common one is SparQL. SparQL is an SQL-like query language and supports the following types of query:

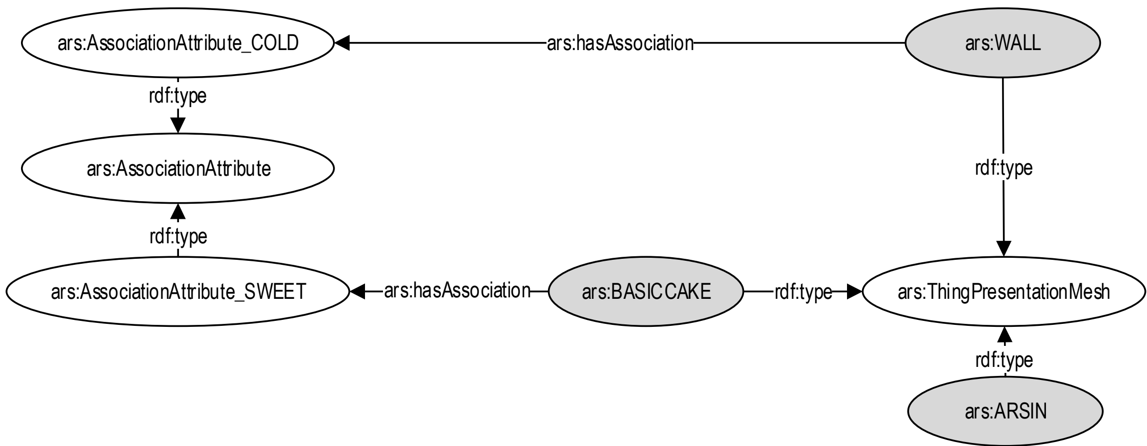
- **SELECT:** Like in SQL n-tuples of query results are returned. Results are returned with their binding. For example if the select all query `SELECT ?subject ?predicate ?object WHERE {?subject ?predicate ?object.}` is applied on the graph of Figure 3.4 the result would look like presented in Figure 3.7. Every part of the triple is returned in connection with the binding that was applied to its place in the statement. Further information on SELECT queries can be found in [SPARQL].
- **DESCRIBE:** This query returns a result RDF graph [KDA11, p. 242]. The simplest possible clause needs an IRI as a parameter: `DESCRIBE <http://ars.org/cake>`, but it is also possible to pass no IRI but a WHERE statement instead: `DESCRIBE ?x ?y <http://ars.org/> WHERE {?x ars:hasAssociation ?y}.` The RDF which is returned may vary as it depends on the publisher of the information [SPARQL]. It is noteworthy, however that a sparkle result graph is a textual description of a graph rather than a graphical one with nodes and edges. For further information please refer to [SPARQL].

- **ASK:** ASK queries provide a Boolean answer, indicating whether a query has an answer or not [KDA11, p. 242].
- **CONSTRUCT:** Like DESCRIBE the CONSTRUCT query returns a graph as result [KDA11, p. 242]. This graph is constructed by substituting variables in the graph template and performing a union on them. CONSTRUCT statements can be used to specify new information based on existing data [AH08, p. 130]. For example it is possible to define that some act contains an image if the image has an association to it: `CONSTRUCT {?act ars:containsImage ?img} WHERE {?img ars:hasAssociation ?act}`.

?subject	?predicate	?object
http:ars.org/cake	http:ars.org/hasColor	"PINK"
http:ars.org/cake	http:ars.org/hasAssociation	http:ars.org/carrot

Figure 3.7: Result of SELECT ALL statement on Figure 3.4

In addition to those query statements SparQL supports **ORDER BY** and **LIMIT** features [AH08, p. 100]. With the first one, it is possible to define the ordering of the result by a binding (this could be for example a date or association weight). With the latter of the two, one can limit the result to a certain number, which could be used on an ordered result set to return only five associations with the highest weight.



[dsTypeCount="3"^^<http://www.w3.org/2001/XMLSchema#integer>;dsType=<http://ars.org/ThingPresentationMesh]

[dsTypeCount="2"^^<http://www.w3.org/2001/XMLSchema#integer>;dsType=<http://ars.org/AssociationAttribute]

Figure 3.8: GROUP BY and COUNT Data Structure Types example

With the aggregates **COUNT**, **MIN**, **MAX**, **AVG** and **SUM** some basic math can be applied to certain contents of the RDF store [AH08, p. 101]. For example `SELECT ?dsType (COUNT(?dsType) as ?dsTypeCount) WHERE { ?s rdf:type ?dsType . } GROUP BY ?dsType` searches for all objects (`?dsType`) which are connected by a rdf:type predicate. A **GROUP BY** clause is applied on the values of the `?dsType` binding, putting all triples with the same `?dsType`

together in some “collection”. On all of those collections a separate COUNT operation is performed. In Figure 3.8 an example graph with the resulting output for this query is shown.

The possibility to **FILTER** and **UNION** results add further options for searching through ones knowledge base [AH08, pp. 107–109]. FILTER may be used to put some constraints on the results, while UNION puts the results of two queries together. All of the presented options can be combined into subqueries, thereby providing many options for the usage in SiMA.

3.2.4 N-Ary Relations

One might notice from the previous subchapters that RDF is characterized by binary relationships, which leaves some design issues [NARY], if someone has to provide meta-information or simply add more than just one value to the same subject:

- **Issue 1:** If an association has a weight or probability (see Figure 3.9) - How shall the relations attributes be *described*?

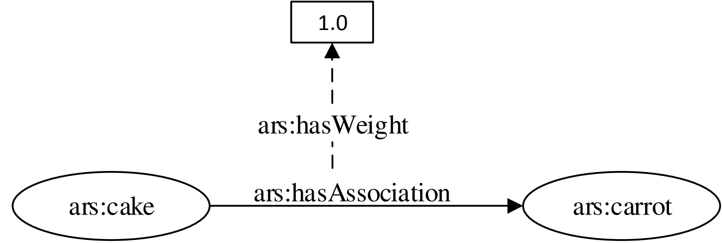


Figure 3.9: Issue 1 - describe instances of relations

- **Issue 2:** If two or more values have to be linked to one subject (see Figure 3.10) – How shall relations between more than two instances (*n-ary relationships)* be represented*?*

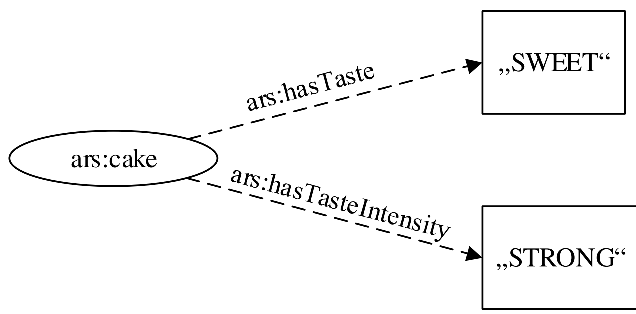


Figure 3.10: Issue 2 - More than one value for one subject

In order to overcome the natural limitations of RDF some design patterns have been introduced by the community [NARY].

Solution 1 – Introduce new classes

In Figure 3.11 the solution to Issue 1 can be seen. In order to represent, the association weight between the carrot and the cake a new association class, combining the weight and the carrot is introduced.

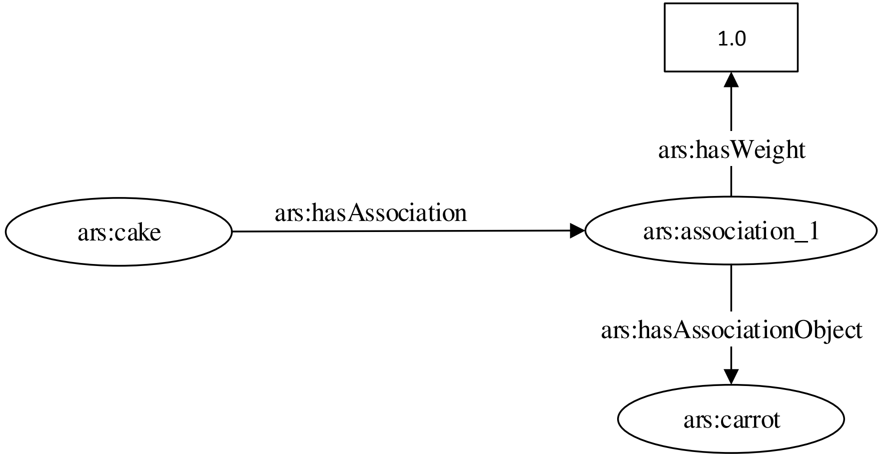


Figure 3.11: Class representation solution to Issue 1

For Issue 2 the solution can be seen in Figure 3.12. In order to be able to reason about a "strong sweet taste" a new class representing a "taste" is introduced and the taste values are assigned. One should notice however that for using the RDF triple store only as a database without reasoning concepts, it would be sufficient to represent the taste of a cake like shown in Figure 3.10.

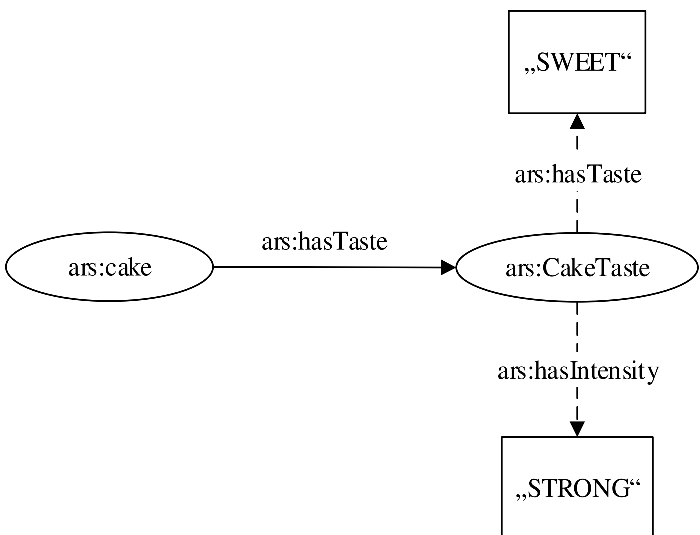


Figure 3.12: Class representation solution to Issue 2

The presented pattern is very close to the SiMA concept, as in Java the data structure representation already knows the concept of associations, containing the association weight and other association information. Also Issue 2 and its solution are close to the concept of the mesh structures which are used in the SiMA implementation.

Solution 2 – Reification

In RDF there is another concept which could be used for presenting n-ary associations, however, it is not recommended by the W3C [NARY]. Reason for not recommending to use the so called *reification* is a conceptual mismatch. Reification is a build-in RDF option to describe RDF statements, which were described earlier in this chapter, by RDF [REIFICATE]. For example, it would be possible to add information about the time when the statement was recorded (see Figure 3.13), or who created it.

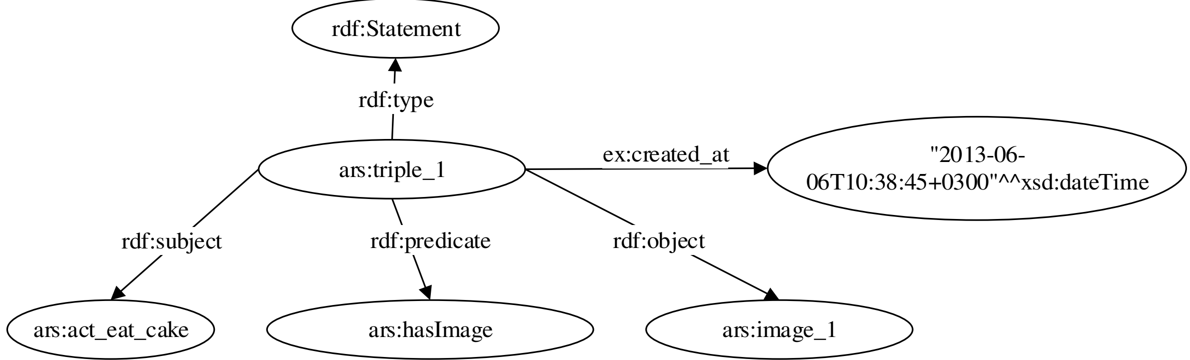


Figure 3.13: Reification example with timestamp

From a conceptual point of view usually n-ary relations do not characterize the statement, but add additional information to it. Therefore it would be more natural to talk about an “association” or a “taste” rather than adding statements about statements.

3.3 Implementation of Database Integration

The following section covers the actual implementation of the concepts presented in this chapter. The first subchapter deals with the project structure, the used frameworks and the general dependencies. In the second subchapter the most important classes and methods are highlighted and explained.

3.3.1 Project Structure

In the course of this work the SiMA project was expanded by a new subproject called `ArsDatabase`. Basically all of this works implementation has taken place in that project, therefore the project structure will be presented in the following chapter in full. The used development language is Java [JAVA] and for the RDF database realization the Java framework Sesame [Bro02, pp. 54–68] was chosen. It provides a sophisticated API which supports creation, parsing, storing, inferencing and querying RDF data. Sesame is in active development and has a large community, which were essential factors for choosing it. Active development is a crucial factor in case that any bugs or other problems are experienced during the implementation process. Furthermore, it is more likely that support for the framework will be available for a long period, which is of high interest for a long-term project like SiMA. A large and active community can accelerate the general implementation process, as it is more likely that someone has already experienced the same problems and therefore a solution can be found without efforts in community forums instead of having to ask questions oneself. Additionally Sesame provides

support for in-memory deployment as well as for local storage and server deployment. This flexibility in storage location could be of relevance in the future and was therefore another reason for selecting Sesame. Other frameworks like for example Allegrograph [ALLEGRO] offer similar features when it comes to community support and stability, but require a server to deploy the database and are therefore limited and would add complexity to the project setup of SiMA. After providing an insight to the general project structure special, attention will be paid to the data structure conversion and the final implementation of the memorization process.

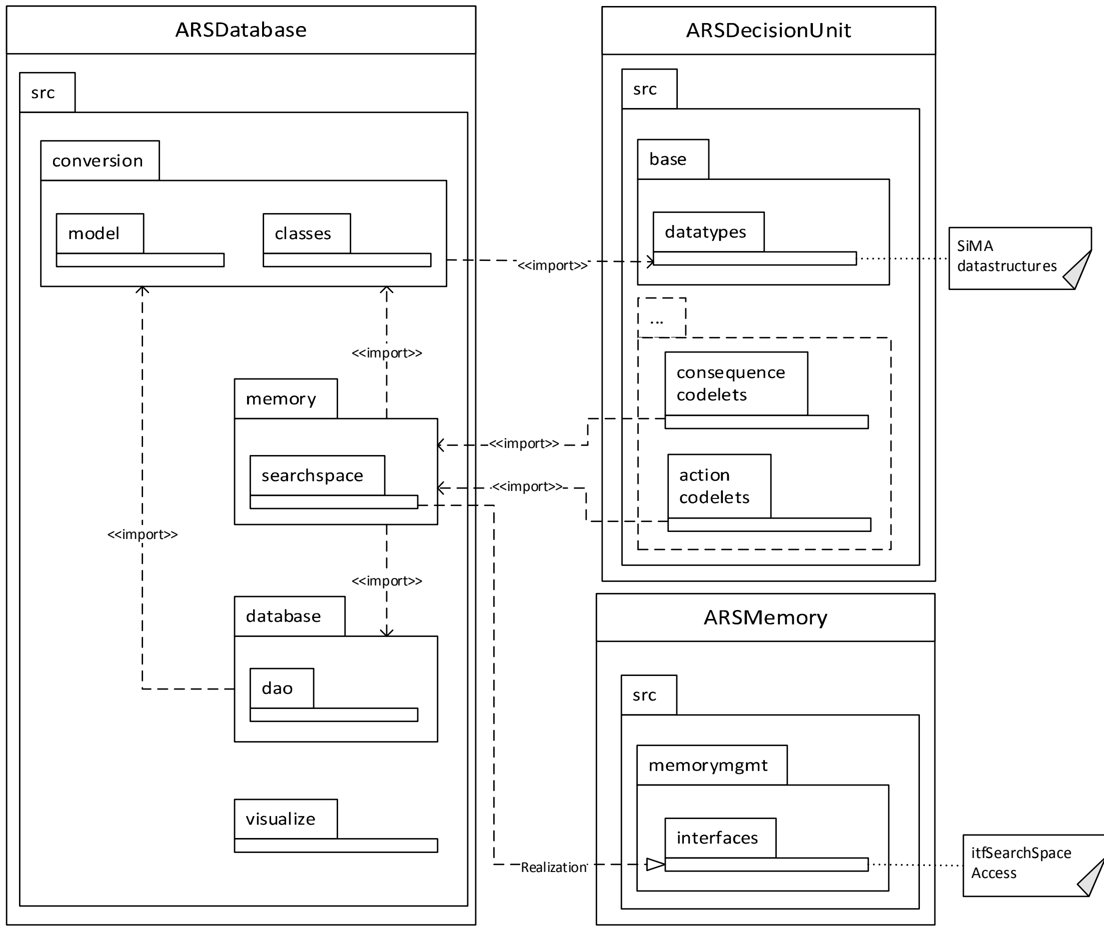


Figure 3.14: Package Diagram ARSDatabase and correlated projects

As the integration of a database introduces a totally new functionality to the SiMA project, with very few dependencies or correlations to other projects, it was decided to introduce a new subproject to SiMA. It would have been an option to put it to the topically fitting ARSMemory, but to provide a better interchangeability of the database it was decided not to do so. Figure 3.14 shows a comprehensive package diagram of the new `ARSDatabase` project and all already existing projects as far as they are relevant. The graphic demonstrates how independent this project really is, as it uses resources from only two of the currently ten subprojects in SiMA. The classes implementing the SiMA data structures (see chapters 2.1 and 4.2 for more details), are situated in the `ARSDecisionUnit` project and form the first dependency of the new project. Further dependencies to this project are the codelets

managing the current goal and the perceived images. More detail on the usage of the codelets will be provided in chapter 5.3. The second project dependency is the interface that handles the access to the long-term memory. It is situated in the `ARSMemory` project and was already used to access the stub long-term memory data which was already mentioned in the problem statement in chapter 1.3. Details on the implementation of the interface will be provided in chapter 5.3.

A closer look at the packages of the `ARSDatabase` project shows that there are four main project parts that can be distinguished from each other. The first part (package `conversion`) deals with the *conversion* of the SiMA data structures to a triple representation. In order to ensure independence from a certain RDF database solution in this part of the project an intermediate triple representation class was introduced. This representation can be passed to the `database` package from where it is translated into a Sesame triple representation and saved to the *database*. The `memory` package contains the *memorization process*, which has to make use of the conversion as well as the database package. Additionally the initial database *migration* class `RDFSearchSpaceCreator` was placed in that package. Due to the implementation of the `itfSearchSpaceAccess` interface it is only necessary to change one line of code in order exchange the database system with the stub memory data for the whole SiMA project. From there on classes continue using only the interface and no further changes to any of the other projects are necessary. Finally a `visualization` package was implemented in order to validate and visualize the results of this work.

3.3.2 Database Integration

In order to handle the communication between the simulation environment and the RDF database the package `database` was implemented. In Figure 3.15 a class diagram of the package is shown. An essential part of this package are the two constant classes, which are supposed to guarantee the correct cooperation of the saving and loading procedures. The class `TripleConstants` holds the translated predicate names from chapter 4.1, while the class `DatabaseConstants` contains information like the URI prefix, other repository data and some prepared queries. Constants are of great importance for any project as they improve the maintainability of the project in the future. The main advantage they provide is that only one position in the code has to be changed if any changes are made. Together with the `SesameConnectionHelper`, which is responsible for managing the repository access, these classes form the top layer of the `database` package.

The `SesameConnectionHelper` provides a method called `getActiveSesameRepository` in which the actual repository solution can be fetched for access. For SiMA three different repository options were implemented during the course of this work. The provided options are a native store, a memory store and an in-memory store. The *native store* is optimized for large databases which are too big to keep them entirely in memory. Data of a native store is directly stored to disk instead of keeping it in memory. In contrast to this store a *memory store* leaves it to the user when to save the data to the store and an in-memory store is for runtime use only and all data will be lost after a run. Which approach is best depends on the actual use case, which is why all three options are provided.

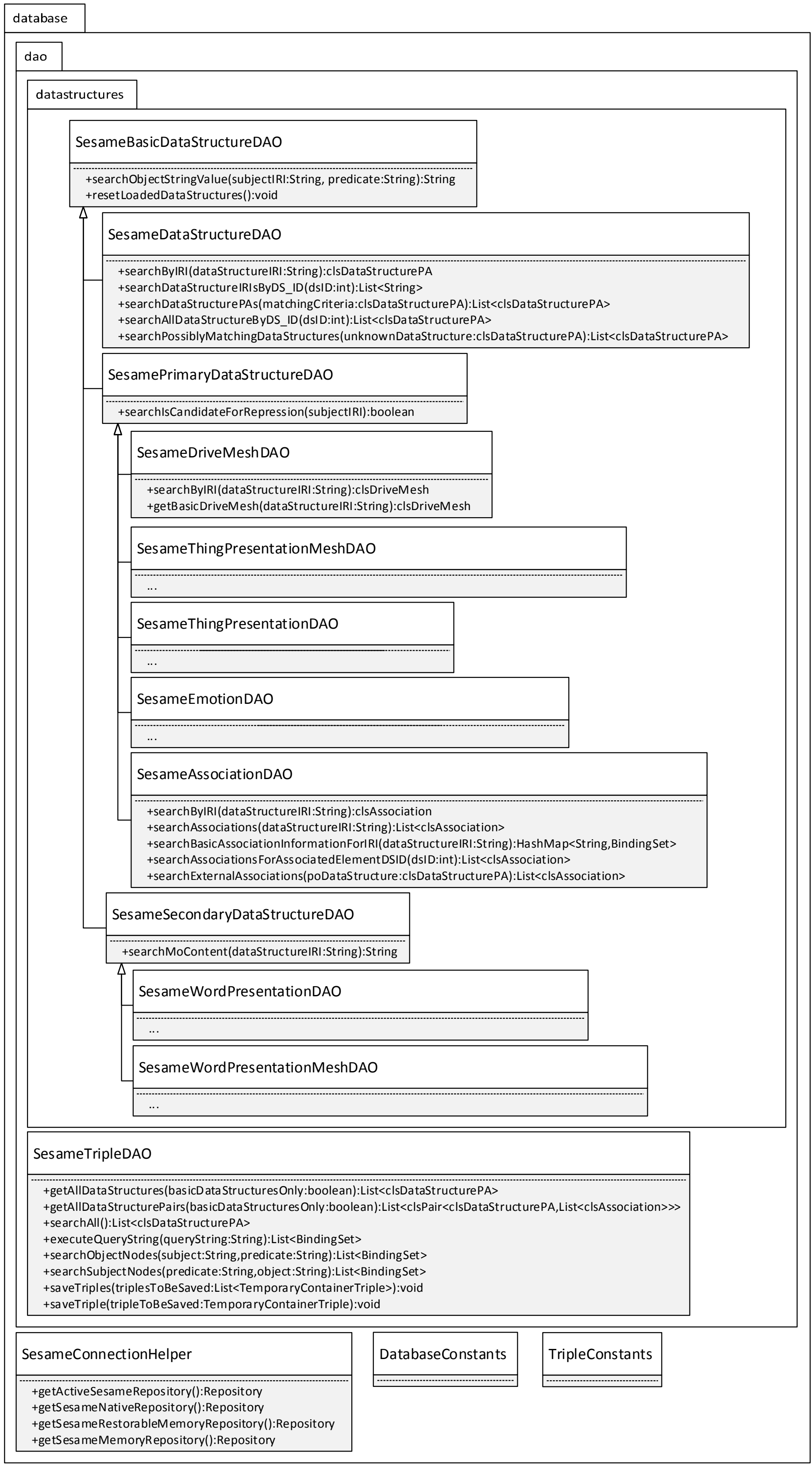


Figure 3.15: Class diagram of the database package

The `database` package contains the sub package `dao` in which the real functionality of the package is implemented. The responsible class for retrieving and saving one or several triples is the `SesameTripleDAO`. In addition to predefined methods for often needed functions, it also provides an option to pass customized query strings to the database.

