A Spatiotemporal Working Memory for Humanoid Robots

Masterarbeit
im Fach Intelligente Systeme
an der Technischen Fakultät
Universität Bielefeld

von: Johannes Wienke
jwienke@techfak.uni-bielefeld.de
Artur-Ladebeck-Str. 89
33617 Bielefeld
GERMANY

Betreuer: Dr.-Ing. Sebastian Wrede
Dr.-Ing. Britta Wrede

October 7, 2010
## Contents

### Abstract

<table>
<thead>
<tr>
<th>1. Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1. Working Memory and Robotics</td>
<td>2</td>
</tr>
<tr>
<td>1.2. Structure of Working Memory</td>
<td>3</td>
</tr>
<tr>
<td>1.3. Outline</td>
<td>4</td>
</tr>
</tbody>
</table>

### 2. Related Work

| 2.1. Memory Frameworks with General Applicability | 5 |
| 2.2. Symbolic Approaches to Cognition and Memory | 7 |
| 2.3. The Egosphere Concept | 8 |

### 3. Requirement Analysis

| 3.1. The HUMAVIPS Scenario: Robot-to-Group Interaction | 11 |
| 3.2. Requirements Arising from the Scenario | 12 |
| 3.3. Technical Requirements | 15 |

### 4. EgoMemory Framework

| 4.1. Representation of Space | 19 |
| 4.2. Integration of Specialized Data: Record Types and Layers | 20 |
| 4.2.1. Implemented Layers | 22 |
| 4.3. Representation of Time: History Model | 23 |
| 4.3.1. Forgetting | 25 |
| 4.3.2. Timestamps | 25 |
| 4.4. Prediction and Correction: Missing Data and Uncertainties | 26 |
| 4.5. Hard Links: Distributed Representations | 28 |
| 4.6. Operations on Time Step Changes | 30 |
| 4.7. Subscription Model | 33 |
| 4.7.1. Subscriptions on Records | 33 |
| 4.7.2. Subscriptions on Hard Links | 34 |
| 4.7.3. Software Realization of Subscriptions | 34 |
| 4.7.4. Typical Subscription Modes for Clients | 36 |
| 4.8. Query Language and Interface | 36 |
| 4.8.1. Desired Queries and Existing Query Languages | 37 |
| 4.8.2. EgoMemory Query Language | 39 |
| 4.9. System Architecture | 42 |
| 4.9.1. Global Structure | 43 |
| 4.9.2. Distribution Patterns | 43 |
| 4.9.3. Middleware Abstraction | 45 |
Abstract

Working memories facilitate cognitive processes by providing a mediating interface between perception, long-term memory, and behavior generation. Therefore, they actively maintain a short-term history of their contents. More recent theories of working memory suggest that the memory system is divided into several slave-systems which maintain different types of information. One example for such a separation is described in Baddeley [Bad03], which proposes two slave-systems, a visual and a phonological one. The visual system is called visuospatial sketchpad. Both memories are controlled by a central executive. This thesis introduces an implementation of a working memory system influenced by the idea of a visuospatial sketchpad and based on an egocentric representation of memory contents. It aims at integrate data of different types and abstraction levels by using layers that maintain records in a spherical coordinate system. Data within these layers can be combined to more complex information through a graph structure. Furthermore, a short-term history is maintained and made accessible by a query language and an event system. The implemented system was integrated into a scenario on a humanoid robot showing its abilities to support common tasks in robotics and human-robot interaction.
1. Introduction

The ability to show intelligent behavior demands a high level of cognitive processing and integration of diverse kinds of information and goals, for humans as well as artificial agents:

Intelligence is the integration of perception, reason, emotion, and behavior in a sensing, perceiving, knowing, caring, planning, acting system that can succeed in achieving its goals in the world. [Alb91, p. 474]

For example, this includes filtering perception and directing attention towards salient stimuli that are relevant for the current goals. Actions must be planned to fulfill these goals, which may involve changing the posture for generating better sensory inputs. Recognizing other agents and objects requires to match stored information with the current inputs and knowledge about previously perceived episodes, and the actions of other agents help classifying their current intents and behaviors.

Cognitive psychology and neuroscience have spent a lot of research effort to investigate how the humanoid brain is structured to meet these requirements. As a result, several models of human memory have been developed. The vast majority of recent models agrees on the fact that the human memory is not a single monolithic system [AS68; Bad03; Tul85]. Instead several subsystems were identified. The actual division of the memory into these subsystems is, however, theory-dependent. Generally speaking two different dimensions of structuring memory can be identified: time and content. The temporal axis separates memory by the duration information is maintained in memory which resulted in commonly known terms like short-term and long-term memory, for instance proposed in [AS68]. On the other hand, the content axis differentiates subsystems based on the type of managed information, e.g. procedural and semantic information [Tul85].

One theory that emphasizes the role of memory as an integral part of human cognitive processing is called Working Memory. The term was first proposed by Miller, Galanter, and Pribram [MGP60] and adopted by Baddeley and Hitch [BH74]. Baddeley defines the term Working Memory as "a limited capacity system, which temporarily maintains and stores information, supports human thought processes by providing an interface between perception, long-term memory and action" [Bad03]. Working Memory as defined by Baddeley can be seen as a short-term memory on the temporal axis, consisting of several distinct storage and processing components, but should not be mixed up with older theories that proposed a unitary view of short-term memory.


1.1. Working Memory and Robotics

Many applications in robotics require complex systems that are capable of performing diverse tasks. Especially when robots are supposed to operate in unstructured environments involving unrestricted interactions with humans, many different strategies are required to form a system that successfully solves tasks. Examples for such systems and applications can be found in the RoboCup@Home league [Rob]. Recent research has focused on how to structure these systems by means of cognitive architectures. Within these architectural frameworks, memories have proven to be an essential part [LLR09, p. 2]. Tasks performed with memories and benefits gained from using memories are twofold: functional and non-functional.

From a functional point of view, the aforementioned tasks of Working Memory defined by Baddeley are of high importance for robotic systems. These systems constantly perceive inputs of different modalities, most of them with high frequencies. Reactive behaviors require that these sensory inputs are processed in a reasonable time to perform actions based on them (perception-action coupling). For example, a mobile robot must be able to turn away from a wall before it collides with it. Also, many sensory inputs and the processing results based on them are probabilistic and contain a high amount of noise caused by the sensors, the processing, or the environment itself. Therefore, a robotic system must be able to filter irrelevant information and combine data and hypotheses of different modalities to achieve more stable results for further processing with reasonable processing times. This task is a natural application for a Working Memory as the memory contains all required information. Besides these reactive behaviors, cognitive systems often contain a lot of knowledge that is persistent over time like classifiers for objects and people or facts about the environment. Planning requires to match the current inputs against persistent information that are often stored in a long-term memory. Sometimes it is desirable to have top-down processing that forms expectations about sensory inputs based on stored examples in long-term memories. These expectations can be represented in the Working Memory as a means of matching the sensory inputs with the expectations using the same representation. In general, the mediating task of Working Memory can be characterized as combining different special-purpose subsystems realizing the aforementioned tasks with their specialized representations into a coherent system, hence consisting of a “bag of tricks” [Cla01].

Besides functional aspects, the existence of (working) memories in robotic systems has a high impact on the software architecture and yields many benefits from a software engineering perspective [Wre08]. The majority of robotic systems is distributed and highly parallel, consisting of several different processes orchestrated into a whole system. Many of these processes operate on the same data that needs to be shared. For example, images from a camera system may be of interest for a depth calculation, object classification, and a face detection. Several components may require the same preprocessing on the shared data. Storing the preprocessing results in the memory prevents duplicated calculation of these results and decreases the system load. Furthermore, many advanced algorithms used in robotic systems do not process their results on a single modality but instead combine different modalities to receive more
stable results. Using a memory yields a way to coordinate data access and sharing between processes in a structured and synchronized way without having to clutter the system with bidirectional connections. Moreover, memories foster data representations of more general usability for different components of the system by providing a global structure for the data. Maintaining these data in a central repository, the memory, allows integrating generic tools with access to all relevant data streams of the system at a single point. Possible applications include learning error detection models on these data streams as proposed by Golombek et al. [Gol+10].

1.2. Structure of Working Memory

The Working Memory model described in Baddeley [Bad03] proposes a separation of working memory into three components as visualized in figure 1.1. The Central Executive is the control system of working memory which is assisted by two storage systems. Baddeley’s first idea was to declare the central executive as a general pool of processing power for the working memory. Later, it was refined to be responsible for attentional control. However, there has not been much research on this part of the working memory and as such the theories are relatively vague.

The first storage system proposed by Baddeley is the Phonological Loop. It is responsible for speech understanding and production by storing auditory memory traces which fade out if they are not rehearsed. The second storage system is called Visuospatial Sketchpad. This storage system is assumed to store visual and spatial information and manage active work with such representation. Tasks involve remembering shape and color information or planning movements. Both storage systems are of limited capacity.

In later revisions of this model Baddeley introduced a third storage system, called Episodic Buffer, with the purpose of forming integrated episodes from multidimensional information.
1.3. Outline

This thesis introduces an implementation of a memory model for humanoid robots influenced by Baddeley's visuospatial sketchpad. Data of diverse representations is represented in an egocentric coordinate system and maintained with a short-term history, hence forming a spatiotemporal working memory.

The thesis continues with a description of different related memory architectures with relevance for the own work in chapter 2. The memory models of these systems are grouped by their specificity and data representation format. Afterwards, chapter 3 gives detailed descriptions of requirements for the developed memory system. Many of these requirements arise from the intended application of the memory in the European FP7 Project HUMAVIPS – Humanoids with Auditory and Visual Abilities in Populated Spaces, hence giving a brief introduction of the project, too. Chapter 4 presents the developed memory system. The description combines conceptual and implementational aspects and explains how they interact to form the system. In chapter 5 the developed memory is evaluated in different scenarios and benchmarks. The thesis closes with a conclusion and an outlook in chapter 6.
2. Related Work

The research community on cognitive architectures has developed various memory-based frameworks and approaches for cognitive processing within the last years. The memory systems developed for these frameworks vary in different ways:

- intended memory function (short-term, long-term, episodic, etc.)
- data representation format
- flexibility of representations
- symbolic, connectionist, or hybrid approaches

This chapter introduces a selection of memory architectures of relevance to the developed framework. The following sections are organized to start with frameworks of general usage and representation forms, mostly motivated from a software engineering perspective. Subsequently, frameworks with more specialized representation forms are described, closing with the most relevant concept for this thesis: egospheres.

2.1. Memory Frameworks with General Applicability

Wrede et al. [Wre+06] and Wrede [Wre08] describe a general memory system called Active Memory developed in the context of cognitive systems. It was used for a vision-based and augmented-reality assistance system in the VAMPIRE project as well as for a mobile robotic platform (BIRON) [SSS08]. The Active Memory is a memory based on the Oracle Berkeley DB XML [Dbx], an embeddable data base for document-oriented storage of XML documents. Thus, data is represented by schema-free XML documents in the Active Memory concept with the ability to attach binary data to documents. The memory server notifies clients, called extrinsic memory processes, on document changes. These include all supported operations: insert, replace, query and remove. Subscriptions on these notifications are content-based, filtering the affected documents by an XPath expression [Ber+07] and additional filters. Data in the Active Memory is always shared between all memory processes. Hence, there exists no ownership of data and no locking is possible. In addition to client-level extrinsic memory processes, intrinsic memory processes can be created. These processes are executed in-process with the server. As there are no requirements on the data except that it can be serialized to XML, the programmer of memory processes is responsible for generating meaningful XML documents that can be shared by several components. Using the same element and attribute names — e.g. for reliability ratings of hypotheses — is one example how documents need to be structured for data sharing. These
ratings can be queried or clients can subscribe on documents containing them through an XPath expression that ignores the remaining document structure. The Active Memory model itself makes no assumptions about the type of conceptual memory it is used for (short-term, long-term, etc.), but a short-term like forgetting process is introduced. Also, no formal model for maintaining a history exists. Timestamps and versioning information can be expressed inside the XML data requiring manual work of the client software.

A second example for a generally applicable framework which uses memories as an integral processing structure is CAST [The CoSy Architecture Schema Toolkit, HZW07; HW08; HW10]. CAST is based on an architectural schema that defines the underlying architecture of a cognitive system. The system, in turn, is decomposed into several subarchitectures, each consisting of more or less closely related processing components that work in parallel and store their processing results in working memories. Each subarchitecture has one working memory. In addition to the memory, every subarchitecture contains a special component called task manager controlling some of the other components in its subarchitecture, which results in a classification of processing components as managed or unmanaged by the task manager. The architecture schema defines rules how components can access the working memories. Per default all components of a subarchitecture have full access to the subarchitecture’s working memory and read-only access to other subarchitecture’s memories. A working memory in CAST is an associative container which maps from a working memory address to an entry. The address consists of the subarchitecture’s name and a unique identifier per memory. Memory entries are program objects (classes) that consist of their address, a type identifier, a version for conflict resolution, and a user data object. They must be describable in the IDL (SLICE) of the underlying middleware ICE. Working memories in CAST provide a set of operations comparable to the Active Memory with the possibilities to add, overwrite, or delete entries. In contrast to Active Memory, no formal query language is available, thus elements can only be retrieved by ID or type (returning all elements of a type). Another feature supported by CAST memories, which can be compared to Active Memory, are subscriptions on changes of memory contents, called change events. These events contain the working memory address and type of the changed record, the performed operation, as well as the component that performed the action. These information can be used to filter notifications. Again, no filters on the user data are possible. CAST motivates memories from a component synchronization and data sharing viewpoint. This is enabled by typically using only one data type in a subarchitecture’s working memory with all components accepting this data type [HW10].

Summarizing the important aspects of both systems, Active Memory and CAST, for the developed memory system shows that both frameworks motivate the use of memories from a software engineering perspective. The ability to share data between processes in an event-based manner, combine their results, and synchronize them are key features of both systems that successfully integrated complex systems. These mechanisms are used as the foundation for the system integrational aspects of the memory framework as described in this thesis. CAST provides a way to handle

specialized data representations, which goes beyond specifically structuring a general
document model like XML, by giving the ability to include user-defined data types.
On the other hand, it lacks the flexibility of easily extending these representations
and filtering subscriptions by their content. A comparable approach of integrating
data types is taken in this thesis but extends it by allowing queries on the user data
similar to the Active Memory system and by providing more specialized processing
logic from a functional viewpoint.

2.2. Symbolic Approaches to Cognition and Memory

A second class of cognitive architecture frameworks presents memories based on sym-
bolic approaches to cognition where knowledge is represented exclusively through
symbols and processing is based solely on symbolic reasoning. One example is Soar
[LC09; Soa], which is intended to be usable for every kind of cognitive problem solv-
ing agent. The general processing loop of Soar is based on the assumption that every
goal-directed behavior can be expressed as a selection of operators based on the sys-
tem’s state and current goals. The state is, in turn, modified by the applied operator
resulting in a new state for operator selection. To represent the current goals and the
system state, which includes inputs from the world, Soar contains a working memory.
This is essentially a set of objects with a unique identifier described by attributes (see
figure 2.1). Attributes consist of a meaning part and a value. If the value refers to
another identifier contained in the memory, an implicit graph of linked information is
created. Objects contained in the working memory must be linked to a state which
represents the current progress in solving a problem and the possible operations to
apply. States are represented in the memory in a similar way to objects using a col-
lection of symbolic attribute-value pairs. An object that is not linked to a state will
be deleted by the system. To represent possible operators, Soar includes a long-term
production memory containing rules about which operators to apply in which situations. The conditions are boolean expressions that are matched against the working memory contents.

