A Concept for Declarative Information Acquisition in Smart Environments

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. André Dietrich

geb. am 17.12.1981 in Kiew (Ukraine)

Gutachterinnen/Gutachter

Prof. Dr. Jörg Kaiser
Prof. Dr. Oliver Brock
Prof. Dr. Andreas Birk

Magdeburg, den 22.11.2016

Abstract

Future smart and intelligent environments are envisioned to be loosely coupled compositions of distributed sensor, actuator, and computation systems, which are able to combine their capabilities, share their information and knowledge, in order to fulfill various tasks autonomously. However, current approaches in these areas do either provide mechanisms for sharing data only or concentrate on the composition of operations/services for predefined problem sets.

This thesis, therefore, firstly analyzes the different notions of data, information, and knowledge to be shared among heterogeneous systems. Based on this distinction, a reference model that is used to relate the previously disparate approaches was developed. It has revealed that most of the concepts and attempts for smart and intelligent environments do only tackle small and restricted aspects.

In contrast to adding a new system to the current state-of-the-art that also operates on one or more of the identified layers, a concept was developed that integrates these technologies from different layers, which allows to combine and access them freely. It treats such compositions of smart information spaces as distributed database, allowing to query, access, and extract all the required information in the desired formats. The developed concept is based on three intermediate steps. At first, cloud-based techniques are applied to form a virtual overlay database and to define a basic organization. In opposition to other approaches, relevant data and information from different entities is therefore not uploaded to a cloud infrastructure but, instead, a cloud is formed by all entities within the smart environment. As a second step and on the basis of SQL-like queries, relevant data for a certain area or task is identified and translated into a precise 3D rigid-body simulation. This local reconstruction of the environment is applied in the last step as a general representation and local knowledge base, from which further representations and information is abstracted. The type of information or representation is also defined within the mentioned SQL-like query. For this purpose, a new and embedded programming language that combines both an SQL-like syntax with new semantics, and declarative aspects with imperative scripting capabilities has been developed.

As a proof of concept, prototypes have been developed for every described step, which can be freely downloaded as open source implementations under BSD-license.

Zusammenfassung

Die intelligente Umgebung der Zukunft, wie man sie sich heute vorstellt, besteht aus einer Vielzahl unterschiedlichster, lose gekoppelter, autonomer Systeme. Dazu zählen Sensoren und Aktoren aber auch Recheneinheiten, die nicht lokal verortet sein müssen. Solche Systeme sollen in der Lage sein ihre Fähigkeiten individuell und aufgabenabhängig zu kombinieren sowie ihr Wissen über die Umgebung und ihre derzeitige Wahrnehmung dynamisch zu teilen. Die aktuellen Ansätze und verwendeten Systeme in sogenannten smarten Umgebungen bieten jedoch nur die Möglichkeit eines dynamischen Datenaustausches in fest vorgeschriebenen Formaten oder konzentrieren sich auf die Entwicklung von Architekturen, die nur auf einen kleinen Problembereich anwendbar sind.

Um eine generische Lösung zu entwickeln, wurde in dieser Arbeit zunächst analysiert, welche unterschiedlichen Formen von Daten, Informationen und Wissen in solchen heterogenen Systemen ausgetauscht werden. Auf Basis dieser Analyse wurde ein schichtenbasiertes Referenzmodell entwickelt, das es erlaubt, bestehende Systeme und Architekturen in Beziehung zueinander zu setzen. Die daraufhin erfolgte Einordnung der wissenschaftlichen Literatur zeigte, dass sich die meisten Arbeiten innerhalb dieser Kategorien nur auf kleine Teilaspekte bzw. Probleme innerhalb von intelligenten Umgebungen konzentrieren.

Das Ziel dieser Arbeit war es nicht, eine weitere spezifische Variante zum derzeitigen Stand der Forschung hinzuzufügen, welches eine oder mehrere Schichten des Referenzmodells abdeckt, sondern vielmehr ein Konzept zu entwickeln, dass es erlaubt verschiedene bestehende Technologien einzubinden und ganzheitlich auf sie zuzugreifen. Hierfür wurde die Metapher einer verteilten Datenbank genutzt, die alle Komponenten einer intelligenten Umgebung einschließt und es erlaubt, nicht nur auf die verschiedenen Daten zuzugreifen sondern auch beliebige Informationen in unterschiedlichen Formaten zu generieren bzw. aus der Umgebung zu extrahieren. Das dafür entwickelte Konzept besteht aus drei Elementen. In der Basis wird ein Cloud-basierter Ansatz verfolgt, der eine grundlegende Struktur und Hierarchie festlegt. Im Gegensatz zu anderen Systemen werden hierbei nicht alle Daten und Informationen unterschiedlicher Systeme in eine Cloud hochgeladen, sondern die Systeme einer intelligenten Umgebung selbst bilden die Cloud. Auf Basis von SQL-ähnlichen Anfragen lassen sich aufgaben- und situationsrelevante Elemente innerhalb der Cloud identifizieren, die in ein präzises 3D Modell der Umgebung übersetzt werden. Eine solche Umgebungssimulation erlaubt es, Systeme im Kontext zueinander zu betrachten sowie erweiterte Analysen durchzuführen und angeforderte Informationen flexibel zu generieren. Die Spezifikation, welche Analysen durchgeführt werden bzw. welche Informationen zu extrahieren sind, wird ebenfalls in der ersten (SQL-ähnlichen) Anfrage definiert. Hierfür wurde eine neue eingebettete und interpretierte Anfrage- und Programmiersprache mit einer SQL-ähnlichen Syntax entwickelt, die deklarative Programmieraspekte mit imperativen Ausdrucksformen verbindet.

Konzeptnachweise wurden für jeden Teilschritt in Form von Software-Prototypen entwickelt. Diese Open-Source-Projekte können frei unter der BSD-Lizenz heruntergeladen werden.

Danksagung

Meinem Betreuer Herrn Prof. Dr. Jörg Kaiser danke ich dafür, dass er es mir ermöglicht hat diese Arbeit zu schreiben. Die Gelegenheit zur Mitarbeit an dem interdisziplinären Forschungsprojekt ViERforES und die Zuarbeit zu KARYON haben mir dabei geholfen, meinen eigenen Standpunkt und meine Sichtweise auf bzw. Herangehensweise an bestimmte Probleme immer wieder neu zu überdenken und zu erweitern.

Ich möchte dahingehend auch meinen Kollegen in der Arbeitsgruppe „Betriebssysteme und Eingebettete Systeme“ danken. Mein besonderer Dank gilt JProf. Dr. Sebastian Zug, der gleichzeitig als engagierter Unterstützer aber auch als Kritiker zugegen war. Die gemeinsame Arbeit an verschiedenen Projekten und die Mitarbeit im „Knowledge 4.0“ Projekt hat mich viel über das wissenschaftliche Arbeiten und Publizieren gelehrt. Mein weiterer Dank geht an meine Kollegen Tino Brade und Christoph Steup für die vielen Anregungen und die Hilfe bei implementatorischen Problemen. Des Weiteren danke ich Dr. Stefan Sokoll und Martin Althuizes für die vielen interessanten Diskussionen.

Weiterhin geht mein Dank an Prof. Dr. Oliver Brock, der mir durch unsere Gespräche entscheidende Probleme und Schwachstellen bezüglich meines Konzeptes vor Augen geführt hat und es freut mich, dass er bereit war, die Begutachtung dieser Promotionsschrift zu übernehmen.

Ich danke meiner Mutter Katharina Dietrich dafür, dass sie mir trotz mancher Schwierigkeiten im Leben alles ermöglicht, mich in allem unterstützt und begleitet hat. Ich danke meinen Großeltern Anneliese und Eberhard Dietrich, die mir immer und in jeder Hinsicht ein großes Vorbild waren. Und ich danke meiner Freundin Sabine Sahr für ihre Geduld, Verständnis und ihre Liebe.

Contents

Figures

Listings

Abbreviations

AmC Ambient Computing

AmI Ambient Intelligence

ANTLR ANother Tool for Language Recognition

API Application Programming Interface

AR Augmented Reality

AST Abstract Syntax Tree

AWDS Ad-hoc Wireless Distribution Service

CAD Computer-Aided Design

CAMUS Context-Aware Middleware for URC System

CAN Controller Area Network

COLLADA COLLAborative Design Activity

CORBA Common Object Request Broker Architecture

CPS Cyber-Physical Systems

CQL Cassandra Query Language

DAvinCi Distributed Agents with Collective Intelligence

DBMS Database Management System

DSL Domain Specific Language

FAMOUSO Family of Adaptive Middleware for autonomOUs Sentient Objects

GIS Geographic Information System

GOLOG alGOl in LOGic

GPS Global Positioning System

GUI Graphical User Interface

HCI Human Computer Interaction

HDFS Hadoop Distributed File System

ID IDentifier

IE Intelligent Environment

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

IP Internet Protocol

IQ Information Quality

IS Information Science

JDL Joint Directors of Laboratories

JVM Java Virtual Machine

LDM Local Dynamic Map

LISP LISt Processing

LOP Language-Oriented Programming

M2M Machine-to-Machine communication

MavHome Managing An Intelligent Versatile Home

MOSAIC fraMewOrk for fault-tolerant Sensor dAta fusIon in dynamiC environments

NASA National Aeronautics and Space Administration

NCAP Network Capable Application Processor

nesC network embedded systems C

NI Network Interface

NoSQL Not only SQL

NRS Network Robot System

ODE Open Dynamics Engine

OGC Open Geospatial Consortium

OpenGL Open Graphics Library

OpenRAVE OPEN Robotics Automation Virtual Environment

OSGi Open Services Gateway initiative

OSI Open Systems Interconnection

OWL Web Ontology Language

PEIS Physically Embedded Intelligent System

PerComp Pervasive Computing

PLUE Programming Language for Ubiquitous Environments

PREDiMAP vehicle Perception and Reasoning Enhanced with Digital MAP

Prolog PROgrammation en LOGique

RDF Resource Description Framework

RDQL Resource Description Query Language

RFID Radio-Frequency IDentification

ROS Robot Operating System

RPC Remote Procedure Call

SensorML Sensor Model Language

SLAM Simultaneous Localization And Mapping

SmE Smart Environment

SOA Service Oriented Architecture

SQL Structured Querying Language

STRIPS Stanford Research Institute Problem Solver

SWE Sensor Web Enablement

TCP Transmission Control Protocol

TEDS Transducer Electronic Data-Sheet

tf transformation

TII Transducer Independent Interface

TIM Transducer Interface Module

TransducerML Transducer Markup Language

TTL Time To Live

UAV Unmanned, Uninhabited, or Unpiloted Aerial Vehicle

UbiBot Ubiquitous Robotics

UbiComp Ubiquitous Computing

UDP User Datagram Protocol

UML Unified Modeling Language

URC Ubiquitous Robotic Companion

URDF Unified Robot Definition Format

URL Uniform Resource Locator

VM Virtual Machine

VTK Visualization Toolkit

XML eXtensible Markup Language

1. Introduction

> "... if technological advances were simply a continuous, linear outgrowth of past technology, we might expect future computing environments merely to comprise more laptops possessing more power, more memory, and better color displays. But the world of information and technology doesn't always evolve linearly. Radical new uses of portable information technology are on the horizon ... "
>
> —Mark Weiser on ubiquitous computing 1993 [221]

From an economic point of view, the near future is likely to look as the scene depicted in Fig. 1.1. Robots and other smart entities enabling ambient intelligence applications are on the run. The Joint Research Centre[0] has therefore launched a series of studies analyzing the prospects and success of technological innovations and their market developments. This series is devoted to give an overview on important technological areas in which it would be important for the EU industry to remain, or to become competitive in the near future. In the report on robotics [79], it is predicted to have tripled the sales in year 2025 (up to 66 billion $) in nearly all fields, in which especially service and personal robots are accounted to have the lion's share.

A similar scenario is envisaged for the field of embedded systems. According to [76], the number of microprocessors (as an indicator for the penetration of embedded systems in our daily life) was already two times as big as the human population before the year 2000. In 2008, there were some 30 embedded microprocessors per person in developed countries [67]. And this development does not seem to reach a saturation point [179].

On the one hand, we will be surrounded by a myriad of heterogeneous systems, probably not directly visible to us, which measure and monitor us as well as aspects of our environments and take control of more and more (minor and simple) parts of our everyday life. On the other hand, there will be highly sophisticated and autonomously acting robots that share with us the same operational areas and interact with us "naturally" and without boundaries. These are complex compounds of actuators, multimodal sensor systems, and software that allow them to perform a variety of very complicated tasks.

This development seems to be pretty straightforward and all too natural, especially if we take a look at the history of robots and their "linear" evolution (see, therefore, the next section). But as pointed out by Mark Weiser, the father of ubiquitous computing (see the main quotation), technology does not always evolve linearly. Thus, the actual revolution was launched (not that radical so far) when new ideas from smart environments and ambient intelligent systems started to penetrate into research on robotics (Sec. 1.3).

[0] It is the in-house science service of the European Commission for providing independent scientific advice, market analyses, and support to EU policy. `https://ec.europa.eu/jrc/ipts`

Figure 1.1.: View on future robotics and ambient intelligent environments

1.1. A Short Overview of the "Linear" History of Robotics

Even though the development of "real" robots was only made possible with the invention of the programmable computer, the idea of robots and the notion about their capabilities is much older [36]. The term robot was derived from the Czech science fiction play "Rossumovi Univerzálnı' Roboti" from 1921, which can be translated as knowing or understanding (also cognitive) universal laborers. Originally, it was used to designate artificial organic people, not robots or better to say automata in the modern sense or in the sense of Fritz Lang's gynoid "Maschinenmensch" from 1924 (Fig. 1.2.3).

Some of the earliest examples in the history of robots were based on purely analog circuits, such as Elmer and Elsie [97]. These two robots with the shape of a tortoises were constructed between 1948 and 1949 by William Grey Walter and were capable of photo taxis (a locomotion that occurs when an organism moves in response to the stimulus light), to find their way to a recharging unit. The primary objective of Elmer and Elsie was to study complex behaviors that can arise just from small number of brain cells. They were also regarded as the first biologically inspired autonomous system.

The Unimate is commonly considered to be the first industrial manipulator (cf. [196]). It was used at General Motors since 1961 to weld and handle die castings. It was intended to replace human workers where tasks are either dangerous, harmful, or tedious. PUMA (Programmable Universal Machine for Assembly) was one of the most successful successor manipulators of the Unimate with six degrees of freedom and only consisting of hinge joints (and so far away from the original idea).

In addition to production scenarios, there are also further industrial applications such as the robot assistant LiSA. It was developed by Fraunhofer [192] and allows it to overlap

human and robotic working areas. An even tighter interaction between robots and humans shall be realized with so-called CoBots (Cooperative roBots), as it was proposed by the JAHIR-Project (Joint-Action for Humans and Industrial Robots), for example [252]. The aim is combine human skills with robots' strength and precision to fulfill tasks they could not solve on their own. Therefore, JAHIR focuses also on observing and understanding of non-verbal communication.

In some cases, assistive robots are also wrongly classified as CoBots, although they only provide a single or a set of services. Examples are the Fraunhofer Care-O-Bots [88] (which operate since 2000 as guides in the Berlin Museum for Communication), the Wakamaru (intended to provide companionship to elderly and disabled people [232]), or even the military LS3 (Legged Squad Support System) that is primarily used to squad equipment up to 180 kg [240]. Another example of a service robot, in this case also called domestic robot, is the Roomba. It performs its vacuum cleaning service without any user intervention and it is controlled by a few simple algorithms (iAdapt Responsive Cleaning Technology, see [233] for more information).

A more complex autonomous robot than the Roomba, but with similar tasks (i. e. collecting dust), is the NASA Mars Rover (Fig. 1.2.13) [33]. Because it has to operate in a hostile environment and an overturning or getting stuck cannot be fixed anymore, it has to plan each of its operations carefully. Although, exploring the surface of another planet seems to be a quite challenging task, there are search and rescue robots on earth whose environment and the demands placed upon their skills are far more challenging. Golem Krang [251], also known as the MacGyver, is a humanoid robot intended to perceive the environment in a way that it can perform rescues by using and combining tools that were found in the local surroundings. This requires an extraordinary awareness and knowledge about the things within the environment — about how to use them, and how to plan actions — which are close to human capabilities. But this is currently nothing more than an ambitious objective.

1.2. Robotic Evolution

The so far presented historical outline is also shown in Fig. 1.2. It is used to depict the commonly described three evolutionary steps in robotics in conjunction with their task- and environment complexity. Starting from simple automata such as the "Digesting Duck" by Jacques de Vaucanson or the "Musician" and other dolls by Pierre Jaquet-Droz's from the 18th century (see Fig. 1.2:11, 12), these automata can be regarded as the zero generation, although their programs could also be adapted due to a change in the gear wheel or roll mechanisms. As a result, the musician doll could even play different pieces of music on a custom built organ by really pressing the keys with her fingers. (See also [120] for a complete overview of the history and prehistory on programmable machines.)

The dots within Fig. 1.2 represent different entities on a quantitative level according to their tasks and their operational environment. While the environment complexity ranges from fixed, well-known settings with constant environmental conditions up to dynamically changing and unpredictable environments, the task complexity covers required algorithms and, thus, also appropriate environmental representations starting from very simple programs and representations going over to highly complex cognitive and learning systems. A simple pick-and-place task in a dynamically changing household environment (Wakamaru Fig. 1.2:6) requires more complex sensing and algorithms than in an unobstructed and well-

known industrial production environment (PUMA Fig. 1.2:3). Certainly, some positions might be changed or adapted, but the linear relationship between task and environment will still be present, which was also surprising for the author, who initially had expected a more scattered plot.

In general, the first robotic generation can be characterized by repeating only fixed sequence programs, intended to accomplish simple pick-and-place tasks or point-to-point operations. Reprogramming, re-teaching, or simply re-configuring is required to adapt to changes of the environment or to alter the task. Sensor systems (in contrast to generation zero) can be integrated, if there are slight inaccuracies within the environment that might affect the working accuracy negatively.

The second generation of robots is adaptive to varying situations (including inaccuracies) and small changes of the environments. Although, they still execute series of preprogrammed operations, they are also able make minor corrections and adjustments of these operational steps. All necessary parameters are gathered with the help of sensing devices that measure different aspects of the surroundings.

Third-generations robots are characterized as smart or intelligent, but there are different opinions according to the degree of intelligence related to different fields of interest. As described in [143], third-generation robots possess a human-like intelligence with a learning ability to adapt autonomously to changing environments. While the basis for these abilities is defined as high sensitivity, in fact they can perceive multidimensional information about the surrounding from a large amount of locally available sensor systems and analyze it in real-time. Or as emphasized in [59], the basis for intelligent behavior lies in the ability of adequate world modeling. All learning, searching for solutions and decision making is firstly applied multiple times in the virtual representation, rather than directly in the real one.

1.3. Robotic Revolution

It seems to be that there is a linear evolution in robotics. More and more advanced systems appear in various areas, capable of solving more and more complex problems within environments of growing complexity. This development goes hand in hand with new kinds of sensor systems, algorithms, and actuators, but also with traditional "autarchic robot design patterns" (as criticized in [151]):

Robots must be fully autonomous. Robots can cooperate with humans or other robots, but they must carry out assigned tasks relying exclusively on their own sensors, actuators, and decision processes.

Robots must operate in non-structured environments. Robots must adapt to environments that have not been purposely modified for them.

As also criticized in [30], in traditional systems, each robot would have to explore and build its own maps of the environment. Without means of creating live and global maps of large environment, there is a duplication of efforts in exploration and the amount of sensor information processing. And a new robot, introduced to the same environment, will also duplicate all the efforts of its predecessors, making the system inefficient.

But, as already mentioned, there are new ideas that started to penetrate robotics and also the concept of the robot itself. Thus, the real revolution currently takes place between the

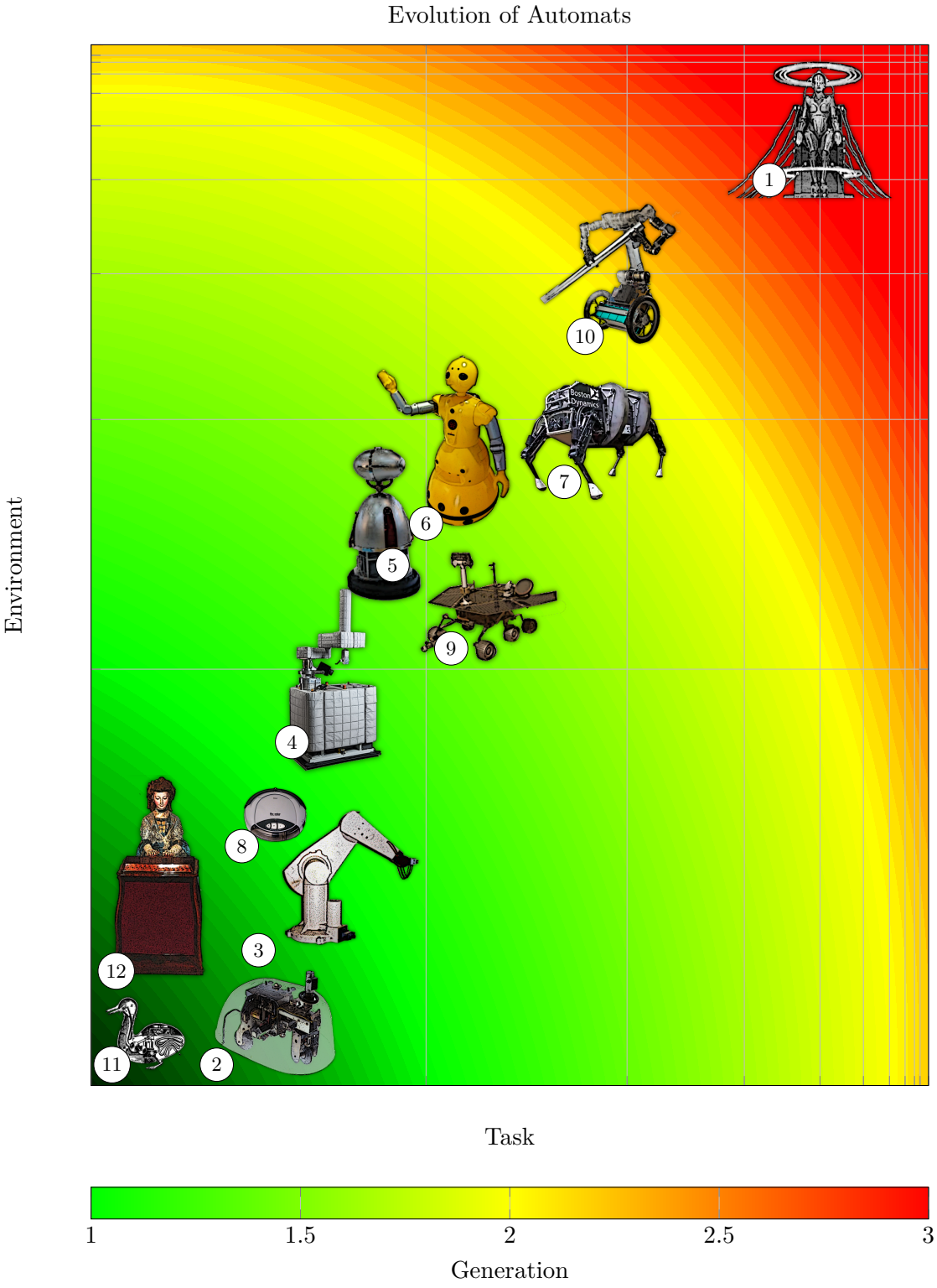


Figure 1.2.: Fritz Langs “Maschinenmensch”[1], Elmar[2], PUMA 560[3], LiSA[4], Care-O-Bots[5], Wakamaru[6], LS3[7], Roomba[8], NASA Mars Rover[9], Golem Krang[10], “Digesting Duck”[11], “The Musician”[12]

present robotic systems drawn in Fig. 1.2 and in Fig. 1.1. A robot cannot be seen anymore as isolated, physically situated, and embodied entity adapting itself to a world tailored for humans. Instead, the robot itself becomes part of a larger ecology of heterogeneous systems, other robots, external sensor systems, "smart" objects, etc., which are able to communicate, get along with heterogeneous information (e. g., numerical, sub-symbolic and symbolic), share their knowledge, environmental perception, and further capabilities. In such scenarios, instrumented or smart environments overcome inherent limitations of mobile platforms and robots offer a mobility dimension unknown to smart environments.

Imagine that the Volksbot in Fig. 1.1 could access the services of the elevator without pushing a button, and obtain a map for the new environment from the systems that reside on that floor, with the newest information on moving humans and other obstacles along with the position of the target person (gathered from available cameras or smart phones). And imagine that this map could further be represented in a format that could be directly applied for localization and navigation based on the local laser scanner; or it simply informs me through my tablet that the packet is for me to pick up...

Although, that is easy to imagine, it does not seem to be accomplishable with today's technologies and applied methodologies.

1.4. The Frog Problem

In 1959 Lettvin published a pioneering study about the physiology and anatomy of the frog's eye Rana Pipiens [140]. Before that, most scientists agreed that an eye transmits a copy of all occurred stimuli to the brain. But Lettvin revealed that a frog's retina already performs some kind of feature extraction. It automatically analyzes an image according to local variations of light intensity, moving edges, standing contrasts as well as a measure of illumination, and it transmits such kind of abstracted information/patterns to the frog's brain which, in response, releases different behaviors:

[1] Scene from Fritz Lang's Metropolis — restored authorized version with the original 1927 orchestral score (124 min.), licensed by Transit Films on behalf of the Friedrich-Wilhelm-Murnau-Stiffung, Wiesbaden, copyright 2002 by Kino International Corporation, New York, USA.

[2] Image number: 10323893_H (4), Credit: Science Museum/Science & Society Picture Library source: `http://www.sciencemuseum.org.uk/images/NonSSPL/10323893.aspx`

[3] Photo from the Integration and Engineering Laboratory at UCDAVIS under PD US ED, source: `http://iel.ucdavis.edu/projects/imc/Hardware.html`

[4] Copyright Bernd Liebl, Fraunhofer IFF, source: `http://iff.fraunhofer.de/en/press/press-releases/2010/robots-get-artificial-skin.html`

[5] Copyright Museum für Kommunikation Berlin, source: `http://www.care-o-bot.de/MuseumRobots.php`

[6] Photo by Nesnad under CC-BY-SA-3.0, source: `http://commons.wikimedia.org/wiki/File%3AWakamaru-fullshot2011.jpg`

[7] Photo under PD US Military, source: `http://www.darpa.mil/Our_Work/TTO/Programs/Legged_Squad_Support_System_(LS3).aspx`

[8] Photo by Larry D. Moore under CC BY-SA 3.0, source: `http://commons.wikimedia.org/wiki/File%3ARoomba_original.jpg`

[9] Picture by Mars Artwork under PD NASA, source: `http://marsrover.nasa.gov/gallery/artwork/rover2browse.html`

[10] Photo by Josh Meister at Georgia Tech under PD US ED, source: `http://www.gatech.edu/research/mediaviewer?pid=160711`

[11] Picture under PD, source: `http://commons.wikimedia.org/wiki/File%3ADuck_of_Vaucanson.jpg`

[12] Photo by Rama under CC BY-SA 2.0 FR, source: `http://commons.wikimedia.org/wiki/File%3AAutomates-Jaquet-Droz-p1030472.jpg`

... The frog does not seem to see or, at any rate, is not concerned with the detail of stationary parts of the world around him. He will starve to death surrounded by food if it is not moving. His choice of food is determined only by size and movement. He will leap to capture any object the size of an insect or worm, providing it moves like one. He can be fooled easily not only by a bit of dangled meat but by any moving small object ... He does remember a moving thing providing it stays within his field of vision and he is not distracted...

So why Rana Pipiens? This work says much about concepts and preconception as well as about biological information processing and its simplicity. But if we substitute the "frog" with a robotic entity and "prey" with the myriad of smart entities (as it was initially introduced) that offer or publish their data within a distributed, smart, intelligent, or any kind of instrumented environment, it can also be used as a metaphor for the main problems that we have in distributed robotics/systems.

Information or data might be accessible within such environments in various forms and variations. It might be hidden behind different interfaces and protocols, stored in different databases or published by various middlewares. Thus, a robot or any kind of application might be surrounded by useful information, but if it does not appear in the right format it cannot be used. Nonetheless, simply making any kind of data accessible by applying one standard communication protocol and one generic format for messages and message descriptions (as it is proposed by most systems that are described within the next section) does not solve this problem either. It is simply the continuation of the same misconception as the one prior to Lettvin's work, since it requires an extraordinary intelligent application (a Maschinenmensch or unknown capabilities of a frog's brain), which gathers and assembles any kind of unfiltered data and transforms it into the required and useful information.

If we look at how distributed applications are developed now, this type of intelligent work is accomplished by humans that pre-configure an entire distributed system as well as the robot, and that have knowledge about the surroundings and the tasks a system has to fulfill. However, this will not be the case anymore within the near future, where systems will have to solve problems in dynamically changing environments with only a vague prior knowledge about the tasks and even less about the systems within the surroundings and their configuration. Although, most of the developed systems so far claim to be generally applicable and even to be easily extendable to solve different concerns, they are not. The very reason for this lies in the "biased" way of developing and accessing distributed systems. Biased in this case means the way of problem solving, which is mostly predefined by the way of programming language and paradigms we apply. In most solutions, an imperative programming paradigm is applied, which forces one to define every step of a solution in detail, where knowledge about the input data (& formats), the configuration of the systems, and the tasks is a prerequisite. But solving problems this way for environments with growing complexity where nothing is granted fails in being generally applicable.

Thus, we can either continue building self-contained systems, which gather information about their surrounding from locally available on-board sensor systems by applying predefined transformations onto predefined data (cf. Robotic Evolution) or, we can start to develop declarative concepts for a distributed world (cf. Robotic Revolution). (See therefore also the essay on programming paradigms in the appendix on page 139.) Applying the declarative paradigm onto the generation process of information enables us to define "What" kind of information is required, instead of defining "How" it has to be generated. Multiple communication middlewares already apply declaration in terms of subscribing for data that is published within a certain format or under a specific topic; however, there

is a tremendous difference between data and information, and as it is introduced within Sec. 2.2.1 in more detail, data refers to the pure amount of available facts, whereby information has a certain value in a specific context. A piece of information can be a certain value, a fusion (which summarizes data of different kind, time, or modality), a map, or even a complex model (representing different relations).

Thus, the main problems regarding the biological metaphor can be formulated as follows:

1. **Lack of Intelligence:** Smart Environments (SmEs) or Intelligent Environments (IEs) (including robots as part of a wide and heterogeneous ecology) do not seem to be smart or intelligent at any rate, they only consist of networked and reconfigurable components and services. How can such a "not intelligent" distributed system be tweaked in order to generate any kind of required information?

2. **Holistic Access to Information . . . :** A robot within a smart or intelligent environment can be surrounded by data, but it is simply not usable and not accessible, if it appears in the wrong format or is hidden behind the wrong interface. How can such information spaces be accessed by a robot and how can the information and data from the robot itself be consumed consistently by the environment?

3. **. . . and Memory Externalization:** A frog does not possess any kind of long time memory, because moving things are only "remembered" as long as they stay within its field of vision. Thus, instead of storing any kind of data from an environment locally that might be necessary in the future, the environment itself could be interpreted as externalized memory, since all required data is contained within. How can such distributed information spaces of "smart components" be organized to create something like a global memory?