Finally there is the package `dao.datastructures` where for all of the described data structures from chapter 4.1 a data access class is provided. Every class offers a method to get a data structure by its IRI. The result of that method is always the corresponding Java object of that data structure. Internally the method fetches the triple representation of the object and fills a new Java instance with the values from the database.

In order to reduce redundancy of code the abstract class `SesameBasicDataStructureDAO` is generalized by all other classes in that package and provides a method to fetch all information that is possessed by any `clsDataStructurePA` in the project. The generalizing classes only have to implement further methods that are needed for their specific data structure.

Another class which is worth further explanation is the `SesameDataStructureDAO` which is mainly a convenience class. The class was designed to provide a general class for managing all data structures and basically renders it unnecessary for most developers to access any other class of this package. It provides a general method to search any data structure by its IRI and will utilize the corresponding access class to construct the correct data type from the database information.

Furthermore, data structures can be searched by their DSID or an example data structure. If an example data structure is passed to the method `searchDataStructure(clsDataStructurePA)` the database is queried for any data structure having either its DSID, DSInstanceID, or moContent in common. If any of the given attributes matches the data structure it is added to the result set.

In this chapter the database selection and integration into SiMA has been covered. First a database analysis and the final selection of a suitable database technology was presented. The rest of this chapter covers the selected technologies features and the implementation in SiMA.

4. Data Structure Conversion and Migration

The integration of a new data store solution into SiMA is a two-step process. As already mentioned in chapter 3.1 the SiMA simulation cannot run without the agent having any basic knowledge about its world. Therefore, after successfully adding the technical dependencies like libraries and implementing the connection to the database, the next step towards a memorizing agent will be to convert the existing file-based data from the static declarative semantic memory to a triple representation. The first section of this chapter covers the automatic conversion provided by the tool Protégé and discusses the applicability for the current SiMA setup. In chapter 4.2 an alternative triple model for a manual conversion of the SiMA data structures is developed. This approach was developed with the goal to store only the necessary information, unlike the automatic approach, which makes extensive use of additional descriptive information, which is not required by SiMA, thereby building up a considerable information overhead. Finally, chapter 4.3 presents the final solution that was implemented to SiMA as part of this work. In the course of this final subchapter all details on the project structure, as well as dependencies to other parts of the SiMA project are given.

4.1 Automatic Migration to RDF

In chapter 3.1 the option of automatically converting the existing Frames file structure in RDF was mentioned. This chapter deals with the conversion result provided by the tool Protégé Frames and its applicability for the SiMA project. In Figure 4.3 a screenshot of the Protégé instance browser and the RDF export options are shown. The instance which is currently shown in the view is again the cake, which was already used as an example in chapter 3.2. When exporting the frames data to RDF it is not possible to export only selected information. If export to RDF is chosen, one is limited to saving all the existing data in two separate files. As can be seen from the screenshot, Frames differentiates between the class and the instance of an object, which is also transferred to RDF by exporting the information into two separate RDF files. The class information of the Frames file is used to generate an RDF schema file (see chapter 3.2.2 for detailed information on schemas), whilst the instance information forms the other file, containing the concrete information.

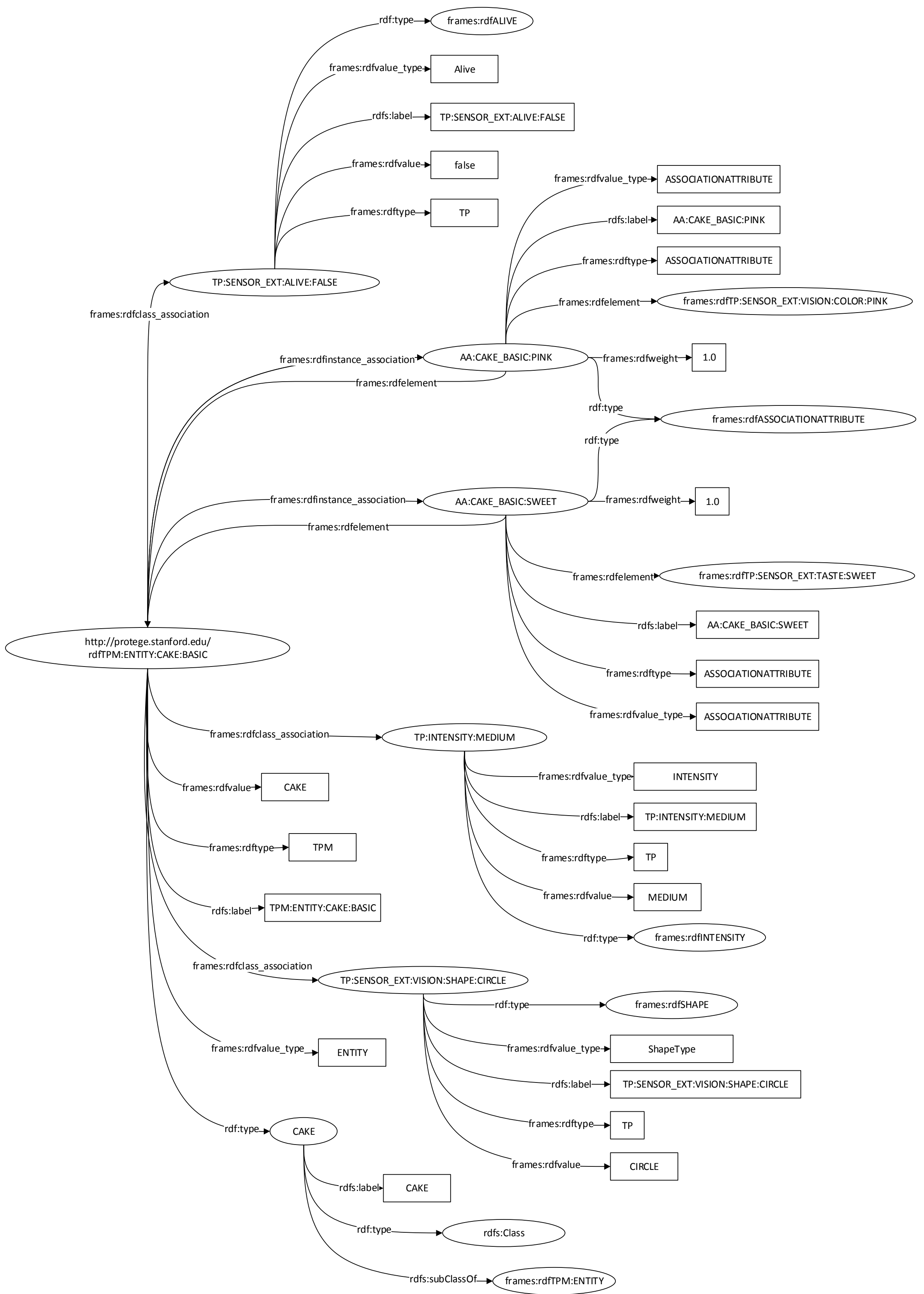


Figure 4.1: Automatically generated RDF representation of cake instance

In Figure 4.1 a graph view, resulting from the automatic conversion of the cake example instance is presented. The resulting output however is not too satisfactory, when it comes to naming conventions or the realization of concepts. For example the class hierarchy between TP, INTENSITY and MEDIUM is mashed up to be represented as an URI with the name “rdf_:TP:INTENSITY:MEDIUM”. This is not desirable at all considering the actual data model concept which is currently used in the SiMA simulation environment. For example the concept of a TP, which equals the class `cls-ThingPresentation`, is not understood as one data structure type, but it is mixed into names and used as a literal. Another flaw is the redundancy of node contents, which partly results from misunderstanding the data structure type concept and partly from the approach to connect every URI with a literal containing its name again.

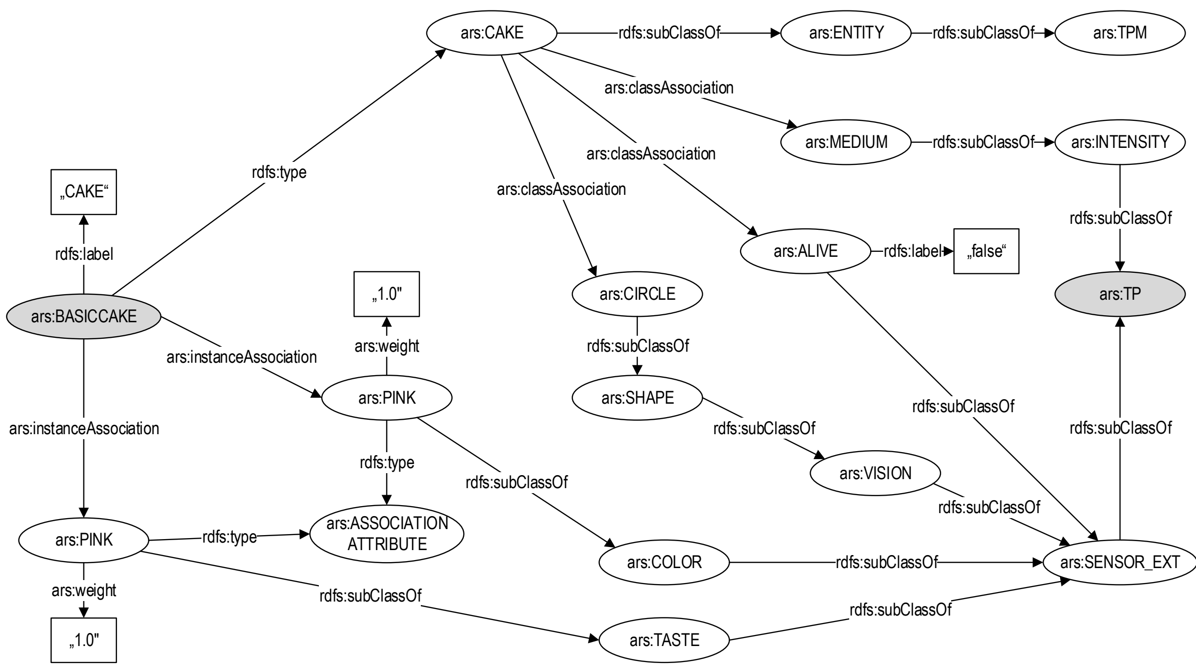


Figure 4.2: Refactored RDF graph

In Figure 4.2 a refactored version of the automatically generated RDF structure is presented. The main difference to the automatically generated version is the transformation of data structure types to unique nodes. On the example of the “ars:TP” node (marked by a grey node in Figure 4.2) one can see the advantage over a literal usage of it. As was explained in chapter 3.2, literal values are not unique and every new literal input creates a new node. If one looks at the conversion only from the Frame side, however it still makes sense even though some configuration options would be favourable. If the later usage of the data is taken into account, however the automatic conversion generates a lot more nodes than are actually needed for the usage in SiMA. If one compares the refactored graph of Figure 4.2 with the automatically generated one in Figure 4.1, it can be seen that nearly twice the amount of nodes has been created in the automatic one. Considering that this is only one example entity of many,

one can imagine the impact of so many redundancies on the performance of the whole knowledge base.

In addition to some flaws that naturally come with automatic generation there are two main reasons to withdraw from using the automatic generation. The first reason for not using the automatic generation is that the generated structure is not in the least similar to the Java representations of the data structures which are used by the simulation framework. Besides the fact that the naming and resemblance of the structures is dissimilar, the parsing process which is done by SiMA at the beginning of a simulation adds further debugging information to the data, which is not contained in the Frames file and therefore the converted RDF data. For example a control variable “moDS_ID” is generated and some “moDebugInfo” values are set. The second reason is that due to the fact that the memorization process will have to make use of the Java version of data structures when storing memories, a conversion between the Java classes and the triple store has to be implemented anyways. As the automatically generated RDF structure differs so much from the Java model, the conversion from Java data structures to the triple store would take more effort than to make use of the SiMA parsing process and convert the existing knowledge base with the same methods as will be used by the memorization process. The original plan to build upon the automatically converted RDF data was therefore abandoned in favour of a Java based solution.

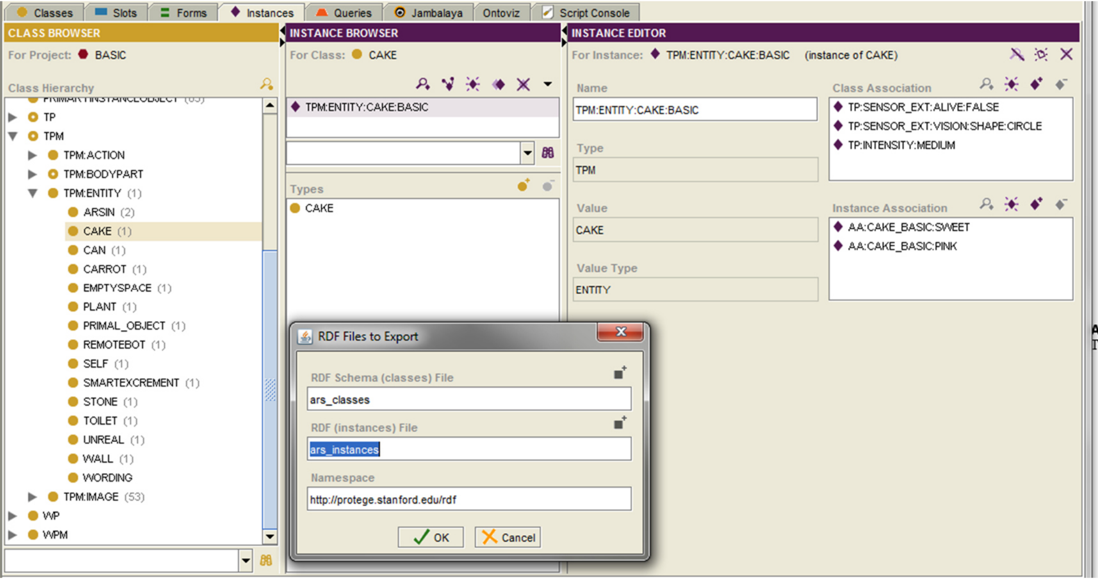


Figure 4.3: Screenshot of Protégé Instance overview and RDF Export

4.2 Alternative Migration Concept

Due to the fact that the automatic conversion produces a considerable information overhead, it was necessary to develop a new triple model instead of using the model which was generated during the automatic conversion. The following chapter presents an alternative triple model to convert the existing Java data structures. This model is going to be used for migrating the static declarative semantic memory as well as for the memory contents generated for the new episodic memory at runtime.

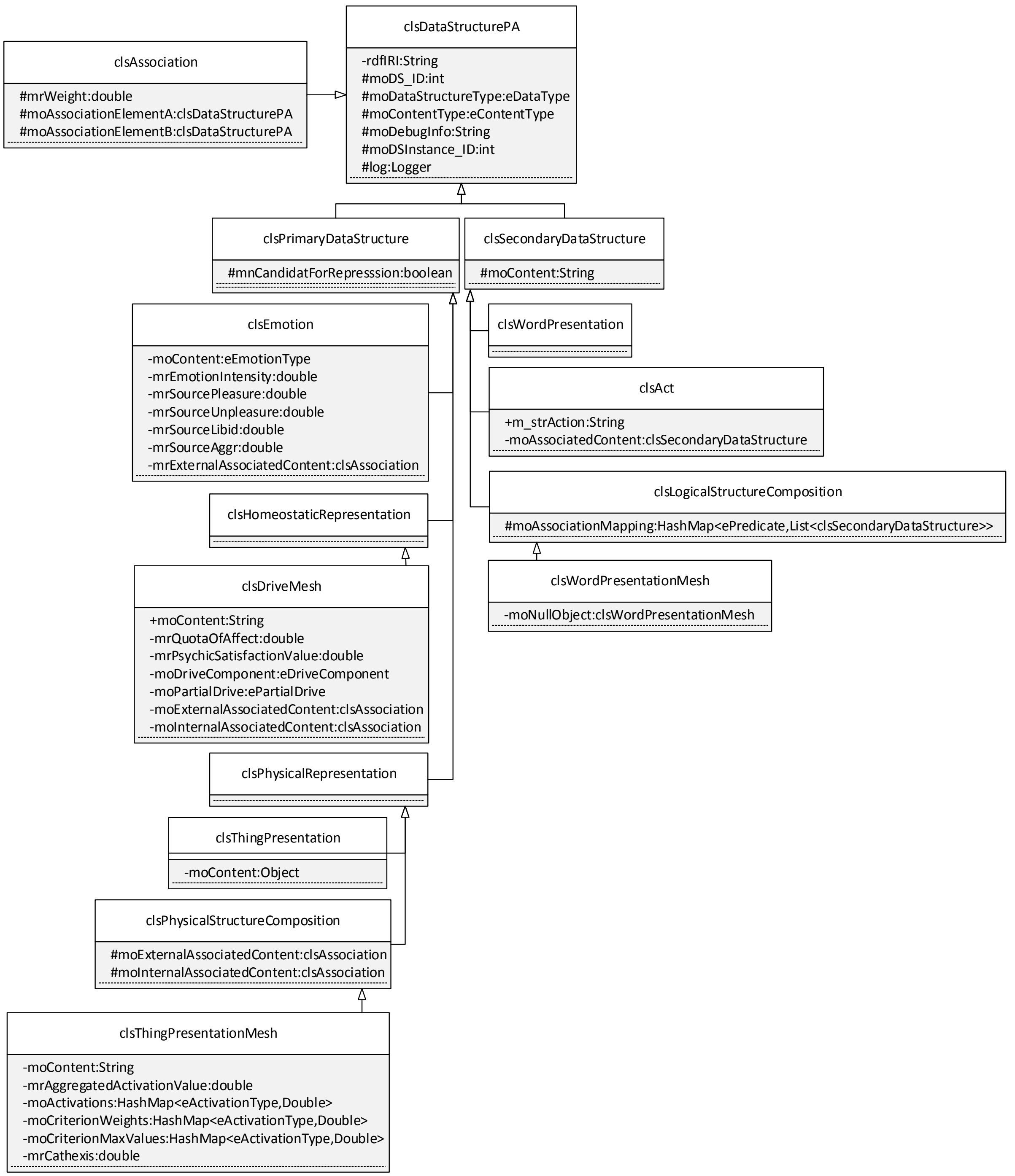


Figure 4.4: Overview of SiMA data structure hierarchy

In the following for every data structure that has been realized in the Java implementation of SiMA a table containing the resenting triples is presented. While the usage of <…> generally marks the occurrence of an IRI the usage of "…" marks a string value that is saved to the database. For the access of a data structure attribute [ds.attributename] will be used and if a list has to be converted to a triple it will be marked by writing get(i) after the data structure attribute. It means that for every item in the

list a triple will be generated. In addition to IRIs and string values the database may also contain RDF specific types, which are presented in the tables without any mark-up. RDF types are at the moment without use for the project, it is merely a question of preparing the database for any future uses like RDF/OWL reasoning.

4.2.1 Primary Process Data Structures

As introduced in chapter 2.1 the primary process is responsible for the unconscious data processing. Its information is processed by the pleasure principle, thus meaning that the priority for processed information is decided by drive demands instead of considering requirements the current situation might bring up [Zei10, pp. 51-52]. Primary process data structures do not contain any logic relations and conflicting information is processed in parallel and passed on to the secondary process without filtering.

Thing-Presentation

The basic data structure of the primary process is the *thing-presentation* [Zei10, p. 49]. In SiMA´s technical definition thing-presentations represent environmental, bodily and homeostatic information and automated motion sequences. The representing class is the `clsThingPresentation` [Zei10, pp. 49–61]. It includes sensorial characteristics of objects, which can be divided into taste, visual, audio, olfactory and tactile information [ZLM09].

clsThingPresentation

-moContent : Object
#moContentType : eContentType
#moDataStructureType : eDataType
#moDebugInfo : String
#moDS_ID : int
#moDSInstance_ID : int

S	P	O
<ds>	<ars:hasMoContent>	"[ds.moContent]"
<ds>	<ars:hasMoContentType>	<[ds.moContentType]>
<ds>	<RDF.type.getURI()>	<[ds.moDataStructureType]>
<ds>	<ars:hasDebugInfo>	"[ds.moDebugInfo]"
<ds>	<ars:hasMoDS_ID>	"[ds.moDS_ID]"
<ds>	<ars:hasMoDSInstance_ID>	"[ds.moDS_Instance_ID]"

Figure 4.5: Thing-presentation and its conversion to RDF triples

As the thing-presentation is one of the basic data structures, it has no complex associations to offer. As can be seen in Figure 4.5 all of its attributes can be converted directly into string values. The *moContent* may contain values like “CIRCLE”, for describing a round shape or “#228373” for the description of a colour, whilst the *moContentType* states what the moContent is about. Examples are “ShapeType” or “Colour”. The *moDataStructureType* states that it is a data structure of the type “TP” and is unvarying for any thing-presentation and the String *moDebugInfo* is for debugging purposes, but will be transferred to the database as well. The *moDS_ID* is used as a data structure identifier for example moDS_ID “12” will always belong to the ShapeType - Circle, whilst the ShapeType - Square has the moDS_ID “354”. For testing purposes the *moDSInstance_ID* was created to identify different instances of a data structure. During conversion tests it turned out that many parts of the code have dependencies on this value and it was therefore decided to transfer it to the database as well. As can be seen the subject of the triples is always the data structures own IRI. As described in chapter 3.2 all predicates are IRIs as well. For the moDataStructureType it was decided to use the RDF.type predicate from the RDF schema which was introduced in the same chapter.