Besides Soar, ACT-R [Bot; Act] is a notable framework that pursues the idea of a unified model of cognition based on symbolic representations. The basic structural idea is that two distinct types of modules exist. Perceptual-motor modules are the interface with the real world, providing perception results and controlling actuators, while memory modules store knowledge about the world and how to perform tasks. Knowledge about the world is represented as facts in a declarative memory using a symbolic representation whereas knowledge of how to perform tasks is stored in a procedural memory. The contents of the procedural memory are essentially the same productions that Soar uses, consisting of patterns to match against the current state and actions that will be performed to modify the state.

Soar and ACT-R represent one extreme form of representing knowledge and processing solely through symbolic reasoning. While this is not the desired overall paradigm for the described memory system, the memory should still allow the realization of such approaches through the flexible representation of user data and access mechanisms.

2.3. The Egosphere Concept

The egosphere concept describes the model of memory with the highest functional specificity regarded for this thesis. It was first introduced by Albus in 1991. He defines an egosphere as follows:

An egosphere is a two-dimensional (2-D) spherical surface that is a map of the world as seen by an observer at the center of the sphere. Visible points on regions or objects in the world are projected on the egosphere wherever the line of sight from a sensor at the center of the egosphere to the points in the world intersect the surface of the sphere. Egosphere coordinates thus are polar coordinates defined by the self at the origin. [Alb91, p. 488]

This concept was later implemented on different robotic platforms with varying aims. Peters et al. [Pet+01] and Peters, Hambuchen, and Bodenheimer [PHB08] describe an implementation which is capable to fuse multimodal data using spatial and temporal information. Moreover, it provides an attention mechanism. The system was tested both on humanoid as well as on mobile robots where it is used for navigation based on the egocentric representation of detected landmarks. In contrast to Albus’ idea of egospheres as a dense map of the world, the framework described by Peters et al., called Sensory Ego-Sphere (SES), uses a discretization of the spherical surface by means of a regular tessellation as depicted in figure 2.2. This forms a graph of connected spatial nodes at each vertex of the tessellation. Data is attached to these vertices. An activation network is used to detect spatial and temporal coincidences of events spreading over multiple vertices. Moreover, it can be biased to direct the attention of processing modules. Conforming with Albus’ theory of egospheres, the
Figure 2.2.: The Sensory Egosphere as depicted in Peters et al. [Pet+01]. The stimulus $S$ is perceived by the robot at the center of the sphere. In the memory it is attached to the vertex $N_s$ that is closest to the intersection point of the line between the sphere center and $S$ with the spherical surface. $\Theta_s$ and $\phi_s$ are the elevation and azimuth angles of the stimulus in polar coordinates and $r_s$ is its distance from the observer at the center of the sphere.

SES transforms the contents of the egosphere on egomotion of the robot. If translations occur, the distance of objects from the sphere’s center must be known. For this purpose and to retain the sensory resolutions, every data element stored at a node contains its exact location as an offset to the vertex. If the depth is not known but the motion of the robot, the movement of objects on the sphere can be used to estimate depth information in a structure-from-motion-like way [Sze10, pp. 343 sqq.].

In addition to the exact position, data stored in the egosphere contains a timestamp that indicates the time of storage and a tag for the data type. The actual data according to the tagged type is attached as a software object and the activation network is parametrized according to the data type.

A more complex cognitive architecture that makes use of the SES was described in Kawamura et al. [Kaw+08]. The SES serves as a short-term memory for sensory results in the architecture. Besides the SES, a long-term memory and a working memory (not comparable with Baddeley’s definition) are used in the system.

Another application for an egosphere was described in Ruesch et al. [Rue+08]. Here, it is used to create a bottom-up attention framework for the humanoid robot iCub based on visual and auditory saliency. Therefore the egosphere consists of several rectangular maps which are indexed by the azimuth and elevation angles of the polar coordinates. Each map represents a stimulus or a combination of several stimuli. This implies that every stimulus and processing result can be represented as a map, which is directly possible for visual saliency. The auditory saliency is transformed to a map by projecting the sound source localization results onto the map with a normal distribution. Both maps are combined by using the maximum at each map location. While Peters et al. maintain a short-term history through timestamped data, the
approach by Ruesch et al. combines new maps with old ones using a decay factor to forget old information, which results in an implicit representation of history. In contrast to the SES approach with a symbolic structure for the egosphere containing explicit data records, this egosphere concept represents knowledge in a sub-symbolic form.

The egosphere idea is of special importance for the described memory system as it introduces the spatial properties of memory contents as a crucial feature respecting the functional requirements on visuospatial memories. The egocentric representation forms the basis for the representation of knowledge in the developed memory system, and the spatial and temporal data fusion aspects are a central requirement for the introduced system. The explicit representation of records in the SES concept is the model chosen for the own work, being compatible with the ideas of CAST and Active Memory. Nevertheless, more subsymbolic representations like the one described by Ruesch et al. [Rue+08] present a concept check for the flexibility of representations and specialized processing logic offered by the developed memory system.
3. Requirement Analysis

Existing systems prove that building cognitive systems, especially in robotics, with shared memories as an integral part of the processing and data distribution logic, provide solutions for many problems in this domain. Nevertheless, limitations exist and were identified while gaining experience with these systems, especially the Active Memory (see section 2.1). While providing a basis for cognitive systems with the ability to be used for every type of memory, this system cannot account for specialized representations required by specific subsystems. CAST, on the other hand, provides a data type specialization mechanism but does not provide content-based query operations and filtering solutions. Neither of these memory systems can account for specialized operations and processing logic which is required by special-purpose subsystems and simplifies their processing. This leads to the conclusion that several distinct and much more specialized memory systems are required to build up an optimal cognitive system, in accordance to what many cognitive psychologists and neuroscience claim. These specialized memory systems, in turn, have to handle the “bag of tricks” approach of current cognitive systems and support components and their specialized knowledge representations.

This chapter describes the requirements which were indentified for the memory system developed in this thesis. According to the separation of concerns in Working Memory proposed by Baddeley (section 1.2), the focus is layed on visuospatial Working Memory tasks. The major part of the identified requirements arises from the intended use of the system in the HUMAVIPS project. Thus, a short introduction on the project is given.

3.1. The HUMAVIPS Scenario: Robot-to-Group Interaction

The idea of HUMAVIPS (Humanoids with Auditory and Visual Abilities in Populated Spaces) is to endow a humanoid robot with abilities that enable him to interact naturally in a group of people. This requires skills like detecting, recognizing, or tracking various people at the same time, determining their state in the group, and to interact with them, either verbally or non-verbally. People who are already talking should not be interrupted while others are predestinated as interaction partners. Moreover, interactions with several people of a small group at once are intended. While humans seem to be able to solve this task naturally, most current robotic systems are bound to one interaction partner at a time. Solving the tasks necessary for robot-to-group interactions from a robotics perspective relies on the ability to fuse data of multiple
modalities. The HUMAVIPS project will, on the one hand, focus on the development of perception algorithms that are able to generate meaningful cues for robot-to-group interaction from audiovisual data and, on the other hand, create models for generating appropriate interactions with a group of people. These tasks will be realized with a software architecture that is based on the ideas of memories and forms another contribution of the project.

The target platform for the project is the commercially available humanoid robot Nao, developed by Aldebaran Robotics\(^1\) (see figure 3.1). It has 25 degrees of freedom, is 58 cm tall if standing and is equipped with various sensors including two cameras, 4 microphones, sonar, position sensors, and several tactile sensors. While the current version of Nao does not possess a vision system that is able to compute 3D information, the final version for HUMAVIPS will be equipped with a stereo vision setup. Besides sensors, Nao has 2 loudspeakers, an integrated text to speech engine, and different LEDs on the whole body for non-verbal communication. A WLAN device allows to distribute the processing and relieve the load on the integrated computer which is a 500 MHz AMD Geode.

3.2. Requirements Arising from the Scenario

Because HUMAVIPS is based on a humanoid robot platform, a memory system used on such a robot must support the sensory structure of humanoid robots. Cameras and microphones as well as other sensors that observe the surroundings of the robot are usually mounted on the head. Representing their data in a visuospatial memory should be possible in a natural way with the least possible amount of coordinate

\(\text{http://www.aldebaran-robotics.com}\)
transformations. On the other hand, using the memory contents in typical scenarios for humanoid robots must present their location information in a way that minimizes the effort needed to perform common tasks, e.g. selecting relevant persons to approach based on their locations. Hence, the first requirement can be formulated as:

**Requirement 1. Humanoid-Aware Coordinate System** A visuospatial memory for humanoid robots must respect their sensory configuration by using a natural, humanoid-centered coordinate system that simplifies registration of information as well as the application in typical scenarios.

In addition to this requirement the chosen coordinate representation must handle location information of different characteristics. While some sensory processes on the robot are only capable of delivering directional information, e.g. a sound source localization, the robot will also be equipped with a stereo-vision setup for the HUMAVIPS project allowing the computation of full 3D coordinates. Depth information can be gathered from the integrated sonar sensors, and motion planning requires to know the full 3D location of effectors. Also the generation of appropriate social behavior depends on the distance of the robot from possible interaction partners. The memory system has to integrate the different kinds of location information without degrading the performance and usability of full 3D coordinates for memory clients.

**Requirement 2. Different Location Information Characteristics** Location information of different characteristics (e.g. directional or full 3D coordinate) must be integrated in the memory without making the application of algorithms on full 3D coordinates more complex.

The strong focus of the HUMAVIPS project on audiovisual processing and data fusion requires a visuospatial memory which is able to cope with subsymbolic data of diverse types. Images are represented as matrices for each frame of the camera while auditory data forms a continuous stream which is normally chunked into small time-spanning data packages. Both types of sensory data must be representable in the memory in a way that supports their individuality while preserving a global structure of the memory which enables clients uninterested in the individual details of the data to still get meaningful information from the global arrangement of memory contents.

**Requirement 3. Diverse Sensory Data Representation** A visuospatial memory must be able to represent sensory data of different modalities and representations in a way that supports the individual data types while preserving a meaningful overall memory structure.

Besides sensory data, the robot will generate various kinds of hypotheses, most notably for persons in the scene. These hypotheses are presumably represented in a symbolic form and must be integrated into the memory as well. Symbolic data has to be integrated in a way that makes use of and supports the overall spatial structure of the memory, too. Moreover, the flexibility of varying contents for different hypothesis types must be given.
**Requirement 4. Integration of Different Data Abstractions**  Besides sensory subsymbolic data, the memory must provide means to represent symbolic data while preserving the overall spatial structure of the memory. In general, data of different abstraction levels must be integrable into one system making use of the same spatial structures.

Symbolic hypotheses maintained in the memory are normally based on perceived sensory information. Moreover, they can be related to each other, e.g. to represent the relations of people in a group. In a general way it must be possible to express these kinds of dependencies or relations in the memory system to build distributed representations of higher value than the single memory elements. Easy access to these representations is required to foster their use.

**Requirement 5. Distributed Representations**  The memory must possess a structure that can express data dependencies and relations to form distributed representations of higher value than the single memory contents. These distributed representations must be accessible for client components.

As the fusion of multimodal data is one of the main goals of HUMAVIPS, a memory system must assist in this task. One property of the memory contents that can give a cue for fusion is their location in space. Hence, the memory must support data fusion components by providing an easy interface to spatial relations of different data streams.

**Requirement 6. Spatial Data Fusion Interface**  The memory system must grant easy access to data-fusion-relevant spatial properties of different memory contents and generate meaningful cues based on these information.

Stabilizing processing results, uncovering otherwise hidden states [McC96] — especially on a robot with limited sensors like Nao — or detecting interesting events for social interactions often requires a history of percepts. Hence, the working memory must include a short-term history of it contents which has to be made accessible for client components.

**Requirement 7. Short-Term History Representation**  A visuospatial memory as a part of the working memory must maintain a short-term history of contents and make it easily accessible for clients.

As earlier stated with requirement 6, the structure of the memory must assist in fusing data of different modalities. In addition to the spatial structure of the memory, the temporal dimension introduced with the short-term history provides another means for data fusion cues.

**Requirement 8. Temporal Data Fusion Interface**  The memory system must give easy access to data-fusion-relevant temporal relations and events of memory contents.
Maintaining a short-term history includes the task of selecting what is preserved in history and what is forgotten. Ideally, this decision is based on properties of the data so that e.g. a model for rehearsal- or activation-based forgetting can be implemented. Also the structure of distributed representations introduced with requirement 5 needs to be regarded.

**Requirement 9. Forgetting**  Forgetting must be supported by the memory in a flexible way that regards properties of the data and their connections.

Acting in a group of people especially requires focussing on important aspects while disregarding less important inputs. A working memory must aid in implementing these attention mechanisms on the humanoid robot based on the memory properties introduced so far. Temporal events can provide valuable information on interesting aspects as well as spatial relations of memory contents.

**Requirement 10. Support for Attention Mechanisms** A visuospatial memory system must support attention mechanisms by providing adequate cues for important events while being able to filter irrelevant information for the clients.

Cognitive processing for solving complex tasks like the robot-to-group interaction in the HUMAVIPS scenario typically requires two different cognitive approaches: Top-down processing generates higher-level knowledge from sensory inputs whereas bottom-up systems generate expectations for the lower-level processing stages based on existing higher-level knowledge. Both approaches are required for solving complex tasks.

**Requirement 11. Top-Down and Bottom-Up Processing** A memory system must provide means to perform either top-down or bottom-up processing.

Many of the perception and processing results in robotic systems are probabilistic and can be interrupted in the temporal dimension if sensors currently cannot perceive objects. E.g. the face of a person in the HUMAVIPS scenario can only be tracked if the robot looks at the face and the person looks at the robot. Nevertheless, having a good estimation where the face currently is helps planning actions even if it is currently not visible.

**Requirement 12. Uncertainties and Missing Data** The memory system must aid in representing and improving uncertainties and dealing with interrupted perception results.

### 3.3. Technical Requirements

Having stated the conceptional and functional requirements on a visuospatial memory, several technical requirements on a software system that implements the desired features exist which need to be addressed. This section summarizes the latter demands.
First to be mentioned is the fact that the memory system serves as a central integrational component of the overall system architecture. Having this in mind, the memory can act as a single point of failure in two ways: it can fail and it can degrade the system’s performance if it is not capable of handling the high load. For both reasons a distribution of the system is desired.

**Requirement 13. Distribution Support** The memory system must be distributable to reduce the possibility of a complete system failure as well as to provide means of scaling with increased load. Distribution must respect that some client processes (e.g. a process providing vision results) may introduce a high load only for certain data types while disregarding the rest of the memory.

Besides the general ability to distribute the system, it must be possible to integrate the memory system with different robotic platforms to be of general applicability. Nearly all of these platforms use different middlewares. Hence, the software must abstract from these middlewares by providing middleware-independent interfaces for inter-process communication.

**Requirement 14. Middleware Abstraction** The memory system must abstract from middlewares that are used for distributing the system to be applicable for different robotic platforms.

The idea of being able to represent data of different types in one system includes respecting different resolution requirements per data type, especially on the temporal axis. A memory system must deal with this requirement to a) provide a history in the required resolution for every kind of data and b) decrease the load on the system and its clients by reducing the temporal resolution for data types where this is not required.

**Requirement 15. Variable Temporal Resolution** The temporal resolution of the maintained short-term history must be adaptable to the requirements of individual data types stored in the system, regarding the resolution and system load.

Robotic systems often require a means of synchronization between different asynchronously processing components [e.g. HW10, p. 5]. As the memory is the central component that integrates the distributed computation of results through representations which are shared between processes, it forms a basis to integrate synchronization mechanisms on a data level.

**Requirement 16. Component Synchronization** A means of synchronizing distributed and in parallel processing components must be provided by the memory system.