4. **Reconfiguration and Representation:** Changing one aspect of the well-known environmental configuration might cause a breakdown of the system. To illustrate, noise in the frogs' environment would probably result in an extinction of the species in that area, because mating is related to sound and touch (although there are sensory systems and probably also filter functions available that could be used for compensation). Thus, the type of environmental abstraction is currently tightly bound to a set of "hard-wired" sensors, fixed transformations, and to the set of predefined tasks. But data from different systems and in different formats could also be used in a dynamic process to generate the same information. Thus, what mechanisms are required in order to generate again an again the same type of information although the environment and its inhabitants change continuously?

This problem statement matches the third challenge that was identified in the overview of ubiquitous and cloud robotics in [49]. It is worth mentioning that the first two challenges in this overview deal with increased autonomy, social awareness and affective interaction, which are strongly related to third generation robotics and, therefore, contradicts the Ubiquitous Robotics (UbiBot) concept in [151]. However, the third challenge deals with engineering problems, which includes the design of new engineering tools and middlewares to create services as plug and play applications, interoperability among robots and smart devices (beyond remote control, voice and web services, and the abstraction of robotic functionalities), extending the perception and actuation capabilities through the network, and the sharing of intelligence (by using cloud-based techniques).

The main objective of this thesis is to develop a concept and mechanisms for declarative information gathering, which enable an entity to access any kind of information (**2.** & **3.**) in any kind of desired format (**4.**) from an instrumented environment, without the need for extraordinary intelligence (**1.**).

1.5. Overview

The next chapter is intended to give a detailed overview of the current state of the art. It starts with a distinction of different terms and notions used in distributed systems, while the following sub-section is used to define the different aspects of data, information, knowledge, etc. Both of them define the basic vocabulary that is used within the following chapters. The second sub-section is furthermore combined with an engineering perspective and defines a hierarchy, which allows it to organize and relate relevant publications according to "What" is shared within a distributed environment.

Chap. 3 addresses the main contribution of this thesis, it describes the concept for a declarative information access bottom-up and it is therefore segregated into three parts. The first part deals with the organization of data within the environment as well as with the organization of entities within, a cloud-based approach is used to form some kind of virtual overlay database that is denoted as the global world model. Within the second part it is described, how data from the global world model can be transformed into a smaller and generic representation, which covers all task-related aspect of the environment with a higher degree of abstraction. In the third part, a declarative query language that allows the definition of any kind of desired information, which is then extracted or abstracted from the local world model, is introduced.

The following implementation chapter is organized in correspondence with the concept. Within the first part it is described in detail, how the distributed and global world model is developed and maintained by applying a Cassandra, a distributed NoSQL database system. The second part explains how relevant aspects of this global world model can be translated into a discrete and precise co-simulation of the surrounding by applying the robotic simulation environment OpenRAVE. And the last part of this chapter is used to present the development and the principles of the new declarative programming and querying language SelectScript.

That this query language can be used to enclose all intermediate steps, namely accessing the global world model, generating the local world model, and abstracting any kind of information in a desired format, is demonstrated in Chap. 5. Therefore, the common task of parcel delivery in a smart factory environment has been chosen, which includes the accomplishment of several sub-tasks and the generation of various different information.

The achieved results of this thesis are discussed within the last chapter, and an outlook onto the future work is provided.

2. Related Work

> "Adapt what is useful, reject what is useless, and add what is specifically your own."
>
> — Bruce Lee

There are currently plenty of different concepts and terms on distribution aiming or suggesting different kinds of interconnection and interoperability. Thus, the next section is intended to give a short overview of these concepts to differentiate between them (cf. Sec. 2.1). The introduced systems put different foci onto different aspects (based on their field of application), and the term notions was therefore used to underpin the very vague (and sometimes missing) definitions and distinctions. Based on these considerations and the lack of an appropriate classification or a suitable attribution in the literature, a new classification scheme was introduced afterwards (cf. Sec. 2.2). It is subsequently applied to differentiate the related approaches from the literature (cf. Sec. 2.3). A summary as well as discussion about the main problems identified within the state-of-the-art is presented in the last section (cf. Sec. 2.4).

2.1. New Concepts of Distribution: Terms and Notions

There are currently different concepts and terms on distribution, aiming or suggesting such a kind of interconnection. Therefore, this section is intended to give a short overview and to differentiate between these concepts.

Among all the new notions and paradigms for distribution, Cyber-Physical Systems (CPS) can be considered as the superset, the most general, and the most spatially extensive concept. It is defined as a collaboration of networked and embedded computational elements used to monitor and control real world physical processes (cf. [137]). But the emergent complexity of systems requires fundamentally new design technologies in multi-discipline areas. Proposed applications range from small-range medical devices and systems as well as home and building automation, up to autonomous cars and intelligent roads, distributed robotics, manufacturing, power grids, and many more. As summarized in [31], even though there are large amounts of funds for projects in that area, the research is at a very early stage.

The initiative on "Industry 4.0" (the fourth industrial revolution) is a German strategic project that mostly arises from the paradigm shift of CPS and that is mainly focused on the manufacturing industry. "Smart Factory" or "Smart Manufacturing", as described in [92], can be considered as the key feature of Industry 4.0, whereby the focus lies mostly in monitoring and developing distributed control application with feedback loops.

The Internet of Things (IoT) in that case is thought to be the connecting communication paradigm. As described in [31], it is only applied to interconnect a variety of "things or object" through unique addressing schemes within the existing Internet infrastructure. It is expected to enable an interconnection of "Smart Devices and Services" that is far beyond nowadays' Machine-to-Machine communication (M2M).

In fact, cyber-physical entities do not necessarily have to be smart: common Radio-Frequency IDentification (RFID) technology, which is often referred to as being a building block of CPS, can be used to enhance any kind of object and to support tracking within well-defined environments. Furthermore, the term "Smart" in this case does not refer to human-like capabilities, it is more related to technological aspects, such as the entity's ability to communicate, to perform a small set of predefined tasks, and to operate autonomously to some extent. For example, a "Smart Sensor" is defined in [117] as: *"... one chip, without external components, including the sensing, interfacing, signal processing and intelligence (self-testing, self-identification or self-adaptation) functions."*

There are varying definitions and notions of smart devices, sensors [228] and actuators [85], or even products [216]. [122] gives a general classification of "Smart Objects" according to real world awareness and interactivity. At its lowest level smart is understood as "activity-aware", which means that an object perceives the world in terms of event and activity streams and each event or activity is directly related to the use and handling of the object (pick up, turn on, operate, etc.). Interaction/communication is bound to static information exchange and log files only. In contrast to this, second layer "policy-aware" objects provide context-sensitive information about the object status as well as work performance, and they also contain an embedded policy model that is used to interpret events and activities. Through this, high-level "process-aware" objects shall understand the complete organizational processes they are part of and can relate the occurrence of external events to these processes. The IEEE 1451 family of standard for "Smart Transducer" [138] is more focused on the standardization of interfaces and communication protocols. A common standard for Transducer Electronic Data-Sheet (TEDS) enables the self-description of systems that can be communicated and, thus, shared with others.

As a summary, it can be said that smartness actually stands for (standardized) networked interconnection and perhaps the offer of a small set of (configurable) services. Although the term "Smart Service" is far more general [153], it is also frequently used to address the capabilities of a smart entity or a composition of multiple smart devices split across the network.

Smart Environments (SmEs) can be accounted as a subset of CPS that are commonly more human centered, examples from the literature are mostly focused on domotics, home and building automation, smart kitchen, etc. A good overview is given in [54], where the authors derive a component structure and give the following definition: *"We define a smart environment as one that is able to acquire and apply knowledge about the environment and its inhabitants in order to improve their experience in that environment."*

Intelligent Environments (IEs) can be further specified as a subset of SmEs and to some extent even more human-centric (one can say personalized), although the transition is smooth. While smart environments are compositions of smart devices, IEs are realized with Ambient Intelligence (AmI). As the interweaving concept of IEs, AmI supports a higher degree of intelligence and autonomy than previous concepts. According to [32] and [157], the behavior of the environment is orchestrated by self-programming preemptive processes (e. g., software agents or multi-agent systems [211]) in order to proactively enhance occupant's experience and support them in their daily life.

All of the concepts, notions for distribution as well as dynamic interconnection that have been presented so far can be interpreted as descendants of the already mentioned Ubiquitous Computing (UbiComp). Mark Weiser was one of the first to realize that computers and computation itself will become invisible and disappear into the background, hidden by a myriad of intelligent components "*... that weave themselves into the fabric of everyday*

life" (cf. [220]). Or as defined later in [221]: "*... ubiquitous computing will be a world of fully connected devices, with cheap wireless networks; it will not require that you carry around a PDA, since information will be accessible everywhere.*" Although formulated in a general and quite human-centric manner, UbiComp predates all of the concepts mentioned so far. Weiser made conclusions about the mobility of device, wireless bandwidth, and the ubiquity of information (and displays). Even though there might be slight differences between the terms Pervasive Computing (PerComp) and Ambient Computing (AmC), they can be applied synonymously.

To end up this consideration and to close the gap to robotics (although, robots can be interpreted as an inherent part of all concepts) there is also a concept combining ubiquitous computing and robotics, called UbiBot. An example often named in this context is the PEIS ecology, which is described in more detail in Sec. 2.3.4.2.13 on page 52, as well as the relatively new concept of Network Robot System (NRS), see [191]. It identifies three types of network robots, visible (embodied robots in a classical sense), virtual (only acting in cyberspace), and unconscious (external sensor systems), whereby every system is offering some kind of service that can be combined at runtime to fulfill more complex tasks (or composed services). The term "Cloud Robotics" was coined in 2010 and refers to the usage of cloud computing facilities to enhance NRS capabilities (cf. [49]).

Because different systems, approaches, or architectures can be easily associated with more than a certain concept, it is quite difficult to relate them. At least all of them seem to be building blocks of CPS. Thus, the next section is applied to introduce a classification scheme, which allows relating and comparing relevant contributions in these areas on a qualitative level.

2.2. Classification Methodology

In the chapter "From Autonomous Robots to Artificial Ecosystems" of "From Autonomous Robots to Artificial Ecosystems" [151], the three different systems are compared with each other, by hypothetically applying them to a hospital scenario. By reading the previous section, it becomes obvious that such a single scenario which could be used to compare all systems and approaches that are covered by CPS probably does not exist. They operate on different "abstract" layers. Thus, instead of trying to identify different metrics and benchmark scenarios, it is more advantageous to identify these layers and to relate to approaches accordingly.

The archetype for this consideration is a pretty well-known Open Systems Interconnection (OSI) model (cf. [223]), as depicted in Fig. 2.1. It is a stacked differentiation method, which might seem a bit far away from it at a first glance, but it has helped to implement another distributed system. Although the TCP/IP stack made the running and became the basis of the Internet, both share the same principles and the OSI model is still used as a device to explain networking in general terms. OSI is a conceptual model for a separation of concerns that defines seven layers of abstraction (in the original version) with characteristic functions and properties for a communication network, regardless of the underlying internal structure, technologies and hardware. It covers various different networking protocols, software, and even hardware devices, such as switches and routers. The seven distinguishable layers are physical, data link, network, transport, session, presentation, and application. Communication is thereby described as a horizontal process between elements of same lay-

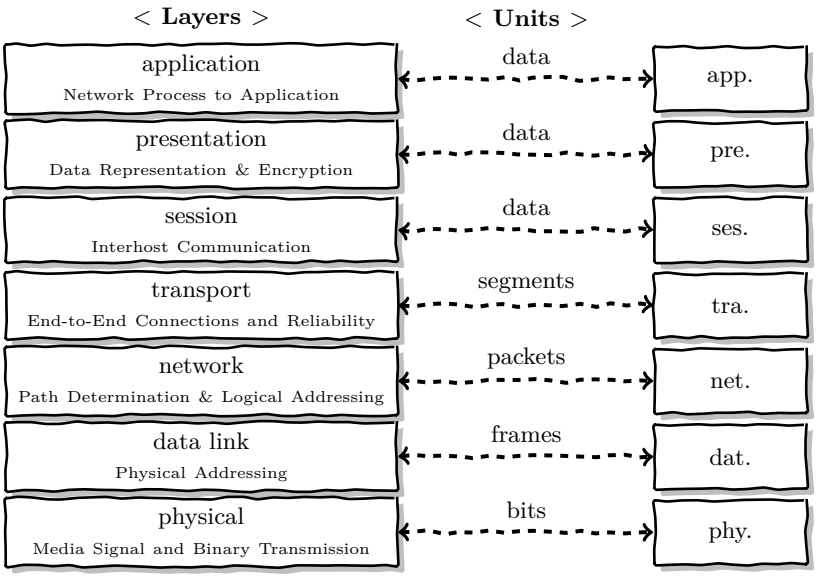


Figure 2.1.: General OSI model as at [223]

ers, throughout which every layer can only interact with the layer beneath, while providing facilities to be used by the layer above.

Thus, the next section is intended to identify different conceptual layers in CPS as well as their common characteristics, by distinguishing what is actually shared. This consideration is made by connecting two opposite directions. Starting from a top-down view on data, information, and knowledge, etc., from the perspective of Information Science, these concepts are examined in concrete terms from an engineering point of view. Additionally, the following section is also intended to define the required terminology (which is applied in subsequent sections and chapters) and, thus, to form a basic vocabulary.

2.2.1. Short Introduction to Information Science

Information Science (IS) itself as a term is relatively hard to define, as it is highly interrelated with various research fields, such as computer science, mathematics, logic, philosophy, cognitive science, economics, communications, social sciences, etc., but in [200] the authors came up with a working definition of IS:

Information science studies the representation, storage and supply as well as the search for and retrieval of relevant (predominantly digital) documents and knowledge (including the environment of information).

According to [40] it is concerned with the body of knowledge, relating IS additionally to the origination, collection, organization, interpretation, transmission, transformation, and utilization of information. In addition to this general definition (information scientists still struggle to find a coherent definition), IS is also concerned about its building blocks and their definitions. These are data, information, and knowledge, which were used "synonymously" (not to say sloppily) not only in the previous sections, but also in our everyday language, and to some extent, also within the scientific literature.

The relation between these three terms is depicted Fig. 2.2. It shows the so-called "Wisdom Hierarchy", a widely accepted concept within IS, which is comparable with the "fault-

error-failure-chain" applied in dependable systems engineering. Similarly to IS, there is still an ongoing debate about their exact definitions of data, information, and knowledge (at least there is consensus about the existence of these first three levels), which is underpinned by the Delphi study published in [230]. That is why the author is referring to the definitions in [210]. These three terms are also applied below to classify different approaches in SmEs and IEs.

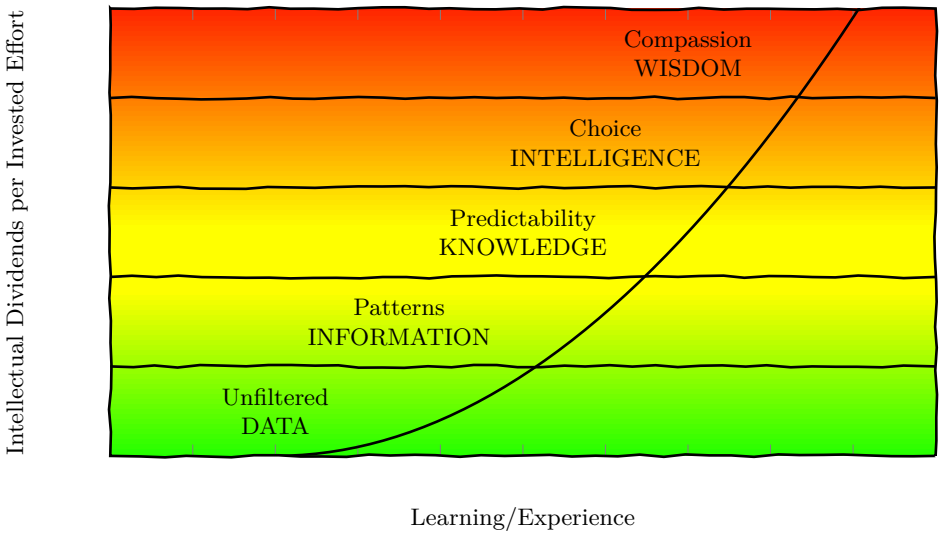


Figure 2.2.: Conventional View on the Wisdom Hierarchy (cf. [172]), with a symbolic curve representing the value and effort of a learning journey

As depicted at the basis of Fig. 2.2, data can be interpreted as the pure amount of simple, discrete, and isolated facts. It describes only partially what has happened, without any judgment or interpretation. Information is something that is derived from data, if it is put into context or combined within a structure. Knowledge arises from information when it is interpreted and thus given some meaning or, put technically, information that has been tested, validated, and codified. Knowledge is thus always associated with some kind of modeling or simulating capabilities. It can thus be used to make inferences or to predict future consequences. While most scientists share this view with slight differences in the details (cf. [230]), the term wisdom provides more potential for debates. Tuomi further integrates intelligence as an intermediate step between knowledge and wisdom. Intelligence is therein interpreted as the ability to choose between alternative actions based on knowledge and when values and commitment are used to guide intelligence, behavior may be said to be based on wisdom.

To summarize all these with a robotics example, data could be represented by a set of laser scans, and then can be detected by patterns like edges, straight lines, curves, etc.; this would represent information and, thus, knowledge could be derived from it through the generation of a map. Intelligence comes into play when trajectories are planned with different algorithms on this basis, while choosing a planning algorithm (and thus also the trajectory) according to certain optimality criteria and constraints could then be considered as wisdom.

2.2.2. The JDL Data Fusion Model

This process model (cf. [89]) can be regarded as a bottom-up view that is commonly applied by engineers as a canonical abstraction and categorization when describing fusion methods. It was established by the Joint Directors of Laboratories (JDL) Data Fusion Working Group in 1986 in order to cope with the lack of a unifying terminology in the sensor data fusion domain and, thus, to improve the communications among military researchers and system developers. Although it was initially developed for the military sector, it has been applied, adopted and refined to comply with several (non-military) areas (see therefore also [198] in which the term (Con)fusion was used, when discussing integral parts of the JDL model).

The general fusion model as it is sketched in Fig. 2.3 comprises five intermediate levels that are required to transform measurement (and context) data into human-interpretable representations, the so-called Human Computer Interaction (HCI). Thus, sources and the HCI represent the very boundaries of every system. Furthermore, it is currently intended for human operators only, allowing them to input commands, request different kinds of information and reports, as well as assessing situations and drawing inferences in order to assist human operators. Along with the representation (by applying multimedia methods, such as graphics, sound, tactile interfaces, etc.), it is also used to guide human's attention to overcome cognitive limitations. In principle, the HCI has to be redefined for smart environments in order to provide a M2M interface. The five steps defined by the JDL as well as a general database management system are described within the following paragraphs.

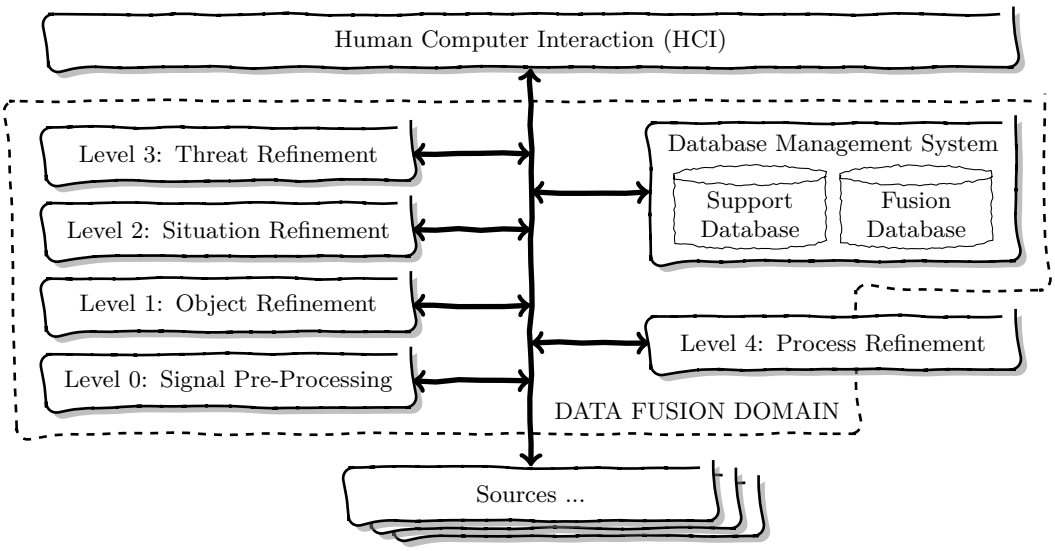


Figure 2.3.: The JDL fusion process model, rotated by 90 degrees to match better with the wisdom hierarchy depicted in Fig. 2.2

0. Level (Signals): It combines the acquisition of input data with its preprocessing. Input data defines are all available signals from local or external sensor systems, observable states, data from remote databases, and to some extent also context information (e. g., time, location, add. attributes). Source preprocessing is intended as an intermediate step that allocates data to the correct fusion process (next level), performs basic feature extraction algorithms, and can send alerts to the third level.

1. Level (Objects): Processes on this level combine lower level locational, parametric, and identity data for refined representations of individual objects. There are four key functions associated with this processing level:

1. transformation of sensor data into a consistent set of units and coordinates,
2. refinements in time estimates of an object's position and kinematics as well as extensions with attributes,
3. assignment of data to objects to enable further statistical estimations, and
4. refinements in the estimation of an object's identity or its classification

2. Level (Situation): It deals with situational representations, by mapping current lower level objects and events into the context of their environment. Additionally, it is focused on relational information (e. g., physical proximity, causal, temporal, and further relations) between objects (and events). Formal and heuristic methods are therefore applied to examine the meaning of level 1 processing results.

3. Level (Impact): Threat refinement, also referred to as impact assessment, is applied to project the current situation into the future and, thus, to plan and draw inferences (the importance of differentiation between abduction, induction, and deduction is further emphasized by Llinas et al. in [145]).

4. Level (Overall-Process): This layer can be considered as meta-process. It is used to monitor lower processes in terms of performance and quality requirements, and to perform process refinements and adaptive data acquisition.

Data Management: It is described as the most difficult part, because of the large and varying amount of data (e. g., signals, images, textual and symbolic data, models, etc.). It therefore deals with functions for effective storing, archiving and querying of fused data (and also to some extent with the management of input data), thus giving access to historical data.

2.2.3. Combination and Further Categorization

The common JDL depiction from the literature was rotated 90 degrees in Fig. 2.3 to present a better correspondence between the two concepts, although a direct match seems to be obvious. Reviewing the literature for a similar connection between the top-down wisdom hierarchy and the bottom-up JDL fusion model did not reveal in any findings. The IS data concept matches the JDL fusion level 0, information matches the JDL concept of an object. A JDL situation is associated with an adequate representation of objects and their relation, which corresponds with the concept of knowledge in IS or, in general, with a model or a world-model in robotics. Deriving predictions or decisions on this basis (probably combined with additional information and rules gained from support or fusion databases) refers to intelligence. As a result, wisdom (the appropriate selection between alternatives) lies behind the data fusion domain (behind the HCI) and requires a human operator.

IS and the JDL fusion model give essential hints about how systems, middleware approaches, or architectures operating in CPS can be divided into different classes or categories, depending on the level of environmental abstraction they are able to share. While IS is more concerned about the concepts and their distinction, the JDL is more focused on algorithms and transformations. The JDL thereby also defines what kind of specific metadata or context information is further required to transfer data into information, information into knowledge, etc. Stemming from the JDL, two additional layers (in addition to wisdom, which is currently far beyond the scope) that are not touched by IS could be identified: sources (originally not part of the fusion domain) and representations (originating from the HCI). Existing approaches in CPS will be related to the identified layers and discussed in detail within the next section:

1. Source-Layer: It is used to share information about the very basis, on sensors, actuators, and databases, which might be specified with data-sheets, interface descriptions, and to some extent also expressed within protocols. Additional information about the location and identifier (as introduced by the JDL) might also be relevant at this layer.

2. Data-Layer: This layer is used to share and access data, which might appear as measurement data, streams, system states, actuator commands, etc. It therefore covers both real-time data (immediate access) and historical data (using database management systems). It might require additional context information about the origin of data and its belonging as well as additional quality attributes.

3. Information-Layer: The previous levels are necessary to derive information. Information might be generated statically from the available data and sources, but it can also be generated on demand by applying different functions from a code basis. Information can be a pattern or a feature extracted, identified or fused from a set of input data (as intended by IS) or a specific entity (as intended by the JDL).

4. Knowledge-Layer: Systems operating on this layer share models. In the robotic context, these are any kind of maps, ontologies, formal or logical representations, etc. (a distinction between different models is presented in Sec. 2.3.4). In fact, any kind of world model used here represents the relations between different lower level information. This means, that this layer is used only for the appropriate representation of certain environments and situations. It therefore builds the basis for the next "algorithmic" level.

5. Intelligence-Layer: It is used to draw inferences for planning and predicting, as well as for decision support, etc. In this case, a robotic entity might query the smart environment directly for trajectory to a certain destination. As mentioned in the introduction, planning does not have to be performed by the robot itself; instead, it can be covered by a certain sets of services resident within the smart environment, the cloud, etc. As revealed in Sec. 2.3.4.2, there are already some systems operating on this layer.

6. Representation: This is not an extension to the previous layer. It is actually a requirement, arising from the HCI and covering all other layers. Since the HCI is dealing with metaphors and the appropriate representation of data, information, knowledge, and also intelligence (e. g., decision support, forecasting, etc.) for humans, different "machines" might require (or request) different representations or formats of data, information, world models, etc.

2.2.4. Discussion

Based on the idea of the OSI model, a new separation of concerns for the application layer has been introduced. And, of course, while most technologies and protocols are not directly designed to meet the requirements of the OSI model, they can be described in terms of how they fit into one or more of the layers. The same is done within the next section, which provides a qualitative rather than quantitative view on different approaches (based on six introduced layers of abstraction). But as in most classification approaches (also in biology) an entity cannot only be related to one class. Rather, it possesses characteristics that have to be associated with different layers. Therefore, the metaphor of a boxplot is also used within the following, to express on which layers a technology is mainly operating (quartiles), where the focus lies (median), and which layers are additionally covered (whiskers), as well as rudimentary aspects (outliers). A comparative overview onto the related work using this classification schema and the boxplot metaphor is given in Fig. 2.15 at page 60.

2.3. Smart Environment Enabling Technologies

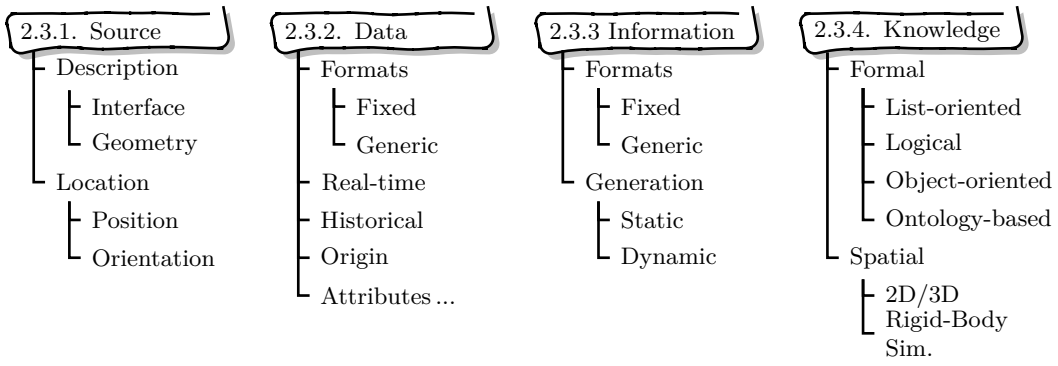


Figure 2.4.: Overview on categories and distinguishing attributes

This section is intended to give a detailed overview on state-of-the-art technologies that enable access to CPS, SmEs, IEs, whether on source, data, information, or knowledge level. The developed classification scheme defines the structure of this section (see also the depiction in Fig. 2.4) and the relevant approaches are placed accordingly. Technologies are described within the sub-section/layer they mainly belong to the most, but additional properties that actually belong to different layers are also mentioned. Fig. 2.15 within the next section shows a synopsis of all cited approaches.

2.3.1. Source-Layer

The survey for this layer will only give a brief overview and it therefore focuses on the most relevant approaches to describe system interfaces and their capabilities. An extensive and very detailed state-of-the-art on system descriptions and interface specifications can be found in the dissertation of Sebastian Zug in [231].

2.3.1.1. SensorML

It is one of the multiple Open Geospatial Consortium (OGC) specification standards as part of Sensor Web Enablement (SWE) initiative. Sensors and transducer components (i. e., detectors, transmitters, actuators, and filters) are therein described as physical processes by uniform XML schemata [165]. Additional non-physical or "pure" processes, which can be treated as mathematical operations, can be described according to the same standard as well. A SensorML process defines inputs, outputs, additional parameters and metadata. These data-sheets are thought to be hosted on every system (or at least be accessible from within the network), which should enable some kind of plug-n-play and auto-configuring capabilities (as originally intended by CPS). Furthermore, metadata such as unique identifiers, names, keywords, capabilities, constraints (i. e., time, legal, and security), documentation, and location parameters enable the mining and discovery of the processes.

Sensor Model Language (SensorML) is thought to be the basis of further OGC standards by enabling interoperability on a system and interface syntactic level so that sensors and processes can be better searched, understood, and thus analyzed and utilized by machines. Although SensorML is also capable of defining complex data types (for parameters and metadata) and the new specification on SensorML 2.0 (cf. [164]) makes extensive use of this possibility, it is actually not intended to describe measurement and actuator data. This is the domain of another OGC standard, namely TransducerML (see Sec. 2.3.2.2), which is further dealing with data exchange and interoperability, preservation of raw sensor data, etc.

Because pure processes (mathematical function), which offer some kinds of (fixed information) services that can be utilized for different purposes from within the net are enabled, it is also necessary to put this approach not only onto the source and data level, but also to extend it into the direction of the information layer (see Fig. 2.15).

2.3.1.2. IEEE 1451 – TIM & TII

The IEEE 1451 is a family of standards for networked smart transducers [101, 138], which is similar to the set of OGC specifications. The defined standards can therefore also be divided into different levels: Those that are used to describe the transducer and its capabilities, and those that define communication properties (NCAP & NI). The Institute of Electrical and Electronics Engineers (IEEE) 1451.0 therefore introduces the already mentioned Transducer Electronic Data-Sheet (TEDS) (see Sec. 2.1), which are XML schemata too. The definitions of the so-called Transducer Interface Module (TIM) and the Transducer Independent Interface (TII) (belonging to this level) include calibration TEDS, transfer function TEDS, command TEDS, location and title TEDS, as well as manufacturer and identification TEDS. These definitions are stored on the TIM hardware element in an electronic memory device and describe functionality that is independent of the physical communications media. These TEDS are more or less optional, in contrast to PHY TEDS, transducer channel TEDS, or meta TEDS, that are used to describe communication capabilities and settings of a transducer.

2.3.1.3. MOSAIC

It is the "fraMewOrk for fault-tolerant Sensor dAta fusIon in dynamiC environments", which is intended to be used as a general programming abstraction for transducers and fusion processes in distributed environments. XML is therefore also applied to define data-sheets

while describing systems and measurement data. But in contrast to the previously described technologies, it integrates advanced mechanisms for online fault and error handling (which are integrated in the fusion process as well). A general classification scheme for measurement malfunctions and disturbances was especially developed. A sensor is thus not only described by its capabilities, location, settings, etc., but also by its proneness to faults and errors, measurement uncertainties, and by extended geometric properties, such as ranges, covering areas, even the shape of the sensor beam.