Thing-Presentation-Mesh

In order to define which thing-presentations belong to the description of one thing, a data structure called *thing-presentation-mesh* is used [Zei10, pp. 49–61]. The thing-presentation-mesh is a pure technical model to connect thing-presentations, therefore representing the concept of an object. In Figure 4.6 a typical instance of a `clsThingPresentationMesh`, which is the corresponding Java class, is presented. The example shows a primary process representation of a cake. Thing-presentation-meshes combine some informal attributes with two lists of associations.

The first list contains internal associations, representing the object characteristics, like its colour, shape or taste. External associations on the other hand represent dynamic information like the object position or distance. It is important to note that associations are used in the primary as well as in the secondary process. Thing-presentation-meshes are also the primary processes’ way to represent images which will be used for constructing experiences in the memorization process.

As can be seen in the hierarchy diagram (

Figure **4.4**) associations are on the same hierarchy layer as the two processes. Associations generally define two elements that have some sort of connection to each other. A good example of such an application of an association is again the cake being associated to the taste “SWEET”. One thing-presentation may be associated to several thing-presentation-meshes, leading to a connection between them. As can be seen in Figure 4.6 every association owns a weight marking the relevance of the association. In total there exist nine association types in the implementation, each marking a different sort of association. For example `clsAssociationAttribute` is used to connect a thing-presentation-mesh to a thing-presentation representing its attributes and `clsAssociationDriveMesh` connects a thing-presentation-mesh to a drive mesh.

The thing-presentation-mesh has many values in common with the thing-presentation as they share the same ancestor the primary process. The thing-presentation-meshes *moContent* contains some sort of object name like “STONE”, “CAKE” or “STOMACH”.

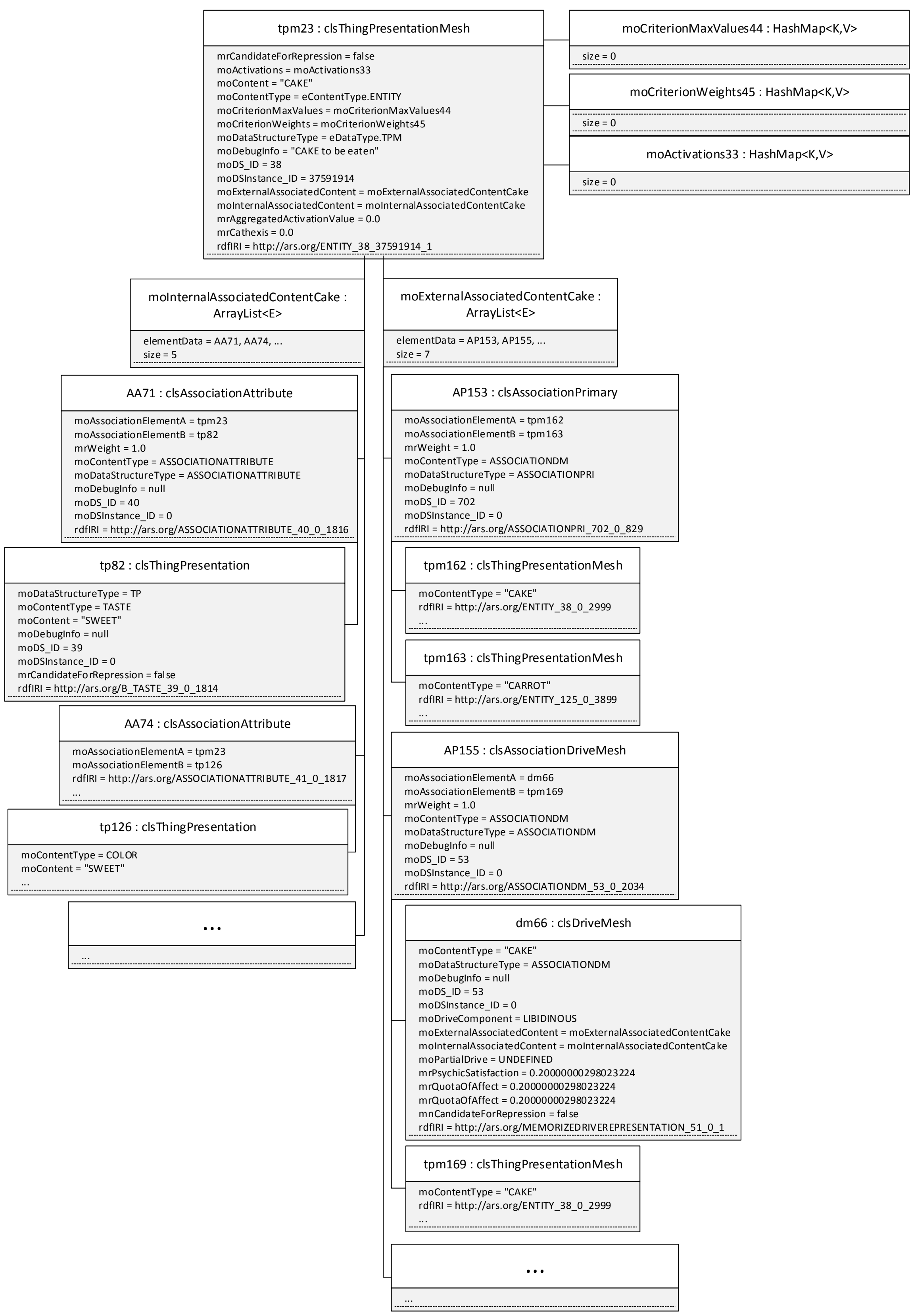


Figure 4.6: Instance of a thing-presentation-mesh representing a cake

The *moContentType* is restricted to one of seven values, which can be processed by SiMA. These values are “RI”, “BODYPART”, “RIREPRESSED”, “ORGAN”, “RILIBIDO”, “ACTION” and ”ENTITY”. They tell the system for example if an object is an environmental object (“ENTITY”) or an “ORGAN” like the agents stomach.

For the attributes *moDataStructureType*, *moDebugInfo*, *moDS_ID* and *moDSInstance_ID* the same as for thing-presentations applies, except that the moDataStructureType contains the value “TPM”. In addition to those single value attributes the thing-presentation-mesh contains some lists. For the RDF conversion of the thing-presentation-mesh it suffices to store a link to the associated objects IRI. Of course the associated object has to be converted as well if the link shall have some meaning. The attributes *moActivations*, *mrCathexis*, *moCriterionMaxValues* and *moCriterionWeights* are internal calculation values and are therefore not converted to triples. The finished conversion model of a thing-presentation-mesh is shown in Figure 4.7.

```
clsThingPresentationMesh
-moActivations : Hashmap<eActivationType, Double>
-moContent : Object
#moContentType : eContentType
-moCriterionMaxValues : Hashmap<eActivationType, Double>
-moCriterionWeights : Hashmap<eActivationType, Double>
#moDataStructureType : eDataType
#moDebugInfo : String
#moDS_ID : int
#moDSInstance_ID : int
#moExternalAssociatedContent : ArrayList<clsAssociation>
#moInternalAssociatedContent : ArrayList<clsAssociation>
-mrAggregatedActivtionValue : double
-mrCathexis : double
```

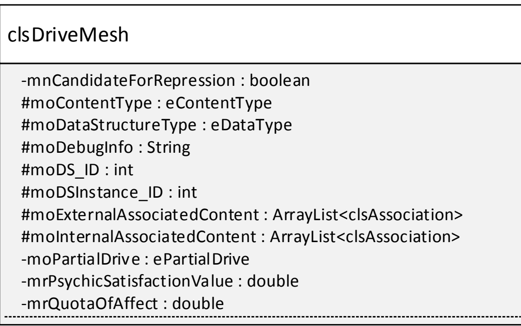
S	P	O
<ds>	<ars:hasMoContent>	“[ds.moContent]”
<ds>	<ars:hasMoContentType>	<[ds.moContentType]>
<ds>	<RDF.type.getURI()>	<[ds.moDataStructureType]>
<ds>	<ars:hasDebugInfo>	“[ds.moDebugInfo]”
<ds>	<ars:hasMoDS_ID>	“[ds.moDS_ID]”
<ds>	<ars:hasMoDSInstance_ID>	“[ds.moDS_Instance_ID]”
<ds>	<ars:hasExternalAssociation>	<[ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasAssociation>	<[ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasInternalAssociation>	<[ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasAssociation>	<[ds.moInternalAssociatedConent.get(i)]>
<ds>	<ars:hasAggregatedActivationValue>	“[ds.moAggregatedActivationValue]”

Figure 4.7: Thing-presentation-mesh and its conversion to RDF triples

Drive-Mesh

The *drive-mesh* also belongs to the primary process and is used to represent what experiences the agent associates with the object. It contains information about the impact of the associated thing-

presentation on the homeostatic state based on the agent's experiences. For example a cake could have reduced the sensation of hunger for one agent, while another one felt sick after eating it. The corresponding Java class is the `clsDriveMesh`.



S	P	O
<ds>	<ars:isCandidateForRepression>	"[ds.mnCandidateForRepression]"
<ds>	<ars:hasMoContentType>	<[ds.moContentType]>
<ds>	<RDF.type.getURI()>	<[ds.moDataStructureType]>
<ds>	<ars:hasDebugInfo>	"[ds.moDebugInfo]"
<ds>	<ars:hasDriveComponent>	<[ds.moDriveComponent]>
<ds>	<ars:hasMoDS_ID>	"[ds.moDS_ID]"
<ds>	<ars:hasMoDSInstance_ID>	"[ds.moDS_Instance_ID]"
<ds>	<ars:hasAssociation>	<ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasExternalAssociation>	<ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasAssociation>	<ds.moInternalAssociatedConent.get(i)]>
<ds>	<ars:hasInternalAssociation>	<ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasPartialDrive>	<[ds.moPartialDrive]>
<ds>	<ars:hasPsychicSatisfactionValue>	"[ds.mo PsychicSatisfactionValue]"
<ds>	<ars:hasQuotaOfAffect>	"[ds.moQuotaOfAffect]"

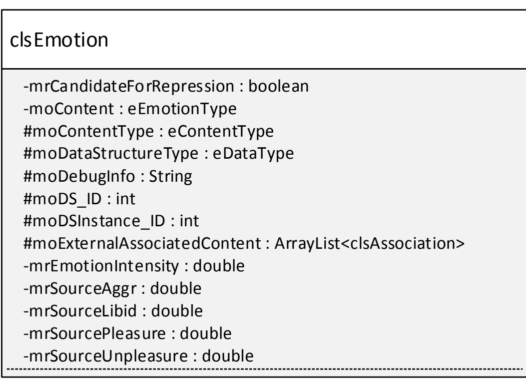
Figure 4.8: Drive-Mesh and its conversion to RDF triples

The attribute *mnCandidateForRepression* is used to mark whether the drive-mesh is an object for suppression in the defense mechanisms, while a drive-meshes *moContentType* states differentiates whether a drive-mesh is a "MEMORIZEDDRIVEREPRESENTATION" or "LIBIDO". Memorized drive-meshes are satisfied needs, while libidinous ones are still to be fulfilled. The *moDriveComponent* tells whether a drive is an AGGRESSIVE or a LIBIDINOUS one. For the same need there is always an aggressive one (for example "bite") and a libidinous one (like "nourish") [SDW+13, p. 6650]. The *mrQuotaOfAffect* states how high the bodily need is, while the *moPartialDrive* identifies the source of a sexual drive [SDW+13, p. 6648]. Its *moDataStructureType* always contains the value

"DM" and for all other attributes one may refer to the explanation of the thing-presentations conversion. Like the thing-presentation-mesh the drive-mesh contains some links to other data structures, which is done by connecting the drive-mesh IRI with the linked data structures IRIs.

Emotion

Finally, there is also a data structure called *emotion* which was introduced to SiMA in order to model the agent´s motivational system and is represented by the class `clsEmotion` [SDW+13, p. 1]. Emotions represent the coordination of bodily needs and the perceived reality.



S	P	O
<ds>	<ars:isCandidateForRepression>	"[ds.mnCandidateForRepression]"
<ds>	<ars:hasMoContent>	<[ds.moContent]>
<ds>	<ars:hasMoContentType>	<[ds.moContentType]>
<ds>	<RDF.type.getURI()>	<[ds.moDataStructureType]>
<ds>	<ars:hasDebugInfo>	"[ds.moDebugInfo]"
<ds>	<ars:hasMoDS_ID>	"[ds.moDS_ID]"
<ds>	<ars:hasMoDSInstance_ID>	"[ds.moDS_Instance_ID]"
<ds>	<ars:hasAssociation>	<ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasExternalAssociation>	<ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasIntensity>	"[ds.mrEmotionIntensity]"
<ds>	<ars:asAggressionSource>	"[ds.mrSourceAggr]"
<ds>	<ars:hasLibidoSource>	"[ds.mrSourceLibid]"
<ds>	<ars:hasPleasureSource>	"[ds.mrSourcePleasure]"
<ds>	<ars:hasUnpleasureSource>	"[ds.mrSourceUnpleasure]"

Figure 4.9: Emotion and its conversion to RDF triples

There exist six basic emotions (anger, mourning, anxiety, joy, saturation and elation) which are generated based on four emotion factors (unpleasure, pleasure, sum of all aggressive quota of affects and sum of libidinous quota of affects) [SDW+13, p. 4]. The emotion factors in turn are created from the

bodily needs and the perceived situation. For memories emotions are of high importance, because memory that is associated strongly to an emotion has higher chances to be activated.

Like the drive-mesh the emotion has a Boolean value *mnCandidateForRepression* for marking suppressible instances. The *moContent* describes the type of emotion like "ANGER, "JOY" and "ANXIETY", All values come from the six basic emotions. Currently all existing emotion have the *moContentType* "BASICEMOTION". The *mrEmotionIntensity* states how strong the emotion is, while the values *mrSourceAggr*, *mrSourceLibid*, *mrSourcePleasure*, *mrSourceUnpleasure* represent the four emotion factors [SDW+13, p. 4]. For the meaning of all other values again, please refer to the explanation of the thing-presentation.

4.2.2 Secondary Process Data Structures

As mentioned in chapter 4.2.1 the secondary process is responsible for resolving contradictory or conflicting information that is passed on by the primary process [Zei10, p. 52]. It is also the counterpart to the primary process, which is following the pleasure principle, as it covers the reality principle. Furthermore, it is also responsible for structuring information and adds temporal local and logic information to the received data structures. In the secondary process goals are formed and decisions are taken, which requires logical structures like the word-presentation.

Word-Presentation

The basic data structure of the secondary process is the *word-presentation,* which is realized in the class `clsWordPresentation`. Word-presentations are basically a set of symbols forming the description of an object [ZLM09, p. 385]. For example, for human beings this could be verbal expressions as well as gestures and sounds. Their structure is very similar to the primary process´s thing-presentation. Like for the thing-presentation a mesh structure is used to connect the word-presentation by associations to other data structures.

clsWordPresentation
-moContent : String #moContentType : eContentType #moDataStructureType : eDataType #moDebugInfo : String #moDS_ID : int #moDSInstance_ID : int

S	P	O
<ds>	<ars:hasMoContent>	"[ds.moContent]"
<ds>	<ars:hasMoContentType>	<[ds.moContentType]>
<ds>	<RDF.type.getURI()>	<[ds.moDataStructureType]>
<ds>	<ars:hasDebugInfo>	"[ds.moDebugInfo]"
<ds>	<ars:hasMoDS_ID>	"[ds.moDS_ID]"
<ds>	<ars:hasMoDSInstance_ID>	"[ds.moDS_Instance_ID]"

Figure 4.10: Word-Presentation and its conversion to RDF triples

The main difference between a thing-presentation and a word-presentation lies in its *moContentType* attribute having nine possible values, namely: RELATION, LOCATION, DISTANCE, CONDITION, CONTENT, INTENSITY, ACTION, POSITION and ENTITY. The *moDataStructureType* is "WP" and the *moContent* could be "CAKE" or the verbalization of a hex-colour defined by a thing-presentation.

Word-Presentation-Mesh

The *word-presentation-mesh* is the equivalent to the thing-presentation-mesh of the primary process in the secondary process. The corresponding class is the `clsWordPresentationMesh`. It also contains internal and external associations, representing information about the object and external information that is associated with it. Associations are used the same way as in the primary process and are explained in chapter 4.2.1. Word-presentation-meshes are also used to represent the concept of *images* in the secondary process and are of high importance for the memorization process which will be implemented in this work.

```
clsWordPresentationMesh
-moAssociationMapping : HashMap<ePredicate, ArrayList<clsSecondaryDataStrucutre>>
-moContent : String
#moContentType : eContentType
#moDataStructureType : eDataType
#moDebugInfo : String
#moDS_ID : int
#moDSInstance_ID : int
#moExternalAssociatedContent : ArrayList<clsAssociation>
#moInternalAssociatedContent : ArrayList<clsAssociation>
-moNullObject : clsWordPresentationMesh
```

S	P	O
<ds>	[ds. moAssociationMapping.get(i).getKey()]	"[ds.moAssociationMapping.get(i).getValue]"
<ds>	<ars:hasMoContent>	"[ds.moContent]"
<ds>	<ars:hasMoContentType>	<[ds.moContentType]>
<ds>	<RDF.type.getURI()>	<[ds.moDataStructureType]>
<ds>	<ars:hasDebugInfo>	"[ds.moDebugInfo]"
<ds>	<ars:hasMoDS_ID>	"[ds.moDS_ID]"
<ds>	<ars:hasMoDSInstance_ID>	"[ds.moDS_Instance_ID]"
<ds>	<ars:hasExternalAssociation>	<ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasAssociation>	<ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasAssociation>	<ds.moInternalAssociatedConent.get(i)]>
<ds>	<ars:hasInternalAssociation>	<ds.moExternalAssociatedConent.get(i)]>
<ds>	<ars:hasAggregatedActivationValue>	"[ds.moAggregatedActivationValue]"

Figure 4.11: Word-Presentation and its conversion to RDF triples

Word-presentation-meshes can have “RI” (Remembered Image), “ACTION” and “ENTITY” as values for their *moContentType*. The attribute *moContent* gives some name for the object or action the mesh describes like for example “CAKE” or “FOLLOW_TARGET”. The attribute *moDataStructureType* contains for all word-presentation-meshes the value “WPM”. The attributes *moNullObject* and *moAssociationMapping* are for internal calculations only and therefore not converted to a triple representation. All other attributes are converted as already described in previous data structures (see thing-presentation and thing-presentation-mesh in chapter 4.2.1).

Act

The central data structure of this work however is the concept of an *act*. An act is used to connect several word-presentation-meshes, which are used to represent environmental situations to a sequence of happenings. In Figure 4.12 a schematic diagram of an act is presented. The act has two functions: It connects data structures and adds additional information like a name for their whole sequence. Its function can therefore be compared to the mesh structures that are used in SiMA. To connect acts and their associated images the `clsAssociationSecondary` is used. In order to distinguish the meaning of those associations they have different content types assigned. As can be seen in Figure 4.12 the connection to an act is marked by the term “hasSuper”, whilst the temporal association between two images gets the value “hasNext”. As the terms imply those associations have a direction pointing from the image to the act and from the temporally first image to the one that occurred afterwards.

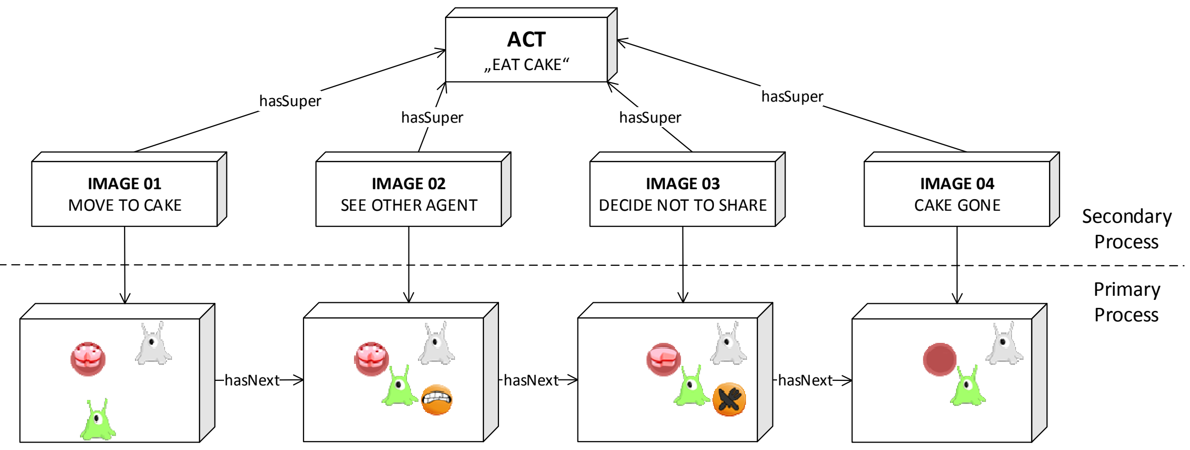


Figure 4.12: Example of an act

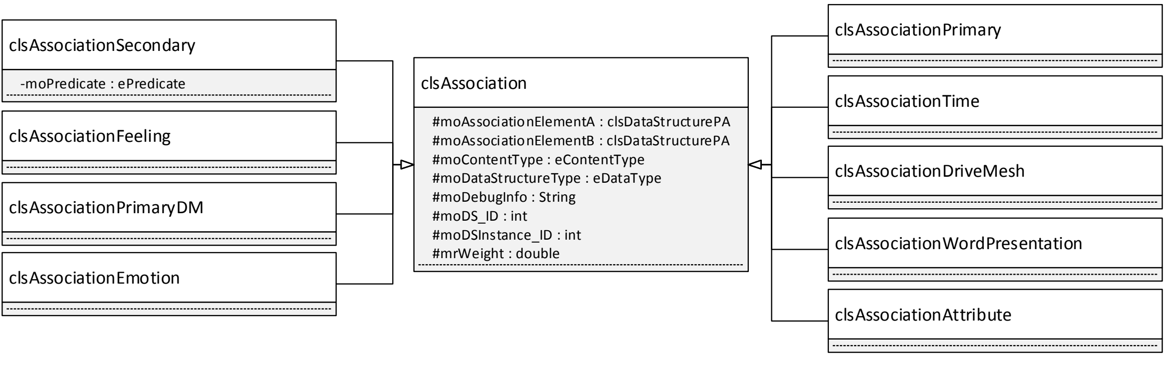
Due to the fact that the act concept is represented by the data structure word-presentation-mesh no separate conversion concept is needed. For further information about the representation of acts in the RDF database, refer to the section about word-presentation-meshes.