The experiences with the aforementioned systems (see chapter 2) have shown that using event-based approaches [Fai06] for memories yields many benefits for robotic systems. These benefits are also desired for a more specialized memory system. Hence, the system must expose an event model that reflects the more specialized structure of the memory.
**Requirement 17. Event System** The visuospatial memory must expose an event-system that copes with the increased complexity of the memory compared to more general systems. This especially includes the spatial and temporal relations of data.

Efficient retrieval of information from the memory is a key aspect for its usability. On the software side this requires a query and filter language that is able to handle the structural dimensions, like location or timing, introduced by the memory.

**Requirement 18. Query and Filter Language** The memory must provide a query and filter language that copes with all conceptual dimensions of the memory and enables clients to efficiently select relevant information from the memory, either by using a query operation or filtering event streams.

From the programmer’s point of view, the software interface of the memory system must be easy to use and still expose all relevant features. Moreover, experiences with the Active Memory have shown that programmers easily tend to ignore features or conceptual ideas if these concepts are not enforced by the interface. Hence, the software interface must reflect and require the features and intended use cases of the memory in its interface.

**Requirement 19. Conception-Aware Software Interface** The software interface of the framework must reflect the conceptual ideas of the memory by requiring their use.
4. EgoMemory Framework

This chapter introduces the framework developed in this thesis, called EgoMemory. After a brief introduction of all concepts, the details of each are explained in the following sections. Because some concepts are influenced by implementational restrictions, the implementation is described in parallel with the conception.

The EgoMemory framework implements a working memory model influenced by the visuospatial sketchpad as described in section 1.2, called spatiotemporal working memory. A brief illustration of the concepts realized in the framework is given in figure 4.1. As depicted here, EgoMemory is a memory using a spherical coordinate system with its origin in the humanoid’s head. Data is stored in different layers that allow customized storage and retrieval mechanisms for different data types. Each record of the memory is stored at a location in space around the robot in a specific layer. To express relations and dependencies of data, links between records can be created by means of a graph structure. A short-term history is maintained using a fixed resolution for each layer and utilizing a prediction mechanism to keep memory contents valid. Besides queries, an event model is implemented that allows access to memory contents using subscriptions. The system is implemented as a C++ library with distribution in mind so that each layer can be executed in a different process if necessary.
Before going into the details of the concepts and implementation issues, a summary of basic assumptions is helpful to understand the following descriptions. Moreover, general principles used while designing the system are introduced.

The \textit{EgoMemory} framework assumes that every piece of data, called \textit{memory record}, has a location in space, either specified through a full 3D coordinate or less precise like directional information. In addition, every record has a type that reflects the data it contains. Each record is stored in a layer of the system which accepts its type and provides specialized operations for it. Records change over time, either because the perceived world changes or processing generates new results. These changes are maintained in a history which is associated with each record. The existence of one memory record through the stored history is called \textit{life line}. The relation of a record to other records in the memory at the same instant of time is expressed through hard links.

A basic design principle for the framework is already visible in this rough description. Whenever there was a decision to either represent information in the overall structure or in the specialized data itself, the overall structure was chosen if this does not prevent the general applicability of the system within the conceptual ideas. This principle directly reflects requirement 19.

From a software engineer’s point of view the system has to be extensible to support the intended specializations. Otherwise every addition would introduce changes in the core system, easily bloating the software. Therefore, interfaces must be provided defining extension points for the intended specializations like layers and the infrastructure must allow the easy addition of behavior through these extension points. The following detailed description of concepts and their implementation mentions extension points that are implemented to support the integration of new behavior.

\section*{4.1. Representation of Space}

The first basic decision to make in a spatially organized memory is how to represent locations in space. The concept of egospheres by Albus proposes polar coordinates with the observer at the center of a spherical surface. Peters et al. [Pet+01], Peters, Hambuchen, and Bodenheimer [PHB08] and Ruesch et al. [Rue+08] have shown that a system working with these conventions can solve robotic tasks successfully. Nevertheless, the work by Peters et al. [Pet+01] on egomotion compensation has also shown that distance information is required to keep contents on the egosphere valid if the robot translates. Besides this technical necessity for complete depth information, requirement 2 requests the handling of 3D coordinates while allowing less precise information. Conforming with the aforementioned general design principle, positions in the \textit{EgoMemory} framework are represented using full 3D spherical coordinates. This does not prevent clients from registering data without complete 3D position information. If only directional information are available, the distance component of the spherical coordinate can be set to a uniform value. If only ordering information are present, the distance can be used as an index.
Using polar or spherical coordinates yields several benefits for registering information in the memory as well as for querying them and provides a solution for requirement 1. Encoding images perceived by a camera mounted on the head of the robot is easy if the camera is centered in respect to the head. Then, positioning images in the memory only involves adjusting the angular components of the coordinates (assuming a simple setup where the camera looks straight ahead). Sound localization results e.g. can be expressed with the azimuthal coordinate component. Typical interaction scenarios for humanoid robots involve changing communication behavior based on the position of possible interaction partners. Especially the distance and the information whether a person is in front of the robot play important roles. Both information can easily be gathered from the spherical coordinate of a record using only a single component.

One basic decision has to be made when using a humanoid-centered spatial memory organization: where to place the coordinate origin and orientation. Regarding the typical setup of sensors for humanoid robots, an expedient solution is to center the coordinate origin in the head of the robot. Otherwise all sensory information must be transformed for registration and retransformed for the aforementioned applications. For the orientation of the coordinate system two solutions are possible. Both can be implemented using the EgoMemory framework. One possibility is to keep the coordinate frame aligned with the head even if the head turns while the other solution is to keep it aligned with the torso of the robot. The first solution simplifies the registration of sensory inputs while the second may have benefits for motion planning and simplified processing to compensate egomotion [Rue+08, p. 964]. All experiments with the framework so far were made with a head-aligned coordinate system as depicted in figure 4.2.

4.2. Integration of Specialized Data: Record Types and Layers

The most fundamental extension point for EgoMemory is the ability to include custom data representations and mechanisms to store and access these representations in a
Figure 4.3.: The basic MemoryRecord class defining the interface for new data types in EgoMemory. It consists of an ID, a position, and three timestamps. Two exemplary classes SpecializedRecordA and SpecializedRecordB extend MemoryRecord and carry additional data.

way that copes with the features of each specialized representation. The ability to include these specializations directly reflects requirements 3 and 4.

To extend the system with custom data types, EgoMemory defines an abstract base class called MemoryRecord which represents the minimal set of information a record must provide (see figure 4.3). Each record in the system has an ID that uniquely identifies the record in the whole system. The ID is a 3-tuple consisting of the layer’s name in which the record is stored, a layer-specific ID, and a time step (see section 4.3 for information on the time step). A SphericalCoordinate expresses the record’s position and several timestamps give information on the record’s life cycle (see section 4.3.2). Records with specialized data are represented by subclasses of MemoryRecord.

Giving the ability to represent specialized data does not solely cope with the complexity of managing these data types. Depending on the data, different storage mechanisms may be necessary or desired. Besides managing data in memory, solutions like using relational or document-oriented databases may yield benefits, e.g. by providing versatile indexing and query operations. Also ideas like using neural networks, e.g. Hopfield Networks [Roj96, chapter 13] for associative storage and retrieval should ideally be possible with the framework. These requirements were addressed in this thesis by separating EgoMemory into distinct layers, each layer acting as a single database for specialized data types with the usual actions. Figure 4.4 visualizes a simplified version of the Layer interface that has to be implemented in order to extend the system with new storage mechanisms. As visible in this diagram, the basic data manipulation operations on layers are create for adding a new record to a layer, update to

Figure 4.4.: A simplified interface for a layer which has to be implemented for extending the system with new storage mechanisms.
change an existing record, and remove for deleting a record (regarding 
\texttt{forgetLifeLine}
see section 4.3.1).

Layers are organized in a hierarchy. From the conceptional point of view this introduces a formalization of data dependencies. Figure 4.5 visualizes the conceptual ideas behind layers. In typical robotic systems, features are calculated on the signal-level sensory inputs like images and in later processing stages symbolic data may be extracted from these features. This makes the feature data dependent on the sensory data and the symbolic data dependent on features and, directly or in turn, also on the sensory data. Layers express these static features of dependency for the system runtime, disregarding individual dependencies of records. Dependencies of individual records can be expressed through linking described in section 4.5.

The implementation manages extension points like layers and their specialized data types with a global registry. This concept is further explained in appendix A.

### 4.2.1. Implemented Layers

During the course of this thesis two basic layers have been implemented, addressing general requirements of feature-level and symbolic data. The feature-level layer stores records directly in-memory providing fast data access and storage but without swapping or specialized indexing.
The symbol-level layer is based on the database mongoDB\textsuperscript{1}. mongoDB is a document-oriented database which uses a binary derivative of JSON, called BSON, to represent documents. In contrast to Oracle Berkeley DB XML used for the Active Memory (see section 2.1), mongoDB runs as a separate server process and supports scalability features like auto-sharding, map-reduce and replication. The server is written in C++ and can be accessed over the network. Drivers exist for many target languages. mongoDB forms an interesting alternative to DB XML because it supports advanced indexing structures like 2D geospatial indexing for retrieving the nearest records based on a 2D location, which may become practical for efficiently indexing the polar angles of the spherical coordinates. A second, highly discussible, advantage of mongoDB is its representation format BSON. Many sources argue that the complex process of generating and parsing XML documents degrades the overall system performance [e.g. Nur+09], whereas JSON or BSON provide a faster solution.

Appendix\textsuperscript{2} gives more details on the implementation of both layers.

4.3. Representation of Time: History Model

Working with sensory inputs and symbolical hypotheses about the perceived world includes updating and changing these memory contents over time. Hence, the contents of memory have a temporal dimension because the world in which the robot acts changes and thus the internal knowledge changes while the robot acts. Representing these changes in the memory and making them accessible yields benefits for processing strategies that are capable of improving their results based on the temporal dimension of inputs and for selecting interesting events which are defined by temporal changes. An integration of a history into a working memory framework thus fulfills requirement \textsuperscript{7} In general two distinct aspects of representing a history must be regarded for cognitive systems. On the one hand, a history can represent the knowledge of the cognitive system at each instant of time resulting in a model where past contents of the memory must not be changed to preserve what the system knew at each time. On the other hand, algorithms may exist that can correct the knowledge of the past later on. To represent this kind of information, experiences in the past need to be modified, but simply replacing these past memory contents destroys knowledge about what the system knew at each instant of time according to the aforementioned aspect of history. \textit{EgoMemory} represents knowledge according to the first mentioned aspect by only allowing changes at the most recent time.

Besides the conceptual advantages of maintaining a history, the management must be computationally tractable and respect the required temporal resolutions of each data type and its application in the overall system as stated in requirement \textsuperscript{15} For these reasons, \textit{EgoMemory} maintains a history with fixed temporal resolutions. Each layer has its own update frequency which controls its lifecycle. As a result history is decomposed into time steps of fixed length. The transition from one time step to another is called time step change.

\textsuperscript{1}http://www.mongodb.org

23
The general system of the *EgoMemory* history preserves the last modification state in a time step before the change. Multiple changes within the current time step are thus not preserved. Only the currently active time step can be modified and past time steps are immutable.

![Figure 4.6: Lifecycle of a memory record through history. Only the last state of the current time step is preserved as history information. The number on the time axis represent the time step index.](image)

Figure 4.6 visualizes these basic ideas of history representation. At one time a client component inserts a record called A into a layer. This layer is currently in time step 1. After some time the time step of the layer changes to 2. Because there was no modification of A in the ending time step, A is preserved in history with the state after insertion for time step 1. In time step 2 the same client performs a modification of A, resulting in a record A'. Short time later another component also modifies A now resulting in a modified version A''. A'' is the last state of A before the next time step change, so A'' is the preserved state of A in the history for time step 2. In time step 3 no modifications occur, resulting in A'' also being the preserved state for this time step. Time step 4 contains the deletion of record A. The resulting state in the history for time step 4 is that record A does not exist in this time step anymore. The dotted green line depicts the *life line* of the record through the stored history.

The described system will not preserve the state A' in the layer’s history as it is overwritten within the same time step. If all changes of records are required in the history either the changes must be maintained in the record data or a higher resolution of the layer has to be chosen.

From a technical point of view, having fixed intervals and a notification mechanism (see section 4.7) presents a means of synchronizing the system as it was stated in requirement 16. For the realization of this concept, the *EgoMemoryId* (see figure 4.3) contains a time step in addition to the layer name and a unique ID. A record is hence only uniquely identifiable in time when all three values are given. On the other hand, this ID structure allows to easily access the whole life line of one record by ignoring the time step part.

Explicitly managing the notion of time also includes providing strategies for temporal collisions. These may occur e.g. if a client reads a memory record, decides that it has to be deleted and instructs the memory to perform this operation. If the time step of the layer has changed between reading the record and giving the deletion instruction, the client would modify the past time step. For this reason many operations accept
IDs with an unspecified time step part to select the most recent record of a life line. Such an ID is in a state called INCOMPLETE.

4.3.1. Forgetting

The ability to store a history of memory states introduces the need for a special operation that removes parts of the history to form a short-term system with limited capacity. Forgetting is a client operation in EgoMemory. Providing this operation on the client-level gives the flexibility claimed by requirement 9 because memory clients have full access to all memory features including using several layers at once for their computations and they can be parametrized externally. If forgetting was an integrated function of the memory instead, it would have to be performed by every layer where only limited knowledge about other layers of the system is present. Moreover, the computation which records to forget at a client-level does not degrade the layer performance and no requirements were identified making it necessary to include forgetting into the processing of layers.

Forgetting in EgoMemory is implemented on the basis of record life lines. The special forgetting method forgetLifeLine of layers (see figure 4.4) removes the record itself and all occurrences of this records in time steps before the given record’s time step. This model assumes a very symbolic approach of representing records in the layers and probably does not cope very well with more subsymbolic layers. Further research how to maintain forgetting for these kinds of layers is required.

The form of forgetting mentioned above is responsible for removing contents of past time steps in the memory and is the only client operation that is allowed to change records in the history of layers. Besides this operation, a second form of forgetting, or more precise attention mechanism, may be required in systems which removes contents from the memory that did not receive a client-side update for a specified amount of time or based more versatile cues for attention (requirement 10). These contents are kept in the most recent time step through the history operations on time step changes. Removing them hence occurs in the current time step and does not change the history. As a consequence, such an attention mechanism can be implemented as normal memory client without the necessity for a special operation.

4.3.2. Timestamps

While the general history model works with time steps as basic unit of time and uses them to represent all operations, it may not be sufficient or convenient for clients to deal with these virtual units. Therefore, memory records are additionally equipped with three different timestamps (see figure 4.3) based on the system’s clock:

**Creation Time** Indicates the time at which the record was first stored in the memory. This timestamp is preserved over time steps and update operations. Hence, it gives a measure for how long a record is already maintained and tracked in the memory.
**Update Time** Indicates the time of the last client-based update on the data. On creation of a record it is set to the creation time. This timestamp is also preserved through history operations. Clients can use this e.g. to implement an attention mechanism which removes records that have not been updated for a long time. Therefore it is one premise to fulfill requirement [10] through the attention mechanisms mentioned in the previous section.

**Insertion Time** Indicates the time at which a record was transferred to the new time step. Every discrete time step of history representation changes this timestamp. Hence, it represents the time at which the prediction or correction of the current record took place (see section 4.4). On first creation of a record in the memory it is set to the creation time. This timestamp can serve as an initial cue for forgetting processes.