The sensor fault description allows the calculation of fault probabilities for sensors and even for single sensor measurements. This information is applied within the selection process, sensor fusion, and to identify appropriate detection and filter methods. MOSAIC therefore has to be accounted to operate mainly on source and data level (see Fig. 2.15), without supporting data transmission itself (which is handled with the help of FAMOUSO, see 22).

2.3.1.4. Summary

As a matter of course, there are many further examples of self-describing transducers that could be mentioned on this level, but the core idea is similar to all. An entity (transducer) is hosting a description of its capabilities and interfaces, to tell other systems what it is and what data it produces. Not surprisingly, these standards vary according to their intended application (e. g., a sensor within a chemical reactor in contrast to a multimedia system) and descriptions can also be hosted externally or within a database, as it is, for example, described for IEEE 1451 virtual TEDS (if no or only little local memory is available).

An application, operating on this level, is thus self-dependently responsible for identifying relevant systems and connecting to it in order not only to access and interpret their data, but also to acquire context information. If there is no general "communication concept" utilized by all entities, then communication parameters have to be described as well (how is data transmitted and encoded). But as already mentioned, the next level is intended to cover these aspects.

2.3.2. Data-Layer

Although "sources" describe the very basis of every smart environment, this abstraction does not seem to be that relevant for many applications. For example, a smart home application that is responsible for the apartment temperature and ventilation is only interested in temperature and humidity data (values in a certain unit), not in the types and configuration parameters of thermometers. Similar to a nuclear power plant that is monitored by a set of inspection robots [39], the only data required are the positions and the degree of radiation (or at least a notification, if a certain value is exceeded).

It is therefore only logical to hide most common aspects behind a common communication infrastructure. Thus, the main focus of this layer is not the handling of systems (sources) and their description, but rather data and its description and distribution, because there has been a lot of research in that area, covering communication middlewares, publish/subscribe systems, shared memories, event-based systems, etc. Similar to the previous sub-section, this part is intended to give only a very brief overview of relevant technologies. A complete overview is given in the thesis by Michael Schulze in [193], where he develops the communication middleware FAMOUSO, which is especially designed for resource limited embedded systems.

2.3.2.1. FAMOUSO

It is a publish/subscribe middleware that was developed in compliance with MOSAIC (cf. [22, 23]) at the department of Distributed Systems at the Otto-von-Guericke-University in Magdeburg. FAMOUSO stands for "Family of Adaptive Middleware for autonomOUs Sentient Objects" and it is intended to offer a seamless data exchange between smart sensors and actuators on different hardware platforms (ranging from 8-bit micro-controllers up to 32-bit PC/Workstations) over a broad variety of communication media and protocols (e. g., CAN, 802.15.4, UDP-MultiCast, and other in-house technologies such as Ad-hoc Wireless Distribution Service (AWDS) [93]).

In contrast to address-based (or "source"-based) communication, the transmission of data is anonymized between communication end-points. Publishers are content producers and subscribers are content consumers. These are roles taken up by distributed applications. Concerning the publish/subscribe paradigm, an application can specify the kind of data they produce and/or consume. All communication-relevant aspects are handled by FAMOUSO, it abstracts data as events, arranges and buffers data according to application requirements, schedules the communication according priorities, collects fault notifications (e. g., absence of periodic events), etc. MOSAIC is required to define the structure of events, to ensure correct message en/decoding as well as correct interpretation. The format of every event is described with an electronic XML data-sheet, covering data types, units, uncertainties, additional attributes, but also communication channel attributes, such as deadlines, periods, and omissions.

2.3.2.2. TransducerML

As already described in the paragraph on SensorML, the "Transducer Markup Language" is the OGC standard for characterizing data (and metadata) that is produced by SensorML systems [166]. It is thus intended to offer an efficient encoding of live and streaming data for the communication layer. To some extent, it seems that the OGC had not been that consequent when defining their standards, because TransducerML also covers some parts that deal with the system description (actually a part of the SensorML domain). This includes information about specific transducer components (model and serial number), their behavior, system calibration and installation information, owners and operators, responses to physical phenomena, sensitivity, and other operating parameters. A future harmonization between the two description languages is thus also indicated as a work in progress in both specifications.

TransducerML is designed to be self-contained and self-sufficient. Any information that might be required for later parsing and processing is captured within the XML data product description. This includes format and types, size, ordering and arrangement, calibration information, units of measurement, precise time-tagging of individual groups of data, uncertainties, coordinate reference frames, etc. All of these elements of the TransducerML system and data description can be automatically tagged, stored for later indexing and cataloging, as well as applied for discovery (but in this case at more data-related level).

2.3.2.3. ROS

Because integral parts of the later concept were implemented with the help of this "framework" (and it was not mentioned in the referenced thesis [193]), this overview is more detailed than the others. The "Robot Operating System" could also be literally translated

as the Swiss Army knife in robotics. It is a relatively new framework, intended to simplify the development of any kind of robotic application by bringing together the latest scientific knowledge. And by this time, it has become a de facto standard in the scientific robotics community (cf. [124]). Robotic applications are no longer designed as single and monolithic processes, but instead as a collection of (distributed) nodes, which perform a certain type of computation in accordance with the Unix philosophy (cf. [189]):

"Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface..."

The development of a ROS node is mainly based on two philosophical pillars (apart from open source ideology), namely peer-to-peer and multilingualism (cf. [175]). To expand multilingualism and to be able to develop specific nodes, used as detectors or filters, we have developed an extension for the statistical programming language R[1] [7]. In contrast to the previous approaches, ROS does not cover system descriptions (on source-level) natively, but it therefore supports multiple communication paradigms. "Natively" in this case means that there are ROS packages, which allow describing the geometry of robots and manipulators. The Unified Robot Definition Format (URDF)[2], for example, allows to define robot models with an XML description. Although it is named after "Robot", it is also possible to describe sensors and scenes. Robots are described in terms of their kinematic structure, dynamic properties, and geometry. The support for sensors currently allows to define cameras and ray sensors with a very limited terminology only (this project is currently dropped and awaiting proposals for new types of sensor elements). Scenes can only be described as spatial relations between robots and sensors, including additional basic geometrical primitives.

Natively supported communication paradigms include publish/subscribe, interprocess communication, service-based communication, and some others. The basis for all communication is strongly typed messages whose formats are described by a simple but effective language-neutral interface definition language (not XML). Starting from primitive data types such as `int32`, `float64`, `string`, `arrays` etc., more complex structures can be derived and also composed to new message formats, an example is shown in Lis. 4.2 on page 93. Every ROS package can define its own message formats. A simple ROS-Desktop-Full installation contains more than 1100 different message formats (and more than 300 service formats) for various purposes. The only difference between the ROS message specification and the service description specification is that a service defines a request and response format.

The ROS publish/subscribe system is, in contrast to FAMOUSO, organized by a central node, the `roscore`. All nodes, therefore, have to register themselves at that master node and advertise their intent to publish data or to subscribe for data (in this case with XML-based RPCs). The communication is then instantiated via unicast/multicast UDP or TCP with the help of the master node, if there is at least one publisher and one subscriber. The ROS publish/subscribe system was initially only intended for single robot applications and thus handled the communication between a limited set of nodes, but there are also additional ROS packages (see the overview in [26]), such as `multimaster_fkie`[3] or `rocon_multimaster`[4] that also enable publish/subscribe via multiple masters. Interprocess

[1] `rosR`: `http://wiki.ros.org/rosR`

[2] URDF: `http://wiki.ros.org/urdf`

[3] `multimaster_fkie`: `http://wiki.ros.org/multimaster_fkie`

[4] `rocon_multimaster`: `http://wiki.ros.org/rocon_multimaster`

communication via so-called `nodelets`[5] (cf. [250]) is conceptually similar to the publish/-subscribe system (topics and message definition). The `nodelets` package hence runs all algorithms in the same process with zero copy transport in between.

Data and communication in this case is abstracted by streams and a very rudimentary message description, compared to TransducerML or MOSAIC. Data streams (logical communication channels) are identified via topics, these are unique strings such as `"LaserScan"`, `"Pose"`, `"CameraInfo"`, `"PickupAction"`, which are used to identify the content of a stream as well as its message format. These message formats for laser scan or pose data are standardized (with no support for units, uncertainties, metadata, etc.). Additional metadata such as time is included as long as there is a time-stamp directly defined within the message description and space requires the application of an additional position transformation system, called `tf` [249]. This additional package can be applied to maintain the relationship between coordinate frames in a tree structure and buffered in time. Data and geometries (representing manipulators, maps, etc.) can be labeled with these coordinate frames and on the basis of the `tf` system transformed (transform points, vectors, and two coordinate frames at any desired point in time).

In some cases it might also be required to request a calculation on a distinct node (e. g., sensor measurement filtering) or request one specific information (e. g., camera parameters) at a specific time slot, or to execute an action (e. g., grasping an object). This kind of remote procedure call is handled via "services" (cf. [259]) or "actions" (cf. [255]), shifting the ROS framework into the direction of the information layer (see Fig. 2.15). Nodes offer their services by using string names, similar to topics in publish/subscribe, and clients can call these services by sending their request in an appropriate message format, awaiting the reply message. The difference between an action and a service in ROS is, that actions generate and send back intermediate results, such as the current status of the manipulator within a grasp process.

Access to the parameter server is often mentioned as an additional communication method, comparable to shared memory. Parameters are stored at runtime in a multivariate dictionary, which is also hosted by the ROS master node. It is mainly applied for simple configuration purposes, but it also allows to inspect and modify the global configuration.

All in all, ROS seems to have support for everything that might be imaginable, whereby the main focus lies on data, covering its definition, standardization, and distribution (see Fig. 2.15). Although entire ROS is probably the largest framework of all CPS enabling technologies, it starts from a very minimal kernel (cf. [175]). It is based on a tiny set of key functionalities and features that can be extended and adapted according to new requirements and in diverse directions.

2.3.2.4. Sensorpedia[6]

Or as it is called in [87] "the Wikipedia for sensors", is a research project initiated by Oak Ridge National Laboratory. Its main objectives lie in the utilization of Web 2.0 social networking principles in order to provide and organize access to online sensor network data and related data sets. In contrast to ROS, it is intended to serve as a mediator for sensor data world-wide, not for a single robot.

At first, sensors and sensor systems have to be registered at Sensorpedia as a new feed, which requires a self-chosen identifier and an Atom-document that contains a general de-

[5] `nodelet_core`: `http://wiki.ros.org/nodelet_core`

[6] Project-site: `https://sites.google.com/site/sensorpedia`

scription (see [86]) of the sensor – the content and some additional context information (e. g., position, update rate, rights, format, type, etc.) to be published afterwards. The data source is registered as a simple URL-link, which is used for direct communication with the sensor. On the basis of these descriptions, sensors can be discovered. All communication is hidden behind the standard Atom publishing protocol (a simple HTTP-based protocol) and the standard Atom syndication format (an XML language), which were originally used for web feeds. Nevertheless, any other standards-based or proprietary protocol can be used for communication between the subscriber and the publisher, and the Sensorpedia API and XML format have therefore been slightly adapted to work with sensor data.

In this sense, Sensorpedia is not directly involved within the communication process, the data or the sensor interface description; instead, it is used as global repository that links to sensor systems with their data and interface descriptions (i. e., TEDS, SensorML or several other OGC standards, etc.).

2.3.2.5. Robopedia

As it was described in [68], it utilizes Sensorpedia for robotic application and provides an extension for mobile surveillance robots, including their sensor observations as well as web-interfaces for the purpose of controlling their actuators. It should be noted that, these web-interfaces are static and are currently intended to be used by human operators only to control robots via the web (offering only limited functionality). For this purpose, every robot runs a single web server, an additional robot communication server, and a process per sensor in order to update Sensorpedia.

2.3.3. Information-Layer

Some approaches capable of transforming data into information, such as ROS with services or IEEE 1451 and SensorML with its virtual processes, have so far been described. As defined in Sec. 2.2.3, information is more abstract than data or a source level description, though, according to IS, every piece of data and requested system description, which has a certain meaning or value in a specific situation, has also to be regarded as information. From the JDL point of view, information is more specific and funded on the data and source level. The data fusion process at this level can thus also be interpreted as a reduction of raw data and used to deduce more expressive values such as the maximum temperature within certain vicinity or average income. This information cannot be measured directly; rather, it has to be generated from a larger database. Indirect measurements can be given as further example (see next paragraph). This kind is required where physical modalities are not directly measurable.

A conceptually higher type of information (without a physical unit), which was denoted as “abstract” in [10], deals with identified object, persons, etc. It is something like a wall, identified within a set of laser scans, a person, a face, or an object, detected within the pixels of a camera frame. Or it is a specific (reoccurring) pattern or shape, which could also be the combination of multiple values, such as an explosion can be identified by the coinciding measured heat, noise, and flash that are above certain thresholds (cf. [125]). I will leave out the field of complex event processing and detection, which is thoroughly covered by the Ph.D. thesis of my colleague Christoph Steup and I will therefore concentrate more on ambient intelligent (smart) middlewares that work primarily on the information layer.

2.3.3.1. Virtual Sensor

The Virtual Sensor, as described in [105, 106, 163], is a frequently cited concept, opposing physical and hardware sensors. They tend to provide indirect measurements of abstract conditions (that, as already mentioned, cannot be directly or physically measured) by combining/fusing data from a set of heterogeneous sensors systems. The "only" presented application in all publications (covering four years of development) is an intelligent construction example. A user thereby requests information about unsafe regions (as circle or ellipse) around a crane. The virtual sensor that is constructed dynamically discovers physical sensors attached to components of a nearby tower crane.

In contrast to the fixed services offered by ROS, IEEE 1451, or OGC, it does not only offer a method to describe the service interface (i. e., input, output, and metadata). Virtual sensors furthermore offer a semantics that allows describing the entire process within a "higher" level of abstraction. This service description is then automatically translated into "low-level" nesC[7], a component-based, event-driven dialect of the programming language C for TinyOS[8] (an embedded operating system). This code can run either locally or it can even be distributed over the network.

Along with to the definition of the input and output formats of the virtual sensor, the process has to be described by a so-called "Aggregator". It is a generic function defining the operations that have to be performed over a possibly heterogeneous set of input data in order to calculate the desired measurement. Required sensor data is then gathered automatically from within the network, based on location and format. Data sources can thus change over time, while the service itself stays stable and is continuously performing. And, of course, the input data can also be acquired from virtual sensors. The fourth part that is required in a virtual sensor definition covers the communication. It is possible to define the frequency of aggregations, which determines how consistent the aggregated value is under actual environmental conditions. The applied communication middleware thereby automatically selects the appropriate communication methods/protocols, based on the requested frequency for continuous queries, but it also has support for one time queries.

2.3.3.2. TinyDB[9]

To close with "tiny" services, TinyDB is a distributed query processing system for extracting information from a network of (smart) TinyOS sensors (cf. [147, 148]). As the name suggests, it is a declarative approach that interprets a network of sensors similar to a database and, therefore, also applies a SQL-like syntax to collect data from heterogeneous sensors within the environment. SQL-like in this case means that the semantics of `SELECT`, `FROM`, `WHERE`, and `GROUP BY` clauses are as in SQL. It thus can be used for filtering, aggregating, etc., but in addition, it also offers further language features (not defined in standard SQL), which have been especially developed to minimize the power consumption in sensor networks. Examples include frequency and lifetime queries, dealing with events, and the creation of Semantic Routing Trees (SRT) for power-efficient information and query propagation.

A SRT allows a node to determine if any of the nodes below it will need to participate in a given query over some constant attributes. Conceptually, it is equivalent to a database index (or virtual overlay network) over attributes created on purpose and that can be

[7] network embedded systems C: `http://nescc.sourceforge.net`
[8] TinyOS Project: `http://www.tinyos.net`
[9] TinyDB Project: `http://telegraph.cs.berkeley.edu/tinydb`

used to locate nodes that have data relevant to the query (see the example in Lis. 2.1). Another index could be created for any other attribute or measured value (e. g., temperature, humidity, sound level, etc.). All sensors therefore possess a description of their capabilities, which are shared among neighbors that are applied within the construction of a SRT. In this sense, every sensor node is a combination of a database and a router able to interpret and answer as well as to forward a query and the response.

Listing 2.1: Creating and SRT on sensor locations in TinyDB, the ROOT annotation indicates the node-id where the SRT should be rooted from

```
1 CREATE SRT loc
2 ON sensors (xloc,yloc)
3 ROOT 0
```

Time is another critical issue in sensor networks. TinyDB therefore offers specific language constructs that allow incorporating times, periods, and durations. The example in Lis. 2.2 depicts such a temporal query on the sound volume. It applies a sliding window that calculates the average volume over the last 30 s with a sampling once per second that is reported every 5 s. Note the special syntax for denoting seconds, additionally hours and days can be denoted.

Listing 2.2: TinyDB sliding window query on the measured sound level

```
1 SELECT WINAVG (volume, 30s, 5s)
2 FROM sensors
3 SAMPLE PERIOD 1s
```

The usage of days in the so-called lifetime queries is useful (see the example below). This query allows gathering a complete measurement series from each sensor that is able to measure temperature over a duration of 30 days. Doing so, every sensor calculates its personal measurement period, based on available energy sources and other query claims so that a node can sustain for that lifetime.

Listing 2.3: TinyDB lifetime query for 30 days on temperature

```
1 SELECT nodeid, temperature
2 FROM sensors
3 LIFETIME 30 days
```

Dealing with events is another feature of TinyDB, which can provide significant reductions in power consumption, by allowing systems to stay dormant until some external conditions occurs, instead of continually polling or blocking on an iterator waiting for some data to arrive. The query language therefore allows it to react to events (Lis. 2.4) as well as releasing them (Lis. 2.5). The example below shows the generated response to an event that is compiled into the sensor node. Every time a motion event occurs, the query is issued from the detecting node. The average volume and temperature are collected from nearby nodes once every 2 s for a period of 30 s.

Listing 2.4: TinyDB released response query to a detected event

```
1 ON EVENT motion-detect (loc):
2    SELECT AVG (volume), AVG (temperature), event.loc
3    FROM sensors AS s
4    WHERE dist (s.loc, event.loc) < 10m
5    SAMPLE PERIOD 2s FOR 30s
```

In contrast to the response to a low-level API event, queries itself can also be applied to signal events (onto which other TinyDB sensors might react). The example below releases a signal whenever the temperature is above a certain threshold.

Listing 2.5: TinyDB release of a "software" event in response to a query

```
1  SELECT nodeid, temperature
2  WHERE temperature > thresh
3  OUTPUT ACTION SIGNAL hot (nodeid, temperature)
```

Compared to the virtual sensor concepts, TinyDB is a great approach that effectively applies a well-known metaphor (relational databases) to an alien field of application (sensor networks and stream processing). It thereby starts from a limited set of functionalities combined with techniques for efficient routing in sensor networks. And in contrast to the virtual sensor concept, TinyDB has been successfully applied in different research projects, such as the Great Duck Island [202] deployment or the Redwood Monitoring Project [208].

2.3.3.3. OSGi & JINI

JINI[10] (cf. [218]) and the OSGi[11] framework (cf. [136, 131]) are both based on the Java platform and are so-called "service delivery platforms" that are used to tackle modularization, collaboration, and service discovery in distributed systems. The OSGi applies a central "service registry" to announce and request services similar to ROS, while JINI applies a "lookup service". This shows already a difference in both technologies, because OSGi targets more closed and static (smaller) environments running on one JVM (whereby a bundle can also be a proxy representing external devices), while in JINI clients and services are expected to run on different JVMs. A service in both cases is represented by a Java interface definition, which means that an application only has to be aware of this "native" Java interface, while all network-related issues that are required to call the remote code are hidden behind the frameworks.

In contrast to fixed ROS services, JINI and OSGi allow services to be dynamically installed, started, stopped, updated and even uninstalled. Additionally, services and clients can join or leave a federation anytime. JINI furthermore enables code mobility according to the Java write-once-run-everywhere property of bytecodes, which means that the service can be executed remotely. Nonetheless, JINI can also load this functionality into a process (locally) even while the process is running. The application is not aware of this; everything is handled by and hidden behind JINI.

The OSGi is also an alliance formerly known as "Open Services Gateway initiative", providing open vendor standards, paving the way for interoperable service definitions, and also setting standards for managing devices remotely. In [136] OSGi is introduced as the enabling framework for "smart spaces", with commercial systems like ProSyst[12] built on top of it, which is also designated as AmI or IoT middleware. Matilda's Smart House[13] is a research project where OSGi is used as a service mediator between smart phones and other wireless technologies to create something like a "magic wands" that help elder and handicapped people to interact with, to monitor, and to control their instrumented surrounding. Atlas, another service delivery platform build on top of the OSGi framework, was combined

[10] https://river.apache.org/
[11] http://www.osgi.org/Main/HomePage
[12] http://www.prosyst.com
[13] http://www.icta.ufl.edu/gt.htm

in (cf. [91]) with CAMUS a "context-aware" middleware for robotic companions (see therefore Sec. 2.3.3.5.2). In [116] it was applied as middleware layer for services provided by actuators and sensors to enable so-called "pervasive spaces".

The virtual sensor concept and service delivery based on JINI or the OSGi framework are conceptually pretty close (although existing in different domains). But still, the developer has to define what services are called and in which order, in contrast to TinyDB, which allows access to information in a more declarative way.

2.3.3.4. Player/Stage

Although not directly an information or feature extraction system, it was described to be applicable in context-aware intelligent environments (cf. [126]), since the Player component itself is a device repository server for robots, sensors and actuators, or in other words, for smart devices. Such devices are abstracted in Player (cf. [84]) by an interface (similar to the previous ones) and a driver, which might be the connection to a physical device, an algorithm (similar to a virtual sensor) that receives data, processes and transforms it, but also a "virtual driver" that creates arbitrary data when needed (e. g., a noise or a random number generator).

As further described in [126], "feature drivers" can be applied as virtual gateways, offering advanced processing, learning, and feature extraction "services" in order to run automatically on streams of raw data. Examples are principal component analysis, independent component analysis, wavelet analysis, Fourier analysis, etc. Based on the input configuration options, given by the user, it is also possible to search for an appropriate cascade of feature drivers.

A mandatory feature for providing such service compositions is context information, covering more than the description of an interface and its parameters. This topic gets briefly discussed within the next paragraph.

To close with Player/Stage or in this case Gazebo [119] as the second component of this project, Gazebo is a 3D robotic simulator (see Sec. 2.3.4.1.14). It is tightly connected with Player and applied for research in multi-agent autonomous systems and high fidelity robot simulations. In the context of this approach it was used to simulate a real robotic environment, the AwareKitchen [126, 186, 185]. The purpose of this simulated environment is not only to visualize and replay data captured from Player drivers but also to develop and test algorithms without using the real hardware.

2.3.3.5. Context-Awareness

Before entering the realms of knowledge representation and to close with information and services, context has to be considered as an additional sort of information that is required by all information extraction processes and that is mandatory for dynamic service compositions (cf. [203]). Especially in smart and ubiquitous environments it is important for agents, services, and devices to become aware of their contexts to be able to adapt themselves to changing situations. Therefore, there exist a couple of systems and even middleware approaches that are dealing exclusively with the acquisition and extraction of context information. Examples include the aforementioned CAMUS that was combined with Atlas in (cf. [91]), the context-aware middleware for pervasive homecare (CAMPH) (cf. [174]), or the Context Toolkit (cf. [61]) as one of the most prominent and earliest instances. The following two sub-paragraphs are used to shortly introduce two totally contrary types of

context systems CAMUS, which is based on an ontology, and the "service oriented" Context Toolkit, based on a small set of information and dynamic context services.

But what is context actually? According to [60], a context can be defined by anything that is relevant to fulfill a certain task, like the task definition itself, sensor measurements, location, proximity or infrastructure, time, physical conditions, and many more. Examples, thus, range from environmental attributes such as noise level, light intensity, temperature, and motion, up to system and device capabilities, available services, user intention, etc. Or as defined in [24], context is any information that can be used to characterize the situation of an entity. An entity includes a person, a device, a location, or even a computing application.

The problem with context is that according to the applied classification scheme, context information somehow belongs to every layer, such as additional information about the location, capabilities, criteria of an actuator (source layer), information about transmitted data such as format, frequency, units, uncertainties (data layer), information about the quality of a service, additional input parameters and settings, inaccuracies (information layer), information about the class of an entity. Whether it is theoretically in the coverage area of a certain sensor has to be accounted as context, but this also requires some kind of knowledge base or even a world model (knowledge layer). This fact becomes also obvious by looking at some surveys and classification approaches on context systems. For example, in [201] context systems are differentiated according to the applied modeling approaches, data structure, and representation. In the simplest form these are key-value models, attaching a certain property to an entity or message (e. g., a location, uncertainty, etc.). They are easy to manage, but cannot be structured in order to provide sophisticated algorithms for context retrieval. More complex representations are based on Markup Scheme Models such as ContextML (cf. [118]), which offers a common XML scheme for describing and exchanging context data between different components. Furthermore, it offers commands that enable context providers to register themselves at so-called "context broker" that are used for discovery. This is in fact quite similar to other technologies mentioned before, but with a slightly different focus. Graphical models such as the Unified Modeling Language (UML) or Object-Role Modeling (ORM) are the third type of contextual modeling according to [201].

2.3.3.5.1. Context Toolkit[14]: This toolkit is one among the first toolkits providing a general concept for building context-enabled applications (cf. [188, 151]). It thereby took over some metaphors that were applied in the GUI programming domain. The basis for this are so-called context widgets, which are considered to hide hardware complexity, offer a general abstraction, and provide reusable and customizable building blocks for context sensing. A widget, as described in [188], can therefore consist of either "generators", "interpreters", and/or "servers". A generator acquires raw context information from hardware or software sensors. An interpreter is thought to transform low-level context information into high-level information or to acquire information from multiple other widgets (e. g., a "meeting" widget could rely on multiple "presence" widgets, while a "presence" widget could rely on multiple generators, abstracting information from batches, image processing, floor sensors, etc.). Taking composition one step further, a "server" widget is applied to collect and store, also in addition to interpreting information from other widgets, which is considered to be similar to a GUI container widget like a frame or a dialog box. An example for this could be a "person finder" widget. Widgets are hardwired classes with predefined attributes, such as the location it is used in and the time of the last activity. Callbacks are triggered, if a change has occurred, informing other widgets or applications.

[14] `http://www.cs.cmu.edu/~anind/context.html`

Neglecting the fact that the Context Toolkit applies event-based communication, while the Virtual Sensor (described in Sec. 2.3.3.1) requires continuous communication, both concepts are actually quite similar and can be applied to solve the same problems. Widgets and Virtual Sensors can be distributed into the network, both are used for abstracting real hardware through services like "presence" or "location" (whereby the hardware itself might change due to migration), and they can be composed to abstract more complex information (interpreters vs. aggregators).

2.3.3.5.2. CAMUS: The Context-Aware Middleware for URC Systems, in short CAMUS, provides a context-aware infrastructure for network-based intelligent robot systems and in contrast to the Context Toolkit, it is an ontology-based context modeling and reasoning approach. The Ubiquitous Robotic Companion (URC) is a new concept for network-based intelligent robots (similar to UbiBots, which were introduced in Sec. 2.1) and is promoted by the Korean government. The main idea of URC is to distribute functional components through the network and to fully utilize external sensors and processing servers.

The architecture of CAMUS, as it was described repeatedly and in very detail in [110, 194, 155, 98], is unfortunately very complicated and to some extent also misleading. Moreover, there are no downloadable implementations that could be used to clarify some misunderstandings. But if it is working as described, it is a very sophisticated system that already bridges into the layers of knowledge representation and intelligence.

In principle, it consists of three major components (cf. [194]), these are feature extraction and mapping, an ontology repository, and a reasoning engine. Feature extraction is handled statically, based on the capabilities of a sensor. For example, a camera sensor can be applied to acquire features such as luminosity, pixel-change, motion-patterns, etc. Changes are transmitted via an event-based middleware PLANET [110], which is also used to inform applications about occurred changes and to interconnect most of the main modules. [110] demonstrates how such monitoring services can be provided by mobile URCs.

A part of the ontology repository is responsible for quantifying such features and translating them into so-called "feature tuples", based on metadata (i. e., device information, sensor access mechanisms, feature-context-labeling rules, and input and output capabilities of pluggable reasoning modules) and applicable functionality. Feature tuples consist of a symbol and an associated probability, for example, the result of a low-level luminosity could be {`(Dark, 0.8), (TotalDark, 0.2)`}. The ontology repository is also responsible for storing these continuously updated streams of features & tuples, which can be interpreted as the current environmental (context) state. Additionally, it also hosts the applied domain ontologies. In CAMUS context is partitioned into five main categories: namely agents, environments, devices, locations, and time. For example, the top-level concept agent can be further "sub-classified" into persons, software agents (robots included), groups, or organizations; it has a certain activity profile. Device ontologies describe the type, a set of offered services, associated profiles, etc. The difference between location and environment is that the first is used to denote the physical surrounding (based on the NASA space ontology[15] [177] that was further extended with concepts such as `BedRoom`, `LivingRoom`, `DiningRoom`, etc., in order to comply with the home domain), while the second is used to encapsulate environmental conditions (e. g., humidity, sound, light, etc.). Since all facts about entities and their relations are already available in an abstracted representation (CAMUS makes use of Web Ontology Language (OWL)), reasoning is pretty simple.

[15] SWEET: `http://sweet.jpl.nasa.gov/ontology/space.owl`

The reasoning engine is the third major component of CAMUS, it is actually a collection of reasoning modules. The authors argue that there are different kinds of logics that would support inferencing. Thus, based on the available facts, it is possible to apply first order logic, temporal or spatial logic, as well as Neuronal or Bayesian networks, fuzzy logic, or (even better) to apply them in combination. The examples below are taken from [98] and demonstrate the application of Jena in querying the "world model" in Lis. 2.6 and in extending the rule base by defining new relations Lis. 2.7.

Listing 2.6: Jena RDQL knowledge query (variables are denoted with a leading `?`) intended to identify meetings a person is going to attend, by comparing a person's location with the meeting venue information. Angle brackets enclose relations (static facts or inferred), see the definition of `test:isAttendantOf` in Lis. 2.7.

```
SELECT ?meeting
WHERE  (?user,  <rdf:type>,           camus:Person),
       (?user,  <camus:hasLocation>,  ?room),
       (?room,  <test:isVenueOf>,     ?meeting),
       (?user,  <test:isAttendantOf>, ?meeting)
USING  camus  FOR  <http://etri.re.kr/URC/CAMUS#>,
       test   FOR  <http://etri.re.kr/URC/TestProject#>,
       rdf    FOR  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Jena is Java framework for extracting data from and writing data to RDF graphs (in this case, the CAMUS ontology repository) and it is able of applying different reasoners in its own (cf. [144]).

Listing 2.7: User-defined inference rule that denotes the new relation (`isAttendantOf`), "similar" to Prolog syntax, whereby all prerequisites have to be fulfilled.