4.2.3 Associations

As was already mentioned in the section about primary structures *associations* are used in the primary process as well as in the secondary process. In the implementation there are nine different association

types, each of them responsible for a different type of connection. The class `clsAssociationAttribute` is used to link a thing-presentation to its thing-presentation-mesh. `clsAssociationDriveMeshes` link drive-meshes to thing-presentation-meshes and `clsAssociationWordPresentations` do the same for word-presentations. The class `clsAssociationPrimary` links `clsThingPresentationMeshes` to each other, while `clsAssociationSecondary` are used to connect `clsWordPresentationMeshes`. Finally there exist the two associations `clsAssociationFeeling` and `clsAssociationTime`. The first one connects a `clsWordPresentationMesh` to a `clsWordPresentationMeshFeeling` (which is a generalization of a `clsWordPresentationMesh`). The latter one is used to connect two `clsThingPresentationMeshes` (representing an image with an object).

As can be seen in Figure 4.13 there are many types of associations, but only the class of `clsAssociationSecondary` adds more information to the data structure. The other data structures differ only in their offered operations and constructors, which is of no relevance for the RDF model.



S	P	O
<ds>	<ars moAssociationElementA>	<ds.moAssociationElementA]>
<ds>	<ars:moAssociationElementB>	<[ds.moAssociationElementB]>
<ds>	<ars:hasMoContentType>	<[ds.moContentType]>
<ds>	<RDF.type.getURI()>	<[ds.moDataStructureType]>
<ds>	<ars:hasDebugInfo>	“[ds.moDebugInfo]”
<ds>	<ars:hasMoDS_ID>	“[ds.moDS_ID]”
<ds>	<ars:hasMoDSInstance_ID>	“[ds.moDS_Instance_ID]”
<ds>	<ars:hasAssociationWeight>	“[ds.mrWeight]”
<ds>	<ars:hasPredicate>	<[ds.moPredicate]>

Figure 4.13: Associations and their conversion to RDF triples

4.3 Conversion Implementation

This chapter presents the actual implementation of the conversion process. The first part of this chapter explains the manual conversion process, along with its most important methods. It further discusses the IRI generation and highlights the important classes and methods. In the second part of this chapter the migration from the old file-based memory into the new database is explained.

4.3.1 Manual Conversion

After deciding to use the Sesame database system, it was clear that some sort of conversion to a triple representation was needed. The package `conversion` was therefore introduced to the ARSDatabase project. In Figure 4.14 a class diagram with the most relevant attributes and methods is presented. In chapter 4.1 the data structures and their conversion to RDF triples in general was discussed. This chapter deals with the implementation of the discussed concepts.

The final implementation is oriented on the hierarchical structure of the data structures which was already presented in

Figure **4.4**. The entry point for a convenient conversion process is the class `TripleStoreUtils`. It provides a `convert(…)` method that accepts as an input parameter the abstract class `clsData-StructurePA`. This class is the most general data structure in the SiMA data structure hierarchy, thus meaning that all data structures can be converted by that method. From there the data structure is passed to the class `DataStructurePAToTripleConverter`, which is located in the package `conversion.classes`. This class has only one task, namely to decide which data structure was provided as an input and to pass it on to the corresponding conversion class. For every data structure of the long-term memory there exists an additional class, which converts its data structure to a java.util.ArrayList [ALIST] of `TemporaryContainerTriples`. As already mentioned in the previous chapter an intermediate triple representation was necessary to stay independent from the concrete RDF database solution that was chosen.

The `TemporaryContainerTriple` provides some basic attributes that allow the correct conversion to the database specific triples, for example an option to save whether the object part of the triple is an IRI or a Literal. There is also a Boolean parameter that allows to mark a “basic” object. The basic object is the topmost object of a number of linked data structures, for example the cake that is actually seen by the agent. Any other object linked to that cake by associations is not considered to be a basic object. Such a distinction is necessary as for search methods only basic objects shall be taken into account.

The last part of the conversion process is the `comparator` package. It was originally created for usage during the IRI generation to overcome some shortfalls of the SiMA implementation. Due to some implementation specific requirements SiMA makes extensive use of a cloning process which makes new instance copies of the data structure objects. Until now this was necessary in order to avoid changing the contents of the HashMap memory. Therefore, two Java instances could still represent the same thing in terms of long-term memory while they are two different objects for a Java program.

However, as will be covered in chapter 6, this approach could not fully reproduce the expected agent behavior and was therefore dropped for a more straightforward one.

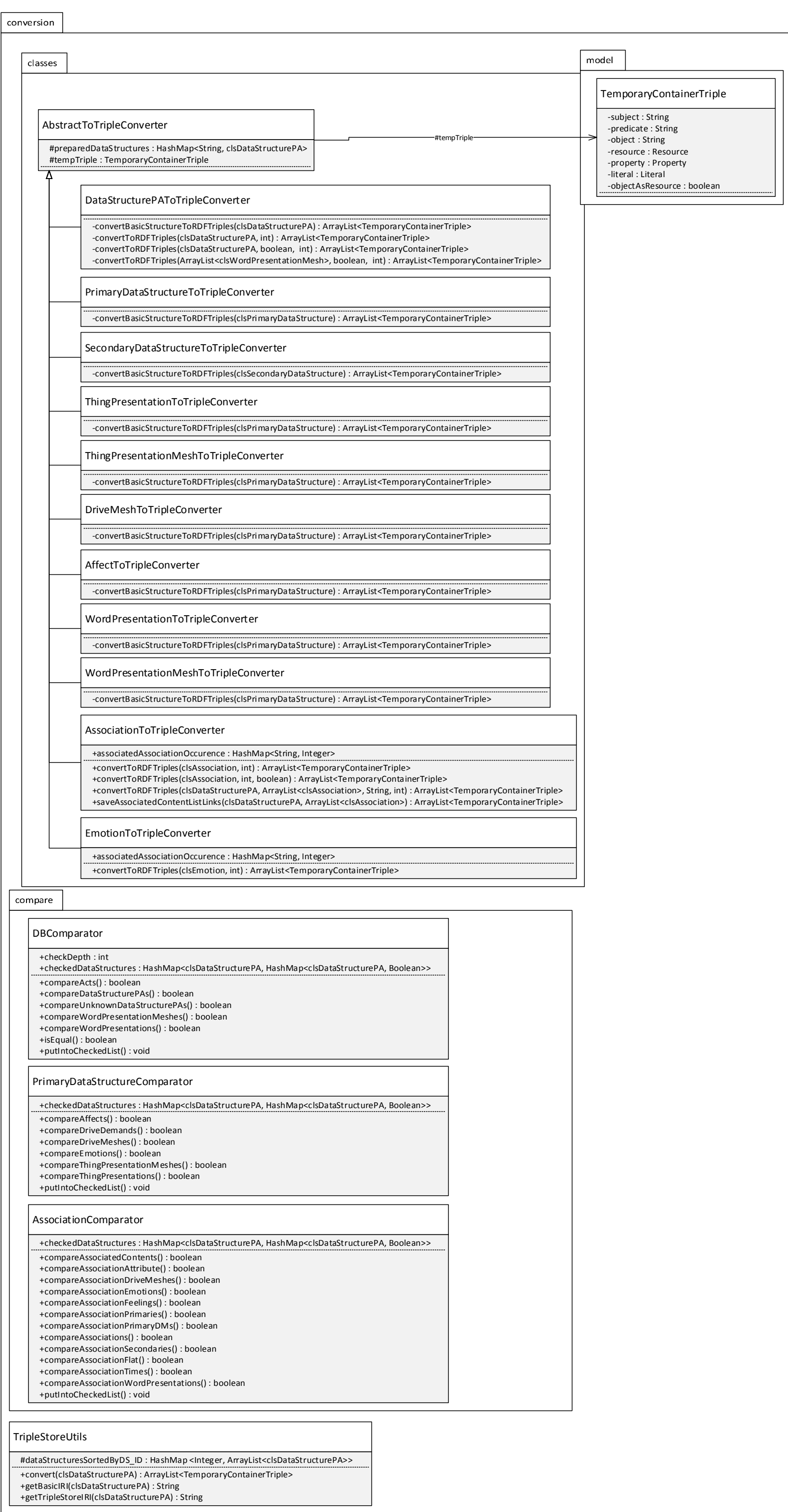


Figure 4.14: Class diagram of the conversion package

In order to migrate the HashMap contents to the new RDF store, two new Integer attributes were added to all `clsDataStructurePA`. The first one is a global counter, keeping track of the number of instances that are created, the second one marks the actual instance´s number. This so called `javaInstanceID` is necessary to differentiate cloned objects from their parent objects. The cloning process was modified to first copy all values from the parent object and then create a new `javaInstanceID` for the clone. Until now there was no real identifier for concrete data structure instances, which made it impossible to create a unique IRI for any SiMA data structure.

```
1:   return O.DataStructureType + "_" + O.JavaInstanceID + "_" +
        O.ContentType + "_" + O.DSID + "_" + O.DSInstanceID
```

Figure 4.15: Algorithm - getTripleStoreIRI(O)

Even though the `javaInstanceID` would suffice to generate an IRI some additional information about the data structure was added to the IRI in order to provide more readability for humans. The final implementation of the IRI generation can be seen in Figure 4.15.

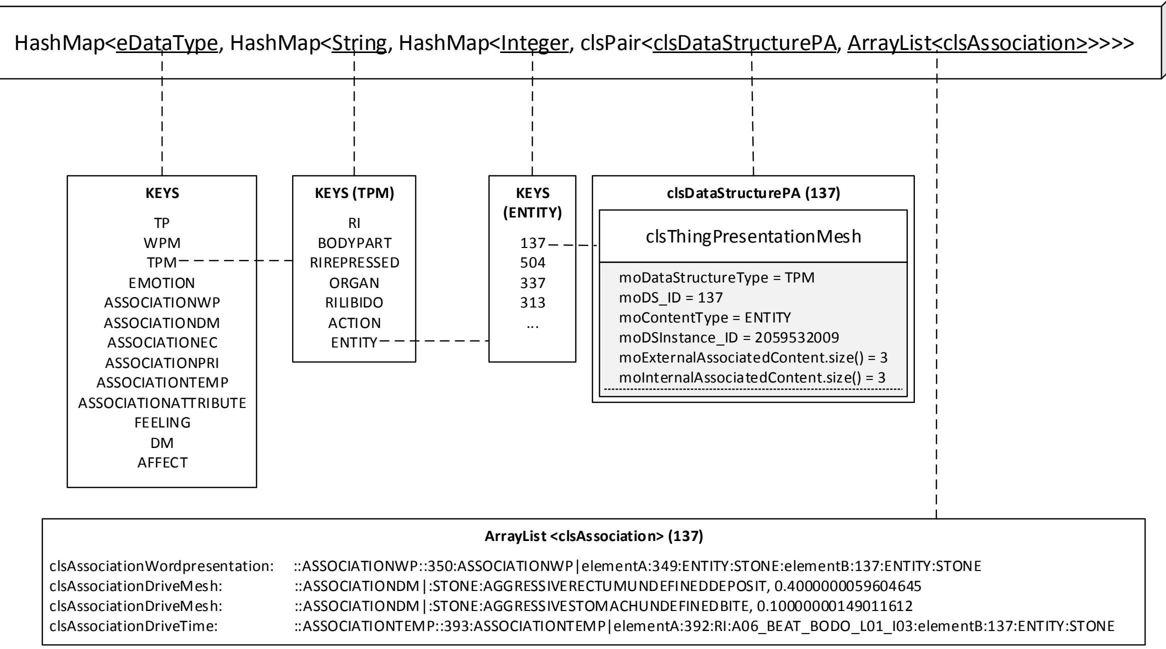


Figure 4.16: HashMap structure of SiMA mock-up memory

In addition to the obvious conversion of the existing data structure attributes some additional information has to be transferred. As already mentioned in chapter 1 SiMA uses a parsed file to simulate the agent's memories. At program startup currently a file is parsed into a complex HashMap structure which is presented in Figure 4.16. This structure commonly referred to as *search space* is currently used during the simulation as a long-term memory mock-up. At program start the ontology file is parsed into a HashMap with a `<String, clsDataStructurePA>` key-value pair. In this case

the key is a unique String representation of the data structure in the value part. For these values a new `HashMap` containing other HashMaps is created. This structure of HashMaps is constructed to provide a fast access during simulation runs and shown in Figure 4.16.

As can be seen the shown HashMap structure was designed to improve the search process during program run. First the data structures of the memory are sorted by their `eDataType` (which represents the data structure type as an enum). Then the data structures are sorted by their content type, like for example “ENTITY”, “ACTION”, “ORGAN” etc. for a thing-presentation-mesh. Afterwards the structures are sorted by their data structure ID. Finally the `clsPair<clsDataStructurePA, ArrayList<clsAssociation>>` which was parsed at program start is put into the Map. This `clsPair` is the real content of the memory in contrast to the previous structure which was only set-up in order to pre-sort the `clsPairs`.

In order to keep the information of the `HashMap` structure, the conversion process has to produce some triples in addition to the structure information. To differentiate the main data structures of the memory from those merely associated to them, all data structures from the `clsPair` are supplemented by the prefix “B_” (B for basic data structure) added to the front of their IRI. This is only done to increase the readability of the database content for humans, it is not necessary for the implementation or functionality of the implemented RDF database. For the database a triple containing the information *“isSearchSpaceContent”* (see Figure 4.17 for the whole triple) is added. This is necessary to differentiate merely associated data structures from the “main contents” of the memory, when searching through the database. To keep the relationship of the `clsPairs` from the HashMap another triple with the predicate *“hasInExternalList”* is used during the conversion. Further triples that have to be added to the database are two counters that are used to store the last value for a DSID and amount of perceived objects. They will be needed for the IRI generation of the memorization process (covered in chapter 5.3).

S	P	O
<basicDS>	<http://ars.org/isSearchSpaceContent>	“true”
<pair.a>	<http://ars.org/hasInExternalList>	<pair.b>
<http://ars.org/DSID>	<http://ars.org/hasValue>	“counter”
<http://ars.org/PerceivedCounter>	<http://ars.org/hasValue>	“counter”

Figure 4.17: Triples containing general information for the database

4.3.2 Search Space Migration and Memory Access

The `RDFSearchSpaceCreator` was implemented to migrate the old search space from the Frames file into the chosen RDF database. To achieve this the original parsing process which is used by SiMA to load the Frames file is utilized to create the original HashMap structure. The content is then passed to the `RDFSearchSpaceCreator` in order to perform a conversion process. First the conversion process iterates through all “basic” data structures and assigns an IRI to them. A data structure is considered to be basic if it directly belongs to the `clsPair` structure. Then, instead of sorting the

`clsPairs` into the HashMap the data structures are converted to triples by the use of the methods that were described in chapter 4.3.1. During that process the "`hasInExternalList`" triple for the relation between the `pair.a` and `pair.b` element is created as well. After all triples are set-up they are saved to the database all together.

Figure 4.18: Class diagram of the memory package

The implementation of the `itfSearchSpaceAccess` interface was closely connected to the process of migrating the old data store into the new triple store, which was already described in chapter 4.3.1. Without this implementation the new database could not be used by SiMA. Therefore the original implementation along with its helper class was copied and adapted to match the characteristics of the new knowledge store. The main change that was necessary to the `RDFSearchSpaceManager` (the new implementation of that interface) was a new link to its helper class, which implements the searching. Both classes can be seen in Figure 4.18. This helper class is called `clsDataStructureComparisonTools` in the original implementation and `DataStructureComparisonTools` in the new implementation that was done in this work. The main difference between both

implementations is that the original one also utilized that class to search through the whole search space HashMap structure, which was described in 4.3.1. In the new implementation the `RDFSearchSpaceManager` already fetches a pre-filtered list of data structures, which is then passed to the `DataStructureComparisonTools` to do the rest of the filtering like it used to do in the former implementation.

The pre-filtering moves some of the `clsDataStructureComparisonTools` processing into queries so that only eligible data structures are considered for the matching algorithm. This querying is a two-step process, as the database is first searched for a really close match and if none is found a more general query is constructed. For the first query an exact match of `moContentType`, `moDataStructureType` is required, whilst for the second one only the `moDataStructureType` has to match. After fetching that list of data structures from the database the `RDFSearchSpaceManager` passes it on to the `DataStructureComparisonTools` where the internal compare methods of the data structures are called in order to acquire a match score. All data structures with a match score beyond a given `THRESHOLDMATCH` are then returned to the `RDFSearchSpaceManager`.

The last adjustment that was necessary for the search process to work was the resemblance of the `clsPair` which was part of the original search space and described in chapter 4.3.1. The process which was described until now fetched all data structures that met a certain match score regarding a specific data structure that was passed to the search method. Those data structures resemble the `pair.a` part of the search space HashMap that was formerly used. The `pair.b` part of the `clsPair` is still missing at that time. As was already mentioned in chapter 4.3.1 a special triple with the predicate *hasInExternalList* was introduced to keep the association mapping between the entity and the associated data structures from the search space. After assembling all data structures with the associations, they are returned to the decision unit as a search result.

5. Memorization Process

The last step towards a proactively experience gaining agent is the memorization process. In chapter 2.3 several simulation environments and their memory management approaches were introduced. The following chapter covers the approach that was chosen to be implemented into SiMA. The memorization process itself consists of two steps. First perceived events have to be stored into a temporary memory from which, after some time (which has to be defined in this work), relevant information is moved to the long-term memory. As described in chapter 4.2 the two main data structures of this process are the word-presentation-mesh (representing an image) and the act. In the following chapter the concepts of collecting images and constructing an act are covered.

5.1 Concepts of the Memorization Process

In [Zei10, pp. 58–59] a distinction between *perceived images* and *template images* is discussed. Perceived images represent the perceived environmental, bodily and homeostatic state and can be seen as some sort of snapshot of the external and internal system state. Template images are mentally created from perceived images and represent patterns for new images. Images are formed from thing-presentation-meshes, thing-presentations and other template images that occur at the same time and have therefore a temporal association between them (shown in Figure 5.1). If a perceived image partly or fully matches a template images the template image is retrieved for further processing. In case that two images match the image with the highest matching grade is selected. The matching process can be seen in Figure 5.1, where Template Image 1 offers two matches and is therefore selected over Template Image 2 which has only one match to offer.

Since the first definitions of template images and perceived images the concept of a general *image* has been implemented in SiMA. Instead of implementing a special data structure in the primary process the thing-presentation-mesh is used to represent simultaneousness between entities. Emotions can only be attached to images in order to tell the agent how he felt in that specific situation. For the construction of acts the word-presentation-mesh of the image is used and associated to acts by the usage of secondary process associations.

As images can only contain a snapshot of the actual situation and do not track any changes [Zei10, pp. 58–59], the concept of an act was defined, but never used in SiMA. In the original concept acts connect

single images to a sequence, by the use of word-presentations [Zei10, pp. 58–59]. The word-presentation is necessary in order to map actions and sequences to a period of time. Figure 5.2 shows the connection between acts, word-presentations and template images in the original concept.

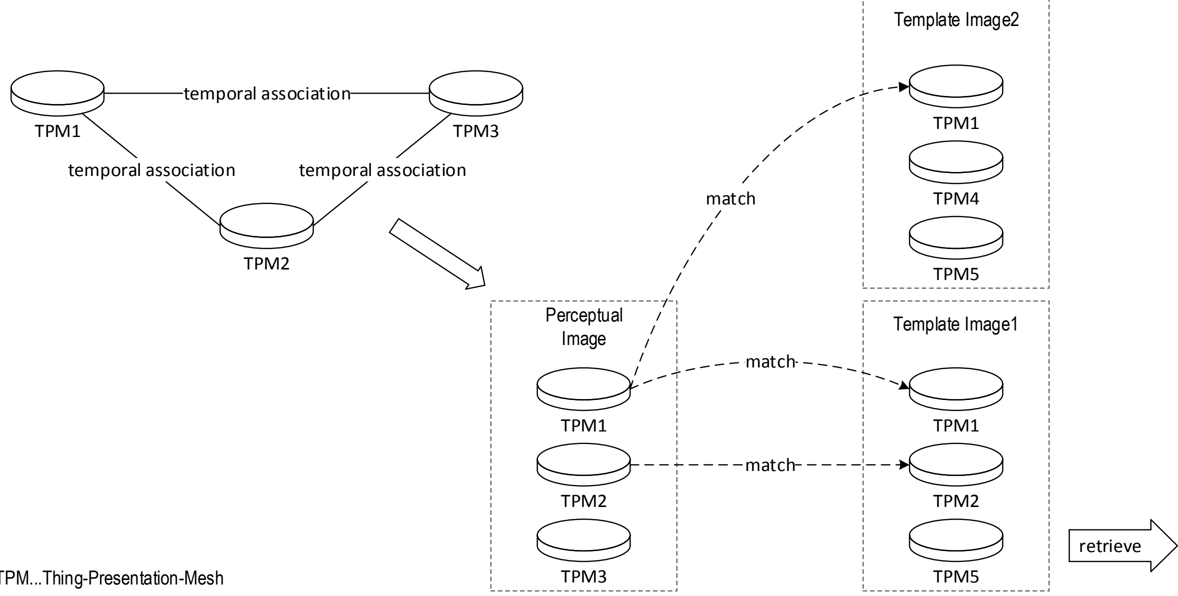


Figure 5.1: Image creation and retrieval based on [Zei10, pp. 57-59]

From Template Image 1 to Template Image 2 the TPM3 has been removed. The two word-presentations are linked by a *temporal relation*, enabling the realization of a reference time and an *action relation*, defining how the situation in the latter image can be reached from the first one.

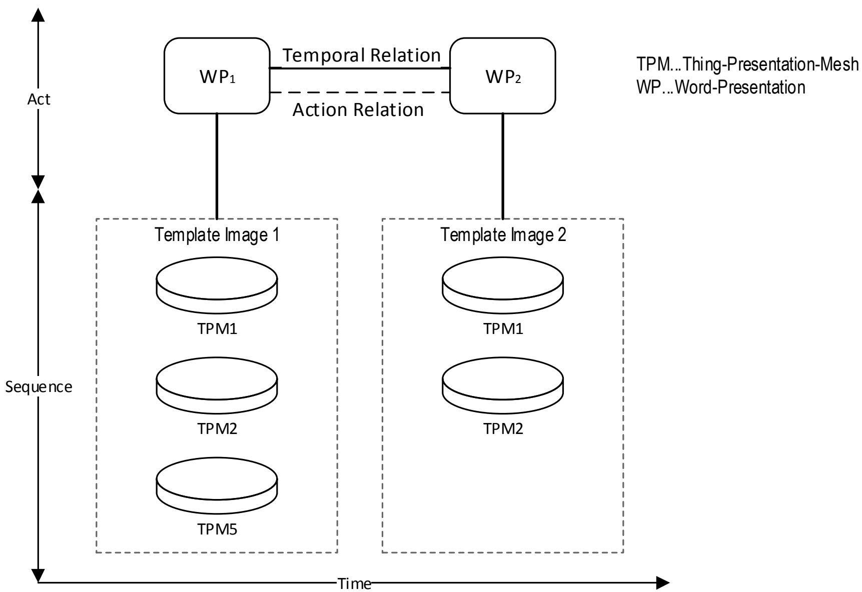


Figure 5.2: Template Images forming an Act based on [Zei10, p. 61]

In the actual implementation acts are implemented only in the secondary process and consist of word-presentation-meshes which are connected to the act and between each other. The word-presentation-mesh in this case represents the secondary process structure of the image. The connection between the word-presentation-meshes implies the temporal appearance of the images, while the connection to the act states that the image is generally part of the sequence the act describes. In order to describe the function of the associations so called predicates are attached to them. To the temporal association between images a predicate with the value "hasNext" is attached. Between the act and the image a predicate with the value "hasSuper" is used.