### 4.4. Prediction and Correction: Dealing with Missing Data and Uncertainties

Having fixed intervals for representing a history introduced the problem of handling missing data for time steps. A client may only be able to deliver data every few time steps of the layer or at certain times perception may not be able at all because the state of an object is hidden to the sensors. Not dealing with these cases would disturb the life lines of records and make accessing the history complex. To cope with this problem, *EgoMemory* uses a predictor-corrector approach [WB95]. At every time step change all records are transferred to the new state using either prediction or correction. These mechanisms create new records for the next time step, thus continuing the record life lines. Prediction of a record takes place if no client modified the record in the ending time step whereas the correction mechanism is called if a client supplied new data by updating the record. The algorithms to predict and correct are highly application dependent, hence form an extension point of the system.

![Figure 4.7.](image)

Figure 4.7.: Extended version of figure 4.6 introducing the predictor and corrector calls at time step changes. \( P \) indicates that the prediction mechanism was used, \( C \) stands for the correction mechanism.

Figure 4.7 illustrates when and how prediction and correction take place. Correction is used after update operations in a time step and prediction in all other cases.
Using the prediction-correction pattern yields benefits for *EgoMemory* beyond ensuring the existence of a record in each time steps. Processing results are often expressed with uncertainties either grounded in the processing itself or the noisy sensor inputs. Egomotion of the robots constantly invalidates old results for the current situation if the update frequency of percepts is low. The prediction-correction pattern provides a *means to ensure data validity* in the memory, e.g. through compensating for egomotion until new percepts arrive from the processing modules (see Peters et al. [Pet+01] for basic equations) or by utilizing tracking algorithms like the Kalman-Filter [WB95] that reduce the overall error based on a statistical noise model. Hence, the pattern provides a model to meet requirement 12.

![Figure 4.8.: Technical realization of the predictor-corrector pattern in *EgoMemory*.](image)

For the technical realization the prediction and correction steps are combined in one class because both steps are usually highly coupled (e.g. sharing the same matrices in a Kalman-Filter). For the sake of brevity the class is called *Predictor*. As shown in figure 4.8, every layer of the system has one *Predictor* that is assigned on application startup. Whenever the layer performs a time step change, it first calls the *Predictor*’s `startPrediction` method. This call can be used by predictors to initialize variables that should be constant for all records in this prediction-correction cycle. One example for the application of this method is a head rotation correction. The predictor can calculate the rotation difference since the last time step change and apply it to all records for the ending time step. As a result, the prediction or correction of all records is consistent even if the computation takes some time. To differentiate between prediction and correction for a given record, the layer passes a flag to the predictor’s `predict` method which is called for every record of the layer in the ending time step. Afterwards, the predictor is informed about the end of the time step change using `stopPrediction`. This hook method can be used in a similar way as described for the start method.

As one can imagine, *Predictors* are called often, linearly scaling with the number of records in a layer and the desired history frequency. Hence, their processing must be fast. Moreover, predictors should not be used to replace real clients of the memory. Both aspects are fostered by restrictions placed on the allowed operations. Predictors always predict or correct one record and must return a new one. Thus, complex operations like splitting one record into multiple hypotheses that possibly use many resources are not allowed. In general, predictors should be able to perform their task with the least amount of external knowledge even though they have access to a restricted set of memory features. E.g. an egomotion correction needs a way to be informed about the motion of the robot that may be part of another memory layer. If a lot additional knowledge is required, chances are high that the process is better suited as a real memory client having access to the full memory features and without compromising the layer’s performance.
Using the predictors at every time step instead of an on-demand calculation of prediction results reflects the need of providing a global system synchronization mechanism through the memory. Clients interested in processing states with a fixed frequency can intercept the prediction and correction results through the subscription model described later. This would not be possible if predictors were called only on demand resulting in an environment that is hard to use with event-based methods. Another aspect is that the constant application of predictors makes the memory system’s performance more predictable because the load on the system caused by prediction is constant assuming a more or less constant number of records and links in the layer. With an on-demand evaluation strategy the load caused by prediction would depend on how clients use the system, making it much harder to scale computational resources appropriately. Finally, using the constant evaluation approach keeps the system in a consistent state at every time making it easier to recover on fault for layers that use a persistent database.

The default predictor installed for every layer simply duplicates the record to predict or correct. Specialized memory record classes should implement a copy-on-write mechanism [Wik10a] if copying their data is expensive.

**4.5. Hard Links: Distributed Representations**

One major advantage of using memories for cognitive systems is the notion of having representations that are shared between all components of a system [Wre08]. This enables components to gather and enrich information in parallel, directly providing their results for all other components of the system which can in turn integrate these results into their processing [HW08, p. 4]. While extending or refining existing records of the memory with new processing results is one solution for this problem, it does not take advantages of the proposed EgoMemory structure. Many processing results used to create or reform contents of the memory (either in a bottom-up or top-down manner) are based on other information found in the memory, probably in other layers than the generated result. Explicitly marking which information was used to generate a result or improve it yields further information for other components that can be in turn integrated into their processing. For example, one component can generate hypotheses based perception results in a general manner and a second component later assigns more domain-specific interest ratings to these hypotheses based on the perceptions used to create them. Copying the source data (e.g. the perception results) into the improved record would have the general disadvantages of duplicating information and may even be impossible because of incompatible data types. Besides expressing these data-driven dependencies, it may also be necessary to express relations between different entities in the memory. In general, a way to express relations between memory contents is needed to fulfill requirement 6 and simplify the implementation of bottom-up and top-down processing approaches (requirement 11).

EgoMemory provides a structure called hard links to meet these requirements. Hard links relate a record of the memory to other records with a user-definable semantic information about the link. This generates a directed graph of connected records.
forming a distributed representation. More formally speaking, a hard link relates one source record with exactly one target record using their IDs and has a meaning information.

Figure 4.9 visualizes the hard link concepts. Links respect the hierarchy introduced by the layers and form an instantiation-level way of expressing data dependencies in addition to the static dependency expression through the layer hierarchy (see section 4.2).

While hard links create a graph based on the memory structure using a 1:1 association of records, sometimes a content-driven approach for linking information may yield more benefits. In chapter 6, some ideas are presented how such a concept could look like.

From the implementational perspective hard links introduce complexity in the system through coupling the otherwise independent layers. An explicit synchronization is required to keep the linking consistent, including different aspects:

1. Conceptually, a solution is required that defines what happens if the target of a hard link is deleted.

2. Layers can have different time steps and their processing is not synchronized. Links spanning over different layers hence theoretically exist in the history of the source record and the history of the target record with different resolutions. Different time steps can lead to inconsistencies if the link is created or removed. One history may contain this change for an earlier time step than the other one.

3. Distribution may result in links where the target layer resides in a different process than the source layer. Processing changes in both layers over the network may introduce latencies which can severely degrade the system’s performance if
a complete locking is used to perform operations on the source and target layer in a synchronized way.

For aspect 1, the solution in EgoMemory is to remove the link if the target is deleted. This decision reflects the basic idea that hard links create a graph in the classical sense where edges in a graph always have a target. As a consequence, hard links can only be created if the target record exists.

The two other problems heavily originate in the complexity introduced by the required synchronization between layers. Having a distributed system in mind, synchronization effort must be kept as small as possible to ensure the overall system’s performance. Therefore, a complete locking as described in aspect 3 is unsuitable. Consequently, EgoMemory does not guarantee the validity of a hard link at the most recent time. This restriction enables maintaining hard links in the source layer and processing changes of links synchronized with the time step changes of this layer. Therefore, the target record of a link is represented in a time step invariant manner using the INCOMPLETE state for the target ID (see figure 4.10). The actual steps needed to maintain the links are tightly integrated into the processing logic performed at every time step change. The next section describes them in detail.

![Figure 4.10.: Representation of hard links in EgoMemory.](image)

### 4.6. Operations on Time Step Changes

Before describing the performed operations, it is necessary to know how synchronization between layers is performed. Synchronization uses the event-based paradigm. Every layer notifies every other layer of which records were created and which records

![Figure 4.11.: The event-based synchronization interface of layers.](image)
were deleted. This enables the layers to reject the creation of hard links if the target record does not exist and their deletion if the target is removed. Figure 4.11 visualizes the interfaces used to communicate the synchronization changes. The Middleware used for distributing these changes must ensure that events are distributed in the order they were raised.

The following description of the processing logic performed at every time step change refers to the two layers that were implemented. In general, the Layer interface does not make assumptions about how and when synchronization with other layers takes place or how prediction is used but both implemented layers use the same mechanism described here. There are certainly differences for subsymbolic layers.

The basic idea is that every layer maintains a list of currently (in the most recent time step) existing records of every other layer. Figure 4.12 visualizes this system. Assuming a layer $A$ that is used by a client. If a new record $R$ is inserted into another layer $B$ of the system, $B$ immediately informs $A$ about the new record through the event mechanism. $A$ inserts $R$ in its internal mirror (otherContents) of other layers. Afterwards, a client creates a link in $A$ that points to $R$. Short time later, $R$ gets deleted in $B$ and this operation is propagated to $A$, too, but enqueued there. At the next time step change of $A$ the enqueued change is processed which results in the deletion of the link and the entry in the mirror.

Figure 4.12.: The synchronization system of the implemented layers visualized for an exemplary use case. Layer $A$ is the analyzed layer and $B$ represents an external layer. On insertion of record $R$ in layer $B$ this addition is propagated to layer $A$ through the event system and $A$ inserts $R$ in its internal mirror (otherContents) of other layers. Afterwards, a client creates a link in $A$ that points to $R$. Short time later, $R$ gets deleted in $B$ and this operation is propagated to $A$, too, but enqueued there. At the next time step change of $A$ the enqueued change is processed which results in the deletion of the link and the entry in the mirror.
synchronization event was not processed immediately by A, clients would often have to deal with errors while creating links, because A would reject the creation of a link targeting e.g. at R. Moreover, this type of synchronization is cheap as it only includes adding an ID to a set. On the other hand, removing links based on synchronization events is a costly operation as it involves accessing the database of the layer. For this reason synchronization of removal events and as a consequence removing hard links is processed synchronously with the layer’s life cycle while changing the time step. Because hard links respect the layer hierarchy this synchronization only needs to respect layers below the own position in the hierarchy. As a result lower-level sensory and feature-level layers, typically requiring a high history resolution, only need to synchronize with a limited number of other layers further reducing the costs of synchronization.

![Figure 4.13.](image)

The integration of this synchronization as well as the overall operations performed at a time step change are visualized in figure 4.13. A time step change starts with correcting the knowledge about links with a target in other layers by deleting all links with a removed target (broken links). This is still done in the ending time step, hence finally correcting the knowledge about this time steps. Afterwards, every record of the ending time step is selected and its prediction or correction for the next time step is created. As links are managed with time steps in the source layer (to have a history of links), the existing links of a record need to be recreated for the new time step. This is an easy task performed in the next step of the diagram, because it only involves increasing the time step of the source record ID by one. While the described mechanisms only require to be informed when a record is created and
removed in the target layer for synchronizing hard links, other layer implementations may need additional knowledge about prediction and correction of preexisting records. Therefore, a time step change ends with sending a cumulated synchronization event for all prediction results. Besides the functional requirements for sending this event, it also increases the fault tolerance of the system. If a layer missed the original creation synchronization event for a record, it gets knowledge about this record at the next time step change of the foreign layer.

This synchronization system ensures the consistency of the ending time step, which will be preserved in the history, with all information received from other layers so far. The process has an asymptotic runtime of $O(\#\text{records in current time step} + \#\text{links in current time step})$ for each time step change assuming a maximally linear generation of iterators for records and hard links in the current time step.

### 4.7. Subscription Model

One fundamental design principle of EgoMemory is to provide access to its contents using the techniques of event-based programming. As already stated in requirement 17, the event system must deal with the complexity introduced by the concepts of EgoMemory. Besides aforementioned synchronization events, EgoMemory defines two types of events of interest for memory clients: events concerning record changes and events concerning hard links.

#### 4.7.1. Subscriptions on Records

The subscription model for memory records must deal with all meaningful operations on memory records defined in the previous sections. As a result, events reflecting these types are defined and clients can subscribe to them:

- **CREATED** Raised directly after a new record was stored in a layer.
- **UPDATED** Raised directly after an existing record was updated in a layer.
- **REMOVED** Raised directly after an existing record was removed from a layer.
- **PREDICTED** Raised during time step change when a record is transferred to the new time step by prediction. This happens if a time step ended without a client update, also including the first time step change directly after creation and even if directly in this time step there was an update.
- **CORRECTED** Raised during time step change when a record is transferred to the new time step by correction. This happens if a client has provided an update during the ending time step and the ending time step was not the creation time step of the record in the memory.
- **FORGOTTEN** Raised if a record is forgotten. If a forgetting call on a layer removes several records of the same life line, the events for the life line are ordered by time step ascendingly. Hence, older records are notified before younger ones.
These events directly reflect the domain operations on memory records and they are raised by each layer. CREATED, UPDATED, and REMOVED inform about record changes in the currently active time step of a layer, PREDICTED and CORRECTED inform about changes made during changing the time step of a layer, and FORGOTTEN informs about changes made to the already preserved history of a layer. Clients can filter the defined events by subscribing only on a subset of types and by restricting the event stream based on properties of records including layer name and specialized content.

4.7.2. Subscriptions on Hard Links

Subscriptions on hard links reflect the available domain operations in a similar way to subscriptions on records with these types:

**LINKED** Raised directly after a new hard link was created in a layer.

**UNLINKED** Raised after a hard link was removed.

Hard links do not change through prediction or correction, only their target and source records change. Also links are tightly coupled with their source record and removed when the record is forgotten. Hence there is no necessity for events on prediction and correction. The same is true for an update event as the semantics of a hard link cannot be changed.

Filtering of hard link events is supported based on the type of the link, the complete source record’s properties including its contents, and the layer of the target. Using properties of the target record for filtering is not possible because links and their subscriptions are maintained by the source layer. As explained earlier (section 4.5), the source layer only knows the IDs of target records in foreign layers to reduce the costs of layer synchronization. Hard link events are raised by the source layer of a link with knowledge only about the target IDs. As a result, it is not possible to perform matching on target contents because the full target record is not available in the event.

4.7.3. Software Realization of Subscriptions

The general model of subscriptions implemented by EgoMemory follows the principle of document-oriented databases. Records and links are the documents of the EgoMemory and clients are informed about changes on these documents if the documents match their subscription specification.

The realization of the subscription interface through publisher and subscriber classes in EgoMemory is an implementation of the observer pattern [Gam+07, pp. 293 sqq.]. Figure 4.14 visualizes the defined interfaces for subscriptions. The upper part of the class diagram visualizes the memory-internal parts used to implement the observer pattern, hence represents the subject. Every layer is equipped with two publishers to differentiate the two domain topics records and hard links. The publisher classes
Figure 4.14.: Interfaces and classes used to realize the subscription model of *EgoMemory*. The upper part of the class diagram displays the classes used by layers to form the subject of the event architecture whereas the lower part contains the classes used at the client level (observer part).

abstract from middlewares to support distribution of the system. The lower part of the diagram illustrates the client side of the observer pattern. Filtering the notification stream from the subject is supported through the subscription classes that define filter criteria. In the observer pattern’s language they define the *aspects* which a client wants to be informed about [Gam+07, pp. 298 sq.]. The subscriber interfaces use separate methods for each different operation as it may become necessary to provide different information for certain operations. Adapters exist for providing stub implementations of these methods or for transforming them to event classes reducing the required effort to implement a subscriber, making the advantages of both approaches available.