```
@prefix camus: <http://etri.re.kr/URC/CAMUS#>
@prefix test:  <http://etri.re.kr/URC/TestProject#>
@include <file:D:/ContextManager-Prototype/RULE/camus_ruld.ckr>

[meeting_attendant:
       (?person   rdf:type            camus:Person)
       (?person   camus:hasLocation   ?room)
       (?meeting  test:hasVenue       ?room)
       (?meeting  rdf:type            test:meeting)
       (?room     rdf:type            camus:Room)
  ->
       (?person   test:isAttendantOf  ?meeting)]
```

Additionally, the Programming Language for Ubiquitous Environments (PLUE) was developed on the basis of CAMUS and Java (cf. [51]). It allows defining so-called event-condition-action rules as an interactive response to context changes that are identified by the CAMUS system. The PLUE example below was taken from [50]. It defines a new task for switching on the light and adapting the room temperature to a person's preferences, if he or she enters the room. The objects marked with a leading `$` refer actually to CORBA objects (not PLANET) and can be replaced by the current context.

Listing 2.8: PLUE exemplary task definition

```
task SmartRoom {
    on ( $place.UserEntered e ) {
        $place.light.turnOn(); }
    on ( $place.UserEntered e ) {
        if ( $place.temperature > e.user.preferred_temp.high ) {
            $place.air_conditioner.turnOn(); }
    }
}
```

2.3.3.6. Summary

The presented systems so far, although quite different, represent only approaches suitable for information generation, extraction, and sharing. As mentioned in the introduction of this section the association of an approach to a certain layer is not always possible, which is especially true for systems such as CAMUS and others that allow to reason about context and other environmental aspects (on the basis of ontologies). Reasoning is one issue that was initially stated to belong to the knowledge layer and above. And of course, RDF, RDQL, etc. is also used to support robotic applications, which is described in more detail within the next section. But the reasoning provided by CAMUS and other approaches on this layer is mainly about deriving supplemental information and supporting dynamic service composition on a very rudimentary level.

2.3.4. Knowledge-Layer

As introduced, this section deals with the abstract concept of knowledge representation, but with a special emphasis on representing the environmental state, thus world modeling. Every system that is somehow interacting with its environment possesses some form of world model whether implicit (as part of the program or code) or explicit (in a shareable representation). Because there are many different types of world models that are applied for different purposes (by algorithms resident at the intelligence layer) such as localization and navigation, learning, reasoning, situation-awareness, forecasting, etc., this section is divided into two parts. The first part is intended to give a short and general overview on common world modeling techniques or representations applied in the robotics domain (i. e., spatial, formal, logical, (temporal)), while the second part deals with approaches that offer a methodology for sharing world models.

The focus herein lies on the representation (what is shared?) and on technologies & concepts (how it is accomplished?).

2.3.4.1. Common Robotic World Models — An Overview

There is a huge amount of complex representations that can be applied for a diversity of problem-solving mechanisms in robotics. The fact that most of the earlier research in "mobile" robotics dealt with navigation and localization becomes particularly clear in the state-of-the-art overview [27] from 1992 and the by the synonymously usage of the terms "map" and "world model".

As elaborately discussed in [207], robotic mapping is still a highly active research area that is far from being fully solved, especially for unstructured dynamic and large areas. A map is simply a spatial representation of the physical environment, which in the field of mobile robotics is acquired through sequences of sensory observations. Thereby, a certain type of map can be generated with the help of a range of sensory systems (e. g., cameras, range finders using sonar, laser, and infrared technology, radar, tactile sensors, compasses, GPS, etc.). On this basis, information is abstracted and composed to knowledge. These maps are commonly used for navigation, path & motion planning, and localization. The following comparison is intended to highlight the most popular formats only, although there are far more different types available (especially in research).

2.3.4.1.1. The Grid Model: This model was developed by Nilsson, during the late 1960 for the Shakey-robot (cf. [159]), as one of two types of environmental abstractions (see paragraph Shakey the Robot on page 35 for the second). It was one of the first applications of an adaptive cell decomposition used for navigation [159, 160]. Hence, the whole area is simply represented as a nested grid of 4×4 cells, which are either labeled as full, partly full, or empty. Partly full cells contain a further (smaller) grid of 4×4 cells, which are labeled in the same manner. This method allowed the representation of the environment with any desired level of detail, while occupying only a minimal amount of memory. In fact, quadtrees (cf. Fig. 2.5a) are similar, the only difference is that they use a decomposable 2×2 grid, whereby newer versions also support label cells with probabilities instead of empty or full (cf. [123]). OctoMaps are thus a continuation for 3D mapping (compare it with paragraph on OctoMaps).

2.3.4.1.2. Occupancy Grid Map: It is probably the most applied format of all. It was initially developed in the mid-Eighties by Elfes and Moravec [156, 70] for mapping-based on wide angle sonar, but it can be used in conjunction with other sensor systems [18]. In contrast to the grid model, occupancy grids maps consist of a fixed set of cells of equal size and each cell is labeled with a certain probability of being occupied or not. The probability values of each cell ranges between $[-1, +1]$, whereby a value in between $[-1, 0)$ mark the probability of being empty, values in between $(0, 1]$ of being occupied, and exactly 0 represents an unknown state (cf. with the example in Fig. 2.5b).

This simple spatial representation has been applied in various areas apart from robotics, where it is used as a standard format, such as in automotive scenarios for modeling mid- and short-range environments [222, 81]. 3D version of occupancy grids are applied in avionics [25, 197] and naval scenarios [139].

2.3.4.1.3. The Generalized Voronoi Diagram: It is a popular environmental abstraction, which, in the context of robotic path planning (cf. [133]), can be used to identify trajectories with a maximum distance to surrounding obstacles (see the example in Fig. 2.5c). Every cell (Voronoi region) of an obstacle is defined by a set of points whose distance to this specific obstacle is less than or equal to their distance to every other obstacle in the environment. Voronoi diagrams can be constructed from various sources, dimensions, and spaces (continuous/discrete) (cf. [107]).

2.3.4.1.4. Potential Fields: This representation transfers the environment into a "world" of forces, whereby attraction is exerted by the goal position and obstacles cause repulsions (see the example in Fig. 2.5d). Interpreting a mobile robot as a single particle within this environment allows solving a couple of navigation problems [83, 167]. Using potential fields in combination with manipulators, which are modeled as an assembly of rigid-bodies connected via joints, allows even to circumvent problems caused by the complexity of a kinematic chain and variable side conditions. The tool-center-point can be defined as the only point that is affected by attractive and repulsive forces — whereby all other parts of the manipulator are affected by repulsive forces only (see also the example in Fig. 2.7). The axis angles of the different joints can, then, simply be read off from the model during the movement and used to control the real manipulator. A major problem of this method is its affinity to local minima.

2.3.4.1.5. Topological Maps: This is another graph-based mapping method, whereby nodes represent landmarks (distinct places within the environment), which are connected via edges (cf. [127]). (Also see the example in Fig. 2.5f).

The most prominent example is actually the London tube map (which was also one of the first). The characteristics of the environment change significantly within this representation: it lacks of scale, distance, and direction, which makes it difficult for a robotic application to decide whether a landmark has been already visited or not. This is also the reason why this graph-based approach is mostly applied in combination with other "metric" mapping methods (e. g., [128, 121]).

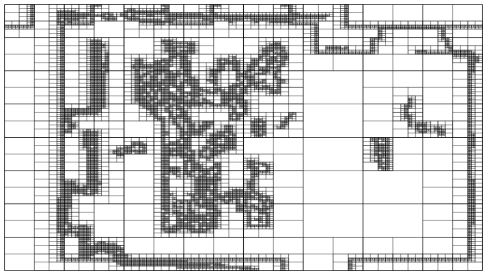
2.3.4.1.6. OctoMaps: They are not, as often mentioned, 3D occupancy grid maps only. Rather, they can be interpreted as a hybrid representation of occupancy grids and octrees (a 3D version of the previously mentioned quadtrees (cf. The Grid Model)). An octree subdivides the space recursively into 8 sub-volumes (cubes) of equal size, after which every volume can be further subdivided into 8 cubes, etc. The resolution of an octree is determined by the minimum cube size (see the example in Fig. 2.5e). In an OctoMap, all of the octree leaves are labeled with a certain possibility of being occupied. The probability of an inner node's being occupied can then simply be calculated as the average, weighted average, or as the maximum occupancy (cf. [225]).

2.3.4.1.7. Point Clouds: If the discretization of the environment should be avoided and a precise environmental representation is more important than memory consumption, then point clouds should be applied. A point cloud is simply a set of points within a (commonly 3D) coordinate system [162], which represents the occupied space. Newer versions such as PointCloud2 [184] are a de facto standard in many robotics applications (e. g., mapping (and SLAM), object identification, segmentation, and surface reconstruction) and can attach even more information to an n-dimensional point as depicted in Fig. 2.5g (i. e., colors, intensities, distances, segmentation results, etc.).

2.3.4.1.8. Shakey the Robot: Historically considered, it is worth mentioning Shakey as the first mobile robotic platform operating in an unstructured and dynamically changing environment and it was pioneering in many ways. The first version of the robot applied two types of environmental representations, a spatial one described in Sec. 2.3.4.1.1 and a property list model [78]. This property list model was used to represent everyday objects as tuples (LISP data structures), containing all relevant features, such as x- and y-coordinates of an object, angle, size, shape, name, etc. Actually, it represented a knowledge base containing all relevant facts about the objects within the environment.

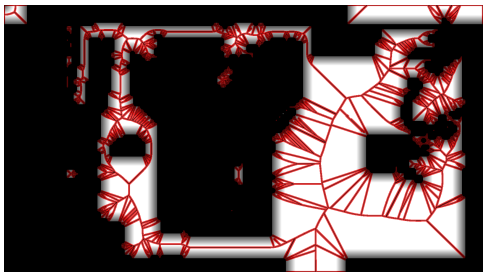
This kind of world model was intended to be used to serve human instructions, for example, "`GO TO A BOX`". It necessitates looking up through all tuples, and identifying an object called "`BOX`" within the current environment. The grid model (see Sec. 2.3.4.1.1) is then afterwards applied to determine a collision free path to the position of the object.

2.3.4.1.9. Lessons learned from Shakey: While the first version of Shakey had to maintain two distinct environmental representations, the second version was based on a logical predicates only. There were five distinct classes (i. e., `type(d1,door)`, `type(f2,wall_face)`,

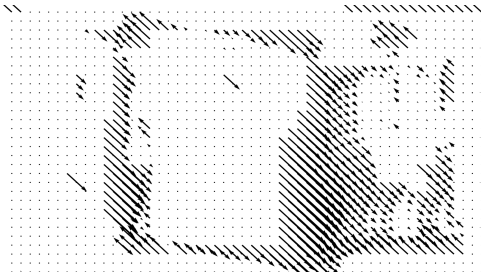


(a) Grid Model & Quadtree

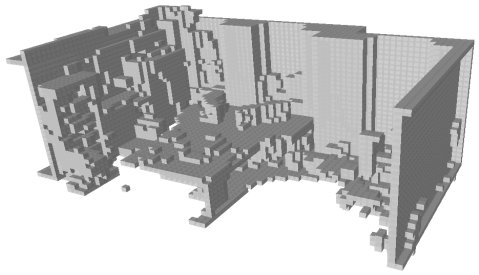
(b) Occupancy Grid Map



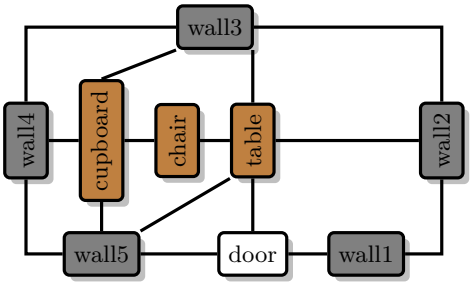
(c) Generalized Voronoi Diagram



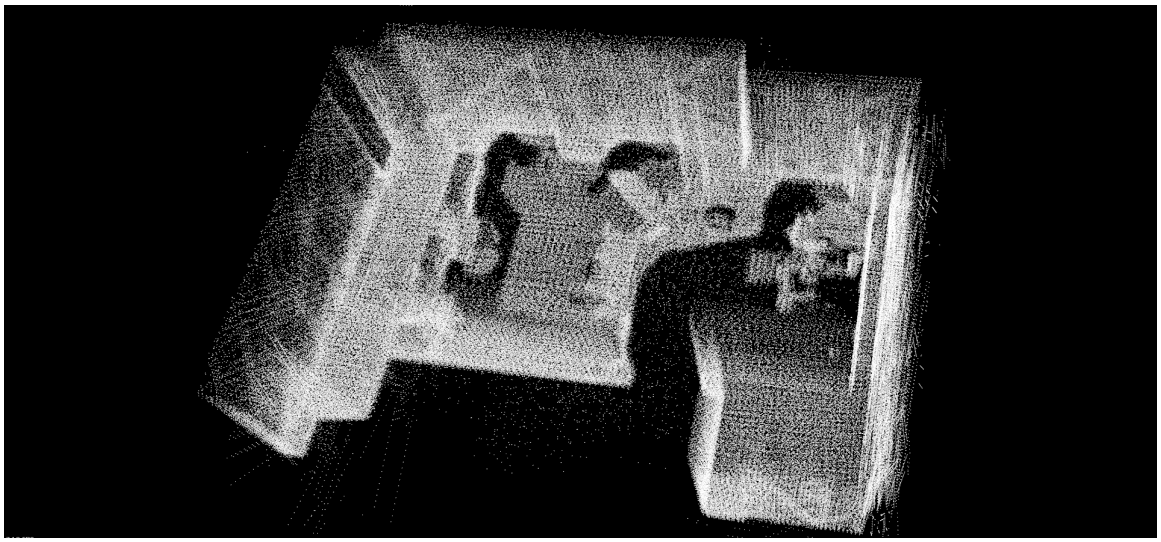
(d) Potential Field



(e) OctoMap



(f) Topological Map



(g) Point Cloud

Figure 2.5.: Different types of spatial world models

`type(r315,room)`, `type(o1,object)`, and `type(shakey,robot)`) used to represent all entities. And further predicates were used to define relations between objects, such as `in(shakey,r1)`, `at(shakey,7.0,11.0)`, `name(o1,box)`, `inroom(o1,r2)`, `joinsrooms(d1, r1,r2)`, `doorstatus(d1,open)`, etc. These predicates are stored in an indexed data structure, which can be directly used as an input for Stanford Research Institute Problem Solver (STRIPS) [75], which is a planning system "similar" to GOLOG (see Sec. 2.3.4.1.11), and which was used to derive sequences of actions in order to convert the current environmental representation into another one where a goal is accomplished. All positions, distances, lengths, and angles are measured with respect to a single base coordinate system. Action routines are responsible to update the statements of the model, such as the position of the robot and its location while moving.

This kind of "holistic" logical representation which integrates spatial information has a couple of advantages. It can be applied either to solve problems such as identifying appropriate action sequences, based on STRIP, or it can be used to reconstruct the spatial model of the environment in its current configuration. It can be used as a more natural representation in terms of human-machine interaction because the applied predicates can also be easily interpreted by humans, which allows formulating facts about the environment that can be easily interpreted and exchanged by humans and Shakey.

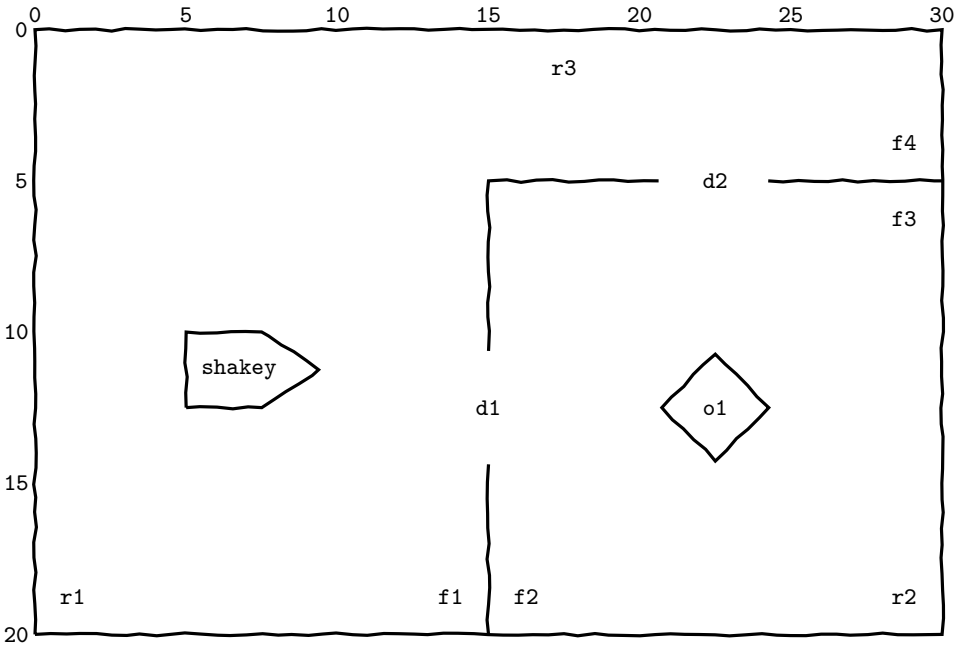


Figure 2.6.: Replica from the STRIPS model presented at the plenary talk "Making Shakey" ICRA 2015.

2.3.4.1.10. Prolog: It is actually not a world model, but can be used for building logical ones and to perform reasoning on them. Prolog is as an acronym, which stands for "PROgrammation en LOGique" and describes a new programming paradigm (i. e., declarative and logical). It was introduced in the early 1970 by Alain Colmerauer [45] and can be seen as the technical counterpart to the theory of mental logic (cf. [99]). Prolog "programs"

are expressed in terms of first order logic by facts and rules. The computation, done by the Prolog-interpreter, is the "simple" search for an answer to a query, on the basis of the defined rules and facts. There are two types of possible results, positive ones, if the answer could be deduced from the knowledge base, and negative ones, which means that the answer could not be deduced. This means that there is a need to extend the Prolog-base with additional knowledge. This lack of knowledge will be handled in more detail within the next paragraph (cf. GOLOG).

But Prolog can surely do more: In addition to its applications in expert systems [52], natural language processing [55], or theorem proving [199], it can also be used to implement further declarative languages. This ability was actually stated to be another programming paradigm, called Language-Oriented Programming (LOP), which was first described in [219]. Prolog is already frequently used in robotics (see, for example, Sec. 2.3.4.2.11) but LOP allowed the development of a couple of new languages. As also sketched within the following paragraphs, to be able to deduce answers on the base of "logical" world models requires a very specific and abstract environmental representation (which can be hard to create and to maintain). Thus, reasoning on maps or and other kinds of knowledge representation is not possible. Nonetheless, as it is shown in Sec. 3.3.2, there are also other methods or concepts that can be used to reason on different kinds of knowledge representations and that also support the idea of LOP.

2.3.4.1.11. GOLOG: Although derived from alGOl in LOGic, it is a famous example of a logic programming language developed with Prolog (cf. [142]). It is based on the situation calculus [152], a sorted second order logical language with equality, which is intended to reason about (robotic) actions and their effect on the environment. Thus, the whole configuration of the environment (including objects and their states) is encoded within logical clauses. Starting from the initial configuration s_0, new situations s_{i+1} can be derived by applying a primitive action a to the current situation with function $do(a, s_i)$. A situation is thereby defined as a history of actions that has been executed, though not by description of the resulting state. Additionally, there are so-called "fluents", which are functions that change their result value, allow the query of current situation. For example, $location(Table, s)$ might result in an (x, y) position, whereas $over(Mug, Table, s)$ returns a true value.

GOLOG is quite frequently applied in the field of cognitive robotics, it is ideal to describe and verify situations as well as to deduce action sequences that result in specific "goal"-situation. But this kind of environmental representation has also some drawbacks. All actions are executed sequentially, which means that there is no support for concurrency (e. g., two robots in one operational area) and it does not take into account the occurrence of external events. Furthermore, this representation does not support continuous actions, because every action is executed immediately, leading to an immediate change of fluent results. For example, the execution of the action $do(lift(Robot, Mug), s_i)$ would instantly change the result of the fluent $over(Mug, Table, s_{i+1})$, although the process of lifting and grabbing is continuous and requires some time. This is why projects like RoboEarth (see Sec. 2.3.4.2.11) combine the logic representation with continuous 3D simulations since the change of real world positions or relations such as $left\_of(Object_1, Object_2)$ happens continuously, which can be better modeled and checked in time-continuous simulations.

2.3.4.1.12. ConGOLOG: This is a more realistic implementation of GOLOG [57], realistic in terms of dynamically changing environments. It is able to deal with incomplete knowledge and it also respects the occurrence of external events (and thus acquires information at runtime). This is a useful feature and allows reaction (immediately) onto changed environmental conditions while working on a task. This ability is made possible due to the support of concurrent processes (which is also the origin of the name) with different priorities and interrupts.

2.3.4.1.13. Further GOLOG Representatives are, for example, IndiGolog [58], an extension to ConGolog, that supports the inclusion of planning and sensing components within incremental and deterministic programs. ReadyLog [73] aims at providing a deliberative component for multi-robot systems under real-time constraints. It has some extensions similar to ConGolog, but it also extends GOLOG in terms of probabilistic projections, non-deterministic actions (which leave certain decisions open), and planning on the basis of Markov Decision Processes. Legolog [141] is a GOLOG implementation for Lego MINDSTORM robots, intended to run on various Prolog implementations with minimal requirements.

2.3.4.1.14. 3D Rigid-Body Models: The application of 3D rigid-body simulations is a widely exploited technique in robotics. In contrast to formal/logical or purely spatial abstractions (maps), the environment is represented more "realistically" in a scene-graph. Objects and obstacles are represented as closed shapes (with colors, textures, and correct spatial relations) and robots as compositions of objects with joints, links, and attached motors. The term more realistically does not only point to the representation, which includes the generation of simulated sensor measurements, but also on the application itself, which means that it can be used to try out something or to check whether a task can be fulfilled. For example, it is possible for a manipulator to try out various trajectories (by emulating motions) before grasping an object. Of course, something similar could also be accomplished with the help of the situation calculus (cf. 2.3.4.1.11) and the following (exemplary) predicate, but only with a lacking environmental representation:

$$\begin{aligned} Poss(pickup(Manipulator,\ Mug),\ s) \iff near(Manipulator,\ Mug,\ s) \wedge \\ \neg\exists\ object\ is\_holding(Manipulator,\ object,\ s) \wedge \\ \neg too\_heavy(Mug) \wedge \\ \dots \end{aligned}$$

Some rigid-body simulation environments are specialized on one type of robot only, but next to the replication of the environment and the robot they also offer an exact replication of the robotic interfaces. This allows to port code directly onto the target platform without any modifications. Such environments are commonly used in the scientific robotics community for studying and learning robotic behaviors. Examples are the UCHILSIM [229] a simulator that was especially developed for the AIBO four-legged league RoboCup, or ARGoS [171] that was initially developed for the Swarmanoid project (cf. [65]).

Others are intended for testing and developing complex motion planning algorithms for various types of robots, such as Klamp't [90] with support for legged locomotion and manipulation, or OpenRAVE [62] that focuses on analysis of kinematic systems. This last simulation environment is described in more detail in Sec. 4.2.1, because it has various advantages and I have used it in important parts of my research.

Similar approaches also exist in the area of industrial factory automation, where simulators are applied during the design process, for rapid prototyping, algorithm and regression testing, remote monitoring, safety double-checking, etc. Some environments allow programming robots in their specific robot language (e. g., ABB Rapid, Kuka KRL, Siemens G-Code, etc.) — also see the supported languages of Workspace 5 [104], and other examples are COSIMIR [80], the WorkCellSimulator [209], or V-REP [180], etc.

However, simulation environments like V-REP and OpenRAVE can further be used to control the real hardware. It is thus applied as motion planner that searches for collision free trajectories in a virtual environment or for an optimal gripping position, and if it succeeds, the same motion is replicated in the real world (cf. [243]). Thus, there is no need for a fixed and possibly optimal trajectory. It is only required to adapt the virtual environment according to the configuration of real surroundings.

2.3.4.1.15. Roy's Mental Imagery: The presented concept in [183, 182] probably had the largest influence on my research, although its main objectives lie in spatial and situated speech understanding and, thus, human-machine interaction. The main idea sounds simple and straightforward: if humans and robots interact in the same or overlapping workspaces, then robots should possess a mental model as close as possible to the human, which allows them to be aware of their situational (spatial & physical) context as well as to take over other perspectives. For example, to be able to properly interpret instructions such as "Give me the blue screwdriver on my left!", a robot must connect the meaning of spatial language with the perceptual and action systems in a situational context.

The environment is therefore represented as a 3D model (similar to the screenshot in Fig. 2.7), which comprises a model of robot itself, a built-in model of the workspace, a model of the human co-worker, as well as models of all objects situated on the work surface. Along with the purely spatial and geometrical representation, objects are also labeled with quantities such as masses, velocities, colors and textures. A combination of OpenGL and the ODE is used to implement this representation. ODE is a widely applied physics engine that consists of two main parts, a rigid-body dynamics simulation engine and a collision detection engine, which means that every object is resident in two engines ("abstractions"): on the one hand, as a "body" in the "world" and on the other hand, as a "geom" in "space". The world or rigid-body simulation engine keeps track of body with masses, forces, friction, positions and orientations, etc., while the "space" is used as a container for the geometrical representation and collision detection. OpenGL was applied for visualization and mental imagery generation (changing camera perspectives).

The model represents the robot's belief in the state of the world. In the original scenario it is updated and maintained with the help of two cameras, mounted onto the robot, and proprioceptive sensors (position and force) that were placed on each joint. The camera systems are used to identify new objects and to keep track of them. Proprioception is used to identify the correct position and orientation of the cameras in relation to the environment; thereby, every movement is replicated within the virtual representation. The model is furthermore persistent, which means that objects are kept in place, even if they are out of sight due to camera movement or movement of the object. ODE is thus applied to predict the positions of the moving object, based on mass and the identified speed of the object.

[15] Tutorial website: `http://www.aizac.info/projects/ode-viz`
Download website: `http://sourceforge.net/projects/ode-viz`

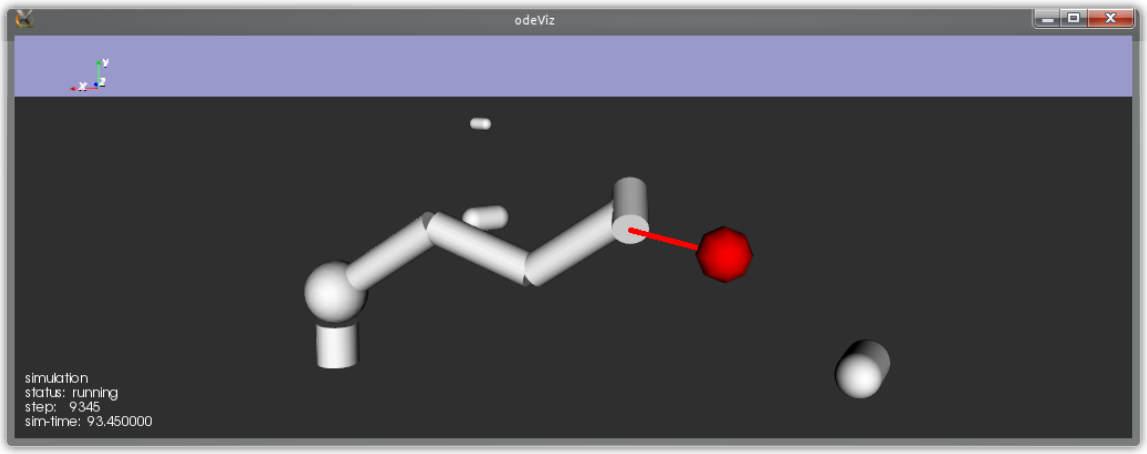


Figure 2.7.: My first attempt of replicating Roy's mental imagery, which was also based on ODE. The simulation shows a model of a Katana manipulator (5 degrees of freedom) whose tool-center-point is attracted by a movable red sphere to indirectly determine the appropriate link states. As in Roy's example, the environment contains additional obstacles. To simplify the visualization I had developed odeViz a visualization library for ODE on the basis of VTK. odeViz[15] automatically analyzes the state and the content of the simulation (world and space) and generates & updates the visualization.

A mental imagery is thus only the visualization/rendering of the entire scene from another perspective rather than from the head-mounted cameras. The robot that is also keeping track of the human within the model is thus always able to switch to the human perspective and to see and interpret the environment "literally" through the eyes of its human co-worker and, furthermore, to ground spoken language/commands into the spatial context. For example, by trying to satisfy the prior command, the robot is now able to identify the right screwdriver, based on color, shape, and position, but also from a set of screwdrivers where another one might be hidden from the perspective of the human.

2.3.4.2. World Models in CPS – Sharing Knowledge Systems

In contrast to the previous general overview on explicit world models, the following examples provide actual approaches that are intended to be shared between different heterogeneous systems such as systems with different sensors, actuators, tasks, and varying environmental conditions, thus real CPS as they were introduced in the first section. This overview actually resembles a colorful potpourri compared to the previous layers, but it also emphasizes that the research in that area is relatively recent and, therefore, relatively specialized in a few application scenarios only.

2.3.4.2.1. Object-Oriented World Model: Furda and Vlacic proposed such a world model in [82] for road traffic environments of autonomous (driver- less) city vehicles. The described concept is pretty straightforward, important aspects of the environment are therefore encoded within objects and classes. Different classes are used to represent a set of different predefined concepts such as vehicles, pedestrians, traffic signs, lanes, etc. An object is thus used as a specialization and represents a specific car, with a 2D position on a certain lane, a certain velocity, etc.

This kind of object-oriented world model is set up with a-priori information obtained from on-board sensors, and also through car2car communication, which means that the awareness about other vehicles (whether they are in front or behind) can be obtained either from the local sensor system, external traffic monitoring systems, or from the communicated position of the front/back car itself. The actual control application is thus cut-off in total from lower level sensor data, component descriptions, etc., operating only on the basis of the abstract world model, see Fig. 2.15.

This type of modeling seems to be appropriate for (hard coded and fast) situation assessment, because all required information is already translated into a simplified structure with well-known semantics. From this point of view it actually mirrors CAMUS, but it provides fewer opportunities for applying further and extended reasoning techniques (which are probably not required in this context.

2.3.4.2.2. Local Dynamic Map: A LDM is conceptually an "Intelligent Transportation System" specific environmental data container, storing all data and information that might be relevant for different transportation scenarios (i. e., driving assistance — cooperative-awareness and road hazard warning, speed management, navigation- and location-based services, communication services, and station life cycle management). It was originally introduced by SAFESPOT[16], a European research project, whose main objectives lie in dynamic cooperative networks between vehicles and the road infrastructure for sharing information. Since 2011 it has become a standard and was defined by the ETSI[17] (cf. [71]). In most cases, an intelligent transportation system is interpreted as a car, but the term itself comprises a huge variety of entities (cf. [71]).

The LDM is thus the very core of the SAFESPOT system architecture, as depicted in Fig. 2.8 in addition the positioning system, the VANET router/message generation, data fusion and the "cooperative" application software. The data fusion unit is the only unit with "write-access" to the LDM, and it maintains the coherence of the LDM via situation refinement, spatial and temporal alignment. It is thus based on a fixed set of information that can be obtained from different sources (car-to-X[18]) such as local on-board sensors, other intelligent transportation platforms, infrastructure units, traffic centers, etc., which are accessed via a dedicated interface to the LDM. The cooperative applications have to be considered to belong to the intelligence level, because they are actually responsible for predictions, situation-awareness/assessment, etc. They define and react to events detected in the LDM (cf. [195]).

But how is knowledge actually encoded within the LDM and queried? Actually, it is a GIS database that is focused on extending digital maps to incorporate real-time environmental conditions. Therefore, only data and information about the real world and conceptual objects is stored, which was previously identified as having an effect on the traffic flow. All information is thus grouped into four general layers: The first (wide range) layer is a (usually static) map representation intended for global navigation only. It is based on NavTeq, OpenStreetMaps or other providers of electronic navigable maps. It has to be mentioned that a multitude of electronic navigation systems already incorporate data from various sources, such as wide range of surveillance sensor systems, mobile devices, etc. [158].

[16] Project website: `http://www.safespot-eu.org`

[17] European Telecommunications Standards Institute: `http://www.etsi.org`

[18] X represents either Car, Home, Infrastructure, Mobile, or Roadside

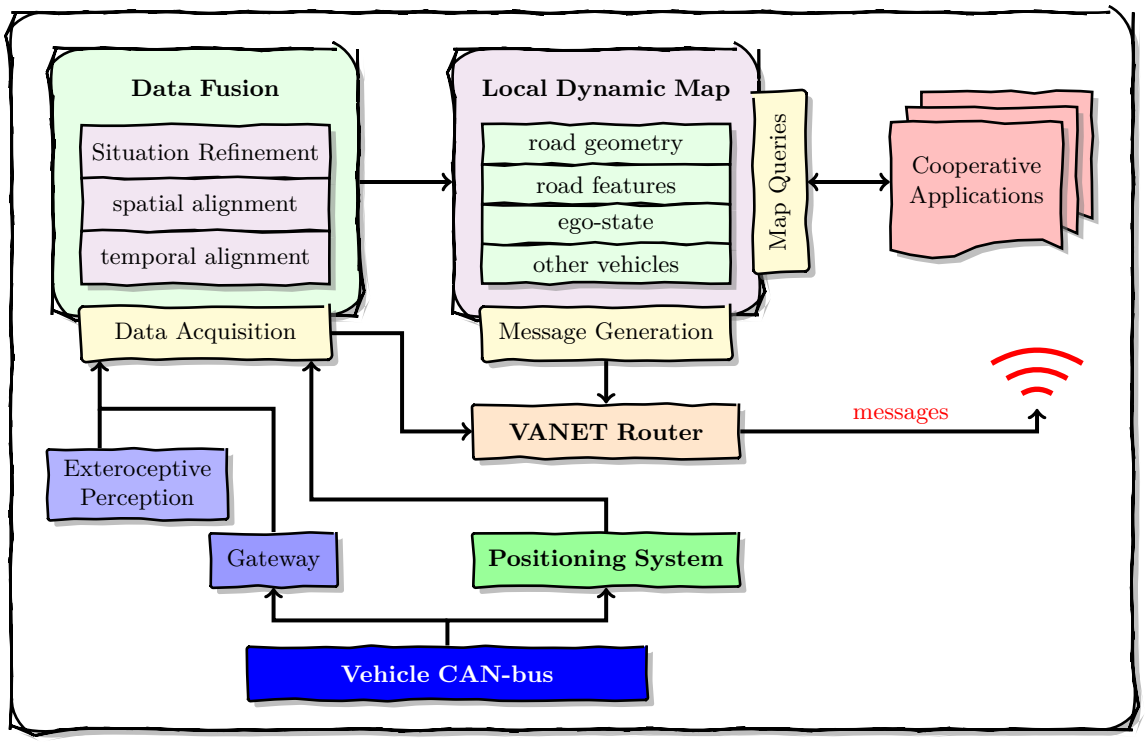


Figure 2.8.: SAFESPOT system architecture, as at [100]

The second layer is intended to store attributes that seldom change, such as speed limits, traffic signs, and visual features/patterns obtained from on-board camera systems, etc. These features can also be used for a better positioning in urban environments, where GPS accuracy might be reduced. The third layer incorporates spatial and dynamically changing information, such as road and weather conditions, information on accidents or traffic jams, etc. And the fourth layer incorporates highly dynamic information about the local environment and the vehicle itself, such as speed, direction, status of different automotive systems (e. g., ABS, ASR, ESP), as well as the positions of other vehicles and their states.

Because it is a database, querying is quite simple by applying SQL statements with a well-known syntax (as it is done by different cooperative applications), and exchanging and updating this kind of environmental representation is thus also pretty simple. All intelligent transportation systems therefore simply have to maintain consistency in their databases. Hence, an LDM has to be interpreted as some "global" world model, containing all relevant and irrelevant information. The already mentioned cooperative applications are responsible for building and updating their own local world models. As described in [195] this is done by repeated and cascaded querying, which unfortunately cannot be achieved under real real-time constraints. The described collision detection can thus only operate in time cycles of 50 ms with a large safety buffer.

2.3.4.2.3. PREDiMAP: It stands for "vehicle Perception and Reasoning Enhanced with Digital MAP" and was an international research project that tried to integrate the vehicle sensor data and external information into digital maps (cf. [48]). Indeed, its objectives are pretty equal to LDM, covering topics such as:

- perception for static map generation
- UAV mapping and disaster management with spatial information
- road edge modeling
- detection and 3D reconstruction of traffic signs

Unfortunately, however, the project did not succeed in fulfilling its objectives and therefore did not generate valuable publications or software artifacts. Its placement in Fig. 2.15 is therefore only weakly adumbrated.

2.3.4.2.4. Distributed Scene-Graphs: In contrast to the previous "automotive" representations, a distributed scene-graph, as introduced in [37], does not define different layers of data and information to be shared. Instead, it defines a structure for storing, sharing, and maintaining knowledge that is spread across multiple robotic entities. As the title indicates, the scene-graph is mainly applied for dealing with 3D data. BRICS3D is the name of the current C++ implementation [239].

The general idea is to have something like a global and common world model which can be applied by different entities and for different purposes such as perception, planning, control, and coordination. The global world is thus represented by a directed acyclic graph of geometrical and hierarchical entities, as depicted in Fig. 2.9. There are two different types of nodes: "leave nodes" are used to store any kind of 3D data, such as point clouds, meshes, primitive shapes, as well as raw and intermediate data (which still has to be processed and might be useful for further analysis), while "inner nodes" are applied to define groups and transformations. There is also an extended version of the transformation node that also deals with uncertainties. Each node has a unique ID attached to it, and additional attributes can be stored as lists of key-value pairs. It is mentioned that this mechanism allows tagging entities with semantic attributes or debug information.

To keep track with moving objects or to make predictions, each transformation node caches a limited history of transformations along with their time-stamps. It is also noted that time is used to define different strategies for when and how to delete geometric and transformation data, and thus to minimize the size of the scene-graph. The so far presented approach is currently struggling with providing automated methods for distribution, thus creating and maintaining local copies of the entire scene-graph, which is quite challenging for larger environments. But first attempts [38] towards this direction provide promising starting points. The system currently does not deal with inconsistencies that may arise due to a large amount of read and write operations from different entities.

2.3.4.2.5. DAvinCi: Inconsistency is not a real problem of the "Distributed Agents with Collective Intelligence" system, because it applies the distributed file system called Hadoop Distributed File System (HDFS) [236] for storing data across multiple machines (files up to the range of terabytes), whose current consistency model is one-copy-update-semantics (updates are written to a single copy and are available immediately). As described in [30], it is a cloud computing framework for heterogeneous service robots. The intended application of DAvinCi was to build live maps (on demand) based on heterogeneous sensor data that is stored within the cloud, but similar techniques can be applied to multi-modal map building,