5.2 Memorizing Experiences

After implementing all the prerequisites, the following chapter finally deals with the memorization process for episodic memories itself. In order to answer the research questions *Research question 2: "How shall the agent decide which memories are to be kept?"* and *Research question 3a: "When shall the act generation be triggered?"* a short evaluation of existing memory concepts with respect to the applicability to the research questions is presented in the following.

In CHREST the memory content is generated at start up [SL07, p. 2]. To simulate different levels of chess mastery time and brain capacity limits can be set to the acquisition of chunks, while scanning through a large database of chess positions from master-level games. During this process often reoccurring patterns are transformed into templates, thereby forming the contents of the CHREST long-term memory. Even though the general concepts of CHREST differ greatly from those used in SiMA, the idea of "strengthening" memories by repeated perception is one that could be adopted in SiMA as well.

The original BDI architecture has no learning mechanisms implemented, however, there are some attempts to implement learning in BDI based systems like for example dMars, which was introduced in chapter 2.3.5. This BDI implementation uses induction on logical decision trees in combination with Lisp files. ACT-R uses production compilation to add new information to its memory [CTN09, p. 109]. Production compilation basically tries to merge two production rules into one rule [ABB+04]. Both architectures use approaches that are so different to SiMA, that any comparison or application of them is pointless.

ICARUS and SOAR both acquire new knowledge by experiencing impasses [CTN09, p. 111]. In ICARUS a new skill is learned, if the selected action can be successfully executed. SOAR offers more complex memory mechanisms, providing chunking, reinforcement learning, episodic memory and semantic memory [CTN09, p. 107]. If an impasse was detected a goal or subgoal is terminated and through chunking mechanisms new production rules are added to the memory. Failing and succeeding leads to rewards or punishments in SOAR, thereby enabling the system to perform reinforcement learning. The described approaches point to other interesting options for SiMA, even though learning mechanisms are not in the current scope of this work. The usage of a goal based approach is probably not only successful when used as a trigger for learning, but may also be used for triggering episodic memorization or when it comes to evaluating the importance of memories at hand. The reward and punishment approach in turn may be translated to the SiMA emotional system by declaring that a

positive emotion may be treated like a reward, whilst negative emotions correspond to a punishment. These emotions and their strength may be used to consider the importance of memories or to indicate that something happened that is worth remembering.

5.2.1 Memorization Trigger

As already discussed the agent perceives images which have to be stored until it can be decided if they are of any interest for saving them to the long-term memory. Therefore, some sort of buffer for perceived images has to be created. To that buffer all perceived images are saved until the system decides that their importance for the future can be calculated. A crucial question when implementing a memorization process is *Research question 3a: "When can the agent evaluate which image is of importance to him?"*. To answer that question one has to evaluate several possible approaches with respect to their applicability to the project at hand:

- **Create an act all n minutes:** Checking after a predefined amount of time, whether valuable information was gathered has the advantage that the condition is very easy to check. However, it seems to be a rather unfunded approach, as memorization in general happens on purpose by learning or by some sort of trigger. For example in [DFZB09, pp. 84–85] the possible importance of emotions for learning is discussed.
- **Create an act if high emotions are experienced:** Experiencing high emotions is a good indicator that something memorable has happened. Emotions may be utilized as a signal whether something is "good" or "bad" for the agent. Depending whether the emotion is pleasant or unpleasant the agent could take the same actions again or try another solution in the future. As SiMA utilizes the concept of emotions, this would be a promising approach, which is similar to the reinforcement learning approach used in SOAR.
- **Create an act if a goal was reached:** As already discussed in the beginning of this chapter, goals are often utilized to mark special events or experiences (like for example in SOAR or ICARUS). Besides emotions agents keep track of their goals and needs. Therefore the most reliable trigger for SiMA seems to recapitulate the experienced situations after a goal was reached. This enables the agent to save only those images to an act that are really related to it. For example, if the agent was hungry, but had to go to the toilet between finding some food and realizing he is hungry, the toilet sequence would not be of interest for an act about finding some food. (Note that going to the toilet could be another goal and form a new act.). For those reasons the memorization process created in this work will build an act every time a goal was reached.

5.2.2 Filtering the Memory

Even though memory for agent systems in general is a much researched topic (see chapter 2.3 for examples), many systems neglect the topic of memory consolidation. According to [SWT12, p. 1008] most systems are developed to store linearly ordered traces of experiences, thereby excluding memory consolidation processes, which transfer and reorganize the contents of the memory, as well as forgetting mechanisms, which discard irrelevant information. It should also be mentioned that the process of storing memories is very dependent on the concrete implementation and also the application area

of a system. For example, there is a huge difference between systems like CHREST (see chapter 2.3.1) which is based on chunks and mainly used in the area of chess and a system like SiMA, which is mainly focused on drives and emotional states. In the following some approaches for deciding the importance of an individual image, in order to answer *Research question 2: "How shall the agent decide which memories are to be kept?",* are discussed.

- **Discard images that are equal to their predecessor:** If no change of situation was experienced he agent can safely drop the perceived image, as no information can be lost.
- **Discard images between two images if the information of the surrounding images is sufficient:** If three images in succession are perceived and the image in between holds no information that cannot be gained through the other two images, the image can be discarded. For example, if the agent is far away from an object in one image, nearer in the next and close in the third image, the agent could as well reason from the information that he was far away first and close later on, that he has moved towards the object.
- **Discard images (or even whole acts) if the emotional importance is too low:** Depending on an agent's personality it could be possible to define certain thresholds for emotions or drives that decide whether an image is important. For example, if the agent prefers to solve a rivalry by evading it and he had to act in an opposite way, he may decide not to remember that event if it made him feel unhappy. Another personality would maybe remember that event, just because it made him unhappy in order to evade such a situation for the future. Again, this approach is similar to reinforcement learning approaches like they are used in SOAR.
- **Discard images (or even whole acts) if the general importance is too low:** Inspired by the approach which is used by CHREST, acts which have high similarities with other experiences are probably of more "daily relevance" than other situations which were experienced only once. For this approach it is necessary to track the age of an experience in order to only delete singular events that happened a long time ago. Otherwise new experiences would be deleted before they could prove their relevance.
- **Refactor acts that are similar to existing ones:** If a similar act has already been recorded, it should be considered to introduce a structure that models only the new information. Like for example thinking about a cake and remembering "I ate one yesterday and the day before yesterday." and "I took it from the fridge.". Whilst yesterday and the day before yesterday are different information, the position of the cake was always the same.

5.2.3 Creating an act

After successfully deciding which image is of importance to the experienced situation an act can be built from the images remaining from the filtering process. For now the only possible filtering mechanism is to sort out images containing no change to their predecessor. This is the most obvious, but also the most important step for a successful memorization process. It helps to keep the memory contents in a reasonable size and by that also reduces the time that is needed for searching through experiences. Further optimizations need additional adoptions to the data structures which is out of scope for this work. For example, it is possible to filter based on emotion intensity or to check whether a similar experience with higher satisfaction values has already been stored. In that case it should be

possible to enrich the actually stored information by additional information instead of creating a totally new one.

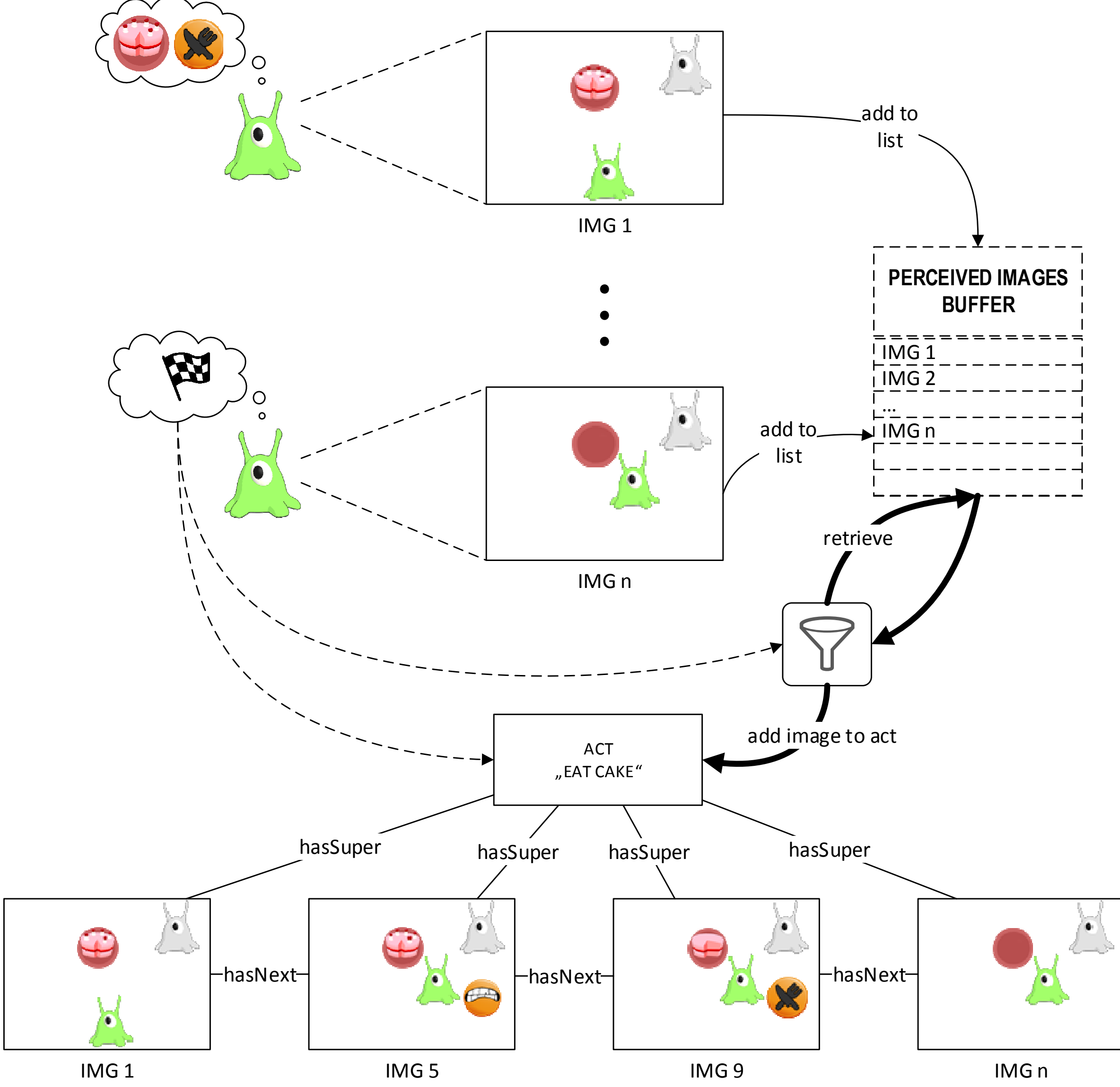


Figure 5.3: Creation of an act

Finally, after sorting out unnecessary images the act can be assembled. In Figure 5.3 the whole process of act creation starting from perceiving the first image until the assembly of the images to form an act is presented. The first step is the agent perceiving some situation and to form a new goal. In case of Figure 5.3 he wants to eat the cake he just saw. He saves the perceived image to a buffer that serves as a small short-term memory for perceived images. In the further progress many images are perceived and stored in that buffer, until the agent realizes he just managed to reach its goal. Then an evaluation of the last experiences takes place. Unneeded images are sorted out and are added to a newly created empty act. First all images get a temporal association with the predicate `hasNext` assigned according

to their appearance in the buffer. After that all images get a hierarchical association with the predicate `hasSuper` in order to link them to the act.

After the assembly of the images in an act the conversion process which was introduced in chapter 4.1 is put to work by converting the act and its images into a triple representation. This representation is then stored in the newly integrated RDF triple store which was discussed in chapter 3. Chapter 5.3 will cover the implementation of the concepts which were presented in this chapter.

5.3 Implementation of the Episodic Memory

After successfully introducing a new RDF triple store into SiMA the last step towards memorizing experiences was the implementation of the memorization process. In Figure 4.18 a class diagram of the `memory` package is shown.

This package contains all functions which are conceptually close to the memory or search space. Even though the conversion of the data structures itself was placed in a separate package the logic of the data store migration was placed in this package into the class `RDFSearchSpaceCreator`. Running this class will result in a new RDF database filled with all the data which was migrated from the original Frames file, which was used to manage the mock-up memory until now. Furthermore this package contains the implementation of the interface `itfSearchSpaceAccess` which is used by SiMA to search through its knowledge base. As this knowledge base has been replaced by an RDF triple store during the course of this work a new implementation of this interface was necessary. Finally the memorization process is located in this package. As can be seen in Figure 4.18 this class is accessed by some goal related action codelets from the project `ARSDecisionUnit`. The relation between the projects can also be seen from the package diagram in Figure 3.14. Further information on the role of codelets for the memorization process is provided later in this chapter.

5.3.1 Memorizing Experiences

For memorizing experiences there is only one class in addition to the already introduced classes of the conversion process. A class diagram representation of the class `MemorizationProcess` can be seen in Figure 4.18. As was already mentioned this class has dependencies on two of the action codelets in the decision unit. Codelets are independent pieces of code that are activated if a certain condition is met. In SiMA they are used to react to situations like the end of a goal or the upcoming feeling of panic. The memorization process makes use of those codelets to get the actually perceived image and to start the act construction if the current goal was reached. The perceived image is sent to the memorization process by the codelet `clsAC_EXECUTE_EXTERNAL_ACTION` and if the current goal is reached the memorization process is informed by the `clsCC_END_OF_ACT` codelet. For that purpose the static `MemorizationProcess` variables `CURRENT_GOAL_REACHED` and `CURRENT_GOAL` are set by the codelet if it realizes its conditions are met. In Figure 5.4 and Figure 5.5 the algorithm for saving experiences is presented.

Every time a new image is perceived the `clsAC_EXECUTE_EXTERNAL_ACTION` codelet passes it to the method `savePerceivedImageToDatabase(I)`, the image is saved and its IRI is stored to a buffer. After that it is checked whether the `CURRENT_GOAL_REACHED` variable was set to true by the `clsCC_END_OF_ACT` codelet. If that is the case the act creation is started. First the new act is labelled with an `eAction` that matches the sequence plot and a new `DSID` is generated for it. Currently `moContentType` and `moContent` are both set to the `eDataType "ACT"`. `De-bugInfo` and `moDSInstance_ID` are not needed by the simulation, but may be set optionally. After that for all perceived images the necessary links of type `clsAssociationSecondary` are created. For the association between `clsAct` and the `clsWordPresentationMesh` representing the image the constant `ePredicate.HASSUPER` has to be used. For linking the images between each other the `ePredicate.HASNEXT` is used. For both associations the `mrWeight` is currently statically set to "`1.0`". After adding the created associations to the images the act is linked to the images as well. Finally the conversion of the act and all images to `TemporaryContainerTri-ples` can be started and the triples are saved in the data store.

```
1:  savePerceivedImage(I)
2:  imageIRIBuffer.add(I.IRI)
3:  if (CURRENT_GOAL_REACHED) {
4:    createAct()
    end if
```

Figure 5.4: Algorithm - memorizePerceivedImage(I)

```
1:  act := createNewBasicAct()
2:  for i := 0 to imageIRIBuffer.size - 2 step 1 do
3:    i1 := getFromDatabase(imageIRIBuffer[i])
4:    i2 := getFromDatabase(imageIRIBuffer[i+1])
5:    link1 := createAssociationSecondary(i1, i2)
6:    link2 := createAssociationSecondary(i1, act)
7:    i1.addExternalAssociation(link1)
8:    i1.addExternalAssociation(link2)
9:    act.addAssociatedContent(i1)
10: end for
11: convertAndSaveTriples()
12: saveTriples()
```

Figure 5.5: Algorithm - createAct()

What might catch attention in this approach is that the filtering of images is already done before the final goal is reached by the agent, which is in contrast to the original concept which was presented in chapter 5.2. The main reason for this is, that at the moment the filtering process does not need information about the act or following images in order to decide which images may be disposed. As was covered in chapter 5.2 the only filtering that is possible for now is to check whether any change has happened between two perceived images, which can be safely done before putting the perceived image to the image buffer.

Another change that was necessary to the original concept is that not the perceived images are kept in this buffer, but only an IRI is kept in a list. This is a workaround and should be changed in future versions of the memorization process. The main reason for this workaround is the fact that the perceived image passed by the codelets is always the same Java instance with changed values. Therefore, putting the images in a list would result in a list containing the last perceived image several times. As it is out of scope for this work to make any changes to the perception processes for now the perceived images are saved right after they passed the filtering process. Only their IRI is stored in the buffer, which is enough to construct the act later on. Even though this solution is for the moment actually better than keeping all the images in the buffer and therefore use up more working memory, this is no permanent solution. Reason for this is, that many improved filtering processes would need to use all perceived images between forming a goal and reaching it to decide which images are of importance. However, until a more sophisticated mechanism is implemented this solution works without any drawbacks.

Resulting from this workaround another problem arises, namely the possibility that the simulation process is interrupted before an act is actually created. Therefore images that are not connected to an act are supplemented with the mark `isBufferObject` “true”. This enables the memory process to remove such unwanted content. The time that was chosen to check for remainders is after saving an act. If the current act was saved the perceived images that were linked to that act are stripped off the `isBufferObject` triple. Then all images still having such a mark are completely removed from the database.

It is possible to load the saved acts back into the simulation by using the same methods, which were used by the conversion process (described in chapter 4.3). However a specific usage of the saved acts was not implemented during this work as this would outreach the scope of one work.

6. Simulation

During this work the first steps towards proactive experience gaining in SiMA were taken. The first part of this work was to integrate an RDF database into the existing SiMA project and to migrate the old knowledge base into it. This included the implementation of a conversion procedure for the old search space structure which is defined by a Protégé Frames file. In order to prove the functionality of the conversion process a test scenario will be introduced and run with both data stores. The conversion may be considered to be successful, if the agent takes the same actions in both scenarios. The second part of this work was the implementation of a rudimentary memorization process. As it is at the moment not possible to test the impact of those experiences on the agent's behaviour, the resulting act will be loaded and analysed in a graph representation only.

6.1 Test Environment

In order to test the newly implemented functions it is necessary to run tests in some simulation environment. For this purpose a basic use case with limited complexity is chosen to prove the correct behaviour of the agent. During execution of the use case the following objectives have to be fulfilled:

1. The functionality of the new RDF database as a knowledge base has to be verified.
2. It has to be shown that the migration from the old file-based knowledge base to the new RDF database was performed correctly.
3. Newly experienced episodic memories shall be saved to the RDF database.
4. In a new run the simulation environment shall be able to access the previously saved memories.

To test all of the listed objectives two separate tests with the same use case are necessary. The separation into two test runs shall ensure that the test results for the two main topics, namely correct implementation of the declarative semantic memory and the first rudimentary episodic memory, are not influenced by possible flaws of the other one. List points 1 and 2 can be assigned to testing the declarative semantic memory and thereby form test case 1. The last two listings form test case 2 and cover those parts of this work which are related to the episodic memory.

To execute the above mentioned use case the multi-agent toolkit MASON is utilized. MASON is a simulation environment written in Java and was developed by the Georg MASON University [Lan10]. In Figure 6.1 an example of the used simulation environment and use case at start-up is presented. The framework provides a 2D-environment, a 2D-physics engine and supports multi-agent systems. It is

possible to step through the simulation, which will be used to compare the agent´s state at specific steps with the two knowledge bases. The implementation of the simulation environment and the scenario set-up were not part of this work, an already existing scenario with the name "UC1 - Standard scenario (Eat) minimalversion" is used.

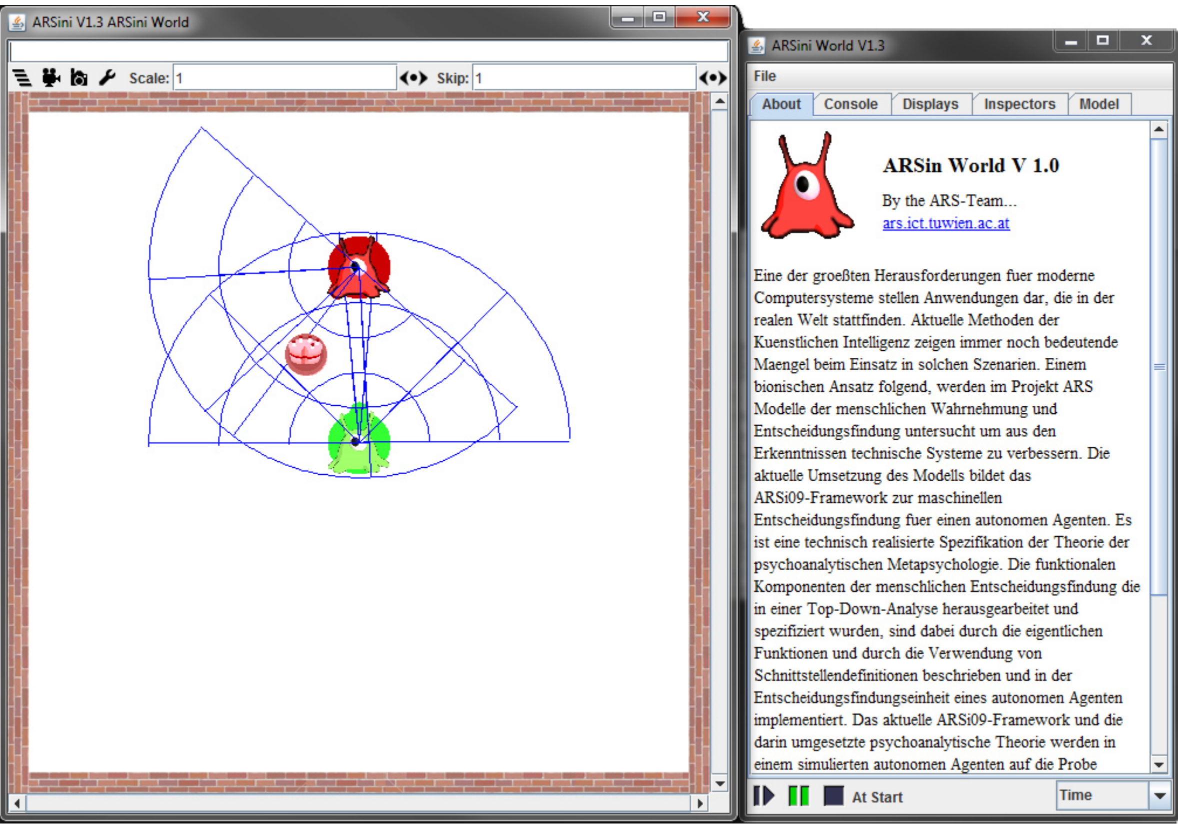


Figure 6.1: MASON Simulation Environment at Start

The scenario setup at start can be seen in Figure 6.1. The environment is surrounded by walls that mark the world boundaries, which the agent cannot pass. The green creature, called Adam, is the active agent, driven by a decision unit, whose actions are of interest for the test outcome. The agent tries to satisfy his homeostatic demands by roaming through the world, searching for energy sources while staying away from dangerous situations. The red creature is referred to as Bodo and its purpose is to bring in some contradictory drives to the agent´s basic bodily needs like for example consuming energy sources. In addition to the agents other entities may exist in a scenario. Example for entities are energy sources or obstacles that have to be overcome in order to reach the agent´s goals. The agent owns some sensors and actuators which enable him to interact with his world. He is able to perform a number of actions of which the relevant ones for the test cases are described in Table 6-1. Each action consumes a certain amount of the agent´s energy, thereby generating an ever increasing need of finding new energy sources. The currently performed action of the agent is indicated by showing one of the symbols from Table 6-1 next to the agent.