Still unanswered is the question how a client actually creates a subscription to one or more layers as the layer classes itself do not provide an interface for subscription. This involves a more general architectural decision about the memory. Having a system separated into different layers may have the consequence that programmers do not treat the system as a whole. This in turn reduces the power of the system. In case of subscriptions, many situations are possible where it e.g. may be of interest to be informed whenever something changes at a certain location in front of the robot disregarding in which layer. If there is no abstraction of the whole system, a client must register itself at every single layer making these use cases cumbersome. As a consequence, every *EgoMemory* client has access to a wrapper class for the whole memory system called *EgoMemoryInterface* that acts as the main system interface (see figure 4.15). Besides knowing all layers of the system and wrapping the domain operations for records and links on these layers, *EgoMemoryInterface* provides methods to register subscribers on the system. Such a unitary view of the system is especially important to express spatial and temporal coincidences through subscrip-
Figure 4.15.: The representation of the whole EgoMemory through a single interface.

tions as they intrinsically span over different layers and provide another important cue for attention mechanisms (see requirement [10]). E.g. Schillingmann, Wrede, and Rohlfing [SWR09] introduces a computational model for generating acoustic packages which makes use of temporal coincidences, hence benefiting from a memory system with abilities to formulate these subscriptions.

4.7.4. Typical Subscription Modes for Clients

Having event types of different semantics and temporal characteristics gives the opportunity to identify modes of how subscriptions on these events can be used. Clients with realtime requirements, that are interested in every change on a record immediately, will typically use the CREATED, UPDATED and REMOVED events of the system. If a client is instead interested in events with a synchronized frequency and possible improvements through prediction and correction, it will use the PREDICTED and CORRECTED events instead of UPDATED. This is essentially the model to meet requirement [10].

4.8. Query Language and Interface

Knowledge that is stored in the memory systems needs to be made accessible for clients. The retrieval of relevant information while disregarding unimportant parts is a fundamental task in cognitive systems, or more general in every data retrieval system, that needs to be supported by a memory [BYRN99]. While the aforementioned subscription model presents one way for client processes to be informed about the memory’s contents, it does not present the correct model for every use case. In addition, a classical request-reply based operation is required that represents the appropriate retrieval mechanism if a client needs certain kinds of information infrequently. For both models a way to express which information is relevant for the client and which is not is required. This specification is called query. For subscriptions the retrieval task is to filter the continuously changing information stream according to the more or less static query specifications. The retrieval task in a request-reply based model is called ad hoc, evaluating varying query specifications against a more or less static set of information [BYRN99]. In any case the formal mechanism to express the query must cope with the complexity of the represented data. In case of the EgoMemory a query must support the memory dimensions location, time and linking.
Expressing a query using a formal language instead of programming language features has a long tradition in computer sciences and yields several benefits for the system from the engineering perspective. Queries expressed in formal languages can become a configuration property of the system instead of being hard-coded, making the system more flexible. Moreover, a human-readable form simplifies their development.

Several mature query languages exist. The next section briefly describes a selection of them and evaluates their applicability for EgoMemory. This evaluation especially includes how well these languages fit the document-oriented nature of EgoMemory where no schema can be assumed for query operations on all layers, because records may itself encapsulate a schema-free document.

4.8.1. Desired Queries and Existing Query Languages

Before describing three selected query languages, some examples of typical queries shall illustrate what a query language must be able to deal with for EgoMemory. Simple queries based on the static record structure are the basic features that need to be easily representable, e.g.:

\[
\text{Select the record from layer } x \text{ with ID } i \text{ and time step } t. 
\]

or including shortcuts for the most recent time step:

\[
\text{Select the record from layer } x \text{ with ID } i \text{ in the current time step.} 
\]

Also using the spherical coordinate structure of the memory must be easy, e.g. in human-robot interaction:

\[
\text{Select all records in layers } a \text{ and } b \text{ that are in front of the robot.} 
\]

Besides these basic operations, layers should be able to provide custom query operations that are tailored to their data types with content-based operations (respecting requirements \( \square \) and \( \square \)), e.g.:

\[
\text{Select all records that contain a saliency image with a compact region of high saliency.} 
\]

Including the temporal dimension of the memory is another aspect to regard. For example sometimes records are interesting if they indicate a sudden change in one value while at other times records should contain a stable value (requirement \( \square \)):

\[
\text{Select all records in the current time step containing a value } x \text{ that is } 5 \times \text{ higher than in the previous time step.} 
\]

\[
\text{Select all records with a value } x \text{ that did not change during the previous and next 3 time steps.} 
\]

Including the linking (requirement \( \square \)) of records in queries could also be used to stabilize hypotheses, e.g.:

\[
\text{Select all person hypotheses in front of the robot that are linked to at least 3 percepts from different modalities since 5 time steps.} 
\]
Besides a way to express these queries in a convenient manner it would be desirable if a potential query language can be used for filtering event streams as well as performing request-reply based query operations on the memory. The following languages were considered.

**SQL** SQL, the Structured Query Language, is the most prominent language for many relational database systems, described by several standards. It is a strong typed language that includes operations for inserting, updating, or deleting data, querying databases, and maintaining their schema. SQL’s basic intent to be used for relational databases with a given schema clearly makes it inappropriate for *EgoMemory*’s document-oriented nature. Besides the obvious paradigm mismatch, SQL provides a basic solution to express temporal relations using sub-queries. The expression `SELECT d.* FROM docs d WHERE value = (SELECT value FROM docs WHERE timestep = d.timestep - 1)` would e.g. select all documents with a constant value since the last time step. SQL has the ability to introduce new predicates through user-defined functions using the `CREATE FUNCTION` syntax. [Wik10b; Sql]

**XPath** The XML Path Language is a query language used to select nodes from XML documents. It provides a relatively easy to use syntax to navigate through single XML documents and relies on their tree structure. Hence, it can be used to navigate through memory records if they are transformed to such a structure. Because XPath does not deal with multiple documents it can only serve to formulate expressions that are matched against single documents. Oracle Berkeley DB XML uses this scheme for querying. Temporal changes over multiple documents cannot be directly expressed because of the single document restriction. They need to be formulated through custom predicates or functions which are supported by XPath. The more recent version 2.0 also supports a for-syntax and can generally be described as a subset of XQuery. [Ber+07]

**XQuery** XQuery is a query language designed to extract data from XML documents. As a functional programming language it is capable to extract information and manipulate them. For selecting nodes from documents it relies on XPath. Temporal relations can generally be expressed because XQuery is a Turing-complete language but the expressions easily grow caused by the lack of special features for this purpose. In general, the underlying document model is compatible with the ideas of *EgoMemory*. As stated earlier, memory records can be mapped to XML-like tree structures and XQuery does not make assumptions about static schemas. Custom predicates can be integrated into XQuery as language functions. [Xqu; Boa+07]

Besides the aforementioned paradigm mismatch, most available implementations of SQL are tightly coupled with a DBMS, hence not usable for *EgoMemory*. For XPath and XQuery several stand-alone implementations like XQilla [Xqi] exist that could be used by *EgoMemory*. Nevertheless, all of the presented approaches have several disadvantages in the context of the *EgoMemory* concepts. All languages were designed specifically for a certain data model. Even if memory records of *EgoMemory* can generally be converted to e.g. the tree structure assumed by XPath and XQuery, this
hinders the integration of new records into the system. If an existing engine that evaluates these languages is used, every record must be explicitly converted to the target representation involving the manual coding of such a conversion with impacts like performance loss through the conversion. Moreover, it prevents the use of query systems provided by a layer’s internal database. If no existing evaluation engine was used, every layer would have to implement the specifications of XPath or XQuery on its own. Even XPath already involves a lot of constructions that need to be implemented to fulfill the specification resulting in an increased programming effort for every new layer. Besides these disadvantages, many of the features offered by the presented languages are not necessary to express typical queries on EgoMemory, but special features to simplify temporal expressions are missing.

4.8.2. EgoMemory Query Language

For the reasons presented in the previous section, EgoMemory does not use a pre-existing query language. Instead a more problem-specific language was designed in the spirit of domain-specific languages [e.g. HT08] which is easy to extend with layer specific functionality and has a small set of domain-specific operations. It is based on the light-weight document language JSON to reuse existing parsing engines and inspired by the query system of mongoDB. Before going into details of the language, the main decision how to represent temporal changes is introduced.

Assuming a tree-like structure of the records, or more generally documents, to match by a query, two general solutions of expressing temporal changes are possible: global or local cursors. A cursor selects one instant of time and performs a matching against the specification given for this instant [CG02]. This procedure may be repeated, hence expressing temporal changes through differences of values for discrete time intervals. The cursor is global if an expression is matched against the whole document at every instant of time selected by the cursor. Listing 4.1 visualizes the difference. A local cursor selects instants of time on the attribute level of the document, hence performing temporal matches only for parts of the document and query. While both approaches can express complex temporal relations, the global cursor approach results in well-structured queries which are easier to maintain and evaluate than local expressions. Several languages using this approach exist [RK04; CG02].

In EgoMemory a query is a JSON array of subqueries which in turn consist of a time specification (the cursor) and an expression to matched against the whole document. This is expressed using a JSON object [Cro06]. Listing 4.2 shows the global structure of a query using the JSON notation. The cursor is defined through the key $t$ and the expression to match is specified by the key $q$. Conceptually, a query is matched against every record of a layer and temporal changes are resolved on the life line of

```
Listing 4.1: Global and local cursors
1 global = at time 1 document.a = 20 and at time 2 document.a = 40
2 local = document.a = (at time 1: 20 and at time 2: 40)
```
Listing 4.2: The global structure of queries in EgoMemory

```
1 {"$query": [{"t": 0,  "q": QUERY},
2    {"t": TIMESPEC,  "q": QUERY}
3    ...]
4 }
```

Listing 4.3: Mapping from memory records to a virtual JSON document

```
1 MemoryRecord = {
2   "id": {"layer": "aLayerName",  
3       "id": 123,  
4       "timeStep": 42},
5   "position": {"distance": 123,  
6       "azimuth": 0.5,
7       "zenith": 2.42},
8   "times": {"creation": 123456789,  
9       "insertion": 123456789,  
10      "updated": 123456789}
11 }
```

this record. The query system only returns one record, hence a query consists of a special subquery which specifies this reference record. Restrictions of the reference record are formulated on its life line. In the JSON syntax for queries the reference record is indicated with the time specification 0. As already visible with $query in listing 4.2 EgoMemory uses mongoDB’s $ notation to express predicates (functions evaluating to a boolean value). Moreover, the query is interpreted as a conjunction of every part if not stated otherwise. The reference record only matches the overall query if every query part matches.

Time specifications can either be expressed in terms of time steps (positive values point to the future, negative ones to the past of the reference record) or through the timestamp information stored in memory records. All time specifications must be given relative to the reference record’s respective value for simplifying the evaluation of these records. Especially for timestamp-based cursors, precise values are normally not known and often temporal restrictions like “during the last 3 seconds” need to be expressed. Therefore, the predicates $lt, $gt, $lte and $gte can be used to express the relations $<$, $>$, $\leq$ and $\geq$. For convenience reasons these operators only select record ranges up to the reference record excluding its value. This means that e.g. the expressions {"updated": {"$gte": -1234356}} selects all records on the reference record’s life line with an updated timestamp in the range [ (reference.updated - 1234356), reference.updated ]. Multiple expressions can be combined to further restrict the cursor range. E.g. {"insertion": {"$lte": 4000}, "insertion": {$gte: 2000}} selects all records that were inserted into the memory (by prediction or correction) during the interval [ (reference.insertion + 2000), (reference.insertion + 4000) ].

Matching on the records contents uses the same mechanisms as presented for specifying the cursor. To match values from records the virtual JSON document structure described in listing 4.3 is used. The presentation of specialized record contents in this...
Listing 4.4: Simple queries and their normalization

1 simple = {"akey" : "hasAValue", "anotherKey" : 42}
2 normal = {"$query" : [{"t" : 0,
3   "q" : {"akey" : "hasAValue",
4   "anotherKey" : 42}}]}

structure is layer-dependent.

Having such a structure defined, the question is how to access elements of sub-objects like the azimuth of the position. mongoDB uses the “dot notation” to access elements of subobject [Mon]. For example the expression position.azimuth selects the azimuth in this notation while disregarding any other children of position. In contrast {"position" : {"azimuth" : 0.5}} would not match the document of listing 4.3 because in this notation both subobjects of position are compared as a whole and they have a different number of children. While this kind of differentiation is generally desirable, it requires an additional step of parsing the dot notation, which cannot be performed by a JSON engine. Therefore EgoMemory currently uses sub-objects to represent children disregarding different numbers or types of children. An expression for this purpose is missing. Thus, to select a record with azimuth angle 0.5 in EgoMemory the expression {"position" : {"azimuth" : 0.5}} is used.

As already explained for the cursor definitions, queries support predicates to perform the matching. A predicate is generally given in the form "$name" : [ARGUMENTS], where ARGUMENTS is a list of arguments for the predicate with the desired length. For convenience reasons predicates without arguments can be given without the empty array and predicates with only one argument (e.g. $gt) can have the argument without the array. A helper class is capable of converting these shortcuts to their normalized representation so that layer implementations can always assume normalized predicates. Another convenience representation converted by this class simplifies queries that only affect the reference time step. They can be given in the simple form displayed in listing [4.4] which will be normalized to the second query in this listing. Layers can include operations tailored to their specialized data by defining new predicates. Nevertheless, a small set of standard predicates is required that allows queries on the basic structure of every record reflecting the spatial memory organization. Besides comparisons like $gt, array operations like $in, and arithmetic operations ($minus) this includes a special operation for angle comparisons $insector to easily overcome the \(0 = 2\pi\) border. Most important is the special predicate $ref which selects the value of an attribute from the reference record, thus allowing expressions relative to the reference record like the one given in listing [4.5]. This query selects every record that has a key with name akey and value test and whose predecessor on its life line has a key val with the value of the reference record’s key val2 minus 10. If no predecessor exists specified with a time step based cursor, the query does not match. The same applies for missing keys, either directly mentioned or in $ref predicates.

Both implemented layers use an evaluation scheme for this language that selects an initial set of records based on the reference record subquery. This initial set
is further restricted by each temporal subquery starting with the restrictions that are closest to the reference record on the temporal axis. Like this the set size is constantly reduced and the query evaluation can stop if the set is empty even before all temporal restrictions are processed. Every temporal restriction is a separate query on the database for the mongoDB-based layer.

The described query language is easy to implement for new layers as it supports only a small but highly domain-specific set of operations. On the other hand, it provides an easy way to integrate operations tailored to the layer’s specific data types by providing new predicates. The temporal dimension is directly integrated into the specification allowing a simpler expression of temporal relations than with existing, unspecific features of other languages (e.g. SQL subqueries). Expressions created with this language are matched against existing documents and do not explicitly formulate the selection of a subset from a collection of documents, hence it is also possible to integrate this language for filtering event stream. Therefore, queries given in subscriptions (see figure 4.14) are specified with the same language. Temporal matching of event streams is currently not implemented but certainly some restrictions on the allowed expressions are required to make this reasonable and efficient, e.g. only allowing time step based temporal restrictions to the past of the reference record.

Currently not designed is the way how to deal with links in queries. The hypothesis is that links are entities which are always subordinated to records because of being maintained with their source record. Hence, it should be possible to express most of the relevant information about the linking through predicates for source and target record. Accessing subgraphs of records on the other hand probably needs completely different mechanisms as described with this query approach because in this case multiple interconnected records are returned instead of only one record for the query approach.

As a result, the query system and language contribute to fulfill requirement 18 but further research is needed to make the linking structure an integrated part of the queries and/or to provide specialized accessors for this purpose.

### 4.9. System Architecture

The following sections describe the overall system architecture of EgoMemory that combines the previously described concepts into a coherent system. Special focus is
set on how distribution of the system is achieved which has a high impact on the system’s design.