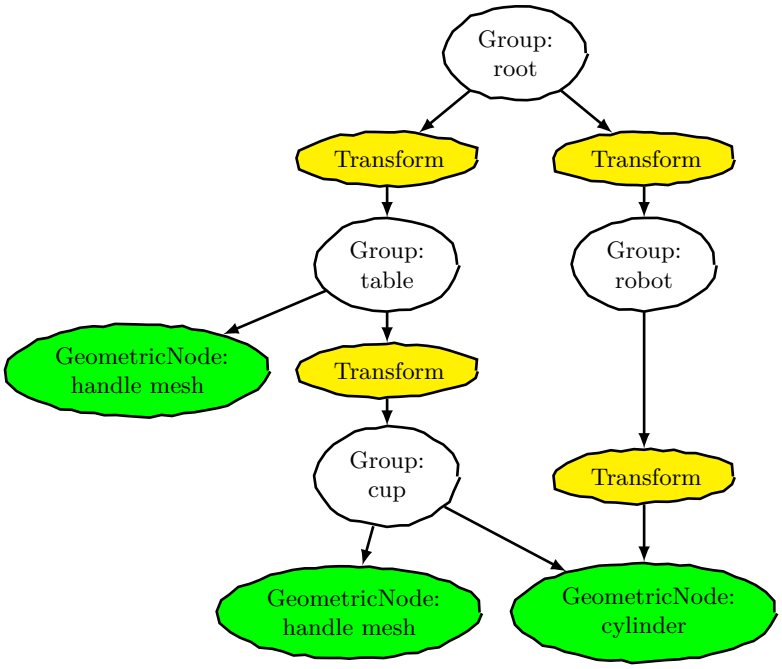


Figure 2.9.: Robot Scene-Graph example as at [239], with leaves (holding 3D data) as well as inner group and transformation nodes.

map segmentation, object recognition, etc. In general, it can be applied for any kind of parallelizable algorithm that processes a large set of input data.

Heretically it can be said that DAvinCi is actually the application of Apache's Hadoop [234] open-source framework (which does all the work) in a robotic context. At its core, Hadoop consists of two main parts, the already mentioned Hadoop Distributed File System (HDFS) distributed storage and MapReduce [235], which allows to process data with parallel and distributed algorithms.

DAvinCi applies the ROS messaging system and implements a ROS collector-node that subscribes to relevant sensor measurements (in the described case these are laser scans, odometer readings, and camera data). These ROS messages are stored in so-called ROS bagfiles (the common ROS file format for storing messages [247]), which in turn are pushed by the collector to the HDFS file system. A particle-based FastSLAM algorithm was implemented in MapReduce to demonstrate the feasibility of the approach.

The DAvinCi framework itself does not possess the ability of relating, selecting, or filtering of data; nor does it support any kind of sophisticated querying. The application that was described in [30] is actually only an implementation (offering one service) examining the possibility of applying Hadoop for map building. This is also highlighted in Fig. 2.15, compared to other approaches at this level. Unfortunately this project is not available online. Nevertheless, there are currently two other ROS projects that make use of Apache Hadoop, the megatree-project[19] for storing point clouds in large out-of-core octrees (cf. Sec. 2.3.4.1.6) and the RoboEarth-project, which is described in detail in Sec. 2.3.4.2.11.

2.3.4.2.6. Cloud Robots as a Service: The authors of [176] promote the idea of combining cloud techniques and a Service Oriented Architecture (SOA) to fulfill robotic services.

[19] `http://wiki.ros.org/megatree`

In their intended approach, a group of robots and a "smart" room share their acquired knowledge over an SOA. Whereby robots act as service provider and consumer and the amount of different services is considered as knowledge. These services are described in data-sheets and are published into a common repository, making them discoverable. If a robot fails to fulfill a specific user request, it can connect to a cloud skill repository and download the requested skill (if available). And the newly learned or updated skills can be uploaded by the robot into the cloud, making it accessible to others as well.

Information from the environment is provided by infrastructure services, but it is not described how information is selected and how the appropriate processing of information is accomplished. The architecture presented is thus very vague (as mentioned, promoting only the idea) and there is no real concept as it is applied in PEIS (see Sec. 2.3.4.2.13) or KnowRob (see Sec. 2.3.4.2.11), which fulfill the intended functionality... According to the applied conceptual hierarchy, the mentioned service-knowledge is actually related to the information level (cf. Fig. 2.15).

2.3.4.2.7. A Robot Cloud Center: As it was described in a few more details in [66], the robot cloud center is actually intended to fulfill similar tasks as the previous approach. It therefore also interprets robots as kind of service providers (so-called "robot cloud units") within a SOA. It also possesses a service repository and registry, but in contrast to the previous work it segregates services into three categories: "common" and "application" services or algorithms on the other hand, which offer general functionalities and helper functions (e. g., face detection, maze following, obstacle detection, map building, etc.); and "hardware" services on the other hand, which offer general access to sensor data and motor drivers.

There are additional components described, though, such as a mapping layer that is responsible for mapping "requested" virtual robot objects to physical robots. User requests are thereby handled by a so-called "cloud panel", which analyzes user requirements, composes new robotic applications based on existing services, deploys newly developed services, and is furthermore used to manage and analyze existing running services on individual robots.

In contrast to the previous idea, there is no notion of a smart environment, which could be applied to extend a robot's perception. Both approaches do not have a holistic concept for knowledge sharing, except data-sheets for service descriptions. Sensor data, information, and general purpose functions are accessible through services, but it is questionable how these parts can be assembled appropriately based on user demands and as prerequisites for robotic task fulfillment. Furthermore, both concepts describe aspects that are already captured, or at least tackled, by PEIS, which predated both, but did not make use of the term "Cloud".

2.3.4.2.8. Sobots, Embots, and Mobots: These different types of UbiBots represent an early but long lasting concept (cf. [113, 111, 112, 115]). It is mainly used for separation of concerns and to simplify considerations about the smart or ambient intelligent environment, all entities are simply considered as robots. Embedded robots (Embots) are sensors systems that offer some kind of "perceptive" service such as positions estimation, vision, sound, etc. Mobile robots (Mobots) are entities that offer some kind of physically acting capability, while intelligence is assumed to be handled mostly by software robots (Sobots). Thus, distributed Embots provide intelligent Sobots with context-aware perceptive capabilities and Mobots act upon service requests in the physical world.

All entities exist within so-called ubiquitous spaces (u-spaces), which consist of both a physical and a virtual environment. Different u-spaces can overlap and, as claimed in [112], interconnection of UbiBots is handled seamlessly with the help of a middleware. Unfortunately, the authors do not provide any details on their implementation and there is no system online available. As described in their middleware overview sections in [113, 111, 112, 115, 114] it is a broker-based system, allowing to make offers for services performed by Embots and Mobots. And as further indicated by the architectural overview on the implemented Sobots, every Sobot is responsible on its own to request sensor information and actuator services. It is mentioned that these services are standardized, but not how. Sensor data has to be translated autonomously into a symbolic representation, into what kind of representation and how this is performed is not discussed. It is furthermore not considered that Sobots might need to communicate, and thus, it is not described what the services they offer. At least in [114] a multi-layer architecture for the "broker" is described. It consists of a central "Device Manager", required to maintain a database with Embots and Mobots. It has a "Task Scheduler" and a "Context Provider", which offer basic services (e. g., user position, temperature, voice commands, etc.) and it also deduces the current situation by integrating all types of sensor information. Additionally, a "Sobot Management Layer" is responsible for transmitting the context information, interconnecting Sobots with Em- and Mobots, and shifting Sobots into other virtual environments.

To sum up, the concept as it has been presented so far might appear to be holistic, but it requires a Sobot intelligence level that is not satisfactory by using today's techniques. Therefore, based on the presented approach, this UbiBot concept actually belongs to the lower information level (see Fig. 2.15), and it does not possess any capabilities to share knowledge.

2.3.4.2.9. DustBots[20]: This European project [190] for urban hygiene is actually "mirroring" the previous concept. It can be regarded as an example of a networked robot system, in which two types of mobile robotic platforms cooperate with external sensor systems to provide different services. These are the DustClean robot (for autonomously cleaning and sweeping pedestrian areas) and the DustCart robot (for door-to-door garbage collection). The internal robot architecture is modular, based on IEEE 1451 (cf. [178]). Sensing modules are abstracted with TEDS and the internal communication is handled with the help of NCAP, see Sec. 2.3.1.2.

A central "supervisor" manages local data gathering and task execution as well as all the communication to the external AmI system and to all local modules. Similar to the Sobot-concepts, the intelligence is not directly located at the robot itself: instead, it is handled by a remote AmI system (cf. [74]). The AmI is a single external application responsible for dispatching robots, "high-level" path planning, and service execution. High-level path planning in this context means that the AmI maintains a complex geo-referenced map along with locations of known users, available roads, and important locations. A robot only receives a small map with the relevant part of the city together with the next goal point. Navigation is then handled with local modules, considering current environmental conditions. Once the goal point is reached, the AmI transmits the next goal point with another map. In this case, the AmI dictates the global robotic states, which are abstracted as action artifacts.

[20] Project site: `http://www.dustbot.org`

The AmI is further responsible for collecting pollution data, its processing, and map creation. As described in [74] all relevant robotic on-board sensor measurements are aggregated by the supervisor module and transmitted in 2 s cycles to the AmI with data related to time and position. This data is then stored within a central database and a user might request an image or table-like representation of current or past pollution for a certain area.

To summarize, what can be said about the DustBot system is that most of the communication is actually on the data level, knowledge is only shared in a fixed map format. Since there is only one intelligent AmI application and the number of predefined tasks is limited, it is probably not required at all to share knowledge between machines (see Fig. 2.15).

2.3.4.2.10. Ubiquitous Network Robot Platform: It is another UbiBot concept (cf. [108, 161]) that tries to combine cloud principles with the idea of networked robots and a SOA (well, there is currently no standardized definitions nor a consistent terminology). According to the NRS notion (as described in [191]) there are three types of robots: visible (embodied robots in a classical sense), virtual (only acting in cyberspace), and unconscious (external sensor systems). In contrast to the So/Mo/EmBot (cf. Sec. 2.3.4.2.8) it does not make assumptions about the degree of intelligence; thus, a visible robot (MoBot) can be as smart as a virtual robot (SoBot).

The ubiquitous network platform is depicted in Fig. 2.10. At a first glance, it is visible that it differentiates between a local and a global platform with similar components. The basis for this separation of equal components is that coordination of robots and services must be provided over a wide area. The local platform thus only covers robots within certain vicinity, whereby the global platform ranges over multiple areas, providing links to several physical points and, thus, several local platforms. Both serve as a kind of middle-layer between the service application and the robotic components at the bottom. All the required data is gathered within five databases:

1. Robot Registry: data on available robots, such as IDs, shapes, mobility capability, transporting capability, etc.

2. Map Registry: “local” data about the service execution environment, floor properties, information about movable and no-go zones, whereby the “global” database provides positional relations among single areas.

3. User Attribute: data on service users, such as IDs, call name, degree of walking ability, sight and hearing abilities (since it is intended to support disabled and older people).

4. Operator Registry: data on robot operators and services, such as IDs and associated skills (operator assistance can be hired, if required).

5. Service Queue: a service repository that also includes metadata about the service itself, such as IDs, conditions when and where invoked, initiation conditions, etc. Services are always registered in the global repository and get distributed from there to the local repositories.

Combining all of this data with current status information about the system that is maintained by the state and resource managers, the entire system represents a quite complex kind of knowledge representation layer itself (which is of course tightly interconnected with message passing and information extraction). It combines static representations of maps and

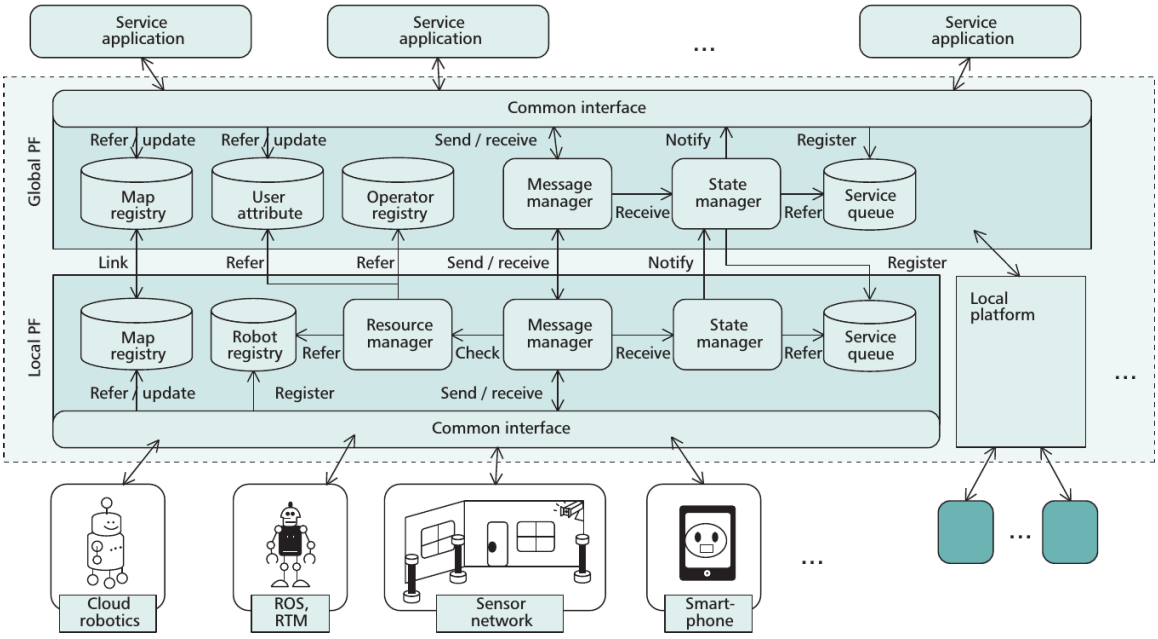


Figure 2.10.: General overview on the ubiquitous networked robot platform, as in [108]

static robot & service & users descriptions with dynamic information about their location, the location of obstructions and target objects. A reference platform implementation is available at [263].

In [205], this platform was combined with the KnowRob system, whereby the task was to answer common customer questions in a "supermarket" scenario. It was used to answer basic questions about food or stationery categories, ingredients, and locations. Robots, when asked for the location, could also point into the direction of the product. The map building approaches of both systems were therefore not combined; instead, this platform served as an "underlay" to KnowRob, abstracting hardware and the execution context, and also forwarding queries to human operators as a fall-back knowledge source.

Thus, questions were answered on the basis of both the KnowRob ontology system and an additional semantic map (see Sec. 2.3.4.2.11), as depicted in Fig. 2.11, which contains instances of products and their locations within the environment (actually a highly precise 3D world model). As mentioned by the authors, this representation can also be coupled with sensors of the environment, like RFID tag readers that could also update the map. This representation is also accessed by the human operator who can update the environment model if the shop layout or some product positions have changed (the KnowRob ontology can also be adopted by a human operator).

2.3.4.2.11. KnowRob & RoboEarth & ...: As depicted in Fig. 2.12, KnowRob [204], Rapyuta [154], and RoboEarth [217] are different parts of a larger system architecture.

KnowRob as described in [204] (and as the name suggests) is a knowledge-processing framework for (mobile) robots. KnowRob itself is an example for a composition of three different (or at least two and a half) representations: a logic-based, an ontology-based, and a semantic map.

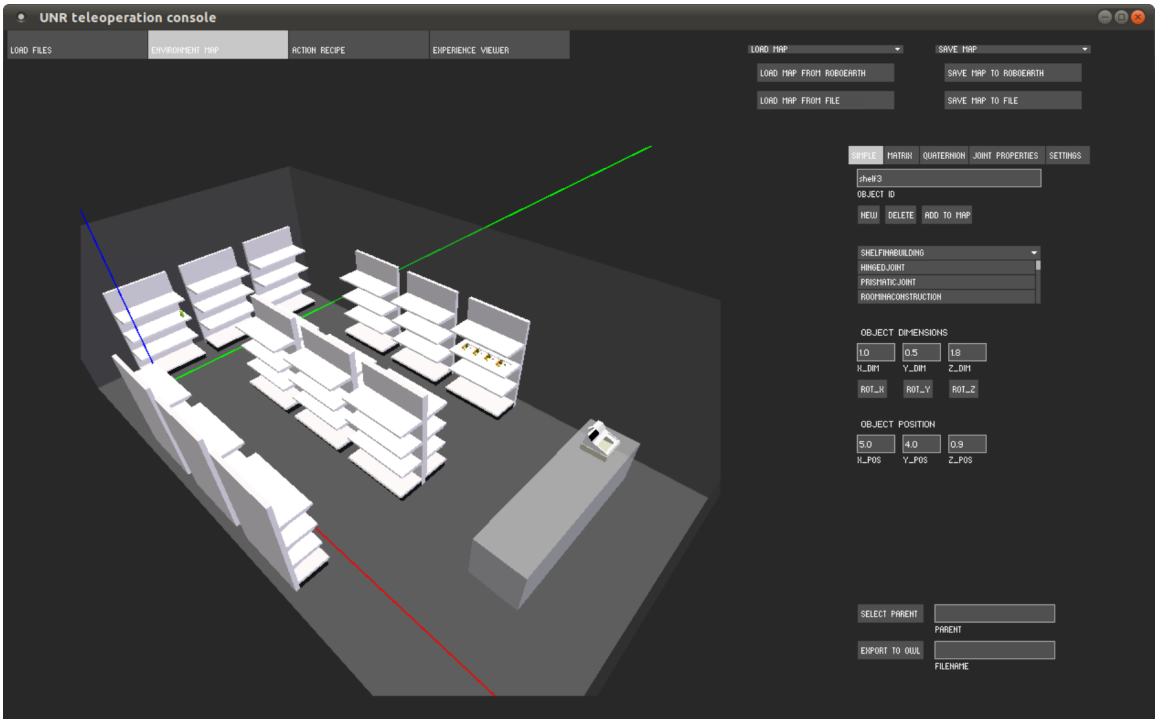


Figure 2.11.: 3D semantic map applied for the supermarket environment, as at [205]

The generation of the semantic map for a kitchen environment was presented in [186], which is a combination of a feature extraction with background knowledge about "kitchens". This so-called "functional mapping" is based on point cloud data, which was segmented in a first step to extract regions with certain geometrical characteristics. These regions can be further abstracted into surfaces. The background knowledge is used to identify/classify surfaces such as tables, cupboards, ovens, etc. that are typical for a kitchen (it simply does not make sense trying to identify objects like cars or tigers). For example, an oven can be distinguished from cupboards by the amount and size of handles and knobs. The resulting semantic map can afterwards be verified by testing it against a ground truth (e. g., positions, links, etc.).

This semantic map (see the example and figure in Sec. 2.3.4.2.10) is used for spatial reasoning to check relations such as $left\_of$, *above*, or *within*, which are required to interpret human orders (similar to Roy's Mental images). But next to the spatial domain, it is also required to check if an object is of a certain type or for what it can be used. For instance, to be able to interpret commands like "start baking" complementary knowledge about objects is required (such as its shape is concave so that it can be used as a vessel, etc.).

RoboEarth can be interpreted as an extension to KnowRob that was originally designed to run on single robots only. The RoboEarth [217] system was therefore introduced to overcome the hurdle for distributing knowledge by applying cloud-based methods. RoboEarth can be considered as a top-down approach, closely related to the questions of what data is required to support KnowRob. Hence, it is used to store data about objects (including CAD models, point clouds, and image data), maps as compressed archives (containing map images and information about coordinate systems), and robot task descriptions (i. e., action recipes in a high-level language). These so-called "action recipes" are further organized in a

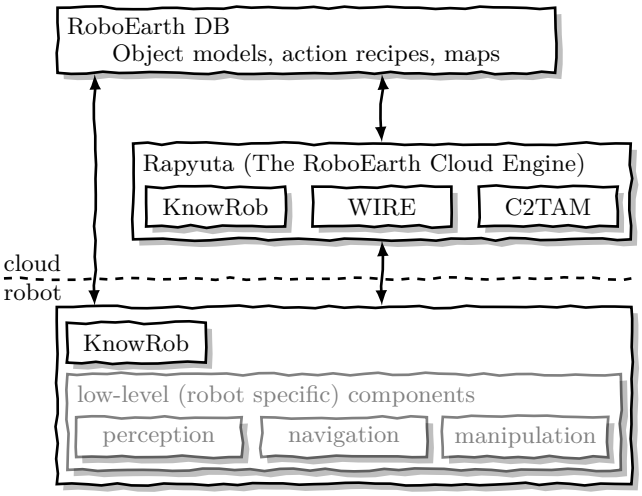


Figure 2.12.: General systems overview as at `www.roboearth.org/software-components`

hierarchic structure and represented by semantic representations of skills that define specific robot functionalities that have to be executed. The database services further provide basic learning and reasoning capabilities, which can be applied by robots to map the descriptions of their skills to the descriptions of actions recipes. Two different database systems are therefore applied. All "simple" data is stored in tables in the distributed database system HBase, which is based on the already mentioned Hadoop (see Sec. 2.3.4.2.5), while semantic data on objects, robots, action recipes, and the environment is stored in the centralized Sesame [261] (an open-source framework that provides extended methods for querying, reasoning, and analyzing RDF data, see also Sec. 2.3.3.5.2).

As further described in [217], the RoboEarth's architecture possesses some additional generic components. One of these components is responsible for appropriate environment modeling, which is segregated into a local and a global part. The local part, executed on the robot, *"merges and associates (possibly contradicting) sensor data from the robot's 3-D perception (e. g., the locations of labeled objects) with tracking data"*. The global part is maintained by the database, *"it updates the current local world model by factoring in suitable prior information... and provides simple reasoning capabilities"*. It is mentioned that this modeling approach is based on the formal world model presented in [181], but I was not able to detect it within the official repositories. Furthermore, the official API descriptions only provide information concerning the database access (cf. [257, 256]).

Another aspect that has to be mentioned in this context is the idea of "web enabled" robots [206], coming from the same research group. It describes the possibilities and advantages of using the Internet (i. e., web pages of online shops, Goggle images and SketchUp, wikiHow, etc.) itself as a knowledge repository for robots. Unfortunately, all has to be translated into a format applicable by robots, which is not brought to a solution yet.

The Rapyuta cloud engine ... while KnowRob is mainly dealing with reasoning in robotic applications and RoboEarth is used to share data, information, and knowledge (i. e., ontologies and maps), Rapyuta is applied for cloud computing. In other words, it is a framework for uploading computational heavy task into a computing facility (cf. [154]). Standard Linux containers are therefore applied to run ROS nodes and services in a virtualized infrastructure.

2.3.4.2.12. MavHome: It stands for "Managing An Intelligent Versatile Home" and is one of the long lasting research projects in the AmI area. Its main objectives (as stated in [53]) lie in the development of an environment that acts as one intelligent and autonomous agent that maximizes the comfort of its inhabitants. This shall be accomplished by perceiving the state of the environment through any kind of available sensor system and by acting upon it through any kind of available device. As previously mentioned, it is one agent, but this agent can be decomposed into sub-agents and so on, whereby every sub<sub>* -agent is responsible for a certain sub<sub>* -task.

An overview of all components of the MavHome architecture is given in Fig. 2.13, and it comprises all elements that are associated with a typical MavHome agent. In contrast to many of the aforementioned attempts, MavHome possesses a very sophisticated internal structure. According to (cf. [226]) each (sub-) agent is composed of a decision, information, communication, and physical layer. The decision layer is assigned to the application elements (see Fig. 2.13), information to all kinds of services (i. e., prediction, data mining, aggregation, and databases), while communication is associated with the applied middleware and different kinds of interfaces, the physical layer is associated with physical devices (sensors/actuators), but also with other lower hierarchy agents.

The process of perception works as follows: available sensor data is transmitted via the communication layer to the information layer, to be stored, processed, and transformed into higher level information that is used in the decision layer. Based on this information and on the learned experiences, derived knowledge, and observations an action is selected. This action is checked in the lower information level, and if it passes, it is communicated to a physical layer and executed.

And it seems to work perfect for some tasks in AmI environments, but these tasks are rather simple compared to the system described in the next paragraph. Perceived elements are actually light, humidity, temperature, smoke, gas, motion, switches, and the tasks to be accomplished in [56, 227, 103, 103] are turning off and on the fan, the lights, the TV, the coffeemaker, the sprinkler, etc. It is working smoothly, data gets interpreted, transmitted, and understood because the system is perfectly configured for the environment. Required data is stored within a central repository, which can afterwards be applied for data mining [103], learning or anomaly detection [102], which is made possible by fixed transformation of data into a logical representation (i. e., fixed states like $CoffeeMakerOn$, $BathLightOf$, etc.). A graph-based model is mentioned in [56] to be used for the spatial representation, which was the reason to list it within the knowledge representation layer in Fig. 2.15. But again, it comes only with limited capabilities (used for simple control of the home automation systems), which makes it hardly applicable for robotic applications, in contrast to the following approach.

2.3.4.2.13. Physically Embedded Intelligent System (PEIS): The PEIS Project [187] is based on the notion that robots and their surroundings have to be viewed as a tightly coupled (symbiotic) ecology with mutual dependency. All entities are thus abstracted by the uniform notion of a "Physically Embedded Intelligent System" — a computerized system capable of communicating and interacting with the environment through sensors and/or actuators. In addition to a complex robotic system, surveillance cameras, fridges, or even RFID labeled objects (e. g., spoons, milk cartons, etc.) are considered to be PEIS objects. And different PEIS objects "can" congregate or share their services and functionalities to fulfill tasks that cannot be carried out by individual systems (at least this is the idea).

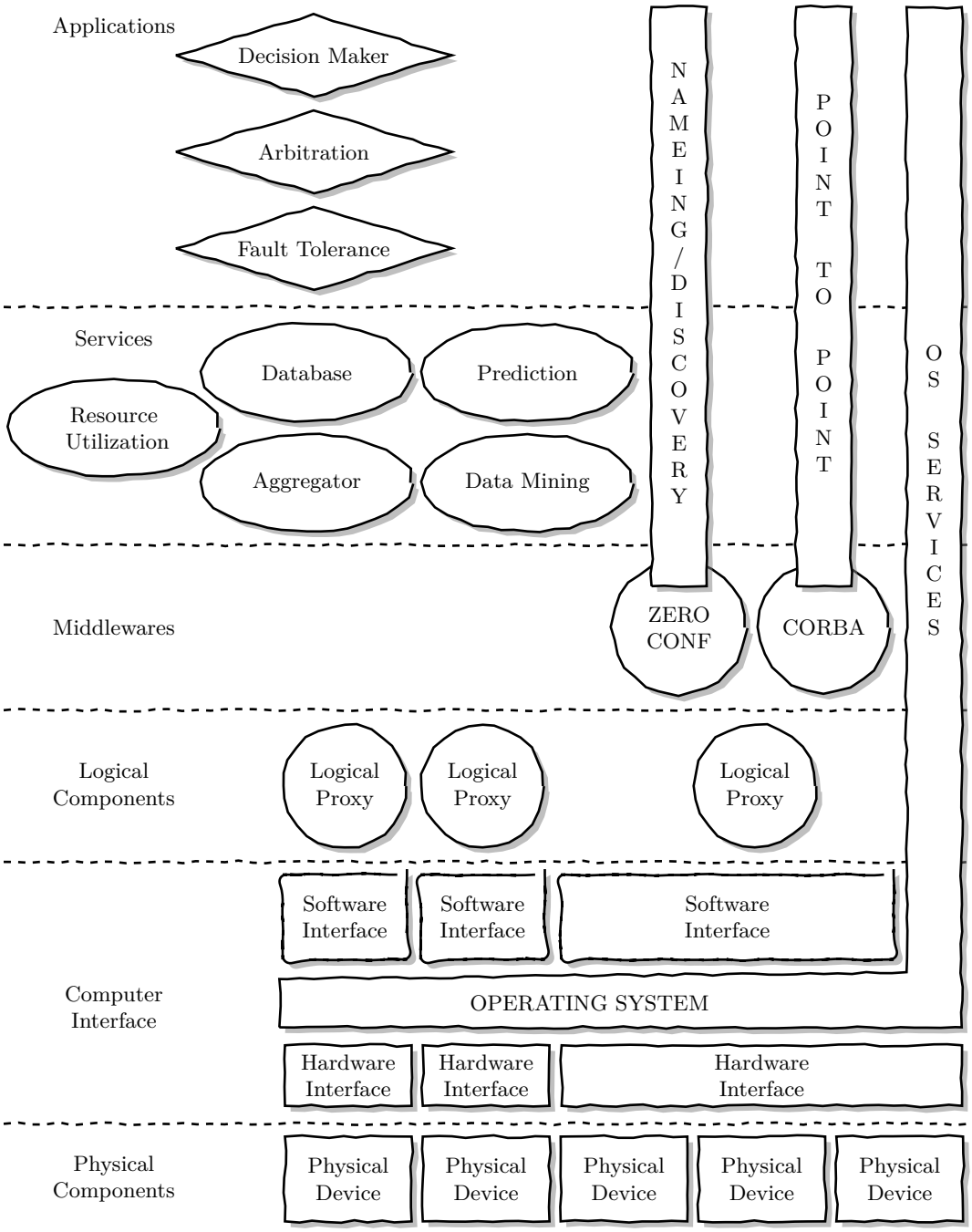


Figure 2.13.: General MavHome architecture, as at [102]

The information about capabilities and offered services of PEIS entities (e. g., object recognition, localization; see [44] for a complete list) as well as the configuration of the whole ecology is encoded in PTLplan [146]. PTLplan is a Probabilistic Temporal Logic planner [109], which, in contrast to the situation calculus (see Sec. 2.3.4.1.11), applies probabilities to actions and situations by searching for a defined goal situation. Based on the available facts, the planner afterwards determines if a certain goal (task) can be accomplished and what the required functionalities/services are. In order to reach a certain goal,

the environment is initially queried for a complete description by using the PEIS communication middleware that combines a distributed tuple-space event mechanism (cf. [43]). Due to this tuple-space, the planner or any other system (robots and spoons included) are always aware of all included entities and the entire configuration of the environment.

While GOLOG, for example, operates only on the knowledge and intelligence level, PEIS is not cutoff from underlying layers and can therefore be applied onto "lower" level problems as well. For instance, perceptional anchoring is the process of connecting identifiers which are used to denote an object (e. g., $Mug22$) with sensory measurements (e. g., red area in a camera image, round shape within a laser scan, etc.). As presented in [134], this method currently works only for PEIS objects. A robot first queries the tuple-space for all "physical representations" tuples (e. g., color, shape, size, etc.) of all PEIS object within the ecology. This information is then compared with the available sensor readings.

2.3.4.2.14. Knowledge Ecosystem: A formal approach to define distributed knowledge bases was published in [150]. It is based on ETHOS (Expert Tribe in a Hybrid Network Operating System) [169], which is comparable to ROS. It introduces the concept of "experts", which are concurrent agents responsible for a specific deliberative or reactive behavior. As depicted in Fig. 2.14 there are three different types of experts, (S) are handling symbolic knowledge, (D) are meant to deal with analogical and iconic representations (i. e., maps, sensory inputs, etc.), while reactive behavior is handled by (R). These experts can be organized and clustered in so-called villages with the help of the ETHOS framework. Hence, it supports loosely coupled communication via publish/subscribe, synchronous access via services, and also shared memories (cf. KB, D1, D2 in Fig. 2.14).

Although ETHOS was previously developed in the context of RoboCup, it has been "conceptually" applied to smart environments in [150]. The entire configuration of the environment including the state of the entities is represented by using description logics, similar to CAMUS (cf. Sec. 2.3.3.5.2) or KnowRob (cf. Sec. 2.3.4.2.11). A new type of agent was therefore required and defined to translate sensor data into symbolic representation.

2.3.4.2.15. A Concept of extended Object-Oriented World Modeling: It is currently a concept only for a complex environment model for autonomous systems [35, 130], with some basic implementation examples. It separates between sensor data, a world model, and knowledge. Sensor data is analyzed with the help of already existing knowledge, and the resulting information is passed to the world model. Knowledge is defined in terms of specific methods and algorithms for analyzing sensor data (according to the applied classification, services at the information layer). The world model consists of objects (labeled with attributes), representing entities of interest. These objects are interconnected in a scene via relations, similar to the distributed scene-graph approach that was described in Sec. 2.3.4.2.4.

The authors do not mention how data, knowledge, and the world model are stored or how scenes, including all details about the environment, are represented. But it discusses the fact that there is a need for a symbolic abstraction/level to describe situations. Probably there is some kind of symbolic encoding which can be directly applied onto the world model.

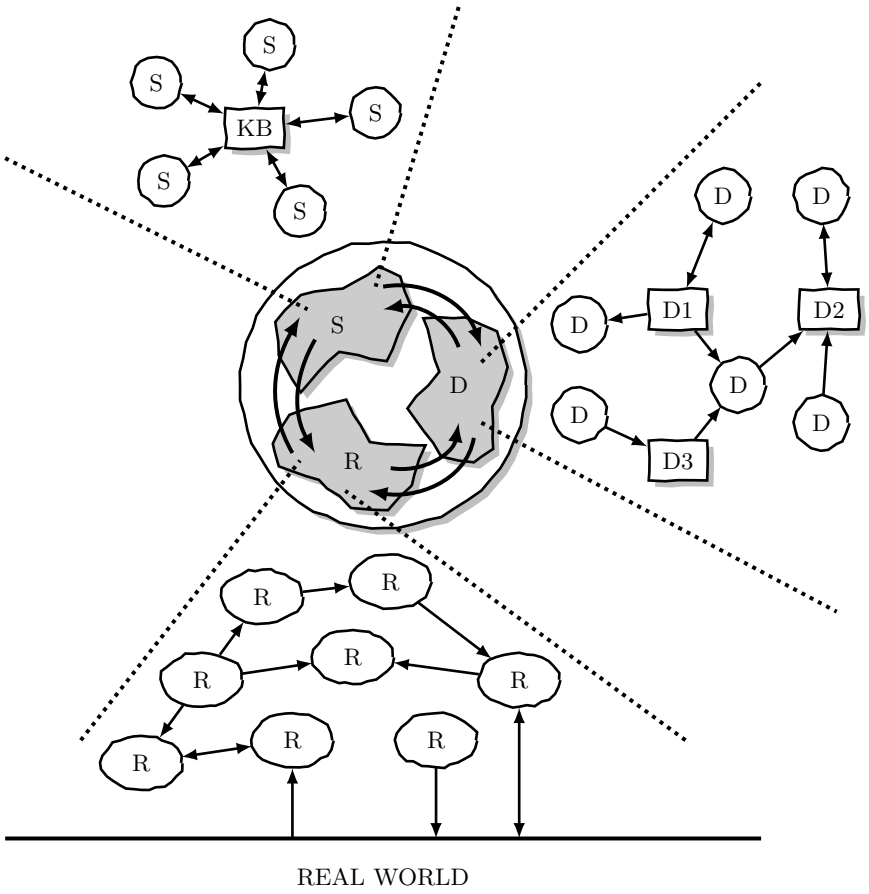


Figure 2.14.: General ETHOS concept of distributed experts, as at [169]

2.3.4.2.16. Summary: Compared to the lower layer reviews, this overview on the related work seems to be more scattered and presents much more divergent approaches. While lower layer technologies up to the information layer already provide a lot of standards with applications or libraries that can be used to translate between source layer descriptions or data layer formats (see, for example, [262]), etc., the approaches on the knowledge layer present mostly single solutions for a few predefined problems. Of course, there still might exist further approaches (without the knowledge of the author) that would conceptually belong to this layer but have not been mentioned in this overview. Nevertheless, the literature review for this layer provides a couple of additional and interesting facts. Firstly, we are still in the collection phase. There is only a small core of relatively young publications, which is furthermore perpetually cited in the related work part of publications that deal with cloud robotics, UbiBots, robots as a service, or simple combinations AmI/SmEs with robots. Although these terms and the developed concepts seem to be promising and some of the presented architectures are quite complex, the presented applications and the solved problems are still at a very early stage. Probably, it is not possible to find a solution that can be explained on six or eight pages, whereby publishing adaptations of previous adaptations seems to be more valuable. Probably it is also too expensive in terms of new and relevant technologies that have to be developed. It might also be too expensive to provide a demonstrator. Or it is likely to be too difficult to change our minds and as well as the commonly practiced approaches.

Thus, if the tasks that have to be accomplished grow in their complexity, we tend to combine different approaches (cf. Sec. 2.3.4.2.10) and also the different world models, which in turn lead to very complex and tightly integrated system architectures. Some approaches can be summarized as supporting technologies, such as LDM, PREDiMAP, Object-Oriented World Model, Distributed Robotic Scene-Graphs, or even DAvinCi, which provide only a method to distribute knowledge. Others such as Cloud Robots as a Service, the Robot Cloud Center, So/Em/Mo-bots, Knowledge Ecosystems, extended Object-Oriented World Modeling, etc., actually provide concepts for large fields of applications, but with vague architectural descriptions. It thus contrasts the complex architectures described for the DustBots, NRS, MavHome, KnowRob&RoboEarth&..., and PEIS that are capable of solving a predefined set of combinable task, but with less complexity if comparing them to the intended task that should be solved by the systems mentioned before. Whereby, KnowRob and especially PEIS seem to provide the more general approaches, and due to their declarative nature they are capable of solving repeatedly new tasks without the need to change or adapt their architectural designs.

The different types of knowledge representation formats that were listed in Sec. 2.3.4.1 can actually be interpreted as inputs to algorithms associated to intelligence layer (i. e., reasoning, planning, situation understanding/awareness, etc.). It thus might be surprising, at first, that there is no further overview given on the intelligence and the representation layer. However, as also visible in Fig. 2.15, there is currently no CPS or cloud-based approach (to the best of my knowledge) capable of solving different types of problems or dealing with the dynamic representation of content. Instead, those systems that slightly touch the intelligence level (e. g., CAMUS, DustBots, PEIS, RoboEarth, or MavHome) do this on the basis of their internal knowledge representation scheme, and therefore mostly reason on actions and services; only the DustBot externalized path planning.

The next section is intended to close the literature review by discussing the approaches, technologies, and their main problem more generally.

2.4. Synopsis – Where are we now?

Although the amount of approaches that has been presented, covering the source and data layer, is little, compared to the approaches operating on higher layers, in fact the former outnumber the latter (cf. [193, 231]). A lot of research has been done in these areas over the past decades, which resulted in many elaborated systems. Fig. 2.15 shows that these are versatile approaches that cover large problem sets in these areas. This becomes particularly obvious by looking onto the standards that have been defined by the IEEE, OGC, ROS, in addition to a multitude of industry standards. Due to these standardizations, source descriptions and data can be easily accessed, transmitted, and transformed (into different formats). Transforming means that a system description defined by one standard can also be translated into another and vice versa (e. g., URDF to COLLADA to CAD, or SensorML to TEDS), similarly to data (e. g., transforming a ROS message in to a MOSAIC message). Thus, interoperability at the source and data level is actually not a real problem anymore. It allows the application and the combination of these technologies widely and for a multitude of applications.

However, turning the attention upwards in the applied hierarchy reveals that this is absolutely not the case for approaches operating at the information and knowledge level. Technologies in these areas appear more scattered and offer limited functionalities. Of course

JINI, OSGi, or ROS offer frameworks which enable the implementation of distributed services, remote procedures/functions that have standardized formats to describe their interfaces; which offer context information along with rendering and code mobility throughout the network (remote evaluation). Taking a deeper look onto ROS, the offered functionality is overwhelming. The current ROS repositories[21] contain more than 2100 packages for various purposes. But how is this functionality used? Generally and very traditionally, human developers seek for the needed ones (i. e., nodes, libraries, services) to solve a particular problem and assemble them as required. It means that we still have to define what functions have to be called, and we not only have to choose the right input parameters, but also have to be aware of the context along with applying those functions in the correct order.

Reminding the intelligent construction scenario, used within the virtual sensor concept on page 26, a user acquires information about unsafe areas in proximity to a crane. It requires one to apply multiple filter, fusion, and transformation functions onto heterogeneous sensor data, the pose of the sensors and the current configuration of the crane, which has to be known and included into the calculation and which, afterwards, has to be presented appropriately to the user. Even a human would struggle to identify all required functions (if accessible) in correct order and with the correct input parameters to generate this information. Thus, simply passing published (sensor/actuator) data to remote functions is not a satisfactory solution. Yet, the virtual sensor concept starts pointing to a more suitable direction, which was carved out by TinyDB even more. Although it seems quite natural to interpret a distributed sensor network as some kind of database (afterwards) and, therefore, to apply an SQL-like query language (that was briefly introduced in Sec. 2.3.3.2 on page 26), it is actually the application of a declarative/relational programming paradigm to a completely new field. It makes requesting for complex information possible, whereby the language is only used to define what kind of information. An "abstract machine" is responsible to deduce this information on the basis of available data, constraints and rules with different solvers (or search strategies) applied in the background. This fact was also emphasized for CAMUS in Sec. 2.3.3.5.2. In the case of TinyDB every sensor node runs an abstract machine to resolve queries and this works also in a distributed manner by aggregating and filtering information (with a fixed and limited set of functionalities) along semantic routing trees.

Imagine we could query the components in the intelligent construction scenario or any other smart environment similarly:

Listing 2.9: Exemplary SQL query for dangerous regions around a crane