Both agents have a certain sensor range, which is marked by three blue half circles surrounding the agent. They mark how far away the perceived objects are in relation to the agent and there is also a distinction between right and left side of the agent.

In this scenario Bodo is not supposed to move, he will only stand near the food source in order to awake associations in Adams decision process. For example, it is possible for a hungry agent that perceives a cake and Bodo to ignore the other agent and eat the cake alone, or share the cake with him. In other cases which depend on the agents prior memories it is possible that it withdraws from the cake totally, as he is too afraid to get too close to the red agent. In Figure 6.1 an energy source "CAKE" which is located at the left side of the agent in a medium distance to it can be seen.

Symbol in Simulation Environment	Description
	Actions "MOVE_LEFT" and "MOVE_RIGHT" Indicate that the agent is currently moving to his left or right side. Those actions are usually performed to reach a certain entity or simply to roam through the environment in order to find objects that will satisfy the current needs of the agent.
	Actions "MOVE_FORWARD" and "MOVE_BACKWARD" Indicate that the agent is currently moving forward or backward. As before, those actions are performed to reach a certain entity or simply to roam through the environment in order to find objects that will satisfy the current needs of the agent.
	Action "EAT" Indicates that the agent is currently nourishing some food source. This action is performed to satisfy hunger and fill the agent´s energy sources.
	Action "BEAT" Indicates that the agent punches some other agent. If the agent feels some sort of aggression against another agent, it is possible that the agent beats the other agent, provided that no repression of the aggression takes place.

Table 6-1: Agent action symbols for use case UC1 - Standard scenario (Eat) minimalversion

When the scenario is started the agent first moves to the left and forward in order to reach the cake which is at start in a medium distance on his left side and therefore already perceived at scenario start-up. He then nourishes the cake and starts to roam around in the environment in search for energy sources. His final move is to beat Bodo. It is noteworthy that the simulation itself will never terminate, however, at the point of beating Bodo the agent stops moving.

6.2 Test Case 1 – Declarative Lexical Memory

To show that the migration of the file-based knowledge base is correctly performed, an execution of the previously introduced use case with both the old and the new implementation is necessary. If the

agent performs the same actions in the same environmental situation, the conversion may be considered to be successful. In order to provide comparable test results, all tests have been run on the same personal computer. The exact hardware setup can be found in Table 6-2.

System	Setup
Operating System	Windows 7 Professional
System type	64 Bit - Operating System
Processor	Intel® Core™ i7-3930K CPU @ 3.20GHz 3.20 GHz
Random Access Memory (RAM)	16,0 GB

Table 6-2: Hardware setup for simulation runs

Every simulation run can be divided into three phases which have to be initiated by the tester. As the performance of those phases is connected to the accessed data store for each test run in test case 1 the time will be measured and presented. The phases of a simulation run are defined as follows:

1. Start simulation: To initiate this phase the tester has to start the Java application. This phase ends when the scenario selection screen is shown to the tester.
2. Scenario initialization: In order to start this phase a certain scenario from the scenario selection screen has to be selected and started. It ends when the selected use case has been loaded and the scenario setup with both agents appears on the screen (see Figure 6.1 for an example).
3. Scenario run: After the scenario has been initialized by the scenario environment, the play scenario button has to be hit. After this the real simulation is running. As already stated earlier in this chapter there is no real end to this simulation phase. Therefore the end of this phase is defined as the point at which Adam beats Bodo and stops moving forever.

6.2.1 Simulation run with original memory

The first step in test case 1 is to run the simulation of use case UC1 - Standard scenario (Eat) minimalversion in order to collect data for comparing the newly implemented features to the former implementation. Table 6-3 shows the time each of the formerly defined phases needed to finish. As can be seen the overall simulation run takes about 2 minutes and 5 seconds.

Phase	Time
Start simulation	1s 43ms
Scenario initialization	20s
Scenario run	1m 44s

Table 6-3: Duration of simulation phases with original memory

In Table 6-1 the agent´s actions during the simulation run phase are listed. They will be used as a reference to check whether the new implementation works correctly, meaning that the agents performs the same actions at the same step. In the table the term “area” will be used in order to refer to the blue

sensor range area around the agent. During the simulation this area changes its angle to indicate the direction to which the agent is currently looking or moving. To describe the actual action symbol which is shown next to the agent the term symbol will be used. In Figure 6.2 the terminology for the further simulation description is depicted. In Table 6-4 the movement of the area are tracked) as well as a change of symbol is tracked.

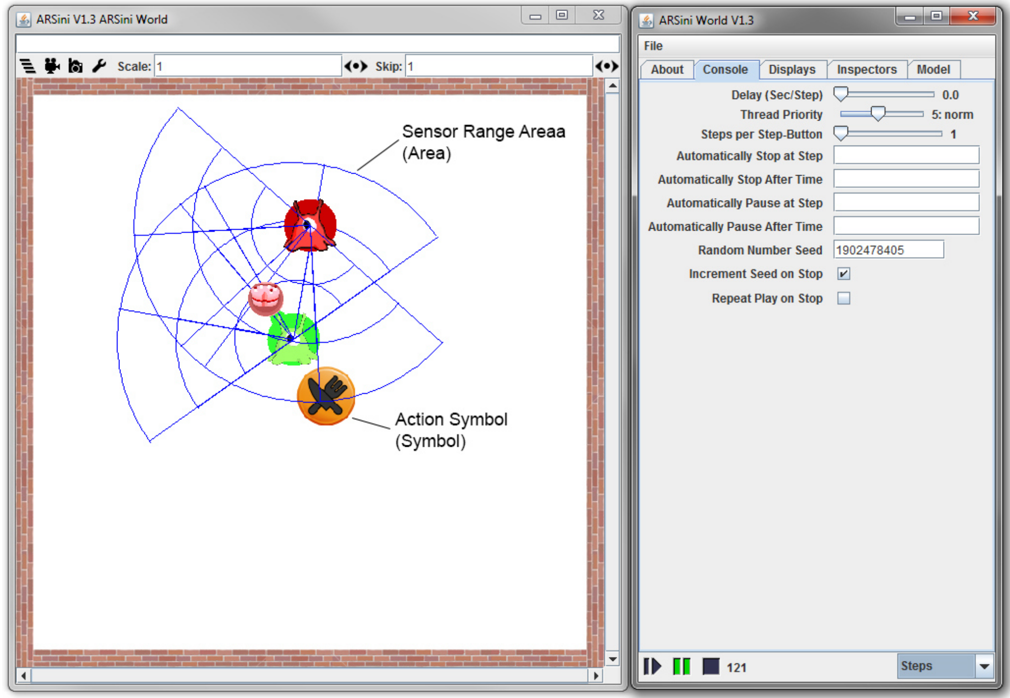


Figure 6.2: Terminology for simulation run description

In Table 6-4 every change of action symbol or movement of the sensor range area is presented. For a TURN_LEFT of the area the short abbreviation TLA will be used for the TURN_RIGHT of an area TRA will be written.

TLA	TLS	TRA	TRS	MFS	MFA	EAT	BEAT
10, 20, 30, 40, 50, 90, 100, 250, 260, 270, 280, 260, 270, 280, 320, 330, 370, 380, 420, 430, 470, 480, 520, 530, 570, 580, 620, 630, 670, 680	11-60, 91-109, 251-290, 321-340, 371-390, 431-440, 471-490, 521-540, 571-590, 620-640, 671-690	230, 240, 250, 290, 300, 310, 340, 350, 360, 390, 391, 400, 410, 440, 450, 460, 490, 491, 500, 510, 540, 550, 560, 590, 600, 610, 640, 650, 660, 690, 700, 710, 720, 730, 740, 750, 760, 770, 780, 790, 800, 810, 820, 830, 840, 850	231-250, 291-320, 341-370, 391-430, 441-470, 491-520, 541-570, 591-610, 641-670, 691-850, 881-890, 911-930	61-90, 110-120, 861-880, 891-910, 931-960	62-89	121-230	961

Table 6-4: Simulation run with original memory – sensor area movement and action symbol changes

To indicate a change of symbol to TURN_LEFT and TURN_RIGHT the abbreviations TLS and TRS will be used. The action EAT will be written down as it is, whilst MOVE_FORWARD will be MFA and MFS for the area respectively the symbol. The numbers in the table mark the respective step in which a symbol is changed or an area movement takes place.

6.2.2 Simulation run with RDF database

After running the simulation with the original memory the simulation settings were changed to use the new RDF database. In order to run the simulation the RDF database first has to be filled with data from the original memory. Therefore the first step before running the simulation is to fill the RDF database with information.

As mentioned in chapter 4.3 the class `RDFSearchSpaceCreator` is utilized in order to convert the original data into a triple representation. The conversion of the knowledge base for UC1 - Standard scenario (Eat) minimalversion itself takes 7 minutes and 53 seconds, saving the constructed triples to the database takes another 37 seconds. Converting the original file-based memory into the Sesame RDF triple store results in 3017298 triples. As was already mentioned in chapter 4.3 it was necessary to fully transform the data provided by the original memory parsing method from SiMA. This approach results in a total of 229537 distinct data structures. Whereas 37566 of them are drive-meshes, 13295 thing-presentation-meshes, 168 word-presentation-meshes, 14277 thing-presentations, 156 word-presentation, 14 affects, 56 emotions, 4 feelings and 164001 are of the data structure type association. After the database is filled with the needed information, the simulation test can be started.

In Table 6-5 the amount of time that is needed to complete the three simulation phases for the use case UC1 - Standard scenario (Eat) minimalversion with the new RDF database is presented. While the simulation start phase is, due to its short duration, not particularly slower, the overall duration of the simulation is about 13 times slower than the original memory implementation.

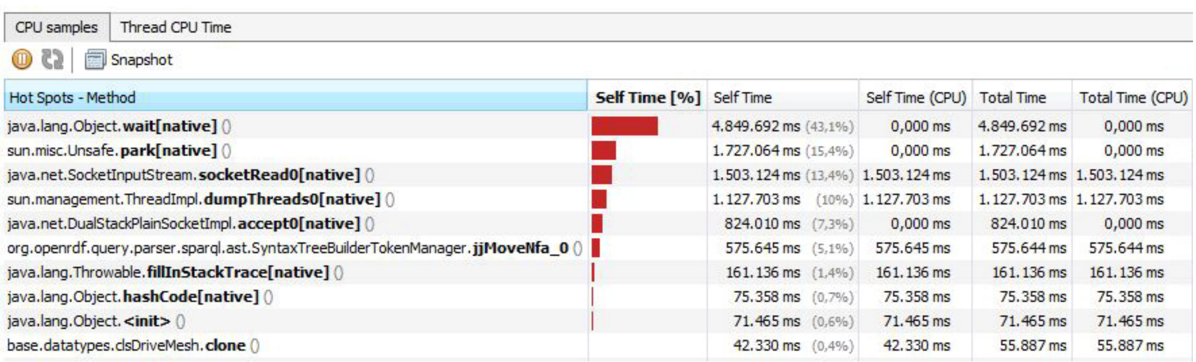
Phase	Time
Start simulation	1s 51ms
Scenario initialization	5m 30s
Scenario run	21m 57s

Table 6-5: Duration of simulation phases with RDF memory

In order to track down the main cause of the performance problems the third party tool VisualVM [VISUAL] was utilized. VisualVM was created for analysing Java applications for analysing application performance, memory consumption, garbage collection and monitor application threads. In Figure 6.3 the profiling of a simulation run is shown. Noteworthy are the two methods `java.net.SocketInputStream.socketRead` with 13.4% self-time and `org.openrdf.query.parser.sparql.ast.SyntaxTreeBuilderTokenManager.jjMoveNfa` with 5.1% self-time. A socket is a communication endpoint between two machines [SOCK] and is used by the RDF

framework to communicate with the database. The method call on the `SyntaxTreeBuilderTokenManager`, which takes the sixth place in the CPU usage ranking and is part of the Sesame Java framework, is another point of interest when tracking down the performance problems.

The analysis results from VisualVM indicate that either the amount of accesses to the database is too much or the sesame database access methods are implemented in a rather inefficient way. Adding a counter to the database access methods resulted in approximately `13000000` database accesses during the simulation run. This would be nearly 600000 calls per minute and could be therefore easily reduced by having some caching mechanisms implemented, as in such a short time no drastic changes to the memory can happen. The large number of accesses can be explained by the fact that all existing data structures of a searched type including their associations are loaded and rated during the search process. This means that originating from its root node every further object that is linked to the root element via its associations has to be traversed and loaded separately, again loading all of its associations.



Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
java.lang.Object.**wait[native]** ()		4.849.692 ms (43,1%)	0,000 ms	4.849.692 ms	0,000 ms
sun.misc.Unsafe.**park[native]** ()		1.727.064 ms (15,4%)	0,000 ms	1.727.064 ms	0,000 ms
java.net.SocketInputStream.**socketRead0[native]** ()		1.503.124 ms (13,4%)	1.503.124 ms	1.503.124 ms	1.503.124 ms
sun.management.ThreadImpl.**dumpThreads0[native]** ()		1.127.703 ms (10%)	1.127.703 ms	1.127.703 ms	1.127.703 ms
java.net.DualStackPlainSocketImpl.**accept0[native]** ()		824.010 ms (7,3%)	0,000 ms	824.010 ms	0,000 ms
org.openrdf.query.parser.sparql.ast.SyntaxTreeBuilderTokenManager.**jjMoveNfa_0** ()		575.645 ms (5,1%)	575.645 ms	575.644 ms	575.644 ms
java.lang.Throwable.**fillInStackTrace[native]** ()		161.136 ms (1,4%)	161.136 ms	161.136 ms	161.136 ms
java.lang.Object.**hashCode[native]** ()		75.358 ms (0,7%)	75.358 ms	75.358 ms	75.358 ms
java.lang.Object.**<init>** ()		71.465 ms (0,6%)	71.465 ms	71.465 ms	71.465 ms
base.datatypes.clsDriveMesh.**clone** ()		42.330 ms (0,4%)	42.330 ms	55.887 ms	55.887 ms

Figure 6.3: CPU usage analysis in VisualVM

In order to find more detailed information about the cause of the performance decrease several possible causes have to be tested in more depth. First of all it has to be tested how much performance can be gained by finding a more efficient way to query the database. Not all queries are equally costly so some research on the exact cost of the individual queries may lead to a great performance gain. Considering the huge amount of database accesses, even slight improvements could have a great impact on the overall performance. Furthermore the implementation of partial loading in order to load only the necessary parts of an object in the search process would greatly improve performance. Another point for improvement could be to solve the problems causing current data overhead. As discussed in chapter 4.3 the original data was converted without optimizations and the database therefore overloaded. It is possible that the database improves on performance if the amount of triples that have to be searched through can be reduced. If that is the case it is maybe possible to split up into several database files for the different memory types (see chapter 2.3). However, for real long time runs this is probably still no satisfying solution. The last possibility is that the RDF framework itself is too slow to be used in such an excessive way. In that case the only solution would be to reduce the amount of database accesses by caching or other strategies.

6.2.3 Results

In this section the final results of comparing the new RDF database implementation to the original approach, which used to parse a file into an in-memory HashMap are presented. Table 6-6 shows the performance decrease between the two simulation runs. As can be seen from comparing the performance of the original implementation in Table 6-3 to the performance of the new implementation using an RDF database, the new implementation causes a noticeable slowdown of the simulation performance.

Phase	Performance Decrease [%]	Performance Decrease [time]	Times slower
Start simulation	0.7 %	10 ms	1.007
Scenario initialization	1550%	5 min 10 sec	16.5
Scenario run	1166%	20 min 13 sec	12.663

Table 6-6: Performance Decrease between the memory implementations

Due to the fact that the start simulation phase slowed down by some milliseconds only, it may be considered to be negligible, as such a difference may result from minor measurement inaccuracies as well. However the performance of the new system in the other two phases shows that many improvements will be necessary until the new system can be seriously used. The scenario initialization phase is now 4 minutes and 10 seconds slower than before, which means a decrease of performance by 1550%. For the scenario run phase it is a decrease of 1166% or 20 minutes and 13 seconds and the overall simulation run is now around 13 times slower than it used to be.

Test Case 1 shows that the new approach is fulfilling objective 1 and 2 (see chapter 6.1). However the outcome of the first test with an RDF database leaves space for improvement. As pointed out in the previous sub-chapter accessing the RDF database seems to be very costly when it comes to CPU usage. It also points out the need for further tests in order to give an informed statement about the usability of RDF for SiMA. Some possible approaches were pointed out in chapter 6.2.2.

6.3 Test Case 2 – Episodic Memory

The following chapter presents the results of the memorization process which was developed during this work. As discussed in chapter 5.3 only the experience saving and the functionality to load it was implemented. Until now it is not possible for the agent to use the experiences he has gained. The reason for this is that too many functionalities of the agent are affected by such a change and would be by far out of scope for this work. In order to test and analyse the results a graph visualization was implemented to the ARSDatabase project. For the act saving simulation again UC1 - Standard scenario (Eat) minimalversion, which is described in chapter 6.1, will be used.

6.3.1 Simulation run with active memorization process

The first step in test case 2 is to run the simulation scenario with an active memorization process. Every time the `clsAC_EXECUTE_EXTERNAL_ACTION` codelet sends a perceived image to the memorization unit this image is saved to the database and marked with the triple `s=<IRI> p=http://ars.org/isBufferObject o="true"`. After saving the perceived image its IRI is stored in a buffer. After storing the image to the database and the buffer, it is checked whether the current goal was reached during the perception of this image. Information about the goal reached state is provided by the codelet `clsCC_END_OF_ACT`. If this is the case, the act construction is started, by constructing first the basic act and then filtering through all the perceived images. The remaining images are then put in occurrence order by connecting them with associations. Finally, they are attached to the act by further associations and then the act and the new associations are saved to the database. In a final step the triples marking the images as buffer objects are removed from the database. After the marker has been removed it is possible to check whether both objectives of test case 2 have been fulfilled.

```
[s=http://ars.org/ACT_3433276;p=http://ars.org/isBasicObject;o="true"]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasMoContentType;o="ACT"]
[s=http://ars.org/ACT_3433276;p=http://www.w3.org/1999/02/22-rdf-syntax-ns#type;o="ACT"]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasDebugInfo;o="ACT"]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasMoDS_ID;o="1055"]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasMoDSInstance_ID;o="0"]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasMoContent;o="ACT"]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasStrAction;o="EAT"]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasAssociatedContent;
   o=http://ars.org/ENVIRONMENTALIMAGE_1000_0_1]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasAssociatedContent;
   o=http://ars.org/ENVIRONMENTALIMAGE_1001_0_2]
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasAssociatedContent;
   o=http://ars.org/ENVIRONMENTALIMAGE_1002_0_3]
```

Figure 6.4: Act representation in the RDF database

As it is currently not possible to prove that the memory can be used by SiMA, only the correct saving can be validated. For this reason all the triples that are generated by the memorization process are stored in a separate test database. By doing so it is guaranteed that all analysed data was produced by the process only. After running the use case with an active memorization process this separate test database contains one act consisting of 54 images. The whole act construct results in 13858 triples when converting and saving it to the RDF database. Saving the act to the RDF database results in a total of 1005 distinct new data structures, whereas 34 of them are drive-meshes, 30 thing-presentation-meshes, 74 word-presentation-meshes, 47 thing-presentations, 59 word-presentations, 6 emotions, 755 are of the data structure type association and finally there is the one act that forms the root node for all of those data structures.

```
[s=http://ars.org/ACT_3433276;p=http://ars.org/hasAssociatedContent;
   o=http://ars.org/ENVIRONMENTALIMAGE_1000_0_1]
[s=http://ars.org/ENVIRONMENTALIMAGE_1000_0_1;
   p=http://ars.org/hasAssociation;o=http://ars.org/ASSOCIATIONSEC_3434189]
[s=http://ars.org/ENVIRONMENTALIMAGE_1000_0_1;
   p=http://ars.org/InternalAssociation;o=http://ars.org/ASSOCIATIONSEC_3434189]
[s=http://ars.org/ENVIRONMENTALIMAGE_1000_0_1;
   p=http://ars.org/hasAssociation;o=http://ars.org/ASSOCIATIONSEC_3434190]
[s=http://ars.org/ENVIRONMENTALIMAGE_1000_0_1;
   p=http://ars.org/InternalAssociation;o=http://ars.org/ASSOCIATIONSEC_3434190]
[s=http://ars.org/ASSOCIATIONSEC_3434189;p=http://ars.org/PREDICATE;o="HASNEXT"]
[s=http://ars.org/ASSOCIATIONSEC_3434190;p=http://ars.org/PREDICATE;o="HASSUPER"]

[s=http://ars.org/ACT_3433276;p=http://ars.org/hasAssociatedContent;
   o=http://ars.org/ENVIRONMENTALIMAGE_1000_0_2]
[s=http://ars.org/ENVIRONMENTALIMAGE_1000_0_2;
   p=http://ars.org/hasAssociation;o=http://ars.org/ASSOCIATIONSEC_3434191]
[s=http://ars.org/ENVIRONMENTALIMAGE_1000_0_2;
   p=http://ars.org/InternalAssociation;o=http://ars.org/ASSOCIATIONSEC_3434191]
[s=http://ars.org/ENVIRONMENTALIMAGE_1000_0_2;
   p=http://ars.org/hasAssociation;o=http://ars.org/ASSOCIATIONSEC_3434192]
[s=http://ars.org/ENVIRONMENTALIMAGE_1000_0_2;
   p=http://ars.org/InternalAssociation;o=http://ars.org/ASSOCIATIONSEC_3434192]
[s=http://ars.org/ASSOCIATIONSEC_3434191;p=http://ars.org/PREDICATE;o="HASNEXT"]
[s=http://ars.org/ASSOCIATIONSEC_3434192;p=http://ars.org/PREDICATE;o="HASSUPER"]
```

Figure 6.5: Image links to act in the RDF database

In this work a basic filtering process was considered and implemented (see chapter 5.3). As was explained in that chapter the current filter process only includes the removal of duplicate images. However the current implementation of SiMA does not need such a filtering process for the memory as it already provides only those perceived images that are different to their predecessor. The current filtering process is therefore unnecessary for the moment, but the existing method could still be extended by one of the filtering approaches that were suggested in chapter 5.2.2.