4.9.1. Global Structure

From a high-level perspective EgoMemory is structured using a layered architecture as visualized in figure 4.16 [Bus+04, pp. 31 sqq.; JBR04, pp. 73 sq.]. A core library (EgoMemory) defines the basic functionality required to set up and use the software and an additional library (EgoMemoryLibrary) provides helpers and utility classes of general applicability, e.g. to implement typical usage patterns. Besides these two libraries, layer implementations exist that rely on the core. All of these parts are packaged as separate shared libraries to allow individual development. The core consists of a utility layer containing e.g. classes for representing coordinates or the registry as described in appendix A and a second layer with domain-specific classes implementing the functionality of the system as described in the previous sections. In the vocabulary of Buschmann et al. EgoMemory is a relaxed layered system because classes from the utility layer are also visible to clients of the EgoMemory.

![Figure 4.16. Global structure of the EgoMemory framework. Every top-level package indicates a separate library.](image)

4.9.2. Distribution Patterns

To support the distribution of the system, EgoMemory uses several patterns to meet requirements [13 and 14]. The Half-Object plus Protocol pattern [BHS07, pp. 324 sqq.; CS95, pp. 129 sqq.] separates an object needed in several address spaces (processes) into interdependent half-objects which synchronize themselves through an internal protocol. Each half-object presents the functionality required in its address space and makes using it more efficient than through a network connection. This pattern is applied to cope with the fact that some layers are heavily used only by one client. E.g. a grabber from the camera may constantly write images to a layer dedicated for this
purpose. In the vocabulary of the pattern, layers are the half objects. They reside in several address spaces that can either be clients using the system (e.g. the grabber component) or specialized address spaces dedicated to host several or all layers of the system, hence forming memory servers. The protocol used by the half objects for synchronization has already been described in sections 4.5 and 4.6.

Besides the ability to integrate parts of the system into clients for gaining performance, the architecture avoids a central server for dispatching requests or synchronizing layers which could become a performance bottleneck. This is possible because of the completely asynchronous operation of every layer without guarantees for consistency at every time (see section 4.5) and the described synchronization approach.

The decision which layer is created in which address space is a part of the system configuration. In any case (same or different address space), accessing a layer should be transparent for the memory client. Proxies [Gam+07, pp. 207 sqq.] on layers are used to achieve the desired transparency. In case of a layer in a different address space, the client accesses the layer through a remote proxy whereas a simple wrapper proxy is created for local communication in the same address space. Figure 4.17 visualizes the distribution ideas.

![Figure 4.17.: Distribution of the EgoMemory framework. Clients access layers through proxies that provide a transparent interface for local and remote calls. Layers can either reside in client address spaces or in a specialized server address space. A layer-internal synchronization protocol keeps the system consistent.](image)

Operations invoked on layer proxies may take long, either because of the operations performed by the layer or the distribution of the system. To avoid interrupting the processing of clients, the Future pattern [JH77], heavily used in the Java Standard Edition [Jav], is applied. Operations return a future object that will hold the result of the operation when it is finished. Before, clients can poll the future for completion or wait on it until the result is available. Like this, clients can either directly wait for the result, use it some time later in their processing, or completely ignore it.
4.9.3. Middleware Abstraction

Communication between different address spaces is performed by utilizing middlewares or directly calling low-level system routines (for simplicity from now on also summarized under the term middleware). To be independent of these communication mechanisms (communication infrastructures) an abstract factory [Gam+07, pp. 87 sqq.] is used to create Middleware-specific implementations of communication objects. It forms another extension point of the system.

![Figure 4.18: The abstract factory to create middleware-specific implementations of communication classes.](image)

Figure 4.18 visualized the interfaces of this factory (CommunicationFactory) and Middleware-specific classes that are responsible for communication. As visible in the methods of CommunicationFactory, several steps are necessary to setup the communication infrastructure for the distributed system. First of all, layers existing in the current address space need to be made available to the rest of the distributed system through method provideLocalLayer. In any case (local or remote) a proxy on every layer of the system has to be created. These Middleware-specific proxies are created by getLayerProxy. Besides the request-reply based layer interface every other communication in the system is event-based. Event-based communication is encapsulate through publisher and subscriber classes influenced by the middleware XCF used in the Active Memory system [Wre08]. Publishers are Middleware-specific by directly calling methods of the middleware to send messages. Hence, they need to be created by the factory through the createXXXPublisher methods. On the other hand, subscribers are only an interface of methods to be called asynchronously from a Middleware-specific port. Therefore, the CommunicationFactory defines methods to register (and unregister) these subscribers.
4.9.4. Configuration

To setup the distributed system, the user must specify which layers exist in the system and how they are distributed across address spaces (processes). This information is defined by the configuration of the memory which in detail defines:

... the address spaces of the system, called *deployment units*.

... the layers existing in the system including their names, hierarchy, types, predictors, and additional implementation-specific options in key-value style. These layer-specific options are e.g. used to set the connection parameters of the mongoDB-based layer.

... the mapping of layers to deployment units. Hence, which layer runs in which process.

... the communication infrastructure used for distributing the system.

In general, a flexible approach based on a naming service for layers and clients is desirable but currently not implemented. Instead, every client of the system must have access to the same configuration (either programatically or through a shared configuration file) and load it at system startup to setup the local representation of the whole system. Figure 4.19 visualizes the classes used to represent the configuration.

![Diagram of classes representing the system configuration.](image)

Figure 4.19.: Classes representing the system configuration.

A configuration is only valid if exactly all layers of the hierarchy are deployed on deployment units. This means every layer of the system must have one and only one machine to be executed on. The method `isConsistent` ensures this requirement.

Having a class representation of the configuration allows to use different sources for the configuration. Besides hard-coding the configuration, it can also be parsed from configuration files of different formats or from a configuration server like the parameter server of Willow Garage’s Robot Operating System (ROS) [Ros].

4.9.5. System Startup

Now that all parts which are necessary to represent the distributed system are introduced, a description of how these parts interact to start and use the system is required. Setting up a usable system and client interface is the responsibility of a
class called `EgoMemoryFactory`. It reads the system configuration, sets up the appropriate communication layer, and provides the local layers before creating interfaces for the client. The tasks that a client needs to perform for setting up the system are illustrated in figure 4.20. After receiving a configuration object from a source like a parser or configuration server, the client configures the `EgoMemoryFactory` with this configuration and the name of deployment unit it will provide to the system. Now that the system is configured, an `EgoMemoryInterface` (see section 4.7.3) can be created and used to work with the memory system. Finally, a call to `shutdown` terminates the system.

A more detailed view of which steps `EgoMemoryFactory` performs to configure the system is given in figure 4.21. The first step to setup the system is the creation of an appropriate `CommunicationFactory` as specified in the configuration. Afterwards, all layers provided by the client’s deployment unit need to be created and connected with the communication infrastructure which includes making them visible to other layers (`provideLocalLayer`). To avoid the necessity for an initial synchronization of layers, the factory now waits until all other deployment units have provided their local layers. If this is ensured, the local layers are started and the configuration call returns after all layers of the system are running, hence providing a fully working system to the client.

The steps performed by `EgoMemoryFactory` to create a new client-level `EgoMemoryInterface` are explained in figure 4.22. Starting with the instantiation of a new interface class, this instance needs to be equipped with knowledge about the existing layers in the system. Therefore, the layer hierarchy is passed in. Moreover, the interface needs to know the `CommunicationFactory` to register client subscriptions. As a last step before returning the new instance to the client, proxies for all layers of the system are passed in from the `CommunicationFactory`. 

![Figure 4.20.: Client tasks to set up the EgoMemory system.](image-url)
Figure 4.21.: Visualization of tasks performed by `EgoMemoryFactory` to configure the system.
Figure 4.22: Visualization of tasks performed by EgoMemoryFactory to create a new client-level EgoMemoryInterface.
5. Evaluation

Several client applications of EgoMemory have been implemented to evaluate the functionality provided by the framework with respect to the requirements, the integration of systems with EgoMemory, and the performance of the framework and its layers. While the first aspects of the evaluation are qualitative, a benchmark set performs a quantitative evaluation.

5.1. Visualization Component

As a basis for the other evaluation setups a visualization component was created as a first prototypical client application of EgoMemory, called EgoMemoryView.

Figure 5.1.: The layout of the visualization component for EgoMemory with two distinct views. The left view displays the memory contents in an egocentric perspective whereas the right one provides a global view. This screenshot shows results from the tracking experiments described in section 5.2 with two persons visible to the robot (different colors). Every person is tracked through several markers with position and orientation for every marker. The coordinate system at every record and its color are added through a custom record visualization.

The foundation built on by EgoMemoryView is the 3D location of memory records in EgoMemory. Records are visualized at their location in space around the coordinate origin using two perspectives. Figure 5.1 shows the basic layout of the visualization component with the two views. The left perspective displays the memory contents in an egocentric way giving the programmer the ability to understand what the robot sees e.g. with its cameras mounted on the head. The right view provides a global
perspective with a free camera as typical for many 3D visualizations. Moreover, it displays the frustum of the left view and the global coordinate system for better orientation.

The visualization component is registered as a record and hard link subscriber on all actions of the memory. It only displays the most recent state of the memory. Graphical toolkits usually can only be accessed from a single UI thread. Therefore, special subscriber classes have been developed that maintain a queue of events received from the memory. They are available in the EgoMemory library for other applications. Figure 5.2 visualizes the conceptual ideas behind this solution. Both subscribers are registered at the memory and a task in the UI thread processes the queued events and transforms them into actions on the visualization.

![Figure 5.2: Conceptual overview of the event processing logic for the visualization.](image)

Figure 5.2: Conceptual overview of the event processing logic for the visualization.

For visualizing links one problem was identified. As shown in figure 5.1, links are displayed as arrows. For this purpose the locations of the source and the target record need to be known but the hard link events only contain the source record. Hence, the visualization component must perform a manual correlation of hard links to their target records which were received on a different stream – the record subscription.

Additional information on how the visualization component is structured to be useful beyond the evaluation can be found in appendix C.1.

5.2. Simulation of Tracking Data

The second scenario was used as a first complex and integrated qualitative test of the system, especially the prediction mechanism. A dataset was recorded through a Vicon\(^1\) tracking system where Nao was used as a guide in a museum. Therefore, the robot was sitting on a table in front of a painting and waiting for visitors. If visitors approached, the robot started to explain the painting, including moving its arm to point at regions of the painting and turning its head towards it. Except from these movements the robot remained static at the initial position. Both, the robot and the

\(^1\)http://www.vicon.com
visitors were tracked by the Vicon system. The visitors had rigid markers on their head, back, shoulders, elbows and hands (see figure 5.1 for the tracking results of visitors). The robot had one marker on its body and one on its head. Tracking took place with a frequency of 50 Hz and recorded the 3D location of the markers and their orientation.

Having the position of the robot and the visitors including orientations, allowed to transform the coordinates of visitors into an egocentric coordinate system centered in, and aligned with the robots head as proposed for the EgoMemory framework. The simulation system performed this transformation for all markers of the visitors and stored every marker as a record in a layer of the memory. Moreover, the orientation of the head was managed in a second layer with a single record at position (0, 0, 0) containing a matrix with the orientation. A third layer was used for person hypotheses which were created based on the marker records. All layers used the in-memory implementation and operated with a frequency of 10 Hz. Simple time-based forgetting processes were installed for every layer truncating the history after 4000 ms, scheduled every 2000 ms. The visualization component was employed to evaluate the results, and the whole simulation operated in one process.

A thread replayed the recorded tracking data and inserted them in the memory with the ability to simulate a slow perception process by dropping frames for the visitor markers but not for the head rotation. This system resembles the Nao's real restrictions where encoder results like the head rotation can be read with 50 Hz or 100 Hz frequency (Nao version dependent, defined by the so called DCM cycle) but perception usually takes much longer. Dropping frames for the visitor markers, e.g. by updating these markers only every second, clearly showed errors if the head of the robot rotated. Markers started to “jump”, as expected, with the frequency of the perception results. Hence, a key experiment performed with the simulation was to test performance of the prediction / correction mechanism to correct these errors. Therefore, a predictor was installed in the marker layer which compensated for the head rotation. During the call to startPrediction this predictor calculated the rotation of the head since the last time step change using:

\[
\begin{align*}
\text{dAzimuth} &= \text{last.azimuth} - \text{current.azimuth} \\
\text{dZenith} &= \text{last.zenith} - \text{current.zenith} \\
\text{last} &= \text{current}
\end{align*}
\]

Each record was afterwards predicted by updating its current position with these equations:

\[
\begin{align*}
\text{record.azimuth} &= \text{record.azimuth} + \text{dAzimuth} \\
\text{record.zenith} &= \text{record.zenith} + \text{dZenith}
\end{align*}
\]

The distance component remained unchanged. If a correction occurred, the position of the record remained unchanged due to the lack of an explicit correction mechanism in this approach.
Besides the simple problem expression with spherical coordinates in this task, the application of the rotation compensating predictor clearly improved the validity of the memory contents. The “jumps” of marker records were much smaller and most of the time only reflected the changed positions of the visitors which were not modelled by the prediction algorithm. On the other hand small inaccuracies and noise in the tracking of the robot’s head orientation were now visible in a constant correlated movement of all marker records at the framerate of the layer, hence at every predictor call. With the selected prediction mechanism the offset of the marker records from their ground truth position scales linearly with the distance to the head.

In addition to the test case for the prediction system, several other proof of concept memory clients have been implemented in the simulation environment. One client generated person hypotheses based on the marker records in the center of all known markers of a person so far (visible in figure 5.1). The person hypotheses were linked to their markers. A person hypothesis was generated if more than five markers of a person were found and removed if less then four markers were available. This process verified the functionality of the event system for building higher-level hypotheses based on other memory contents and the possibilities to generate hard links.

A second process notified whenever a person entered or left the view of the robot by defining a sector on the azimuthal dimension of the spherical coordinates. This process needed to maintain a list of persons who were visible at the moment for identifying when a person left or stayed in view for every update event. A temporal query (as shown in listing 5.1) could easily remove the necessity for the client-side list maintenance task but is currently not possible due to the lack of temporal matching in event streams.

A third memory process was implemented that used the link structure to assign a validity rating to the person hypotheses. As a stub implementation, the number of linked markers was used as the rating. Therefore, the process was registered as a hard link subscriber on the person hypothesis layer and updated the validity of a person hypothesis every time a hard link event was received. The implementation demonstrated that such a use case is generally possible with the framework but the required programming logic is cumbersome due to the lack of a specialized way to retrieve or even be notified with complete subgraphs of the linking structure. Listing 5.2 displays which steps were necessary to implement this use case. Resolving the linking

Listing 5.1: Exemplary queries to detect when persons enter or leave the view

```json
start = -20 * Pi / 180
end = -start
enter =

{"$query" : [{"t" : 0,
              "q" : {"position": {"azimuth":
                               "$insector": [start, end]}}},
             {"t" : -1,
              "q" : {"position": {"azimuth":
                               "$insector": [end, start]}}}]}
```
Listing 5.2: Using the linking to change record properties

```cpp
// same method used for unlinked
void ValidityRater::hardLinked(
    const HardLinkSubscription &subscription,
    MemoryRecordPtr sourceRecord,
    const HardLink &link) {
    updateRating(link.getSource());
}

void ValidityRater::updateRating(EgoMemoryId &personId) {
    try {
        MemoryRecordPtr r =
            memoryInterface->getRecord(
                personId.incompleteId())->get();
        boost::shared_ptr<PersonRecord> personRecord =
            boost::dynamic_pointer_cast<PersonRecord>(r);
        assert(personRecord);
        set<HardLink> links =
            memoryInterface->getHardLinks(
                personId)->get();
        personRecord->validity = links.size();
        memoryInterface->updateRecord(r);
    } catch (FutureTaskExecutionException &e) {
        reportError("Validity rating failed: " + e.what());
    }
}
```
structure involves an additional call on the memory to `getHardLinks` (line 20). If properties of the link targets were required, another step would have been necessary to resolve the link target IDs to real records. Each of these steps introduces the potential for concurrency problems because records can be deleted in the meantime from concurrently operating memory processes. This actually is the case if the hypothesis generation removes a person hypothesis which includes sending unlinked events for the remaining marker record. The code above then fails because it cannot resolve the source ID of the links anymore. Hence, a more integrated solution for resolving links or selecting subgraphs is required to simplify this use case.