```
1  SELECT places FROM crane_region on ground WHERE hazardous_potential>0.01 ...
```

Approaches at the knowledge level that enable querying or reasoning (e. g., on context, actions, services, etc.) can also be applied to standard frameworks and solvers that use central knowledge bases, such as CAMUS → Jena; RoboEarth → Sesame; ContextToolkit, LDM, PREDiMAP, MavHome → MySQL; Ubiquitous Network Robot Platform → PostgreSQL and PostGIS; except PEIS that utilizes PTLplan on a shared memory. LDM and PREDiMAP thereby offer only complex containers for any kind of data that might be required in an automotive application (relations between data are represented indirectly by the database structure). The application itself is then responsible to identify and extract relevant information.

[21] Package overview for ROS indigo: `http://www.ros.org/browse/list.php`

Approaches that apply different knowledge representation formats from logical or formal up to more complex ones (e. g., semantic maps, occupancy grid maps, complex 3D models, etc.) commonly make use of distributed databases or distributed file systems (i. e., DAvinCi, RoboEarth, Robotic Cloud Center, Cloud Robots as a Service). Nonetheless, as already mentioned, these are fixed representations of a certain type according to the supported task. There is currently no system, concept, or approach that allows to switch or translate between different knowledge representation formats. Therefore, translating between different 2D maps is probably as easy as translating data or source descriptions on lower layers, even if it necessitates neglecting the mounting height of the sensor systems that was used to generate it. This is also true for other knowledge representations that are quite similar to each other. However, a concept of a global or central format/representation from which dynamically different kinds of knowledge representations formats or information can be extracted does not seem to exist. The only concept that tries to put all data into a common spatial and temporal context is the Distributed Robotic Scene-Graph, which was presented in Sec. 2.3.4.2.4. But similar to other approaches, it does not possess any kind of interface to request information or knowledge in a specific format; rather, the application has to deal with the available data and information. Imagine that it would be possible to extend the previous select statement from Lis. 2.9, and add something like a request format:

Listing 2.10: Extension to the query in Lis. 2.9 by request formats