In order to validate the correct linking between act and the images as well as the triples containing information about the act, all contents of the test database are then parsed to a human readable file. Figure 6.4 shows the triples that represent the act and its link to the images. As can be seen the act has its attributes set conforming to the intended values which were discussed in chapter 5.2.3. Also the links to its associated contents are defined correctly. Also the links to its associated contents (marked by the predicate hasAssociatedContent) are defined correctly, which can be seen exemplary in the last 4 lines of Figure 6.4. All other data structures, like for example the word-presentation-meshes, which are used to represent the images, have been used in test case 1. Thus, it has already been demonstrated in chapter 6.2 that they can be stored and used as expected. In Figure 6.5 one can see by example that all images have internal associations specifying their parent act and their successor image, as it was

defined in preliminary work on acts fore SiMA [Zei10, pp. 58–59]. Therefore, objective 3 from chapter can be considered to be satisfied as far as it is possible to test this requirement in the current implementation.

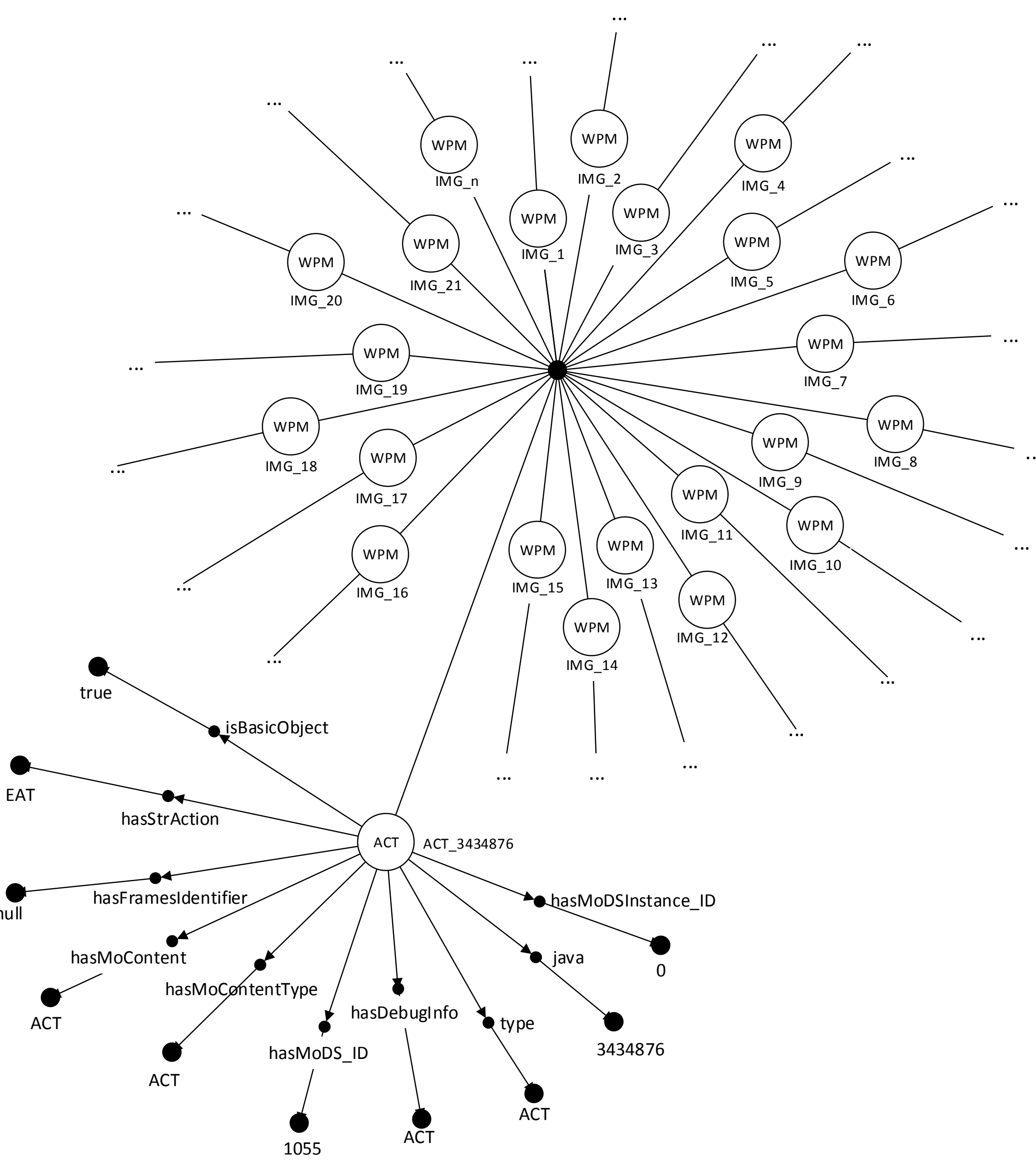


Figure 6.6: InfoVis graph presenting the act and its images

As mentioned in the beginning of this chapter a graph visualization package was introduced into the ArsDatabase project in order to be able to analyse the saved memory data. This package provides functions to produce JSON (JavaScript Object Notation) files out of the SiMA data structures or alternatively out of RDF query results. JSON is a data-interchange format which is easy to read for humans and machines [JSON]. For displaying the JSON file as a graph the JavaScript InfoVis Toolkit

[VISU] was used. It is a JavaScript based graph visualization toolkit providing several different graph visualizations. For the usage with the SiMA data the “ForceDirected” graph type was chosen. It supports edge directions, which are used to represent the association directions. Additionally, it is possible to customize the node appearance to represent the different data structure types. Thus providing a sufficient customizability and an easy producible input format, making the InfoVis library an obvious choice for the purpose of presenting and validating the results of this work. Figure 6.6 shows the graph resulting from the memorized act with a node expansion depth of 1. The node expansion depth can be used to restrict the graph depth. In the implemented JSON generation it is possible to pass certain IRIs, which will be drawn in the graph even though they exceed the provided graph depth. This is a useful option if one is only interested in specific information. If due to the graph depth restriction not all information is presented nodes with the label “…” are drawn to indicate that there exist more nodes than actually shown.

In order to show that the memory contents can be accessed by the simulation, the newly implemented database access methods are utilized to fetch contents from the memory. They are then passed to the visualization toolkit in JSON format to make the results visible for this analysis. For the sake of readability the image labels in the graph do not show their database IRI, the prefix “IMG” and their occurrence number is used instead. As can be seen in the graph the final act is linked to its 54 images and each of them is of the data structure type word-presentation-mesh (WPM). It can also be seen that the act´s attributes, like the `strAction`, the `moContent` and its `type` are loaded correctly from the database into the Java application. Figure 6.7 shows `IMG_01`, which is the first perceived image during the simulation run further expanded. Every image has two internal associations of type `AssociationSecondary` attached, one of them representing the connection to the act, the other one linking the image to its successor. As discussed in chapter 5 they are marked with the predicates `HASSUPER` and `HASNEXT`, which can be seen in Figure 6.7 as well. This figure also depicts how the data structures are linked to each other. `ACT_3433276` is linked to `IMG_1` by having it defined as its associated content and the image itself is linked to the act by its `AssociationSecondary` declaring the act to be its parent data structure. Again the methods which were already validated in test case 1 are utilized to load the remaining data structures. Therefore, it is implied that the whole act structure can be loaded correctly and objective 4 is fulfilled.

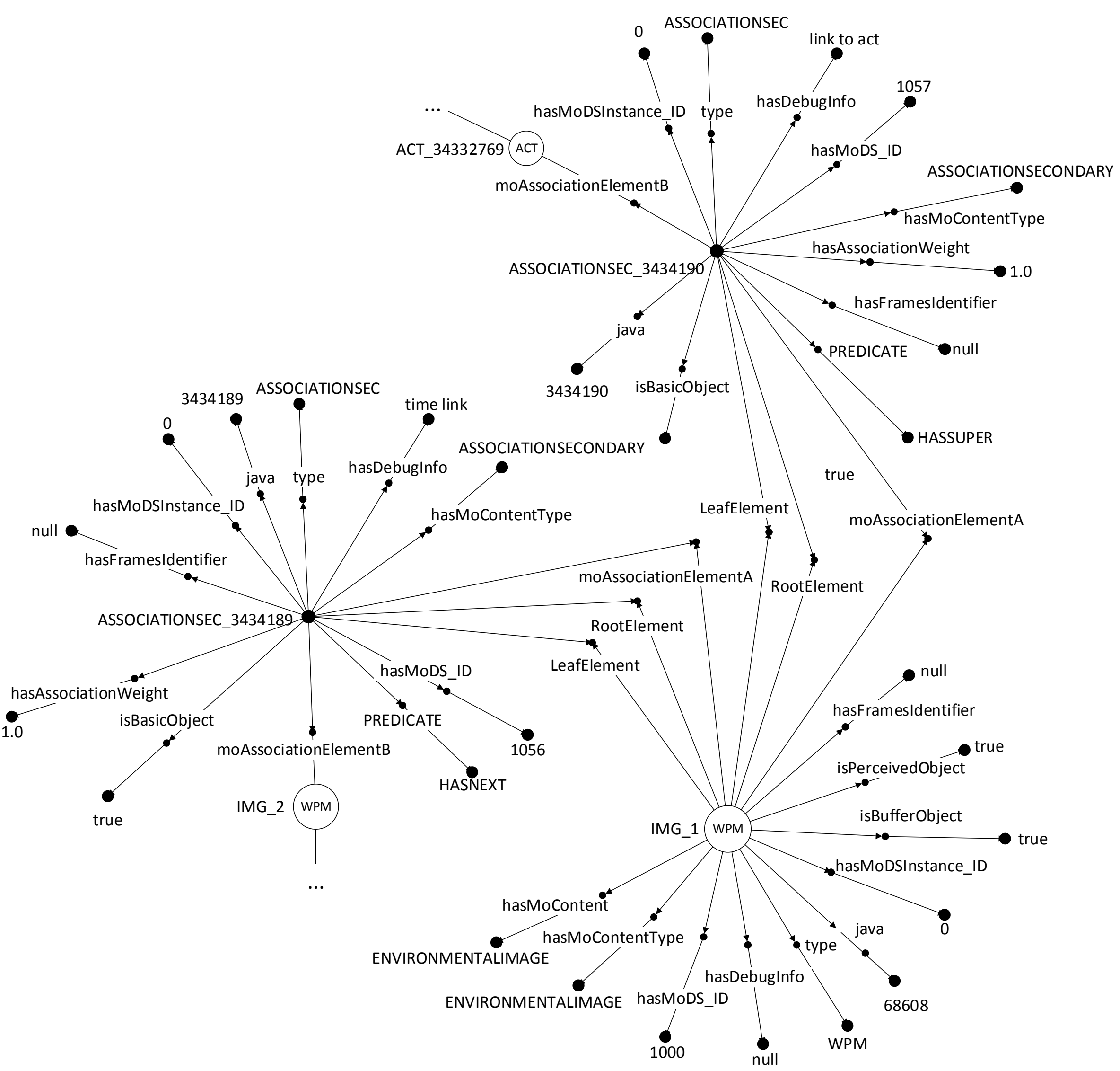


Figure 6.7: InfoVis graph presenting the image and its act related associations

6.3.2 Results

This section gives a short summary of the results of the simulation runs with the first memorization process that was ever implemented to SiMA. Focus of this chapter are the fulfilment of the objectives 3 and 4 which were defined in chapter 6.1.

The third objective was to save an episode consisting of several images to the database. This objective could be tested by querying the database contents after a finished simulation run, showing that the act and all images with the according links were written to the database (see chapter 6.2.2 for more details)

Finally, the fourth objective was to show that it is possible for SiMA to access and load the saved data again. To show this the act was loaded by the newly implemented database access methods and then parsed into a JSON file in order to prove that the data can be accessed by the provided methods.

Concluding it can be said that all requirements for test case 2 have been fulfilled. However, it is necessary to integrate the episodic memory to the SiMA decision process to make further statements about the practicability of the current implementation.

7. Conclusion and Future Work

In this work the first steps towards a proactively experience gaining autonomous agent have been taken. The file-based knowledge storage has been replaced by an RDF triple store, which enables the SiMA system and consequently any agent to permanently store information. As a further step the old knowledge store was migrated in order to preserve the already existing information and thereby the agents ability to react to its environment and thereby gain new experiences. Based on this, the first process for memorizing experiences has been implemented into SiMA. In the following the results of this work will be discussed, followed by an outlook about future tasks.

7.1 Conclusion

In chapter 1.3 the main research question for this work was formulated as: *"Which features need to be implemented, in order to enable SiMA to collect experiences?"* and in the course of this work first answers to this question and the resulting sub-question were found. In the following the findings of this work shall be discussed and supplemented with first suggestions on how to further improve the implemented solutions.

The first task that came up with the implementation of a long-term memory in SiMA was the replacement of the old data store solution with a new one, which is able to permanently store newly collected information. Replacing one knowledge store by another, usually brings up many questions. For example the necessity of a change, which was in this special case out of question as the previous solution simply lacked the functionality of a permanent store. A more complex question was the substitute itself, which was formulated within research question 4: *"Which of the existing database paradigms are most suited for SiMA?"*. In chapter 2.2 the range of options for this decision were presented and discussed and in chapter 3 an RDF triple store solution was selected. The final decision was based on preliminary work and some additional research work. After the first implementation of such an approach a first summary about the chosen solution is possible.

The decision for an RDF triple store introduced many interesting options to SiMA, but also some intermediate disadvantages and new challenges. One of the reasons to choose an RDF database was the option to possibly migrate the data easily with an automatic convert and maybe some minor adjustments. As pointed out in chapter 4.1 the data resulting from the automatic conversion was not exactly fit to be used in SiMA and was therefore dropped, meaning that research question *5 ("How shall the current knowledge base be migrated?"*) was still to be answered. The final solution in this

work was to make use of the conversion methods, that had to be implemented for the memorization process and combine them with the parsing process that was used by SiMA at program start. However the resulting conversion is not flawless, which is partly due to the general complexity of the structure and partly due to the unmaintained state of the file and also the SiMA code in general. The recursions in combination with the cloning problem (see chapter 4.3.1) made it rather complicated to make out, which data structures have to be put into the long-term memory and which are only to be used inside the execution run. For the future, it would be recommendable to refactor the knowledge base as well as the code using it. At this point, however the new permanent store brings a possible improvement to the system, as for the future cloning will not be necessary anymore. The data structures fetched by the search procedures already represent new instances (in Java terms) and therefore changing them will not alter the memory contents. In case that the data structure has to be saved to the memory as a new instance only the IRI has to be changed.

As expected (see chapter 1) the current integration of an RDF database into SiMA brought no immediate improvement for the project. Except for the ability to permanently store data and an option to finally solve the cloning problem, which consumes a lot of processing time in the current implementation, in its current utilization the chosen solution even seems to be a downgrade for the system. As discussed in chapter 6.1 the simulation runs takes 1166% longer than in the previous set-up with an in-memory search space implemented through a HashMap structure.

As already mentioned in chapter 1.2 some of the reasons for the change, were to prepare the system to gather much more data than it is actually the case, therefore tests with long runs and data gathering have to be started in order to make an informed decision whether to use RDF in the future. Another fact that has to be kept in mind is that actually the data structures and their utilization in SiMA are not optimal for the chosen solution. First of all, the fetched data should be reduced to what is actually needed, (for example only the basic thing-presentation-mesh instead of all its associations, including their associated elements and all information that is attached to them. Secondly, it should be also considered whether the data structures as they are currently set-up should be used. Maybe it would be possible to have the data structures only mapped in an RDF graph and not to use the Java implementation of those data structures at all.

If no changes to the data structure management in the Java implementation seem to be considerable, it is maybe better to change to a more general approach like a relational database. The data structure as it is currently defined is not requiring any functionality regarding unstructured data. However, this would have been the main argument for the usage of a NoSQL solution. As discussed in chapter 2.2, a relational database system will be much likely faster and easier to query. Therefore, if only predefined data structures are in use, the RDF triple store solution should be changed to a relational solution.

As the current memorization process is only a first proof of concept, many improvements are still possible. First of all the filtering process is far from optimal, as only the most basic considerations are taken into account. Therefore a fully comprehensive answer to research question *2: "How shall the agent decide which memories are to be kept?"* could not be found within this work. Due to a lack of complicated use cases even the basic filtering (see chapter 5.3) turned out to be unnecessary as no two images are equal in the current use cases. In order to stay true to the concepts used in SiMA, it would

be a necessary step to first work out a technically feasible concept of psychoanalytical theories for the memorization process and realize the results in SiMA.

After applying the basic filtering process, the perceived data is collected until the agent reaches its goal. This is the moment that was chosen to construct an act and at the same time answer to research question 3a: *"When shall the act generation be triggered?"*. During the simulation runs and also in retrospection this seems to be a suitable solution for SiMA, as it works perfectly together with the technical concepts like codelets, as well as with the general concepts like goals and emotional motivation. The construction of an act is mainly based on the usage of secondary associations and was thoroughly explained in chapter 5.3. For now it seems to be a reasonable approach to answer research question 3b: *"How shall the act construction be implemented?"*, but it is highly possible that adjustments to the concepts of an act and also its construction will be necessary when their usage is finally implemented in SiMA.

In summary, this work was an important first step towards an agent able to memorize and recall experiences. The newly implemented RDF triple store opens up many new options for the future of SiMA. The flexibility of the database system allows future developers to store and retrieve the model or structure they need without the necessity to apply any changes to the database itself. The implemented experience gathering is a first proof of concept for the implemented models and data store, however the usage of experiences and also more effective experience filtering are still to be implemented. This work has pointed out several optimization requirements, beginning with the data structure usage in general to more sophisticated act generation and also the missing act usage. The next chapter will therefore discuss some options for future development that may offer solutions to the appealed problems. Additionally, some general optimizations and further steps are discussed.

7.2 Future Work

The discussion in chapter 7.1 showed that the concluding results of this work not only gave some much needed answers to fundamental questions, but they also brought up many new questions and tasks. The concluding chapter of this work will therefore cover the newly arisen challenges and give an outlook on the work that is still to be done.

Caching

In the previous chapter the necessity to rethink the data loading approach, in order to improve the performance of the overall system was discussed. As mentioned in chapter 2.2, database solutions have different specializations, and provide advantages and disadvantages depending on the data that has to be stored. From that follows that the replacement of one database technology by another, leads to a loss of the previously enjoyed advantages. However, for the situation in SiMA it is not necessarily an absolute truth. In order to speed up the new search space solution, a possible approach could be to complement the system with a *cache* for long-term memory elements. This would reuse the idea of an in-memory store and all its advantages, by using both a permanent and nearly infinite storage, as well as an additional store that makes certain data available on a speed aspect.

As of 2015 caching is a state of the art technique to reduce traffic and ultimately increase speed and performance in various systems with a small sacrifice of data accuracy, depending on the settings of the cache [TG01, pp. 95-97]. Caches are commonly used in web environments by web browsers to avoid loading the same images and texts over and over and therefore reduce internet bandwidth. They are also used on the server side, the webservers, to be able to serve websites faster, without the need of generating a web resource from scratch for every single http call, which would take, more central processing unit (CPU) processing time. Here an analogy between a CPU and the SiMA memory can be found. The CPU that can in many ways be seen as the „brain“ of a computer, is also confronted with the problem of permanently loading data from a storage system that responds in a way slower fashion and would be significantly slowed down by it if it would load every single instruction set, or „word“. To avoid such problems modern CPUs use highly sophisticated methods to decide whether data is worthy to be loaded or kept to the, in comparison relatively small CPU cache. One that should be mentioned is the „locality principle“, which says that data stored in neighbouring locations in the main memory is very likely related to the currently needed data and might be needed for execution in the near future [TG01, p. 96, p. 320]. CPUs even utilize different layers of caches to handle various levels of data forms and importance, which creates a hierarchy that could be drawn as a pyramid, with the fastest and smallest cache that is closest to the CPU at the very top (e.g. CPU registers) and the slowest but largest form of data storage (e.g. hard drives) at the bottom [TG01, p. 99].

While there is certainly a lot more depth to caching at the CPU level, this was just a very short excursion to emphasize the analogy between the SiMA system and the CPU, both functioning as the brain in their environment with high speed demands in a data heavy environment. Thus it might be possible to take some of the in modern CPU implemented solutions as a proof of concept that makes it worthwhile to take a look in whether and how caching would benefit the current triple based memory of SiMA. If, like previously suggested, it is possible to reduce the memory loading to get only needed parts of the data structure, the implementation of an “intelligent” cache would reduce the resulting increase of memory accesses, thereby contributing to even better performance. Intelligence in this context means that it has to fulfill various requirements depending on the data at hand. First of all, just keeping everything in the cache would be obviously counterintuitive, because it would slowly migrate the system back to its old state, where it only had an in-memory database, with all the downsides that have been pointed out in chapter 1.