Another problem visible in listing 5.2 is the requirement to downcast records to their specific representation. This task is inconvenient and only necessary because a generic memory and layer interface is used for different data types. On the other hand, the generic interface gives the flexibility to use the in-memory layer for every data type if no special indexing is required. With a layer interface templatized to return a special data type this flexibility would not be possible, especially with the global wrapper class `EgoMemoryInterface`. A templatization of layers would result in a complex metaprogram in many cases.

### 5.3. Demonstration Scenario Human-Robot Interaction

![Diagram](image)

Figure 5.3.: Architectural overview of the demonstrator. Green boxes indicate modules provided by the Nao SDK accessed over the network. Orange boxes depict modules that grab data from the robot and store them in `EgoMemory`. Blue boxes generate behavior based on the cues found in the memory. The dashed arrows indicate the dependencies of components.

The third scenario developed to evaluate the system involved the Nao robot. Several methods were implemented enabling the robot to start a conversation with a human. This included tracking faces by moving the robot’s head, focussing salient regions if no faces were visible, and greeting and saying goodbye to people approaching and respectively leaving the robot. All of these tasks were integrated based on `EgoMemory` and visualized by `EgoMemoryView`. Figure 5.3 gives an architectural overview of the demonstrator. The upper part of the diagram shows modules provided by Nao’s SDK.
NaoQi. The modules are accessed over a network protocol whereas the rest of the demonstrator is a single application based on the NaoQi module architecture running on a remote computer. Orange boxes indicate modules that are used to access sensory data from Nao, eventually extract features from these sensory data, and store them in EgoMemory. The FaceGrabber module reads face detection results generated by a NaoQi module and stores these detected faces in the memory. To have 3D data in the scenario without having a stereo vision system the size of the detected face patches was converted into a depth coordinate using a fitted function. Appendix C.2 describes this approach more detailed. The full 3D information of the faces were inserted into a face layer of the memory system. A second module called HeadPosGrabber read the joint information of Nao’s neck and stored them in a special layer for this purpose. These information could be used to additionally enable the rotation compensating predictor as described in section 5.2. The third module called SaliencyCalculator calculated the visual saliency on the camera stream received from the robot using the model proposed by Itti, Koch, and Niebur [IKN98]. The resulting saliency map was stored in another layer of the system using a special record type which accepted images. All three layers of the memory system used the in-memory implementation.

The two blue modules on the right side of the diagram generated behavior based on the memory contents. Therefore, both modules were registered as memory record subscribers. TrackingControl implemented a behavior to focus a visible face of the robot and follow the face when the person moved by adjusting the head orientation of the robot. A simple strategy was included to recover if the person moved too fast and the face detection lost the face. If this happened, the controller turned the head in the direction of the last movement of the face. This behavior was performed using the subscription mechanism of the memory with subscriptions on the face layer. If no face was visible at all, the robot turned into the direction of the most salient region in its camera image every few seconds. To select the recent saliency image from the memory the query mechanism was used instead of the subscription mechanism because the saliency images were required less often then they were generated. The second behavior module, called ApproachChecker, greeted and said goodbye to persons approaching or leaving the robot. Faces were used as the cue for this behavior, too. Due to the lack of temporal event matching in streams the memory was frequently polled for records that indicated an approaching or leaving person. The whole logic, e.g. for detecting an approaching person, could be expressed with the temporal query printed in listing 5.3. This query selects every document in the memory which represents a face closer to the robot than 100 cm and which was further away in the preceding time step. Detecting a leaving person could be expressed by exchanging the two subqueries.

The described scenario demonstrates a more complex use case for the memory system. The detection of approaching and leaving persons illustrates how the temporal dimension of the memory can be used to select events that otherwise need to be detected by the components itself, hence simplifying the component code. While the solution with polling the memory for these events generally works, temporal event matching for subscriptions certainly further reduces the client code necessary to perform such tasks.
A more severe problem which was identified through this scenario is the way in which clients have to handle inserting and updating records. The history representation of EgoMemory relies on tracking to have meaningful life lines. On the one hand, it may be possible that clients are not capable to perform tracking at all. In this case an automatic association mechanism in the layers could be used to create meaningful life lines. Imagine e.g. a person detector capable of detecting multiple persons at a time without recognizing them. Assuming that people do not move fast and the algorithm operates with a high frequency, a simple way to build up an association of faces over time could be created by means of distance and direction of motions compared to the last frame. On the other hand, if algorithms can provide tracking results the way they fill the memory with contents usually looks like:

```plaintext
if data is tracked:
   select recent record from memory
   update record contents
   update record in memory
else:
   create new record
```

This essentially describes the task of associating a client-internal tracking ID with the memory ID structure and selecting the create operation for new hypotheses and update for already tracked ones. Both tasks, association-based on metrics and ID association, could be simplified by providing filters on the insertion process of the memory. In cases where the algorithms cannot provide tracking information at all,
a filter can generate associations to previous frames. In cases where tracking information like a tracking ID is available in the data provided by the algorithms, the filter can perform the association from tracking IDs to memory records depicted in the pseudocode above. In both cases this would reduce the client logic to a single operation “provide data”.

If no tracking can be achieved for certain layers at all, most likely signal and feature layers, additional mechanisms are required to access the history of a layer. These mechanisms cannot rely on the life line concept of single records. Instead, the whole layer forms a life line which needs to be inspected.

5.4. Performance Benchmark

Besides the aforementioned qualitative evaluation scenarios, several benchmarks of the two implemented layers (in-memory- and mongoDB-based) were performed. Layers are the building blocks of the distributed architecture and perform or initiate all domain operations of the memory system. Hence, they are the determinant for the overall system performance. The performed benchmarks shall give an insight which applications are possible with both layers and how the layers react to increased load.

The plots for the benchmark results are collected in appendix C.3 including a detailed description of the test.

As a general summary, the in-memory layer is significantly faster than the mongoDB-based layer but does not provide the flexibility of using an arbitrary document as user data with full indexing support. Both layers show the same asymptotic behavior for all tested operations. While inserting records into the in-memory layer is a logarithmic operation, regarding the total number of records in the layer (insertion in C++ std::map, see figure C.4), this behavior is not clearly visible in the benchmark results for the mongoDB-based layer. Nevertheless, the database server manages index structures as well. Hence, it can be assumed that the same asymptotic behavior for insertions exists. Figure C.5 illustrates that the effect of the logarithmic complexity can be neglected for typical use cases as the insertion time nearly scales linearly with the number of inserted records in a previously empty layer. A third benchmark illustrates the influence of the data size on the insertion time (see figure C.6). To generate comparable results the record type of the mongoDB-based layer was used for both layers containing a user-defined BSON document of varying size. The size of this document in bytes was measured in comparison to the insertion time. As expectable, the insertion of the in-memory layer is unrelated to the data size whereas it increases linearly for the mongoDB-based layer. The in-memory layer only manages pointers that do not change with the data size whereas a conversion and network transmission is required for the mongoDB layer.

The retrieval of records by ID is indexed in both layers. Figure C.7 displays access times depending on the layer’s sizes. Once again a logarithmic complexity is visible. For the in-memory layer this is the complexity achieved by std::map whereas the mongoDB documentation gives no information on the complexity of indexed queries.
All these results clearly show that the in-memory layer is significantly faster, especially for the retrieval case with two orders of magnitudes.

A second aspect which was analyzed is how the temporal dimension of each layer influences the processing time. The most important factor is the time required to perform a time step change, limiting the history resolution of a layer. Figures C.8 and C.9 clearly show that this time linearly scales with the number of records and the number of hard links. Moreover, it is nearly unrelated to the history size (figure C.10).

Assuming a target history resolution of 30 Hz, no linking, the default duplicating predictor, records without user data, and only rare client requests, the in-memory layer can handle \(~2500\) records in each time step while the mongoDB layer only handles \(~130\) records for such a resolution. Also in this case the mongoDB-based layer is slower by one order of magnitude. One reason for this is the way prediction is performed. All records of the ending time step are fetched from the server, converted to their class representation, predicted and then reconverted to BSON for storing the prediction result. This includes a high network load and conversion time. The mongoDB server allows to execute JavaScript code, hence performing the prediction through server-side JavaScript code would remove the necessity for the transmission-conversion cycle of records to predict and could speed up the mongoDB-based layer.

Besides the time step changes another important aspect of the temporal dimension is the evaluation of queries with temporal restrictions. The worst-case for the evaluation of a temporal query exists if all temporal restrictions of the query match and thus need to be evaluated. Hence, the benchmark for this aspect generates queries which always match one record but with a varying number of temporal restrictions on the record’s life line. To have comparable data for both layers, the queries operate on the base structure of memory records using the time step in the ID as a controlled restriction. Listing C.1 shows how these queries were generated. The reference ID and time step were randomly chosen using the range of time steps which guarantees a match of the whole query. Figure C.11 illustrates the results of this benchmark. The query time increases linearly with the number of temporal restrictions but is neglectable in case of the in-memory layer compared to the overall query time. The mongoDB-based layer is much faster as it makes use of the index structure for the reference query, whereas no indexing is used by the in-memory layer.

In general, these benchmark results show that the mongoDB-based layer will hardly be able to cope with soft real-time requirements like a target history resolution of 30 Hz. Further improvements are necessary, e.g. the server-side calculation of predictions and corrections. The in-memory layer is much faster and applicable for real-time requirements but lacks a versatile way to make use of indexes for queries.
6. Conclusion and Outlook

The software developed for this thesis presents a new model for a spatiotemporal working memory for humanoid robots. Memory contents are stored using their location in a spherical space around the robot. The system is capable of integrating data of different types and abstraction levels and relations of memory contents can be expressed through a graph structure forming distributed representations. A history is maintained which can be accessed and used to formulate queries with temporal restrictions and a predictor-corrector pattern retains the validity of memory contents. These functionalities are integrated into a distributable software framework respecting the varying requirements of complex robotic systems.

Several evaluation scenarios have proven the applicability of the implemented approaches. The predictor-corrector pattern was successfully used to compensate for egomotion of the robot, and the history and query language were able to select interesting events for human-robot interaction defined through temporal changes of percepts. Besides these functional aspects the event system was tested successfully as a basis of all evaluation scenarios, and creating and accessing the linking structure is possible. Nevertheless, the scenarios lack a real application for the linking. Effectively creating and using distributed representations requires a much more complex system as it could be created for the evaluation.

On the other hand, several enhancements and open issues were identified during the implementation and evaluation of the system which require further research.

Implementing layers for the EgoMemory system has revealed several problems of representing and accessing the data for a) temporal queries and b) efficient lookup of connected records through the linking structure. For representing temporal data, research on temporal databases, either relational or document-oriented, can give ideas for a future realization [JS99; Nor03]. Efficiently accessing the record graph formed through linking has overlap with the research on graph databases [Gi94; AG08] and semantic web technologies like RDF and its query language SPARQL [Bec04; PS08]. Besides giving ideas how to represent data layer-internally, the research in these areas can give hints how to improve the layer interface for clients and extend or simplify the existing query language. Especially research on efficient representations of temporal data for the system is a key factor to improve the performance of the currently implemented layers. With regard to the linking, a second type of links is envisaged which creates a more content-driven association of records. These soft links associate a record with a set of targets specified by a content restriction, hence forming much more dynamic distributed representations. One exemplary use case for this type could be to let the memory automatically tag possible candidates for further
evaluation by client processes. Also the problem of correlating link events with their target records identified for the visualization component needs to be solved.

Regarding the representation of knowledge in the memory several further research questions arise. The spatial layout of the memory requires a way to represent location information which are overlayed with uncertainties. These probabilistic features need to be integrated into the query language and subscription model. A second property of the spatial layout, which is currently not covered by the memory, is shape information of records. Representing information about the shape of records can aid in fusing data spatially by providing intersection tests in subscriptions, resulting in new operations to integrate into the query language (requirement 6). However, these tests introduce a completely new type of complexity in the system by relating records of different life lines. With the current system different life lines are independent except from being linked. Nevertheless, an efficient implementation for correlating records of different life lines is highly required beyond shape intersection tests. Temporal data fusion also requires this correlation and is an open issue with the current system (requirement 8). The existing synchronization system between layers can serve as a basis to allow these types of correlations. Sending timestamps in addition to the record IDs enables layers to perform temporal correlation of other layer’s contents with their own records and additionally sending shape and location information enables spatial correlation. These new features as well as the existing temporal matching have to be integrated in the subscription model for filtering event stream, hence including the complex event processing use case in the system. The integration can take advantages of existing research in this area [Esp; Dek07].

Another uncovered use case is how to handle multiple hypotheses for the same memory content. The system could include mechanisms to support splitting up life lines of records into multiple hypotheses and fuse several existing hypotheses into a single through an explicit model. Hence, a more versatile management of history and queries is required. The current query system needs to be enhanced to support life lines with multiple predecessors and successors.

To solve the problem of inserting data without tracking into the memory, mentioned in section 5.3, the proposed filter structure has to be implemented. Many other ideas can also be realized with this concept which simplify the client processing. Hence, it will likely form a new extension point of the memory system.

Besides the addition of features to the system more complex scenarios are required to analyze how the presented concepts can be used in robotic systems. A set of usage patterns needs to be extracted including aspects like typical subscriptions, processing pipelines, and the generation and application of linked records. Reusable memory processes like forgetting and attention mechanism have to be created to provide a framework easily usable in new systems without having to reinvent the wheel. This includes research on how forgetting can be implemented in a more versatile way as presented with the time-based forgetting process for the evaluation scenarios. The structure of the memory provides many features like linking which can give better cues for forgetting. The same is true for attention mechanisms which remove unmaintained contents from the memory. In addition to cognitive sciences, ideas on forgetting and
attentions processes in *EgoMemory* can also be gathered from research on garbage collection in programming languages where a comparable problem of releasing unref-erences memory areas is addressed [Wil92].

From a more global perspective patterns of integrating the specialized memory system presented in this thesis in a complete memory framework — with other working memory systems, long-term structures, and, for the HUMAVIPS project, a global map of the world — are required.

Despite many new research questions, the memory system created for this thesis is already capable of solving robotic tasks successfully and provides a basis for further research and development on the aforementioned topics.
A. Extension Point Mechanism

_EgoMemory_ utilizes a global registry to make extensions known to the system. Listing A.1 shows an idealized form of the _Registry_ class that is used to register extensions in the system. The registry contains the global knowledge about the registered extensions of a specific extension point which is expressed through the template argument. Using the Singleton design pattern [Gam+07, pp. 127 sqq.] reflects this global system configuration knowledge. E.g. to make a new memory record type known to the system, a new instance of _MemoryRecordFactory_ must be registered in _Registry<MemoryRecordFactory>_ using _addRegistree_ (line 12). The same counts for new layers by registering instances of _LayerFactory_. Registered classes, called _registrees_, are accessible through a unique string using _getRegistree_ (line 21). For initializing layers at system startup, this string is read from a user-supplied configuration. Additionally methods exist to discover all registrees. These methods are not printed in listing A.1.

The question that is not yet answered is how extensions, in C++ shared libraries, are actually registered in the system. Besides manually registering them at application startup, requiring to rewrite the startup code everytime a new extension shall be added, two other approaches are possible in C++: link-time and runtime extension. _EgoMemory_ has integrated support for link-time registration. This means that an extension is available to the _EgoMemory_ system through linking a shared library containing the extension classes to the binary which creates the memory instance. The framework provides macros that use compiler-specific features like GCC’s _attribute_ ((constructor)) syntax [Gcc] for achieving such a registration. Using a link-time integration of components prevents rewriting code but does not remove the necessity to call the linker. Another desirable solution is to discover extensions at runtime using _dlopen_ [Dlo], completely removing the need to call the compiler or linker. While this solution is desirable and compatible with the registry concept, it is currently not implemented.