```
1 ...
2 AS OccupancyGridMap with z-pos, resolution, ...
3 AS TopologicalMap with ...
4 AS List ...
```

Whatever it is, return this information in specific format so that a certain algorithm (probably from the intelligence level) or application can deal with it. The most obvious benefit would be that such a practice does not affect any existing system, and there is no need to change or adapt the code of a system in order to comply with new data, information, or knowledge representation formats. It is furthermore something that actually belongs to the top layer “Representation”, which is responsible for the appropriate presentation of content. This capability is commonly associated with UbiComp and AmI in relation with HCIs and different types of displays (which is also underpinned by the overview in Fig. 2.15). But as mentioned earlier, this layer can be envisioned in a much broader context, enclosing each of the layers below in the sense that any kind of data, information, knowledge, solutions to problems (e. g., trajectories, action sequences, etc.), which belong to the intelligence layer, could be presented in various different formats and as indicated in Lis. 2.10.

Summarizing the related work at the knowledge layer, most of the approaches discussed so far represent highly specialized systems — specialized in terms of their application and supported knowledge representations. Thus, the more general an approach gets the more complicated its architecture becomes. This fact becomes obvious by the complexity of the depicted architectures of MavHome, RoboEarth, the Robot Cloud Center, the Ubiquitous Network Robot Platform, or ETHOS. It has to be mentioned that these architectural maps only present very high-level sketches. Every newly introduced feature requires an adaptation of these systems, the integration of new elements and storage of data, information, or knowledge. Thus, the complexities grow and systems become even harder to maintain and adapt.

To conclude this section and as well as the chapter, there is one remarkable similarity to the OSI model (introduced in Sec. 2.2.4) and the state-of-the-art depicted in Fig. 2.15 that has not been mentioned yet. As defined by the OSI model, communication is only possible between entities that operate on the same abstract layer, and the same seems to be true for the approaches that have been presented so far. As in the earlier stages of the Internet, different approaches coexist next to each other, requiring additional interfaces/transformations to be able to access elements of a foreign architecture. Furthermore, it cannot be predicted anymore at design-time for an application, what type of metadata, data, information, knowledge, and in what representation format might be required to accomplish a certain task. A certain measurement value or a sensor description might be required also at the knowledge representation or intelligence layer (for decision making). Thus, accessing elements from lower or higher layers is mandatory, though it needs to implement an additional set of interfaces, which is actually contradictory to an original layered approach.

Literally put, everything might be required in every imaginable format. Thus, the goal of this thesis is to develop a concept and a prototypical implementation that actually enables access to any kind of metadata, data, information, etc., from a distributed smart environment in any desired format. A declarative approach has therefore been designed and will described within the next chapter, which actually allows it to access any kind of information with simple SQL-like statements as they were exemplified in the two listings before.

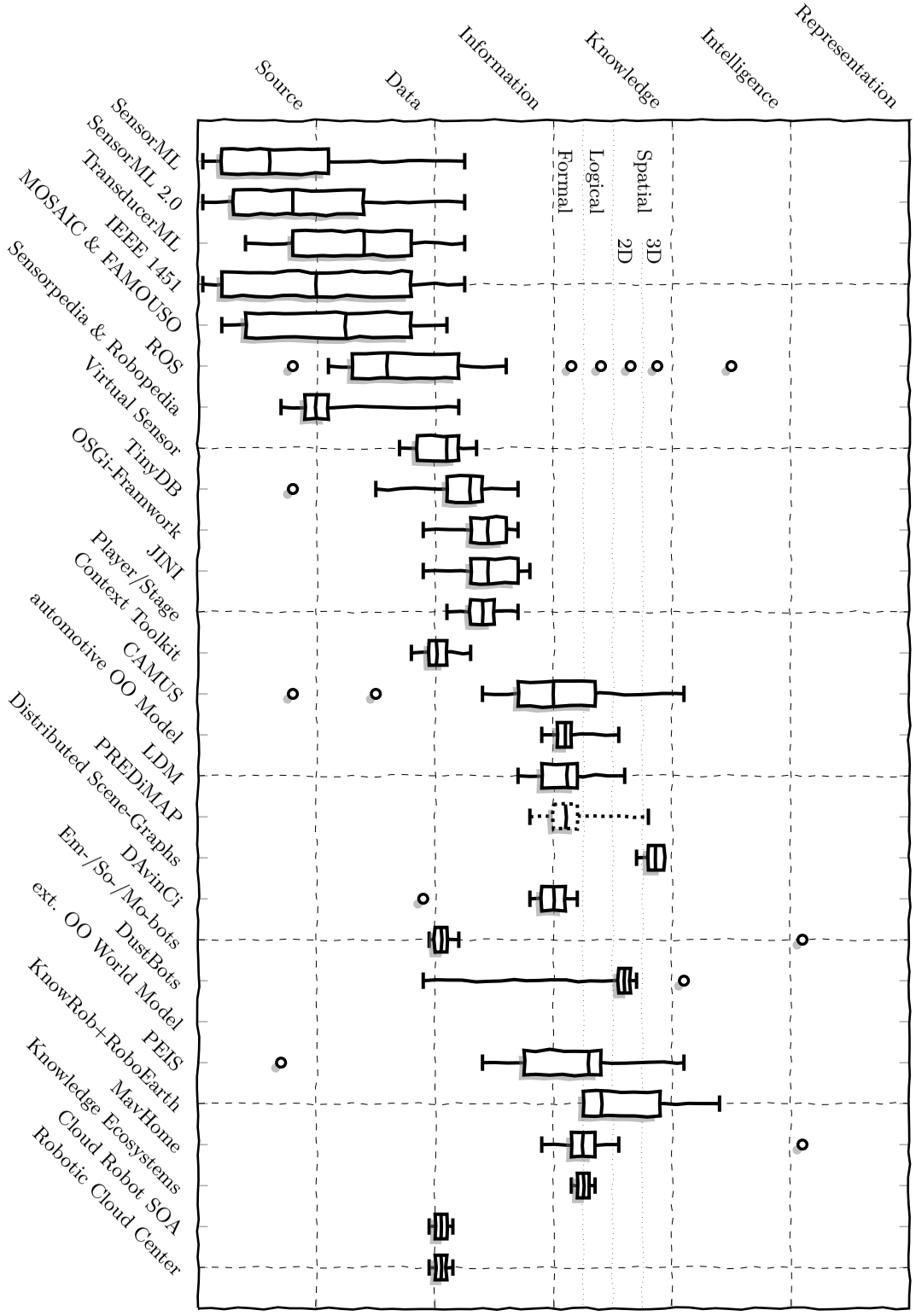


Figure 2.15.: Layers covered by the related work, see the exact definition of every layer in Sec. 2.2.3 on page 17

3. Concept

> "... if you look on the surface you see complexity, it looks very non-mathematical...
>
> ... think not of what you see, but what it took to produce what you see..."
>
> —Benoît Mandelbrot on fractals and nature

As it has already been introduced with the frog problem in Sec. 1.4, the goal of this thesis is not to add an additional boxplot to the current state-of-the-art; nor is it to develop a system that fully overlaps with all of the layers presented before (I doubt that this is even possible). Instead, the main objective of this thesis is to offer a concept that allows us to integrate different technologies from different layers and to access and combine them freely. This holistic access mainly covers the four problems carved out in Sec. 1.4. According to the previous chapter, elements of interest have been differentiated into source level descriptions, data, services, information, world models, etc., whether they are stored somewhere or accessible in "real-time" (problem 2 & 3). Furthermore, it requires the ability to present or to generate the desired piece of "information" in the required format (problem 4). And the major problem (1) is how this can be achieved without an extraordinary intelligent application.

Fig. 3.1 is used a starting point for the following considerations. It sketches the current situation as well as its main problem. There is currently no general concepts or interface that allows the query of such CPS, SmEs, IoT, AmI, etc.; instead we are overpowered by the myriad of available interfaces, data (and their formats), and functionality. And, therefore, we still develop applications that deal only with a fraction of it.

As indicated by listings within the previous section and by the "slightly more" detailed descriptions of systems that adopted the database semantic for their purpose, I took over some of these ideas and applied them to another domain. Why should it be impossible to treat such composed smart information spaces in the same way as a "distributed" database? It would require at least three things: some kind of basic order or organization, some format to define requests, and some kind of abstract machine that is responsible for processing these requests and delivering demanded information.

The main question mark is (cf. Fig. 3.1) how a distributed environment of CPS can be queried as if it would be an ordinary database? Very early conceptual considerations have been discussed in [1] and [3], the solution presented and updated here consists of three intermediate steps. These steps are described in a bottom-up fashion within the following sections: therefore, the following enumeration is intended to provide a short overview on every step. For each step there exists a separate implementation that can also be applied independently. Since every step is a conceptual part with its own implementation, such that every solution can also be applied on its own, it is recommended to read the following sections in conjunction with their implementation in Chap. 4. The section numbering of this chapter, therefore, overlaps with the section order within the following chapter.

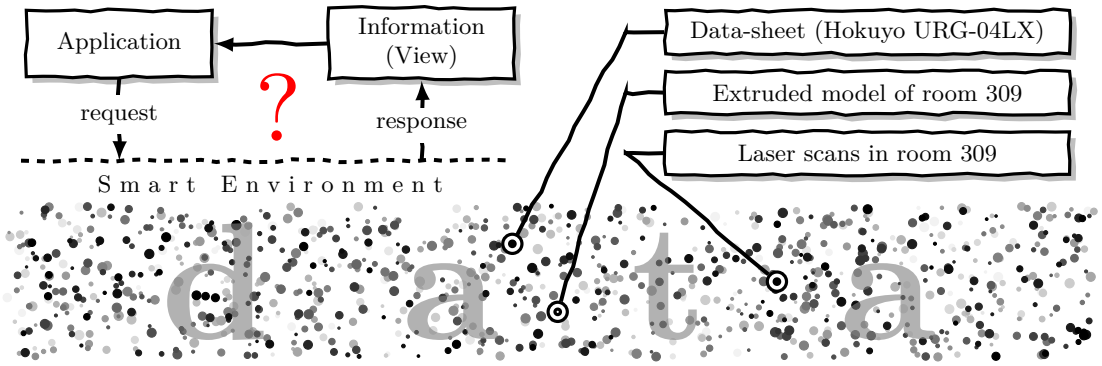


Figure 3.1.: The dots are used to mark any kind of appearance of source level descriptions, real-time sensor or actuator data, context information, maps, historical measurements, etc. within a smart environment that is continuously changing. The dashed line represents an interface to this distributed environment, which might be accessed by an application to request for specific information in a certain representation.

1. **Organization:** This initial section deals with the problem of organizing data and information in a distributed environment of CPS in a way that all of it can be searched and accessed. To track this problem down, a cloud-based approach is applied. It is used in two ways: as a holistic store for any kind of data and as a virtual overlay structure that represents the entire environment. In the simplest sense, this structure can be described as distributed scene-graph that reflects the global state of the environment. It therefore can be related to the robotic scene-graph approach that was presented in Sec. 2.3.4.2.4 but, in contrast, it provides more complex methods to handle and access data along with information as well as methods for dealing with consistency.

2. **Idealization:** If the previous step was about the development of a global world model, this part is used to describe how local world models can be extracted from it. These local models reflect the idea of Roy's mental imagery that was described in Sec. 2.3.4.1.15. A precise 3D rigid-body simulation is applied to mimic the real environment for a certain region of interest. Entities within the local model (e. g., sensors, robots, humans, etc.) can be connected to their real world counterparts in order to replicate their behavior online (the information of how this can be achieved is taken from cloud). Additionally, all objects within can be associated with supplementary semantic information. This "ideal" replica of the environment is thought to represent the most general model and serves as a common knowledge base for the next part.

3. **Extraction & Abstraction:** The previously reconstructed local model of the environment (updated with real-time data) is used in this part for any kind of analysis and thus information extraction. It is furthermore applied as a source from which different kinds of world models can be abstracted, not only the spatial ones listed in Sec. 2.3.4.1.

   As discussed in Sec. 3.3.1.1, from the point of data analysis and interpretation, there is no real difference between a database and a simulation. The SQL syntax was therefore adopted and extended to be applicable on simulations at runtime. All the

complexity, that is required to generate information and models, is hidden behind an abstract machine (language interpreter). This section hence, mainly deals with the development of a general language concepts and, furthermore, with the idea of how SQL can be applied for reasoning tasks in order to solve similar problems as Prolog, but with the running simulation as its knowledge base. Another point why reasoning has become such an important feature of this new language is that it allows deducing sequences of transformation, fusion, and filter functions. It is similar to the identification of action sequences for robots, which is actually well-known in robotics since the times of the situation calculus (see Sec. 2.3.4.1.11); however, reasoning on the application of ROS functions and services is currently unknown.

That these three steps are afterwards applied in reverse order is demonstrated in Chap. 5. At the end, only one select statement is required, whereby the local world model is generated automatically and in accordance with that statement.

3.1. Organization & Access

If you imagine that every mobile or immobile robot, every smart sensor, or a simple application possess its own private data storage with its own structure and own formats (e. g., binary, XML, JSON, etc.), the discovery as well as searching and accessing this data becomes a tough challenge, especially when new entities are introduced or others leave. To deal with this problem, a couple of systems either upload their data to a centralized or distributed repository, or have switched to a distributed file systems, such as the already mentioned MavHome, RoboEarth, or DAvinCi, etc. Nevertheless, only worthwhile and predefined data is thereby uploaded, which still describes a centralized solution, from an application point of view.

The goal at this stage should not be to enforce the upload of all (previously defined) data to an external cloud-based infrastructure, but instead to create a cloud that integrates all the entities of a smart environment. Such a kind of holistic data store was proposed in [12]. Wherein, every entity maintains its own local database and stores its own data locally for its own purpose. All of these local databases are organized in a bigger cluster, which allows the query of and access to data of external systems. Similar to other approaches, it is also based on a cloud-based infrastructure, but one that has not been introduced to robotics (cf. Sec. 4.1.1). The main problem thereby was that simply storing data does not offer any benefits. Instead, some kind of organization is required that allows to interpret and query data in a global context. This organization was published in [10] and pretty much reflects the layers that were applied in the previous chapter to categorize the related work.

3.1.1. Categorization

The categories depicted in Fig. 3.2 can be interpreted as a kind of bootstrap for the organization that is described within the next sub-section. This categorization was introduced in [10] also as a point for critics to related approaches, which pretend that simply sharing data is enough to build up any kind of CPS application. Instead, it is merely the basis (or from another perspective, the tip of the iceberg).

At this stage the term “data” is used because it relates to the sheer amount of data (literally everything) that can be stored and accessed without any relation. Every kind of data can be assumed to belong into one of these four general categories. Metadata (literally

data about data) is used for any kind of source level descriptions (i. e., sensor data-sheets, robot description formats, data definition formats, etc.). Abstract data is used to classify any kind of map, identified object, etc., thus everything that cannot be directly measured but has to be identified or recognized, in contrast to raw and virtual data. Raw data refers to any kind of sensor measurement, actuator command or status information, as well as any kind of position information. Sensor fusion results or data transformations that are typically associated with functions at the information level (applied onto lower level data) can be associated to virtual data, such as the average or maximum temperature of a room. It is, for the most part, the application of mathematical or physical laws, in contrast to abstract data (such as recognized furniture) that, according to the state-of-the-art classification, also belongs to the information layer. The classes of raw and virtual data, furthermore, appear in two flavors within a distributed environment, either as real-time data that has to be accessed and analyzed online or as historical data that has been stored. That, is to say, historical data should be directly accessible from the cloud, through which only links can be provided to real-time data.

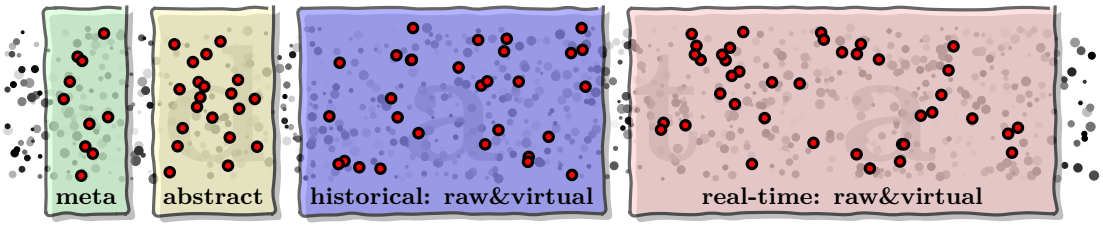


Figure 3.2.: General categorization of data into four integral parts

3.1.2. Organization

Providing a global structure might look easier than providing a global data store (see the implementation in Sec. 4.1), but it requires every entity to store its data and information according to a common pattern. The previously defined categories are now applied as simple containers (locally) for storing all associated data. The simplified entity relationship diagram in Fig. 3.3 depicts the entire organization, which can be applied to any underlying database system — whether centralized or distributed. Metadata was split into two "tables" that are used to store general data-sheets for sensors (e. g., TEDS, MOSAIC, etc.) and robots (e. g., COLLADA, URDF, etc.) only. This does not contain any context information about the location of entity, its identifier, orientation, or how its data can be accessed. In the same way abstract data is split into one table for location information (of any type) and the other for data about identifiable objects. Raw and virtual data require multiple tables; in the ROS-world this would require one table per topic and per producing node.

None of these containers so far stores any kind of contextual information; context is brought to them by the central table complex. Although linear and distributed, this table's structure is hierarchical. Every complex entry possesses at least one unique identifier, a complex base, a relative pose (relative to the base, which also defines the origin of coordinates), and a type and an entity identifier that is pointing to either metadata, abstract data, or another complex entry. Thus, any kind of entity is thus represented by a complex

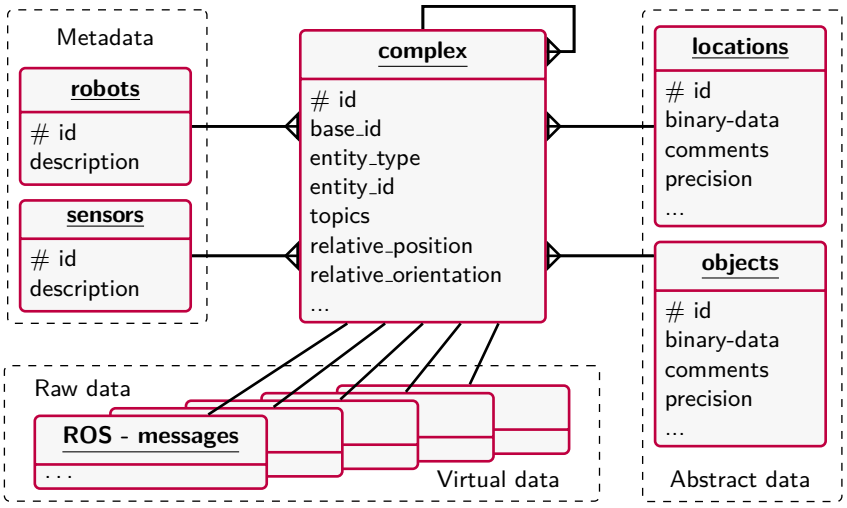


Figure 3.3.: Organization of data with the help of table complex as the global link, whereby raw and virtual data only mark accessible historical data and metadata is separated into two parts to distinguish between acting and sensing entities

entry, to which any kind of further additional information can be attached, such as pose uncertainties, semantic information, as well as published topics or offered services (if it is a sensor or a robot). In this case, these entries are used to provide links to real-time data and services, similar to Sensorpedia & Robopedia described in Sec. 2.3.2.4. An entire mobile robotic platform would thus be represented by multiple complex entries: one for the mobile platform, an additional manipulator as well as all on-board sensor systems which have to be represented as individual "complex" entries, and which form a sub-tree together, as part of the entire structure.

The principle that is sketched in Fig. 3.4 and, as already mentioned, pretty much reflects the notion of a distributed scene-graph as it was described in Sec. 2.3.4.2.4. But in contrast, the application of a distributed database system for maintaining the global world model seems to be more promising. It offers more and sophisticated methods for maintaining consistency (which indeed can be fine-tuned according to different requirements, see Sec. 4.1.1). It already possesses different methods and syntax to query data. And it probably allows using better and more effective methods for replication, if an entity with its partial knowledge about a certain area of the environment leaves the cloud. But as mentioned previously, this requires every participating entity to store its data accordingly and to update it continuously.

In contrast to other data classes, raw and virtual data has to be stored in multiple tables, one table per topic and producer. It is necessary to be able to cope with the large amount of produced data as well as its diversity, and to reduce replication efforts. In contrast to other tables, every stored message has to be labeled with a time-stamp, which allows it to keep temporal relations. Raw and virtual data require a direct association with their producers (not a spatial relation), in such a way that a video stream can be associated with a certain (complex) camera, laser scan with the producing scanner, or a set of position information with the mobile robotic platform. The implementation for this is briefly described in Sec. 4.1.2 and is actually more complicated than the implementation of this overlay structure, because being able to query any kind of stored sensor, position,

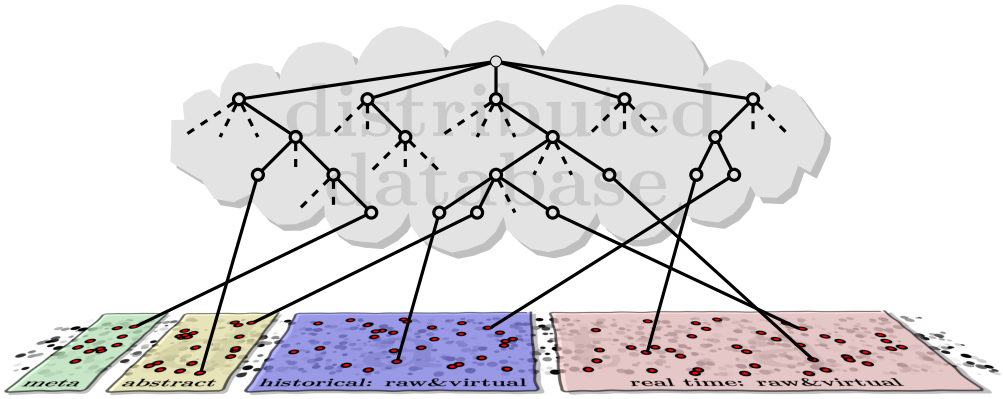


Figure 3.4.: Principle of hierarchical organization to maintain, store, and link any kind of data within a smart environment

robot status or configuration data, etc. (raw & virtual) requires a translation of the binary data stream into a format that can be queried. An adaptable method for this was presented in [12], which automatically generates a table structure that reflects the message format and decodes and encodes messages accordingly.

One benefit (in addition to the distribution) of applying cloud-based data stores, in this case, is that they do not require such a strict predefined structure as most relational databases do. Such systems are commonly based on other design principles than fixed tables for organizing data, such as key-value stores, document stores, etc. Thus, not every detail has to be known during design-time and not every entry has to have a specific format. Rather, the intended structure has to be known. As already mentioned, there might be multiple formats or description formats for sensors or robots. A cloud-based approach allows the storage of any of them, while letting the next layer or application decide which one of these should be used. As described within the next section, everything that is required to query this database is some kind of bootstrap, such as the identifier of a robot or its type, the number of a room, a position, a recognized marker and everything attached can be reconstructed by querying the hierarchy upwards or downwards.

3.1.3. Summary

This part introduced the notion of a holistic data store and a distributed scene-graph for organizational purposes, which is actually a composition of data and information stored on various different entities that form a smart environment. It thus also provides an infrastructure for deriving local world models, which is described within the next section. To the “complex” graph, therefore, might be referred to as a global world model, but it has several drawbacks and this requires a transformation of the available data into a local and more precise model. Although there are primitive spatial and temporal (as primary keys) relations included to structure the components of the smart environment, it does not allow switching perspectives (as it was imagined by Roy’s Mental Imagery), nor does it take into account how the current working step of a manipulator affects its geometry, or how mobile robotic platforms and transformations of the environment change the sensing areas or affect sensor readings; it does not support reasoning about action sequences, situation-awareness, or to make predictions. As described further, there is a difference between “historic” and

real-time data, for the second the overlay structure does only provide links. Continuous queries or callbacks known from the active database concepts (see Sec. 5.4) would be far too complex, expensive, and would not provide a sufficient level of detail. From this point of view, it makes perfectly sense to reconstruct local world models for a smaller area from which an information is required and to connect them to the smaller set of available real-time data, which is produced within them.

3.2. Idealization

> Idealization is the process by which scientific models assume facts about the phenomenon being modeled that are strictly false but make models easier to understand or solve. That is, it is determined whether the phenomenon approximates an "ideal case" then the model is applied to make a prediction based on that ideal case ... If an approximation is accurate, the model will have high predictive accuracy ...
>
> —Wikipedia

Most of my early research as a member of the research group for distributed and embedded system (EOS) dealt with supporting task in developing new communication middleware concepts (cf. [22, 23]) and with the abstraction of sensors and their fault analysis in distributed environment (cf. [2, 15, 16, 17]). The research area afterwards shifted to the application of more complex world models and simulations as tools to perform sensor measurement validation, plausibility checks, to provide some form of virtual redundancy or forecast environmental changes, where previously applied methods based on signal analysis reached their limits (cf. [5, 20, 8]). As I realized afterwards, all of these approaches presented only very specialized solutions to very specific (and well-known) environments (cf. [4]). Thus, the first genuine problem was the creation of such models, as it was described within the previous section, the second dealt with the extraction of information along with further abstracting of local world models.

This section is used to describe how local world models can be reconstructed and also what kind of knowledge representation formats are thereby used (cf. Sec. 2.3.4.1). Nevertheless, the aspect of previous research on sensor data validation is still valid and, therefore, also discussed within this section in conjunction with some of the main problems that arise while dealing with local world modeling.

3.2.1. The Local World Model

I propose the application of a 3D rigid-body simulation, which is similar to Roy's notion of a world model or to the model applied by KnowRob, with a scene-like representation of the environment, including all inhabitants (an early conceptual work on this was presented at [8] and further worked out in [3]). As it is presented within the next section, a rigid-body simulation can be interpreted as the most general representation of not only other 3D or 2D maps, but also formal and logical representations, and various kinds of information can be abstracted. But not only can information required by other machines be abstracted, as described in [13], but representations for human co-workers, by using AR techniques, could be applied to support a tighter interaction between humans and robots as well. It requires a visualization of the robot's intention and next working steps, in order to turn off a robot's

black box behavior. This approach was later picked up by former project partners[1]. The next section is therefore dealing with issues of information extraction and the generation of more fine grain environmental representations.

Since the global world model already possesses a hierarchical structure, translating it into smaller scene-graphs is actually not that complicated. Everything that is required is a "bootstrap", a "complex" entity identifier, which might be a room, a position, an identified robot, a marker, or if the entity itself is already represented within the global model (by a complex entry), the ID of its base, etc. As depicted in Fig. 3.5, the reconstruction of the scene can be organized recursively in two directions: upwards, whereby, the ID of an entry has to match the current base ID and downwards, by querying for the base IDs of other entries. (See, hence, also the brief description of the appertaining implementation in Sec. 4.2.2 as well as the listed screencast that demonstrates how the virtual overlay structure is accessed and queried.)

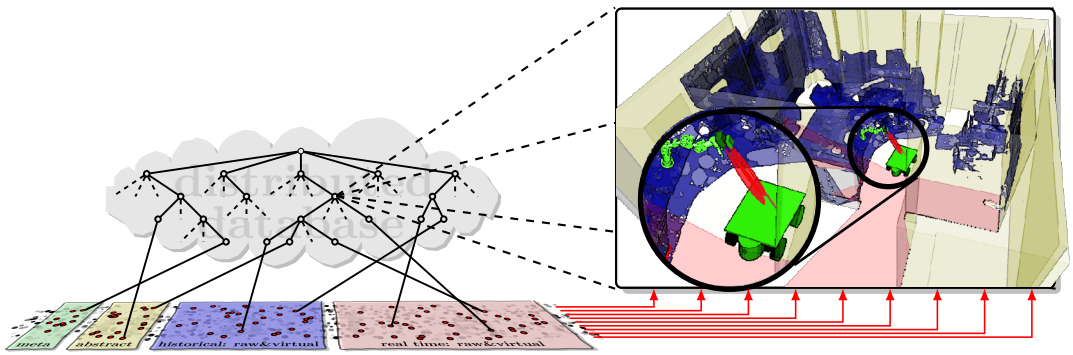


Figure 3.5.: Reconstruction of a scene from a distributed scene-graph with an extruded map of the environment (yellow), robots and sensors (green), stored Kinect scans (blue), and online accessed real-time sensor readings (red), the red arrows mark the connection of the model to required real-time data.

In contrast to a "simple" 3D representation, a rigid-body simulation distinguishes between the entities within (e. g., robots, sensors, furniture, walls, etc.) and allows the attachment of semantic, physical properties, offered services, topic names, pose uncertainties, etc. to every object, which is stored or linked to within the virtual overlay database. In fact, the reconstruction of such a local world model is not a real technical challenge and there are plenty of simulation environments that can be used to maintain a local world model (see the overview in Sec. 2.3.4.1.14). But in contrast to Roy's ODE environment, which is a physics simulation environment only, it is more valuable to apply more complex tools. Tools that also support the integration, and thus the simulation, of different sensor systems, an integral part that is left out by all approaches, presented in the related work at the knowledge layer in Sec. 2.3.4. The next section is thus used to present some of the benefits that arise due to the application of simulated sensor systems. As described in Sec. 4.2.1 on page 97, the robot simulation environment OpenRAVE was utilized and adapted for this purpose, but the general system architecture also allows it to apply different simulation environments.

[1] A cooperation with the Fraunhofer IFF in the ViERforES (Virtuelle und Erweiterte Realität für höchste Sicherheit und Zuverlässigkeit von Eingebetteten Systemen) project. See also the description of the EXCELL project: `http://www.iff.fraunhofer.de/en/business-units/robotic-systems/echord-execell.html`

As mentioned within the introduction to this chapter, the local model should also be applied to replicate any movement, any state change, and also every sensor measurement that appears within the real world. This requires connecting virtual robots and sensor systems to their real world counterparts, as it is indicated with the red arrows within Fig. 3.5. Thereby, all required communication settings should be read from the "complex" entry that represents a distinct entity. Different mechanism in the background can subscribe for or request data via various communication systems and update the local model accordingly. Thus, the reconstructed model itself can be used as local knowledge base maintaining the spatial relations between different entities, but it can also be applied as a general interface to different types of sensor systems. Hence, an application does not have to cope anymore with communication aspects; everything can be hidden behind the local world model. The only thing required is an interface to real and simulated measurements. How this is accomplished in described detail is in the implementation for OpenRAVE in Sec. 4.2.2.2.

The appearance of stored raw and virtual data can only be linked to the model. Note that data such as camera streams or Kinect depth scans cannot (and do not have to) be analyzed directly just as recognized objects (in stored or online accessed real-time data) which are directly integrated into the local model. Additional transformation-, fusion-, filter-, etc. functions have to be applied afterwards, based on the type of requested information. This issue is discussed within the next layer.

3.2.2. Application

As introduced at the beginning and revealed in earlier publications, the application of rigid-body simulations for explicit world modeling has several advantages, but it also causes some problems. Both aspects are briefly described within the following paragraphs. See also the following play-list to get an impression on some of the main aspects discussed in [3]:

`http://www.youtube.com/playlist?list=PLgJeoIw_8oS5ORCWmaZtAD9fX3QpcmOlY`

3.2.2.1. Benefits of a Local World Model

As already introduced and previously discussed in [4], precisely simulated environments in conjunction with simulated sensor systems can be utilized in various ways. On the one hand, it offers some kind of the ability to perform **virtual measurement** as virtual sensors can be attached everywhere if their specific measurement formats are required (e. g., distance values, laser scans, point clouds, etc.) as input to a certain algorithm (cf. [3]). Complex hulls within the virtual world can be also defined as safety zones and applied as collision detectors with virtual objects that have a real world counterpart. It thus enables quite complex transformations, while the correctness of the model is maintained with the help of other sensor systems that combine real and virtual measurements. Thus, if real sensors are also represented within the virtual world, it offers some kind of **virtual redundancy** that can be used in two ways, and which is highly dependent on the quality of the virtual environment representation (it is discussed within the next sub-section). It allows generating intermediate sensor measurements with arbitrary frequency that indeed only provide approximations to the real value. The elapsed time between two real measurements should therefore always be integrated as a weighting factor. On the other hand, virtual and real measurements can be used to perform some kind of **plausibility checks**, as it was described in [5] for a manipulator with different head mounted sensors. An error margin was thereby estimated that defined acceptable deviations between real and virtual

measurements. If there is a discrepancy between real and virtual measurements which cannot be explained by measurement noise (described within the sensor data-sheet), a system will have to reason about the source, whether the environmental representation or some sensor measurements is faulty. Probably, dismissing some of the sensor systems results in a more consistent configuration, concerning the virtual representation and measurements compared to real measurements. It is thus also possible to deal with environmental properties or changes (which occur in highly dynamic surroundings) that might affect a sensor measurement (cf. [8]). For example, the effect of different surfaces such as reflection or absorption on light-based or ultra-sonic-based distance sensors can be identified as easily as ambiguous measurements, which are caused by exceeding a specified measurement range. In fact, these are no faulty measurements at all — a precise environment model, replicating those measurements would gain the same results and could still be utilized. But by additionally replicating any action within the environment model a system becomes aware not only of the influence of its local behavior on sensor readings (e. g., due to self-occlusion), but also on changed sensor readings due to the behavior of external systems (e. g., by virtual replications of robotic movements/trajectories, etc.) or due to changed environmental conditions (e. g., a newly placed box or a closed door, etc.). The effect of external actions or status changes does not have to be observed with local sensors, it can be observed within the virtual representation (cf. [3]).

Thus, it allows the identification of disturbed sensor systems, which further enables an advanced **sensor selection** as a basis for higher level fusion and transformation as well as the **integration of external sensor systems** to it (cf. [20]). Using external sensor measurements for a better position estimation by applying motion sensors, surveillance cameras, laser scan and distance measurements also from other mobile robots (which continuously publish their positions) or for other tasks in smart environments (e. g., the detection or recognition of humans and objects) requires the precise knowledge of what sensor systems are available, what is their relative position and orientation (with probably known uncertainties), as well as what their coverage area is. The model can thus be searched in order identify relevant sensors that monitor a certain area or to identify an appropriate set of sensor systems for a given task. As already described, the model shall also be used to access the data of external sensors.

Additionally, since real physics engines are applied and objects are also labeled with physical properties, a persistent model can be used to make **predictions** about future states of the environment (or collision) even if objects leave or roll of (as discussed in Roy's mental imagery, see Sec. 2.3.4.1.15) an area that is monitored by different sensor systems, their new positions can be estimated based on the physical simulation. Similarly, it is also possible to foresee possible collisions by maintaining a forwarding simulation for a certain time horizon.

3.2.2.2. Problems, Difficulties, and Challenges of a Local World Model

There are a couple of problems that are quite hard to grasp, but fortunately they have been already widely investigated and discussed in related areas, such as in the research fields of Data Quality [34] and Information Quality (IQ) [29], which can be related to previously introduced information science (cf. Sec. 2.2.1) and rely on similar distinctions between data and information. IQ defines a couple of metrics that can be used to measure the value or to quantify the confidence of particular information, or in other words: how good does an information reflect a certain aspect of "reality". It is thus also associated with the

relevance for a decision making process. Since data is defined as the basis for information, the quality of data has a direct effect on derived information. An approach for the online evaluation of the quality and validity of data, which also incorporates the application of fusion, filter, and transformation algorithms, was developed by the working group EOS [41, 42]. It allows identifying valid data at runtime, based on a predefined properties vector. But both approaches correspond in the statement that poor data quality cannot result in good IQ.

The same applies to the quality of the local world model that is derived from a common data source that represents the entire environment. There is currently no metric and, in addition, there will probably not be any metric describing the **incompleteness** or completeness of data and, thus, also of the local model. Therefore, the model always comprises all environmental data and information for a certain vicinity that is directly accessible. There is currently no other selection process applied than the one that uses the spatial relations. What else might be relevant or not to solve a certain task or to deal with a specific problem requires some kind of description, which is introduced within the next section. But the problem of incompleteness remains, and there is no way of identifying what might be missing or to reason about the degree of completeness. At least **inaccuracy** can be handled to some extent, by associating uncertainty values to pose information, sensor systems, or measurements, etc. In Sec. 4.2.2.3 it is described how arbitrary information can be attached to an entity within the simulated environment, but there is currently no simulation environment that is capable including these uncertainties into the simulation process. Additional methods have to be implemented, such as flow simulation, virtual spraying, sealing, etc., which can be seen in the overview at: `http://www.fcc.chalmers.se/software/ips`

Another problem that is difficult to deal with is **inconsistency**. Inconsistency might arise due to several factors. There is the perceptional anchoring problem, which was already introduced in Sec. 2.3.4.2.13 for the PEIS ecosystems, but that is also known in IQ. In IQ it can be related to **ensuring convergent validity** [29], which is defined as the extent to which multiple items that measure the same construct are or can be correlated. Thus, an object that was identified by multiple robotic systems does not necessarily have to be represented as one "complex" entry within the overlay database and, thus, also within model. In combination with a weak consistency level, this might result not only in an incomplete or inaccurate but in a **wrong representation** of the environment (with multiple appearances of one object). Literally speaking, different systems might have a different view on the same object or on the environmental configuration (based on different sensory systems and different points of view). Such inconsistencies can leak into the global environment model, but the connectivity of entities (different time-stamps) within a distributed environment can also cause inconsistencies. Inconsistency due to disconnected systems can be handled to some degree by underlying database system. As it is described in the implementation in Sec. 4.1.1, eventual consistency might be the best choice for distributed systems of dynamically changing (connected/disconnected) entities. Thus, not every data/information is accessible immediately and a model will therefore present an incomplete and probably not actual representation of the environment.

But how could the **model quality** of or the **confidence** in it be measured? One way would be to identify the degree of correspondence between the previously introduced virtual and real sensor measurements. An accurately simulated environment should produce similar measurement results as the real one. The amount and diversity of sensor systems could be also used as a weighting factor. Furthermore, **reputation** could be introduced as an additional quality measure, which can be associated with the entity that was responsible

for producing and storing data or information (based on the accuracy of earlier insertions). And of course, the associated time-stamp to information can be utilized as an additional value to describe the **timeliness** of a model.

3.2.3. Discussion

This conceptual part was used to demonstrate the principles and describes the potentials and problems of applying a rigid-body simulation to replicate the local surroundings. So far there have been also only weak requirements for a simulation environment defined, a scene-graph structure to maintain all components of the environment and the capability to simulate robots and sensors as well. As already mentioned, OpenRAVE was chosen due to several reasons (see Sec. 4.2.1), but there also might be others. Depending on the task and probably also on external environmental conditions (e. g., fog due to spraying, different ground conditions (working on soil), temperature and humidity can affect sensor measurements, etc.) another simulation environment can be more useful and provide more sophisticated simulation capabilities and other APIs. Nevertheless, the described problems will still remain.

A very nice fact at this point is that it is more or less irrelevant what environment is used, how information is organized, and how it can be interfaced. The simulation environment is used as a basic framework for the next steps. It can thus be replaced without affecting the application that is using it for analysis, forecasting, abstraction of other environmental representation, or to simply extract certain information. But how can it be irrelevant? The next section introduces a new declarative query language that abstracts the underlying environment and allows querying any kind of discrete operating simulation with an SQL-like syntax, whereby an abstract machine is then responsible to produce the requested output.

3.3. Extraction & Abstraction

> Change your language and you change your thoughts.
>
> —"Karl Albrecht"

Within this section, the notion of a holistic query language as well as the grammar and syntax of such a language are introduced. In fact, there is a need for such a language due to several reasons [9]. I firstly realized this when the interfaces to an applied local world models and to the developed filter and transformation functions (see Sec. 4.3.1) grew in their complexity and became hard to maintain. Furthermore, there is a selection process involved in nearly every extraction of information or in the application of filter and transformation function, since they are not applied onto the entire local world model but only on some "selected" parts or aspects. Many parameters for these functions have to be set dynamically, depending on predefined requirements, the environmental configuration or other context information.

As recognized in [35, 130] (see also Sec. 2.3.4.2.15), a scene-graph presents a complex structured world model that requires some kind of semantic/syntax to be able to define complex situations within, including temporal aspects along with spatial ones in order to support situational-awareness.

Conceptually, there are three directions imaginable for dealing with these problems: A fixed set of templates allowing the extraction of predefined sets of information. A probably more advanced solution would support basic and imperative scripting elements, enabling a higher flexibility according to parameter settings and sequence changes (order of execution of template-based filters/functionality). A declarative approach would push the effort towards an abstract machine or algorithm, allowing to define what is required rather than to define how it is achieved. Since none of these solves the problem of extracting information alone, a combined approach is proposed, whereby the main emphasis is put on developing a declarative query language, which also comprises imperative aspects, allowing the identification of the appropriate combination and parametrization of functions/templates.

Thus, if we interpret a local world model as the underlying knowledge base, it should also be possible to query it with a syntax and semantics that is similar to the access of an "ordinary" database. With the listings 2.9 and 2.10 in Sec. 2.4 on page 57 I already tried to introduce the SQL syntax as a convenient approach. Within the following section it is described why this approach is not that unfounded as it might sound at first and how it can benefit by integrating also imperative programming elements. It is rounded out by a definition of language requirements and a brief overview on syntax and grammar. Most of these topics have been introduced already in [6] and to some extent also in [9].

The third part of this section deals with an extension of the language in order to support reasoning, which was presented in [11]. Reasoning with the help of a SQL-like syntax might sound even more bizarre, but as revealed later on, it is quite reasonable and it enables the definition of even more sophisticated queries.

3.3.1. Concept of a Holistic Query Language

3.3.1.1. Databases vs. Simulations

SQL or "Structured Querying Language" [47], is a so-called 4th generation declarative language, which nowadays has become a standard for defining, storing, manipulating, and querying data in databases (cf. [69]). And although databases are in general not directly portable across different Database Management Systems (DBMSs), and different DBMSs were developed for different purposes with different and specific implementation details (which is similar to the amount of world models and simulation environments), SQL offers a standardized interface to access all data and relations. Even non-relational DBMSs, so-called NoSQL-systems (Not only SQL, for a broader overview see also [246]) try to copy or (re-implement) the syntax and the semantics of SQL because of its expressiveness and wide acceptance, well-known examples are Cassandra's CQL (see, for example, Lis. 4.4 on page 94) [242] or HBase with Pheonix [260].

Although it has turned into a standard in the database community, SQL does not only cope with data; instead, it offers an abstract method for defining, manipulating, and querying relations between "entities". From this point of view it can be said that there is actually no real difference between a database and a simulation from the data point of view, both represent a side of the same medal. As depicted in Fig. 3.6, databases are usually used to store all relevant aspects of a system or a process, whereby a simulation is applied to replicate a system's behavior for the purpose of studying or forecasting. Databases can also be used to derive important simulation parameters, or data from simulations is stored for further analysis in a database. Thus, if both are applied to describe the same system (e. g., the weather, strength and dynamic analysis, traffic-flow, a library rental system, etc.), then

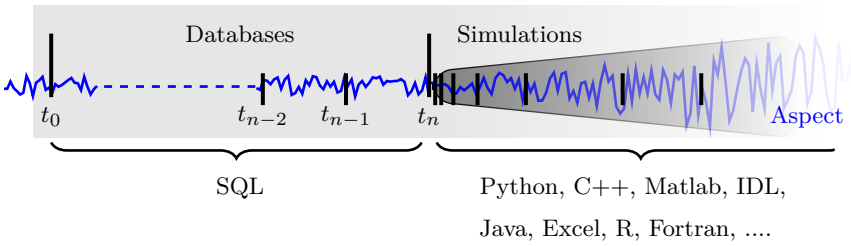


Figure 3.6.: Database entries can be interpreted as representations of a finite set of system clippings, which start with a first measurement at time t_0 and range to the latest measurement at time t_n (now). A simulation is applied to predict future system states or outputs. These future "measurements" can be produced for arbitrary small or big time steps, whereby every value is additionally affected by a growing discretization error over time.

the data that is stored within a database will commonly mirror the data that is reproduced by a simulation (e. g., temperature and humidity according to time and space).

No one would nowadays try to query a database by directly accessing all data sets and analyzing or filtering them manually with an imperative or procedural programming language. However, this is still true for most of the simulation environments applied in robotics (as well as in other fields). Would it not be nice to query both sides in the same way and with the same syntax, such as a weather database/simulation as it is listed below?

Listing 3.1: Identification of European cities whose average temperature is above a certain threshold