To avoid a cache blow-up, it needs to know which data is of importance and might be needed in the near future and which has not been used for an amount of time and is therefore probably not needed any longer. Taking this even further an implementation of the mentioned *locality principle*, could be even more beneficial for such a system. Even though the functioning of SiMA certainly differs from the principle that a CPU uses, where it is assumed that neighbouring memory addresses are probably related, while SiMA does not work on the memory address level, which is already managed for it by the CPU, the key aspect of the mentioned approach lies in „neighbouring“. The used RDF triple store is essentially nothing else than a web formed by associations with neighbouring entities. This similarity makes it highly likely that while computing an act, neighbouring images will be needed at a very high likelihood in the near future. So when a call is set to the SiMA memory to load a specific image of an act, we can only load that specific image and while it is processed, the cache now might decide to make calls in parallel to retrieve images that follow the currently processed one and store them to

its memory for eventual following search calls. Thus, when the agent decides it needs one of these images, it has already been loaded and can immediately be served. Later on the cache might decide that it is time to drop those images from its memory again to make room for other entities. To make informed decisions about which information is to be dropped, the system could use timestamps and simply remove data after some time, or maybe an activation counter should be implemented. Every time a data structure has been used, the counter is increased, thereby marking that the data structure is possibly more important than a more recent one that was only needed once. These are just examples of how such a caching system might benefit the SiMA system in the long run and what would make the implemented migration from a full in-memory store to a triple store a valuable and much needed approach.

Threading

In order to make the caching approach work as a performance improvement, another concept which is also very important in modern computer science will be needed. Currently the SiMA memory only runs as a single process on the CPU, and all its tasks are worked up in a sequential manner, meaning that only after one task has finished its calculations the next task can be started. As the system´s memory is further developed, its behavior is getting more sophisticated but also more specialized. This results in many different tasks, which are not all connected by sequential dependencies and could therefore be processed in parallel. Task splitting makes use of so called *threads* that are defined on the programming level. A thread is a small process within another bigger process, which performs a specific self-contained task. A basic computer program starts at its entry point and is executed line by line till its end. If threads are used, it is possible to outsource some of the main programs work into threads. Depending on the execution system these threads can be worked off quasi parallel or really parallel, if multiple cores are available on a system. The term “quasi parallel” means that all processes are striped into parts and execution time is divided between the processes, each one getting short instances of processing time before they get swapped out and replaced by another process and vice versa. While at a first glance this seems like no change to just executing processes in sequential order, in reality processes often have to wait for external resources such as memory contents or user input, and during such times it is ideal to remove them from the processing queue and make room for more urgent calculations, that are currently not blocked by external resources [TG01, pp. 489-490]. Splitting tasks to work in parallel would be not only a necessity for the implementation of a sophisticated caching solution as described above, it could also benefit the memorization process. During the discussion about caches and the benefits they could bring to SiMA the use of a thread was already implicitly talked of. In fact, without threads it would not be possible for a cache to fetch data in advance while other data is processed. In the case of a cache the threading procedure, as it was already described, is rather clear: Every time a data structure is fetched and processed, the cache starts a parallel thread that searches for data structures which are connected to the one that is currently processed. For the usage of threading in the memorization process some further considerations are necessary. In principle, one can assume that the system should not have to wait for the memorization process to store its experiences, in order to resume with taking actions. However, if the search as well as the memorization process have to access the memory at the same time, some sort of conflict management will be needed, in order to prevent so called *race conditions*. The term race condition describes if the success of an

operation is dependent on which thread finished its task first. The fact that threads usually take a variable amount of time to complete makes it nearly impossible to determine if the state of the system will be correct at all times, and inconsistencies might occur [TG01, pp. 494-495]. For example, if the data retrieval task in SiMA starts to collect a mesh data structure from its memory while the memorization task starts to change the associations of this mesh while they are still loading, an inconsistency occurs. In order to prevent inconsistencies Java (and also other programming languages) offers several tools, like the usages of semaphores. Semaphores are basically non negative integer values that are used by different threads to take and release program resources or variables for exclusive use only. If a thread is in need of an already taken resource, which is marked by such a semaphore, it simply waits until the resource is free again. [TG01, p. 495]. Java offers this kind of synchronisation for Java functions via the synchronized keyword that will handle exclusive access on static methods.

Memorization Algorithms

Chapter 5.2 covered several possible options for the implementation of memorization algorithms in SiMA. However, only a basic memorization trigger and filtering process were implemented. At the moment memories are only stored if the agent reaches its goal and the filtering is only able to recognize duplicate images occurring after each other. Future implementations should therefore consider the possible effects of emotions on memorizing experiences. The first possible application of emotions to the memory is to use experienced high positive or negative emotions as a memorization trigger as was described in chapter 5.2.1. Also the filtering process should be further improved in order to avoid that the memory is going to exceed the calculating limits of the simulation environment and hardware. This leads to the second application of emotions as they could be used as a relevance indicator for perceived images and less relevant images could be filtered out during the filtering process (see chapter 5.2.2). By doing so the quality of the memorized episodes will very likely improve.

In addition to the topics which were covered in chapter 5.2 cleanup be considered to implement a memory clean-up, in order to avoid that the memory is going to exceed the calculating limits of the simulation environment and hardware. Spoken in a less technical view, it is probably necessary to emulate the human's ability to reprocess his memory during sleep [GB04, p. 679]. On this topic much research has been done, but as for the psychoanalytical theories that form the base for SiMA´s decision process, it is not the task of a software engineer alone to design a technically feasible, but psychoanalytically correct model of this process. As SiMA is based on psychoanalytical theories it would be again advisable to cooperate with psychoanalysts, in order to find a psychoanalytically based model for the memorization process.

Learning

As was already appealed in chapter 1 learning is absolutely obligatory for agents that shall be able to autonomously react in changing environments. The first requirement, the ability to store outcomes of actions and situations, for such a learning agent has been satisfied within this work. For the future, it is therefore to consider the implementation of a learning procedure in SiMA. Beginning with the first procedural languages, like LISP or PROLOG machine learning became an important research topic in computer science [Lor14, p. 433]. One approach that became very popular for agent systems is the so called *reinforcement learning*. It is based on the idea of learning things by being rewarded or punished

for doing them. This approach is well suited for agent systems as they usually include some sort of evaluation whether the current actions brought a positive or negative result, which can be utilized to initiate a learning process. In [DFZB09, p. 84] reinforcement learning was already considered to be a valid approach for an emotional driven agent like it is the case in SiMA, as emotions can be easily used as an evaluation factor for the success of a situation. Emotion-related learning is even considered to be more flexible than common stimulus-response models as emotional learning is a two-step learning process and also emotions can trigger several processes at the same time [DFZB09, p. 84]. The first step of emotional learning is to react to a stimulus in a certain way. In a second step the agent may learn to behave in a certain manner in order to strengthen or evade the produced emotion. The emotional abilities of SiMA agents were already discussed in chapter 5.2.1 and even though it was decided not to use them for a memorization trigger, they seem to be a good foundation for a learning agent. It may be important to notice that the fact that they are not used as a memorization trigger does not influence the usage of emotions for learning. It is for example possible to check for the emotions attached to an image in an act independent of the cause it was created. As motivation and emotion have always been an important part of SiMA [SDW+13, pp. 1-6], reinforcement learning based on emotional states seems to be a promising approach for future implementations.

Memory Limitation

Further improvements that are possibly necessary in the future is a memory handler that keeps the memory “clean”. In [SWT12, p.1007] the importance of *limiting the memory* of agents acting in real-time environments is stressed. Agents that have to react in a reasonable time, in a complex environment usually lack the time to consider all of the perfectly stored information. The implementation of a forgetting mechanism or at least some sort of activation level as it is used in SOAR (see chapter 2.3.4) will be therefore a necessary next step in SiMA. Otherwise the existent knowledge might overwhelm the agent’s decision process, as it is highly possible that too many equally important or matching experiences can be found after collecting experiences for some time. Possible solutions for limiting the memory are to implement some *time stamp* or *usage factor*, that tells the agent how often the memory was already accessed or how far in the past the experienced situation took place, would probably improve the decision process with respect to long-term memory usage. Forgetting is also an important part of the human’s brain functions [SWT12, p. 1007], here again the cooperation with other scientific disciplines should produce some further concepts that can be introduced to the SiMA system. For example, it should be considered to split the memory up into declarative, semantic and episodic memory and implement a *consolidation process* that is assumed to transfer information from the episodic memory to general facts in the semantic memory, from where the information can be accessed faster [SWT12, p.1007. This process is also assumed to shape the memorized experiences in order to improve recall performance.

Memory Editor

For the future use of the new long-term memory it would be an important step to implement a graphical *memory editor*. Such an editor is absolutely necessary in order to set-up new use cases and was part of the former file-based approach within the Protégé Frames Framework, which is of course lost for the new RDF triple store solution. A memory editor for SiMA should be able to represent the RDF

memory graph and allow to edit nodes inside the graph, as well as in an attribute window. It should also be possible to modify the triples in the database. For an effective usage, it should be possible to filter all of the mentioned views by data structure type, IRI and SPARQL statements. Due to the huge amount of data that is contained in such a memory, it is absolutely obligatory to provide a good user interface with options to show and hide elements freely. For that task tree views with collapsing options (for example based on subject IRI in the triple editor) or additional filter options like block lists, containing IRIs or data structure types that shall not be shown are highly recommendable. The usage of data structure depth has already proven itself valuable while implementing the visualization package (see chapter 6.3). In an even further step it would be interesting to implement OWL reasoning for the data. Another important feature would be the ability to construct memory parts and assign them to different “memories”, in order to avoid a relapse to the problems with unmaintained data, which were mentioned in chapter 4.3. It would also be an important step for the multi-agent system if separate memories could be created and assigned to different agents. This leads to another future task, namely the implementation of real multi-agent memories. At the moment all agents access the same memory, which is not realistic and also not practicable if the caching and threading approach is going to be implemented. Then in a further step agents should be able to learn from each other, like it was already implemented in the BDI implementation dMars (see chapter 2.3.5)

This work extended SiMA by an RDF database system, thereby providing the option to permanently store experiences during runtime. Furthermore the existing declarative semantic memory was converted to the new permanent store. In a final step the first proof of concept for saving episodic memories was implemented into SiMA. However, it is still a long way to go until the final goal – an agent that can proactively gain and effectively use its episodic memory - is reached.

Literature

[ABB+04] John R. Anderson, Daniel Bothell, Michael D. Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036–60, October 2004.

[AH08] Dean Allemang and James Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition, 2008.

[Bar10] Daniel Bartholomew. SQL vs. NoSQL. *Linux Journal*, 2010(195):4, 2010.

[BHS03] V. Bonstrom, A. Hinze, and H. Schweppe. Storing RDF as a graph. *Proceedings of the IEEE/LEOS 3rd International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices (IEEE Cat. No.03EX726)*, 2003.

[BLS+11] Laurent Bonnet, Anne Laurent, Michel Sala, Benedicte Laurent, and Nicolas Sicard. Reduce, You Say: What NoSQL Can Do for Data Aggregation and BI in Large Repositories. *2011 22nd International Workshop on Database and Expert Systems Applications*, pages 483–488, August 2011.

[Bro02] Frank Broekstra, Jeen and Kampman, Arjohn and van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *The Semantic Web – ISWC 2002: First International Semantic Web Conference Sardinia, Italy*, pages 54–68. 2002.

[CTN09] Hui-Qing Chong, Ah-Hwee Tan, and Gee-Wah Ng. Integrated cognitive architectures: a survey. *Artificial Intelligence Review*, 28(2):103–130, February 2009.

[DALL12] Nate Derbinsky, Ann Arbor, Justin Li, and John E Laird. Algorithms for Scaling in a General Episodic Memory (Extended Abstract). 2012.

[Deu11] Tobias Deutsch. *Human Bionically Inspired Autonomous Agents*. PhD thesis, Vienna University of Technology, 2011.

[DFZB09] Dietmar Dietrich, Georg Fodor, Gerhard Zucker, and Dietmar Bruckner. *Simulating the Mind - A Technical Meuropsychoanalytical Approach*. 2009.

[DGLV08] T Deutsch, A Gruber, R Lang, and R Velik. Episodic Memory for Autonomous Agents. pages 2–7, 2008.

[DLG+04] Mark D'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System. *Autonomous Agents and Multi-Agent Systems*, 9(1/2):5–53, July 2004.

[DSBD13] Dietmar Dietrich, Samer Schaat, Dietmar Bruckner, and Klaus Doblhammer. The Current State of Psychoanalytically-Inspired AI - A Holistic and Unitary Model of Human Psychic Processes. *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE*, pages 6666–6671, 2013.

[GB04] Steffen Gais, Jan Born. Declarative memory consolidation: Mechanisms acting during human sleep. *Learning & Memory*, pages 679-685, 2004.

[GHES05] Alejandro Guerra-Hernández, Amal El Fallah-Seghrouchni, and Henry Soldano. Learning in BDI multi-agent systems. *Lecture Notes in Computer Science*, 2005.

[GL10] Fernand Gobet and Peter Lane. The CHREST Architecture of Cognition: The Role of Perception in General Intelligence. *Proceedings of the 3d Conference on Artificial General Intelligence (AGI-10)*, pages 1–6, 2010.

[GLC+01] Fernand Gobet, Peter C R Lane, Steve Croker, Peter C-h Cheng, Gary Jones, Iain Oliver, and Julian M Pine. Chunking mechanisms in human learning. 5(6):236–243, 2001.

[HFsP+04] Alejandro Guerra Hern, Amal El Fallah-seghrouchni, Informatique De Paris, U M R Cnrs, and Capitaine Scott. Distributed Learning in Intentional BDI Multi-Agent Systems. 2004.

[KDA11] Atanas Kiryakov, Mariana Damova, and Ontotext Ad. *Handbook of Semantic Web Technologies*. 2011.

[Lai08] JE Laird. Extending the Soar Cognitive Architecture. In *Proceedings of the 2008 Conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, pages 224—-235. IOS Press, 2008.

[Lai14] John E Laird. The Soar 9 Tutorial. 0, 2014.

[Lan10] Roland Lang. *A Decision Unit for Autonomous Agents Based on the Theory of Psychoanalysis*. Phd thesis, 2010.

[Law14] Ramon Lawrence. Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB. *2014 International Conference on Computational Science and Computational Intelligence*, pages 285–290, March 2014.

[LC14] John E Laird and Clare Bates Congdon. The Soar User â€™ s Manual version 9.4.0. Technical report, Computer Science and Engineering Department University of Michigan, 2014.

[LCT11] Pat Langley, Dongkyu Choi, and Nishant Trivedi. ICARUS User's Manual. Technical report, Institute for the Study of Learning and Expertise 2164 Staunton Court, Palo Alto, CA 94305 USA, 2011.

[Lea10] Neal Leavitt. Will NoSQL Databases Live Up to Their Promise?, 2010.

[LGS09] Peter C R Lane, Fernand Gobet, and Richard Ll Smith. Attention Mechanisms in the CHREST Cognitive Architecture. pages 183–196, 2009.

[Lor14] Iris Lorscheid. Learning Agents for Human Complex Systems. 2014.

[NLA07] Andrew M Nuxoll, John E Laird, and Ann Arbor. Extending Cognitive Architecture with Episodic Memory. pages 1560–1565, 2007.

[PDH+07] Gerhard Pratl, Dietmar Dietrich, Gerhard P Hancke, Walter T Penzhorn, Senior Member, Justin Li, John E Laird, Ann Arbor, Ben Goertzel, Cassio Pennachin, Nil Geisweiller, Richard L. Smith, Peter C.R. Lane, Fernand Gobet, Vergleich Pm, Zertifizierung Pmi, Kognitive Leistungen, Shiwali Mohan, James Kirk, J I L L F A I N L Ehman, J O H N L Aird, P A U L R Osenbloom, Editorial Board, and Phoebe Chen. A New Model for Autonomous, Networked Control Systems. *2008 Second UKSIM European Symposium on Computer Modeling and Simulation*, 3(1):21–32, September 2007.

[Rus03] Gerhard Russ. *Situation-dependent Behavior in Building Automation*. PhD thesis, Vienna University of Technology, 2003.

[SDW+13] Samer Schaat, Klaus Doblhammer, Alexander Wendt, Friedrich Gelbard, Lukas Herret, and Dietmar Bruckner. A psychoanalytically-inspired motivational and emotional system for autonomous agents. *IECON Proceedings (Industrial Electronics Conference)*, pages 6648–6653, 2013.

[SGL09] Richard L. Smith, Fernand Gobet, and Peter C.R. Lane. Checking chess checks with chunks: A model of simple check detection. 2009.

[SL07] Richard Ll Smith and Peter C R Lane. An Investigation into the Effect of Ageing on Expert Memory with CHREST. 2007.

[Sto10] Michael Stonebraker. SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10, April 2010.

[SWT12] Budhitama Subagdja, Wenwen Wang, and Ah Tan. Memory formation, consolidation, and forgetting in learning agents. *on Autonomous Agents,* (June):4–8, 2012.

[TCM03] Douglas G Turnbull, C M Chewar, and D Scott Mccrickard. Are Cognitive Architectures Mature Enough to Evaluate Notification Systems? *International Conference on Software Engineering Research and Practice (SERP 2003),*, (June), 2003.

[TG01] Andrew S Tanenbaum and James R Goodman. *Computerarchitektur - Strukturen, Konzepte, Grundlagen (4. Aufl.).* Pearson Studium, 2001.

[VBPG09] Rosemarie Velik, Dietmar Bruckner, Peter Palensky, and Envidatec Gmbh. A Bionic Approach for High-Efficiency Sensor Data Processing in Building Automation. pages 1949–1954, 2009.

[Vel08] Rosemarie Velik. A Bionic Model for Human-like Machine Perception. 2008.

[VMS07] David Vernon, Giorgio Metta, and Giulio Sandini. A Survey of Artificial Cognitive Systems : Implications for the Autonomous Development of Mental Capabilities in Computational Agents. 11(2):151–180, 2007.

[WDM+12] Alexander Wendt, Benjamin Dönz, Stephan Mantler, Dietmar Bruckner, and Alexander Mikula. Evaluation of Database Technologies for Usage in Dynamic Data Models. In Joaquim Filipe and Ana Fred, editors, *ICAART 1*, pages 212–224. SciTePress, 2012.

[WTN+] Hai H Wang, Samson Tu, Natasha Noy, Alan Rector, Mark Musen, Timothy Redmond, Daniel Rubin, Tania Tudorache, Matthew Horridge, and Julian Seidenberg. Frames and OWL Side by Side. pages 1–4.

[ZDI+08] H Zeilinger, T Deutsch, Member Ieee, B Müller, and R Lang Member. Bionic Inspired Decision Making Unit Model for Autonomous Agents. pages 259–264, 2008.

[Zei10] Heimo Zeilinger. *Bionically Inspired Information Representation for Embodied Software Agents*. PhD thesis, Vienna University of Technology, 2010.

[ZLM09] H Zeilinger, R Lang, and B Müller. Bionic Inspired Information Representation for Autonomous Agents. pages 383–389, 2009.

Internet references

[ACTR] "ACT-R 6.1 Software", available http://act-r.psy.cmu.edu/software/, accessed 2015-05-13

[ALIST] "Class ArrayList<E>Java SE Documentation – API", available http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html, accessed 2014-11-30

[ALLEGRO] "AllegroGraph", Franz Inc., available http://franz.com/agraph/allegrograph/, accessed 2014-11-30

[FUKU11] "AKW Fukushima: Die „Tapferen 50" an der Strahlenfront", SPIEGELONLINE, http://www.spiegel.de/panorama/gesellschaft/akw-fukushima-die-tapferen-50-an-der-strahlenfront-a-751070.html, accessed 2015-01-21

[FUKU14] "Erster Reaktor in Fukushima ist gesäubert", derStandard, http://derstandard.at/2000009664839/Meilenstein-in-Fukushima-Erster-Reaktor-ist-gesaeubert, accessed 2015-01-21

[HASHMAP] "Class HashMap<K.V>", Java SE Documentation - API, available https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html, accessed 2014-11-30

[IRI] "Internationalized Resource Identifier", WIKIPEDIA, http://de.wikipedia.org/wiki/Internationalized_Resource_Identifier, accessed 2015-02-24

[JAVA] "What is Java?", Oracle, https://www.java.com/en/download/whatis_java.jsp, accessed 2015-02-24

[JENA] "RDF", W3C, http://www.w3.org/RDF/, accessed 2015-02-24

[JSON] "JSON", JSON, http://json.org/, accessed 2015-06-08

[MEMORY] "How Human Memory Works", Mohs, Richard C., http://science.howstuffworks.com/life/inside-the-mind/human-brain/human-memory.htm, accessed 2015-03-05

[NARY] "Defining N-ary Relations on the Semantic Web", W3C, http://www.w3.org/TR/swbp-n-aryRelations/, accessed 2015-03-11

[NEO4J] "Neo4j", Neo Technology, Inc. available http://neo4j.com/, accessed 2014-11-30

[OWL] "OWL 2 Web Ontology Language", W3C, http://www.w3.org/TR/owl2-overview/, accessed 2015-02-21

[RDFQUERY] "RDF query language", WIKIPEDIA, http://en.wikipedia.org/wiki/RDF_query_language, accessed 2015-02-24

[REIFICATE] "RDF Reification", W3C, http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#reification, accessed 2015-03-11

[ROBO11] "Überforderte Roboter im Atom-Einsatz", ZEITmagazin, http://www.zeit.de/wissen/2011-04/roboter-katastrophen, accessed 2015-01-21

[SIMA15] "Simulation of the Mental Apparatus & Applications", Project website, http://sima.ict.tuwien.ac.at/, accessed 2015-05-16

[SMAKI14] "SmartKitchen", Project website, http://smartkitchen.ict.tuwien.ac.at/, accessed 2015-01-09

[SOCK] "Class Socket", Oracle, http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html, accessed 2015-05-22

[SPARQL] "SPARQL 1.1 Query Language", W3C, http://www.w3.org/TR/2013/REC-sparql11-query-20130321, accessed 2015-02-24

[VISU] "JavaScript InfoVis Toolkit", JavaScript InfoVis Toolkit, http://philogb.github.io/jit/, accessed 2015-06-07

[VISUAL] "VisualVM", VisualVM, https://visualvm.java.net/, accessed 2015-05-22

[W3CRDF] "RDF", W3C, http://www.w3.org/RDF/, accessed 2015-02-24

[W3SCHOOL] "RDF", W3Schools, http://www.w3schools.com/webservices/ws_rdf_reference.asp, accessed 2015-02-24

[WIRED] "THE THREE BREAKTHROUGHS THAT HAVE FINALLY UNLEASHED AI ON THE WORLD", Wired, http://www.wired.com/2014/10/future-of-artificial-intelligence/, accessed 2015-02-21

Declaration

Hereby, I declare, this present work has been drawn up without inadmissible aid of third parties and without usage of other than mentioned resources. Further sources or indirectly appropriated data and concepts are identified by stating the source.

This work has not been presented to other examination procedures, neither nationally, nor in foreign countries, in the same or in a similar form.

Vienna, 16th July 2015

Verena Himmelbauer

Verena Himmelbauer