Figure A.1 displays the defined extensions points of the _EgoMemory_ system. In addition to them, the visualization component (see section 5.1 and appendix C.1) defines an extension point to include new visualization mechanisms for specialized memory records.
Listing A.1: Registry class for extensions

```cpp
template <class R>
class Registry {
private:
  Registry() {}
public:
  static Registry<R> *instance() {
    static Registry<R> *inst = new Registry<R>);
    return inst;
  }

  void addRegistree(R *r) {
    if (registreesByName.count(r->getRegistryKey())) {
      throw invalid_argument("There already is a registree with key \\
                              + r->getRegistryKey() + \\
                              + ");
    }
    registreesByName[r->getRegistryKey()] = shared_ptr<R>(r);
  }

  shared_ptr<R> getRegistree(const string &key) const {
    typename map<string, shared_ptr<R> >::const_iterator
      it = registreesByName.find(key);
    if (it == registreesByName.end()) {
      throw invalid_argument("There is no registree with key \\
                              + key + \\
                              + ");
    }
    return it->second;
  }

private:
  map<string, shared_ptr<R> > registreesByName;
  static Registry<R> *inst;
};
```

Figure A.1.: Extension points defined by the EgoMemory system.
B. Implemented Layers

This appendix chapter describes the implemented layers regarding the way how data is stored and made accessible for clients.

Both implemented layers are subclasses of a base class, called LayerImplementation, which encapsulates the common behavior across both layers. This includes many argument checking routines, subscription and synchronization handling, and a template for the steps performed at every time step change (see section 4.6). The template method pattern [Gam+07, pp. 325 sqq.] is applied and LayerImplementation is a part of the core library to be reusable for new layers.

B.1. InMemoryLayer

The InMemoryLayer stores records and links directly in the memory of the process using no database or file backend. Records are maintained in two indexing structures. One map indexes records by their ID allowing fast access by ID which is the common case for all operations on single records, especially when dealing with hard links. A second indexing structure organizes records by their time step. A map associates each time step with the set of records contained in this time step. This structure is used to achieve a reasonable performance on time step changes where an iteration over all records of the ending time step takes place. Hard links are indexed by their source record's ID reflecting the common lookup case. An indexing by target ID may become necessary if many links are used. The synchronization between layers constantly needs to remove links if the target record is deleted, hence a lookup by target ID is required. Queries are performed by a separate matcher class which allows to reuse the matching code for subscription handling.

B.2. MongoLayer

The MongoLayer maintains records and links in a mongoDB database. mongoDB allows to separate documents into several collections for optimizing batch operations. This layer uses two collections, one for records and one for hard links. Every record of a life line is stored as a single document in the database. Several indexes on the database server are used to speed up record lookup by ID and time step and hard link lookup by source ID and the whole link. Queries are converted into queries on the mongoDB server. Temporal queries require as many database queries as temporal subqueries exist.
C. Evaluation

C.1. Visualization Component

The visualization component \textit{EgoMemoryView} introduced in section 5.1 was designed as a general tool for \textit{EgoMemory}. It uses Qt 4 [Qt4] for setting up the graphical user interface and the Visualization Toolkit – VTK [Vtk] for 3D rendering. Both frameworks are opensource, cross-platform compatible, and widely available. VTK provides a very high-level interface for rendering a wide variety of types, including geometrical primitives, images and 3D surface or point cloud data. Compared to other rendering and game engines in C++ the VTK interface is easier to use.

To be usable as a general visualization tool for \textit{EgoMemory} the component is packaged as a separate library that can either be used in-process or as a standalone application via the memory distribution mechanisms. As a general tool it must cope with the flexibility introduced by the memory. This specifically means to support tailored visualizations of custom data types used in the system. Therefore, an extension point is defined using the registry concept described in appendix A. Figure C.1 shows the interfaces for this extension point. Implementations of \textit{VisualRepresentation} encapsulate the operations required to setup and maintain a specialized visualization of data on a VTK renderer. These representations are created and updated by abstract factories called \textit{VisualMapper}. \textit{VisualMappers} are arranged in a chain of responsibilities [Gam+07, pp. 223 sqq.] which is used to select the instance that visualizes a given record. When a new record needs to be displayed the visualization component calls the \textit{canMap} function of every \textit{VisualMapper} while iterating over the chain until a mapper indicates its applicability for the record. If no mapper returns \textit{true}, the default visualization is used. \textit{VisualMappers} are registered in the registry. Figure C.2 gives an example how the integration of images types into the visualization through the extension point could look like.

Figure C.1.: Interface used to add custom visualizations for special data types.
C.2. Calculating Distance from Face Patch Sizes

The face detector provided by the NaoQi framework returns a squarish bounding box for detected face patches. To obtain distance information the size (width and height are always the same) of this bounding box was used. Therefore, a dataset with two different faces was recorded associating distances with their bounding box sizes. Afterwards, a function of the form shown in equation \[C.1\] was fitted to the average of both samples using the nonlinear least-squares (NLLS) Marquardt-Levenberg algorithm implementation of gnuplot \[WK^+\]. The resulting fit is displayed in figure \[C.3\]. Solving the fitted function to \(\text{dist}\) results in two solutions. The second solution of equation \[C.2\] can be dropped as it returns negative results for the intended input range. Hence, the first solution is used to calculate distances from face patch sizes.

\[
\text{size} = p \cdot \ln(u \cdot \text{dist} + v \cdot \text{dist}^2) + q \tag{C.1}
\]

\[
\Leftrightarrow \text{dist} = -\sqrt{\frac{u^2 + 4v \exp \left(-\frac{\text{size} - q}{p} \right)}{2v}} \quad \wedge \quad \text{dist} = \frac{u + \sqrt{u^2 + 4v \exp \left(-\frac{\text{size} - q}{p} \right)}}{2v} \tag{C.2}
\]
Figure C.3.: Dataset to calibrate the distance calculation from face patch sizes and the fitted function on the average of both samples. The size of the bounding box returned by the NaoQi’s face detector is given in radians as the used angle of aperture in the camera.
C.3. Benchmark Results

This appendix part contains the plots describing the layer benchmark. All plots were generated on an otherwise idle Intel® Core™2 Duo CPU with 3.00 GHz and 4 GB RAM. The mongoDB-based layer used a database server on the same machine that was accessed over the loopback network device. To include the network communication time in the benchmark all measurements were based on the real time (instead of user or system times). The return value of \texttt{times()} was used to calculate this processing time in seconds by dividing through \texttt{sysconf(SC_CLK_TCK)}. Both layers operate single-threaded and the benchmarks were performed by only a single thread. If not stated otherwise, all tests were performed with a minimal record structure only consisting of the required attributes and for the mongoDB-based layer an empty user document. Only the default duplicating predictor was used.

![Graph showing time required to insert records into an empty layer](image)

Figure C.4.: Time required to insert records into an empty layer.
Figure C.5.: Time required to insert 50,000 records in a layer consecutively. After each set of insertions the layer is not cleared. The first measurement for the mongoDB layer is not displayed due to an initial indexing effort.

Figure C.6.: Time to insert 20,000 records depending on the data size. The mongoDB layer record class was used for both layers (InMemoryLayer accepts every record) and filled with a BSON document of varying size. The size of the document was determined using the `objsize` method of `BSONObj`.
Figure C.7.: Time required to retrieve 10,000 random records by their ID (indexed in both layers) depending on the layer size. Records exist only in one time step. The first measurement for the mongoDB layer is caused by the initial creation of an index.

Figure C.8.: Time required for one isolated time step change depending on the number of records in the layer. Averaged over 100 trials.
Figure C.9.: Time required for an isolated time step change in a layer with 1000 records and a varying number of randomly created hard links. Averaged over 100 trials per number of links.

Figure C.10.: Time required for a time step change in a layer with 1000 records averaged over 50 trials. The layer is not cleared after each time step change. Hence, the number of records in the whole layer stored in history constantly increases. The first sample for the mongoDB layer is not plotted because the database initially created an index resulting in a much higher processing time.
Figure C.11.: Time required to perform a temporal query based on the number of life line restrictions in the query. In any case the layer contained 100 records that were predicted in 100 time steps using the duplicating predictor. Hence, every record existed in every time step. Temporal queries were performed with time step based restrictions on the time step attribute of each record. For every number of time step restrictions 100 random queries were generated and performed 10 times. This cumulated execution time was averaged for the plot. The queries were constructed to always match one single record in the layer. Listing C.1 displays the structure of these queries. Please note the different axes for both layers.
Listing C.1: Temporal queries for the benchmark

```json
query: {
  "$query": [
    {
      "q": {
        "id": {
          "id": 24,
          "timeStep": 45
        }
      },
      "t": 0
    },
    {
      "q": {
        "id": {
          "timeStep": 44
        }
      },
      "t": -1
    },
    {
      "q": {
        "id": {
          "timeStep": 43
        }
      },
      "t": -2
    },
    # more temporal restrictions as necessary
  ]
}
```
Glossary

**BSON** Binary JSON, a binary-encoded serialization of JSON-like documents.

**Communication Infrastructure** Adapter for a middleware used to distribute the *Ego-Memory* system.

**DBMS** Database Management System, a program that maintains a database and makes it accessible to clients.

**Distributed Representation** A distributed representation is one in which meaning is not captured by a single symbolic unit, but rather arises from the interaction of a set of units [Eli04a].

**Egomotion** Motion of the robot in the environment. Egomotion causes the sensory inputs to change even if the environment remains static.

**Event** Notification sent by a subject of the observer pattern to interested clients.

**Extension Point** An interface of a software system that allows to add functionality (extensions) without changing the system itself.

**Frustum** A truncated pyramid cut off in parallel to the base. In computer graphics the frustum of a camera defines the 3D area rendered by the camera.

**GCC** The GNU Compiler Collection, a collection of compilers for C, C++, Objective-C, Fortran, Java, and Ada.

**Hard Link** A structure of the *EgoMemory* which relates one record with exactly one other record in the system, also called link.

**IDL** Interface Description Language, a programming language independent way to describe interfaces of software components.

**JavaScript** a dynamically typed, object-oriented scripting language best knows for its application on websites.

**JSON** JavaScript Object Notation, a light-weight standard for a text-based data-exchange format derived from the syntax of the JavaScript programming language.
Layer One database of the EgoMemory system providing storage and access mechanisms tailored to a special data type managed by the system.

Life Line The existence of a record in the EgoMemory system through time steps of the history representation.

Long-Term Memory The permanent memory store accessed after a considerable gap between the presentation of a stimulus and its recall [Eli04b].

Memory Process A software component using the client-level interface of the EgoMemory system.

Memory Record One piece of data stored in a layer of the EgoMemory system and maintained through history operations, also called record.

Middleware Software that connects multiple processes of a potentially distributed system.

mongoDB A document-oriented database using a binary JSON derivative as representation. The server is written in C++ and supports replication and sharding. JavaScript is used as the primary client language, e.g. to create map-reduce jobs.

NaoQi The SDK used on the Nao robot.

Oracle Berkeley DB XML A schema-free, embeddable database for XML documents with query support through XPath expression.

Predictor A part of the EgoMemory system that implements a predictor-corrector pattern to transfer memory records at the time step change to the new time step.

SDK Software Development Kit, a collection of tools used to create applications based on certain software packages or hardware platforms.

Short-Term Memory A part of the human memory in the Atkinson-Shiffrin model which maintains information long enough to use it. Various theories exist about its structure, partially with a unitary view of the STM.

SQL Structured Query Language, a language to express queries on relational databases.

Subscription The registration of clients on subjects to be informed about changes through events.

Time Step A discrete interval of time maintained as history information with a single state in a layer of the EgoMemory system. Also used to express the length of these intervals.
**Time Step Change** A point in time at which a layer of the EgoMemory system preserves its current state in its history.

**UML** Unified Modeling Language.

**Working Memory** A theory about the part of human memory which maintains information in the short term. In the context of this thesis it refers to the description given by Baddeley [Bad03].

**XML** Extensible Markup Language, a set of rules for encoding documents in a structured, machine-readable form.

**XPath** XML Path Language, a query language to select nodes from XML documents.

**XQuery** XML Query Language, a query language for collections of XML documents based on XPath.
Bibliography


[Rob] RoboCup@Home. URL: http://www.ai.rug.nl/robocupathome/ (visited on 08/23/2010).


List of Figures

1.1. The three-component model of working memory .................................. 3
1.2. Visualization of SOAR's representation of knowledge ............................. 7
2.1. The Sensory Egosphere ........................................................................ 9
3.1. The humanoid robot Nao ...................................................................... 12
4.1. Implemented concepts in EgoMemory .................................................... 18
4.2. Proposed coordinate system for EgoMemory ......................................... 20
4.3. MemoryRecord interface for new data types ........................................ 21
4.4. Layer interface .................................................................................... 21
4.5. Conceptual ideas of layer structure ........................................................ 22
4.6. Memory record lifecycle through history ............................................... 24
4.7. Memory record lifecycle with prediction and correction .......................... 26
4.8. Realization of predictor/corrector pattern .............................................. 27
4.9. Hard link concept ............................................................................... 29
4.10. Representation of hard links in EgoMemory .......................................... 30
4.11. Event-based synchronization interface of layers ....................................... 30
4.12. Example for the synchronization system .............................................. 31
4.13. Actions at time step changes ................................................................ 32
4.15. EgoMemory interface ......................................................................... 36
4.16. Global structure of EgoMemory framework ........................................... 43
4.17. Distribution of the EgoMemory framework ............................................ 44
4.18. Abstract factory for Middleware-specific implementations .................... 45
4.19. Classes representing the system configuration ....................................... 46
4.20. Client tasks to set up the EgoMemory system ....................................... 47
4.21. Tasks performed to configure the system ............................................. 48
4.22. Tasks performed to create a client-level interface .................................. 49
5.1. Layout of the visualization component .................................................... 50
5.2. Conceptual overview of the event processing logic for the visualization 51
5.3. Architectural overview of the demonstrator ........................................... 55
A.1. Defined extension points ...................................................................... 64
C.1. Interface used to add custom visualizations for special data types ...... 66
C.2. Integration of specialized data types in the visualization ...................... 67
C.3. Distance calculation from face patch sizes ............................................ 68
C.4. Insertion time with empty layer ............................................................ 69
List of Requirements

1. Humanoid-Aware Coordinate System ........................................ 13
2. Different Location Information Characteristics ..................... 13
3. Diverse Sensory Data Representation .................................. 13
4. Integration of Different Data Abstractions ........................... 14
5. Distributed Representations ............................................. 14
6. Spatial Data Fusion Interface .......................................... 14
7. Short-Term History Representation ................................... 14
8. Temporal Data Fusion Interface ....................................... 14
9. Forgetting ..................................................................... 15
10. Support for Attention Mechanisms .................................... 15
11. Top-Down and Bottom-Up Processing ................................ 15
12. Uncertainties and Missing Data ....................................... 15
13. Distribution Support ..................................................... 16
14. Middleware Abstraction ............................................... 16
15. Variable Temporal Resolution ......................................... 16
16. Component Synchronization ............................................ 16
17. Event System ................................................................ 17
18. Query and Filter Language ............................................. 17
19. Conception-Aware Software Interface ................................ 17
Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig angefertigt und keine weiteren als die angegebenen Hilfsmittel und Quellen verwendet zu haben.

Bielefeld, im August 2010

______________________________

Johannes Wienke