```
1 SELECT cities, inhabitants, mean_temperature, ...
2 FROM cities_in_europe
3 WHERE mean_temperature > 33.0
4 ORDER BY inhabitants DESC ...
```

Or simply to analyze network traffic, which is either stored in a database or multiple log-files or generated by a network simulator (e. g., ns2[2] or ns3).

Listing 3.2: Exemplary SELECT-statement for an ns2 simulation

```
1 SELECT average(throughput), average(packet_loss) FROM network_communication
2 WHERE routing == 'RIP' AND fail_rate > 0.05 ...
```

As already discussed in [11, 6] and also presented by some approaches in Chap. 2, SQL-like query languages have already been introduced to other fields than databases. For example, the language integrated query (LINQ) [170] extends some of the .NET languages to apply SQL queries to relational databases as well as to arrays. It is similar to other approaches in the Java world, such as the Java Query Language (JQL) [224] or SQL for Java Objects (JoSQL) [238], among others and with a reduced syntax. But there are no such attempts that are applicable to query simulations online. Searching the literature for relevant publications, for querying world models, or simulations, or at least scene-graphs with the required expressiveness yielded only in one publication [135]. Even the authors of this paper note that they could not find related approaches to their developed query language for simulated mesh data, which is called MeshSQL. It allows defining queries for mesh-based physics simulations. In contrast to the approach presented in this thesis, the results of a simulation are stored within a database, according to time and space. MeshSQL is thus a real extension to SQL1999 [69], which is specialized on mesh data only.

[2]Network Simulator 2: `http://www.isi.edu/nsnam/ns`

This query language is intended to enable researchers to interactively explore simulation data, to identify new and interesting effects. MeshSQL therefore offers temporal, spatial, statistical, and similarity queries, which require different types of return values, i. e. simple values, surfaces and slices in different formats, which is also a requirement for my approach.

But before proceeding to the next part, due to several reasons there was initially also a discussion of applying a Prolog-like syntax (cf. [9]). Prolog and SQL are "quite" similar and appeared both during the 1970's. Both are declarative languages and the Predicate Logic, used by Prolog (logic programming), is a subset of the Relational Calculus implemented by SQL (relational programming). Furthermore, a subset of Prolog has been already applied successfully as a database query language, called Datalog [46]. Since the focus initially was more on extracting facts, relations, different abstractions and not on reasoning, SQL seemed to be the better choice, also due to its expressiveness and its widespread usage and acceptance. But as already mentioned, the developed query language was later extended to be applicable to reasoning problems too (see Sec. 3.3.2.3).

3.3.1.2. Language Requirements

As already introduced and also depicted in Fig. 3.7, the main idea is to put a thin abstraction layer on top of the local environment model similar to standard SQL, which allows to apply the same queries over and over again, even when the underlying DBMS changes. As presented later in Chap. 5, this approach even allows concealing most of the effort that are required for accessing the overlay database and model generation. Fig. 3.7 shall further give an impression on the capabilities of the newly developed declarative query language named SelectScript. Select due to the primarily applied statement and Script because of its extended scripting capabilities.

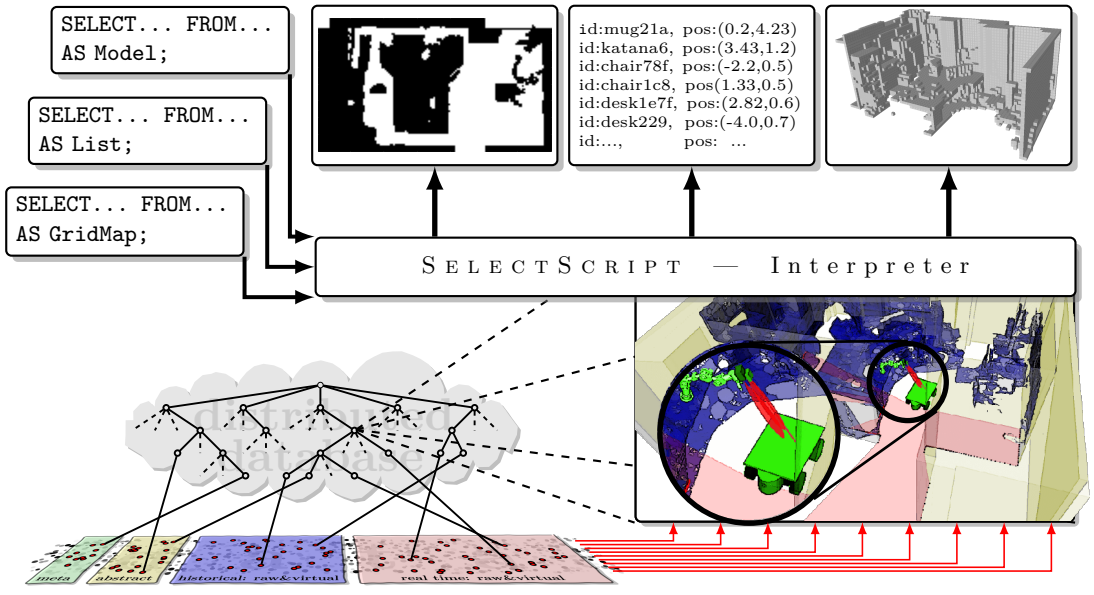


Figure 3.7.: Interpreting the local environment model as an implicit and dynamically changing knowledge base should allow the application of an interpreter and a SQL-like syntax to define what kind information should be extracted from it

If the generated responses depicted in Fig. 3.7 appear somehow familiar, this is because they were previously already presented in Sec. 2.3.4.1 on page 36. All of the presented world models there (except the topological map) were generated with simple SelectScripts and the transformation functions presented in Sec. 4.3.1. To get a further impression on the language and its application (before continuing), take also a look at the YouTube play list that was used to demonstrate parts described in [6]:

`https://www.youtube.com/playlist?list=PLgJeoIw_8oS5bgn94qFAVo88FRpXRtqpN`

3.3.1.2.1. Dynamic Interpretation: Since models are constructed dynamically, on the basis of the information stored within the underlying overlay database, and simulating a variable and continuously changing environment, not all possible combinations of tasks and associated queries are known at design-time. It is therefore more appropriate to apply an interpreted language (as it is also true for most declarative languages). It should allow creating and answering new kinds of queries dynamically or adapting existing ones, based on the current context.

3.3.1.2.2. Adaptability & Extendibility: As already introduced, there might be different types of co-simulated local environment models. Thus, adaptability denotes the requirement of porting the language/interpreter to different "discrete" simulation environments.

As it was shortly introduced in Sec. 2.3.4.1.10 on page 37, Prolog can be interpreted as an example for Language-Oriented Programming (LOP), since on top of it different languages have been implemented with support for very specific types of logical world models. LOP was summarized at best in [64]:

"*... I want to be able to work in terms of the concepts and notions of the problem I am trying to solve, instead of being forced to translate my ideas into the notions that a general-purpose language is able to understand (e. g., classes, methods, loops, ...). To achieve this, I need to use domain-specific languages. How do I get them? I create them ...*"

As it is later revealed in Sec. 4.3.2, there is actually no need to provide a whole implementation for one world simulation environment. A "whole implementation" means a complex software to grant access to another complex software. Instead, it is more beneficial to provide a small language stub for effective querying, which can be extended afterwards to comply with different environments, required functionality, and probably also with different programming languages. It thus should be possible to derive new "dialects" for different underlying systems. From this point of view it can also be applied as a meta-programming language, which allows combining existing functions dynamically to generate desired information (cf. Lis. 3.4 on page 78). Thus functions should provide a basic API, allowing to interface the underlying system effectively.

3.3.1.2.3. Basic Scripting Capabilities: In contrast to SQL and its derivatives, the language should at least provide basic scripting capabilities. Since not everything can be solved with basic `SELECT` statements only, a combination of declarative and imperative aspects offers the opportunity to define simple and expressiveness queries. Reoccurring patterns and sequences of commands could be used to define procedures in order to reduce the complexity of queries.

3.3.1.2.4. Situation-Awareness: As previously reviewed in Sec. 2.3.4.2.15, Belkin et al. emphasized the necessity for appropriate queries on scene-graphs and a definition on a symbolic level. They argue that even the simplest situations can be defined by a single symbolic attribute value of an object (e. g., a person is smiling). Prolog or any other formal representation, therefore, seems to be more suitable for defining and checking situations, but as shown in Sec. 5.4 a SQL-like syntax can be applied to accomplish this task as well.

Rather than continuously querying (poll) the simulated environment, the system should provide callback mechanisms for an event-based notification, allowing the encoding of a situation once and attaching it to the currently simulated environment so that every time a situation is detected, a specific callback is executed to inform a subscriber.

3.3.1.2.5. Inclusion of Temporal Aspects: A scene-graph commonly depicts only the spatial environmental configuration for a certain point in time. An exception to this is provided by the distributed scene-graph that was described in Sec. 2.3.4.2.4, wherein transformation nodes also store a local history of transformations. A query language should thus also provide the possibility for defining temporal queries in order to include time into the analysis process. And, therefore, to be able to query for example: since when is something true, for how long has it been the case, are there temporal patterns, etc. The same is true for the previous point, defining situations, which also requires the ability to incorporate temporal aspects.

Within the presented approach, this was solved with a new concept for temporal variables. These variables store the result of a query for a certain period of time along with their time-stamp (cf. Lis. 3.9). These variables can be further reused and queried, with the same syntax as a local robotic world model is queried (see Sec. 3.3.1.3 and also the example in Lis. 5.16 on page 130).

3.3.1.2.6. Coping with Different Representations: As already mentioned, the locally applied world model is utilized as the central knowledge (fusion) base from which all required information shall be extracted as well as other knowledge representation formats. These formats thereby cover all logical & formal and spatial 2D & 3D representations that have been listed to some extent in Sec. 2.3.4.1. Thus, the language requires some kind of mechanisms and a special syntax, allowing to request information and knowledge in different formats (the previously mentioned templates). It has also to be extendable and adaptable to meet new requirements.

Applications and algorithms in this sense do not have to be adapted or altered to make use of different input formats. Instead, it only has to be requested in the desired format so that all the details about conversation and transformation are handled in the background of language (abstract machine). In the language concept that is described within the following section, this is accomplished by the altered application of the SQL keyword `AS` and the following statement, which denotes the requested format and needed parameters (cf. Lis. 3.7).

3.3.1.3. SQL Adaptations to Comply with Simulations

This section is intended to introduce the envisioned language and its expressiveness. It is furthermore intended to demonstrate that most of the previously described requirements can be quite easily solved or in other words expressed with the help of a SQL-like syntax. Lis. 3.3 depicts a simplified version of the SelectScript grammar (there are actually more

rules for evaluating arithmetic or Boolean expressions, etc.). However, it also depicts five language features that were already mentioned, but described in more detail here.

SelectScript covers only a subset of SQL, namely the possibility of defining `SELECT` queries as listed in lines 42 - 62 in Lis. 3.3 and the possibility of defining `PROCEDURED`, see line 65. As it is visible, a query covers the basic `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `ORDER BY`, `LIMIT` and `AS` expressions. Thereby, only the application of the keyword `AS` was altered to support the request of different response formats. The usual column names, which are used to define the return values of a query, are defined by the identifiers of an expression, such as the name of a function called (line 42), or specific built-in functions. Functions are determined by an identifier and parentheses (line 38). As already mentioned, functions are defined externally within the host programming language and are only linked to SelectScript (see Lis. 4.17 on page 110).

A single `SELECT` definition can thus look like the one listed in Lis. 3.4. If there are multiple parameters to be passed to a function, then the keyword `this` is applied to mark the correct position of the element that was identified by the `WHERE`-clause. Furthermore, it is also possible to apply nested function calls. Here, the additional parameters can also contain complete expressions, `SELECT` statements, etc. (see line 35 in Lis. 3.3). The definition of the language allows it furthermore to apply nesting nearly everywhere.

Listing 3.4: Basic `SELECT` expression with function calls as a column substitution

```
1  SELECT fct_1, fct_1(this), fct_2(p1, this, p2), fct_3(fct_1(this)*2, p3) ...
```

The keyword `this` is furthermore applied to keep track of different sources, defined in the `FROM` expression. If there are multiple sources defined, then the name of the source is concatenated with `this` to mark the position of elements for evaluation within a function or another expression, as depicted below. Or if, for example, the Cartesian product has to be evaluated, local variables can also be defined within the `FROM` expression (see line 3 in the listing below).

Listing 3.5: Application of the `this` pointer

```
1  SELECT fct(this) FROM struct ... ;
2  SELECT fct(struct_A.this), fct(struct_B.this) FROM struct_A, struct_B .....
3  SELECT fct(A.this), fct(B.this), C.this FROM A=struct, B=struct, C=(SELECT ...
```

The `WHERE` is used identify relevant elements; and only if this expression evaluates to true, the elements are selected and passed over to the next expressions, or they are ignored otherwise. If no `WHERE` expression is defined, all elements are handed over automatically. The same functions that are applied in other parts can also be applied here as well as in the `GROUP BY`, `ORDER BY`, or `LIMIT` sections, which implement the same functionalities as in ordinary SQL.

Listing 3.6: Application of `WHERE`, `GROUP BY`, `ORDER BY`, and `LIMIT` expressions

```
1  SELECT .. FROM .. WHERE fct_1(this) < fct_2(this, p) AND (( ...
2                          GROUP BY fct_3(this), ...
3                          ORDER BY fct_4(this), ... ASC # or DESC
4                          LIMIT 33*3
```

Furthermore, there is an additional keyword `AS` that has already been mentioned in the last paragraph of the previous section, and that is defined in line 58 in Lis. 3.3. It is applied (in contrast to its common SQL usage) as a key element of the language to define the

Listing 3.3: Simplified overview on the SelectScript grammar[3]

```
# a script is a set of ex, each statement is closed with a semicolon            #
  <SCRIPT> ::= (<STMT> ";" | <COMMENT>)+                                        #
#-------------------------------------------------------------------------------#
  <STMT> ::= <ASSIGNEMENT> | <SELECTION>     #                                  #
         |   <PROCEDURE>   | <EXPR>          #                                  #
         | "(" <STMT>";"(<SCRIPT>";")+ ")"   # def. of a sequence of statements #
#-------------------------------------------------------------------------------#
# dynamically typed variable assignments, with two types of variables:          #
  <ASSIGNEMENT> ::= <ID> "=" <STMT>          # standard                         #
        | <ID> "{" <STMT> "}" "=" <STMT>     # temporal - with time horizon     #
#-------------------------------------------------------------------------------#
# allowed expression (identifiers are used as variable or function names)       #
  <EXPR> ::=  <uOp> <EXPR>                   # unary operators                  #
         | <EXPR> <bOp> <EXPR>               # binary operators                 #
         | <ATOM>                            #                                  #
#-------------------------------------------------------------------------------#
# Boolean, compare, and arithmetic operators (evaluated with precedence)       #
  <bOp>  ::= "+"  |  "*"  |  "=="  |  "<"  |  "<="  |  "%"  |  "AND"  |  "OR" | #
             "-"  |  "/"  |  "!="  |  ">"  |  ">="  |  "^"  |  "XOR"  |  "IN"   #
  <uOp>  ::= "-"  |  "+"  |  "NOT"                                              #
#-------------------------------------------------------------------------------#
  <ATOM> := (<ID> "." )? "this"              # pointer                          #
   | <FUNCTION>                              # function call                    #
   | <ID> ("{" <STMT> "}")?                  # variable call, eg. Nma_12 {time} #
   | "[" <PARAMS>? "]"                       # definition of a list             #
   | <STMT> "[" <PARAMS> "]"                 # slice operator for lists         #
   | see Lis. 3.10 on page 81                # scripting extensions ...         #
   | "(" <STMT> ")"                          # parenthesis                      #
   #---------------------------------------#  literals:                        #
   | <Bool>                                  # T(rue), F(alse), 0, 1            #
   | <Int> | <Float>                         # ... -1, 0, 1, 2 ... | -3.14159   #
   | <String>                                # "enclosed by quotations"         #
#-------------------------------------------------------------------------------#
# list of parameters, applied by select statements, functions, lists, or by     #
<PARAMS> ::=  <STMT> ( "," <STMT> )*         # additional AS representations... #
#-------------------------------------------------------------------------------#
# function call with and name and arbitrary parameters                          #
<FUNCTION> ::= <ID> "(" <PARAMS>? ")"        # identifier(p1, p2, (33*4), ...)  #
#-------------------------------------------------------------------------------#
# querying with the possibility for specialized return values, which is defined #
# by the final keyword "AS" ...                                                 #
<SELECTION> ::= "SELECT" <PARAMS>                                               #
   "FROM" <PARAMS>                                                              #
   ( "WHERE" <STMT> )?                                                          #
   ( see Lis. 3.13 on page 85 )?             # reasoning extensions ...         #
   ( "GROUP" "BY" <PARAMS> )?                                                   #
   ( "ORDER" "BY" <STMT> ("ASC"|"DESC")?                                        #
             ("," <STMT> ("ASC"|"DESC")?)* )?                                   #
   ( "LIMIT" <STMT> )?                                                          #
   ( "AS" ( "val"                            # only the first value             #
          | "list"                           # array representation (sequence)  #
          | "dict"                           # default representation (table)   #
          | "dummy"                          # for execution only, nothing ret. #
          | <ID> ( "(" <PARAMS>")" )? )?     # enabling extensions              #
#-------------------------------------------------------------------------------#
# procedures are internal function definitions, parameter-passing is allowed    #
<PROCEDURE> ::= "PROC" ("(" <ID> ("," <ID>)* ")")? ":" <STMT>                   #
#-------------------------------------------------------------------------------#
<COMMENT>::= "/*" * "*/"                     # internal or multi line or single #
           | "#" * <EOL>                     # line comment (until newline) ... #
```

resulting format of a query, which might be a single value, a list of values, a dictionary, or nothing (`dummy`, if only the expressions have to be evaluated). The dictionary is also the default format and preserves the expression identifiers as keys in order to generate a SQL-like table (see also Lis. 3.14).

But it is also possible to request something totally different, such as an occupancy grid map (see Lis. 5.12 on page 126), Prolog clauses (see Lis. 5.18 on page 131), or a map showing the sensor coverage of an area (see Fig. 5.2d). Further expressions can be integrated in order to extend the language according to various requirements, in addition to the incorporation of new functions and additional structures. The optional parentheses after the representation format can be used to pass additional parameters to the method that is used to create these representations (in fact, they can also contain further (nested) expressions or queries that are automatically evaluated before passing).

Listing 3.7: Requesting different formats with the keyword `AS`

```
1  SELECT .. FROM .. AS representation(..); # dict(ionary), list, value, or ..
```

As depicted in line 2 in Lis. 3.3 a script can contain multiple statements (and comments, equal to SQL, line 67) and intermediate results can also be stored persistently in variables (see definition in line 9 in Lis. 3.3). This allows to define more complex query scripts by applying the resulting values, stored in a variable, as inputs to other queries (see therefore line 2 in the listing below). In this context, persistently means that the values stored in a variable cannot only be accessed within the same script, but also later by other scripts as well.

Listing 3.8: Application of basic variables

```
1  var = (2*3+3.141592)^0.5;      var_list = ["string", fct(2), var, [...]];
2  var_struct1 = SELECT ...;      var_struct2 = SELECT ... FROM var_struct1 ...;
```

In order to be able to define temporal queries, such as at what point in time something has become true, or for how long has it been so (in some cases it might also be necessary to define complex temporal sequences), a new type of variable was defined (see the definition in line 10 in Lis. 3.3 and the application in the listing below). Temporal variables are defined with curly braces. Such a variable allows keeping previous results of a script over a certain period of time and can be applied as a base for further queries (see the example in Lis. 5.16 on page 130). The curly braces are thereby applied in multiple ways, within a variable assignment they define a maximum time horizon, and empty braces define an “infinite” time horizon. Temporal variables are especially useful when dealing with situations and callbacks. As later presented Sec. 5.4 the applied callback mechanism allows it to register a script, which is then continuously evaluated in the background. Because of the persistence, the content of such a variable is continuously altered every time the script gets evaluated. The content of these variables can then be accessed as follows. Calling a temporal variable without braces returns only the last stored value. It thus appears as if it would be an ordinary variable. Calling it with empty braces returns all stored values. A negative value defines a time horizon dating back from the current time, the selection in line 2 is thus applied onto all stored values within the last 2 time units. And finally, it is also possible to call such variables with positive values within the braces, which are used to return a value for a certain point in time.

Listing 3.9: Application of temporal variables

```
1  var_temp1{2.5} = SELECT ... ;  # with time a horizon of 2.5 time units
2  var_temp2{} = SELECT ... FROM var_temp1{-2} ... ;
```

But what is the result of a script that contains multiple queries and variables, and in which there is no keyword such as `return` (used in other languages)? In fact, the decision about this was quite simple. The result of a script is always represented by its last statement. Thus, if there are multiple `SELECT` statements, then the last one defines the return value (identified by the keyword `AS`). But the last statement can also be single placed variable, the result of a function call, even a variable assignment or a newly defined list, which allows to place relevant values freely and, thus, to generate complex return formats.

3.3.1.4. Additional Extensions

But additional elements, not commonly known in SQL, have also been introduced to define more elaborated queries (see Lis. 3.10). Examples are the `IF` expression that can be used even within a `WHERE` expression to change its result. `print` can be applied for extended logging, see therefore also the example in Lis. 3.12 on page 83. There are further built-in functions such as `eval` that enables reflective programming by dynamic evaluation, `to` enables the redefinition of identifiers, `mem` and `del`, which can be used to deal with memory issues, and `help`, which is intended to provide a basic help-desk on functions. (For their application see also Sec. 4.3.3 on page 110).

Listing 3.10: SelectScript extensions to support hierarchical queries

```
66 # additional helpers that can be placed everywhere, see the example in Lis.5.8  #
67 "IF" "(" <STMT>                          # evaluates to true or false           #
68          ( "," <STMT>                    # then ...                             #
69            ( "," <STMT> )?  )?  ")"      # else ...                             #
70                                                                                 #
71 # print out arbitrary log information with this inline function, the last       #
72 # statement defines also the return value ...                                    #
73 "print" "(" <STMT> ("," <STMT>)* ")"                                           #
74                                                                                 #
75 # return memory relevant information about variables and procedures ...         #
76 "mem" "(" <ID>? ")"                                                            #
77                                                                                 #
78 # deletes a variable and return its value                                       #
79 "del" "(" <ID>  ("," <ID>)* ")"                                                #
80                                                                                 #
81 # evaluate a piece of code                                                      #
82 "eval" "(" "SelectScript-commands as strings" ")"                              #
83                                                                                 #
84 # change the identifier/function-name within a SELECT expression, it is used to #
85 # change the keys if a dictionary is requested, such as:                        #
86 "to" "(" <String>, <STMT> ")"  # SELECT to("a",f(a.this)), to("b",f(b.this))    #
87                                                                                 #
88 # return all function names (if empty) or information about a specific one      #
89 "help" "(" <STRING>? ")"                                                       #
```

3.3.1.5. Sequences and Procedures

The definition of a statement in Lis. 3.3 on line 6 allows the definition of sequences, which can be interpreted as some kind of sub-program execution. Thereby, the last statement in the sequence also defines its return value. Only one global lexical scope is assumed, and thus, variables defined within a sequence or within the main script persist unless they are deleted explicitly.

Procedures are thought to be used as substitutions for more complex statements. They can be stored in variables and be accessed like a variable, whereby their definitions are evaluated at first to generate the result value. To provide a basic methodology to define

"function-like" expressions, procedures can also be defined with input parameters. Input values are thereby not associated with variables, but instead with `this` pointers.

Listing 3.11: Sequences and their application with procedures

```
1  result = (a=1; b=2; a+b;);
2  print( result );              # outputs 3
3
4  p = PROC: a+b;                # procedure is stored within a variable
5  a = 98;
6  print( p );                   # evaluating p first, before outputting 100
7
8  p = PROC (a, b):(a=2*a.this;
9                   a+b.this;);  # also stored as a variable
10 print( p(2,2) );              # output: 6
11 print( a, b );                # output: 4 2
```

3.3.2. Reasoning

3.3.2.1. Graph and Table Equivalence

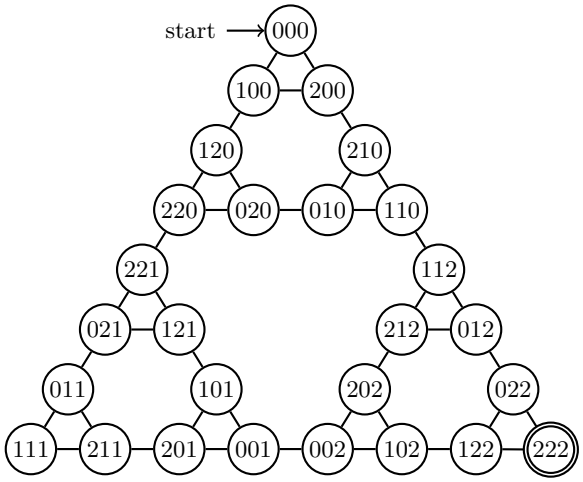
So far reasoning has been left out from previous considerations, although it has been mentioned to be one of the key features. SelectScript has been designed to be a declarative extension to other programming languages, but reasoning is not that well supported. Although queries such as the following can be easily resolved with the help of SQL and a large "table" that contains all possible combinations:

- What area is monitored at most by external sensor systems?
- Which areas are less frequently traversed by mobile robotic platforms?
- What combination of sensors offers the best fusion quality or validity?
- Which group of n robots is closest to a certain point?
- ...?

As it was already described in [11] and as depicted in Fig. 3.8, most of the common search problems can be represented as either a graph/tree or as a table. Both contain all possible combinations ... search paths. It is obvious that a graph-like structure consumes less memory than a table, whereby a graph needs to be traversed every time in order to find an appropriate solution. A table that contains all "paths" can be explored more efficiently, by applying different caching or indexing strategies and filters in the background, e. g. by directly requesting the row with the last entry 222.

The table that is depicted in Fig. 3.8b was generated with the help of the SelectScript shown in Lis. 3.12. The script actually generates only one result, which is listed in line 21. The additional `IF` expression, the variables `final_tower` and `count` have only been introduced to generate logging output that shows all valid sequences of steps and the resulting tower configuration (in the (xxx) notation). (Note that invalid combinations result in an empty list and the functions `move` and `showTowers` are externally defined). The variable `moves` contains a list of all possible steps, which are aggregated in the `FROM` expression to a Cartesian product[4] and, thus, a very large table. `move` is applied recursively and is applied onto each (also invalid) combination of steps.

[4]The minimum number steps required to solve the Towers of Hanoi problem can be calculated with the formula $2^{disks} - 1$



(a) Connected undirected graph

0001	000	100	120	020	120	020	120	020
0002	000	100	120	020	120	020	120	100
0003	000	100	120	020	120	020	220	221
0004	000	100	120	020	120	020	220	020
....	...	...	...	...	...	...	...	...
0693	000	200	210	110	112	012	212	012
0694	000	200	210	110	112	012	212	112
0695	000	200	210	110	112	012	022	122
0696	000	200	210	110	112	012	022	222
0697	000	200	210	110	112	012	022	012
....	...	...	...	...	...	...	...	...
1084	000	200	100	200	100	200	100	120
1085	000	200	100	200	100	200	100	000
1086	000	200	100	200	100	200	100	200

(b) Table of possible paths

Figure 3.8.: The Tower of Hanoi problem with three disks. Both representations can be used to identify a valid path and thus the intermediate steps to get from configuration 000 to 222. The numbers mark the positions of the disks (tower) and their order is used to mark their size (starting with the smallest).

Listing 3.12: Solving the Towers of Hanoi with SelectScript – vanilla approach

```
moves  = [[0,1], [0,2], [1,0], [1,2], [2,0], [2,1]];
count = 0;

SELECT [m1.this,  m2.this,  m3.this,  m4.this,  m5.this,  m6.this,  m7.this]
  FROM m1=moves, m2=moves, m3=moves, m4=moves, m5=moves, m6=moves, m7=moves
 WHERE [[],[],[3,2,1]] == ( final_tower = move(m7.this,
                                               move(m6.this,
                                                 move(m5.this,
                                                   move(m4.this,
                                                     move(m3.this,
                                                       move(m2.this,
                                                         move(m1.this,
                                                           [[3,2,1],[],[]])))))))
      ) AND IF ( final_tower != [];
                   print(count = count+1,
                         showTowers([[3,2,1],[],[]],
                                    [m1.this, m2.this, m3.this, m4.this,
                                              m5.this, m6.this, m7.this])),
                   True )
    AS list;
# result: [[[0, 2], [0, 1], [2, 1], [0, 2], [1, 0], [1, 2], [0, 2]]]
```

The table and graph/tree equivalence is actually nothing new as well as trying to identify a solution to the Towers of Hanoi problem with the method that was presented here. It is furthermore not very efficient and requires way too many instructions. But as described after the next section, with some slight adaptations, the database metaphor can be applied more efficiently onto common search and reasoning problems. It furthermore offers a method to utilize different search strategies and methods for optimization, without the need to change the problem description. But before that, we need to clarify why declarative reasoning is so important.

3.3.2.2. The Declarative Paradigm & Future Requirements

A short essay on the distinction of programming paradigms can be found in the Appendix on page 139, in which different paradigms are related to the ways that we use to describe problems in order to get a solution. And for a couple of problems it is more convenient to formulate them declaratively. But if we look at the mostly applied languages in ROS [7] or the used paradigms in the overview that has been presented in Chap. 2, problems are mostly solved in the imperative way to some extent. Imperative programming has proved to be perfect for lots of applications, but as also criticized, for example, in (cf. [241]) it is simply wrong to apply dogmatically the same old "principles" onto new environments and problems such as in CPS. Due to their complexity and the dynamics, not all possible combinations of available sensor systems, transformations/services, required world models, etc. can be foreseen and thus hard wired at design-time. Or as discussed in [11], ROS offers an enormous amount of functionality, but how is it used? We search for the ideal combination of nodes and modules and try to connect them in the correct order.

Although the application of reasoning methods to solve different problems (also in robotics and CPS, by utilizing different DSLs) might not be that new, it is new for the area of appropriate environment perception. Thus, identifying the appropriate set of sensor systems and the appropriate sequence of transformations (services) to apply is actually quite similar to reasoning about action sequences in robotics (cf. Sec. 2.3.4.1.11). Sarcastically speaking, we are still programming in assembly, but of course on a higher abstraction. Thus, trying to tackle some problems in CPS in a declarative way is actually more beneficial and there have already been first attempts to go in this direction.

The Robot Perception Architecture (RPA) developed by Nico Hochgeschwender (cf. [96, 95, 94]) is a declarative approach in which parts of a robots perception architecture, as depicted in Fig. 3.9, are defined as explicit components that can be reconfigured, modified, and validated. Based on the task requirements (left hand side results in Fig. 3.9) that are expressed in the Robot Perception Specification Language (RPSL) the most appropriate perception graph from a set of predefined perception graphs can be identified and selected at runtime. This is accomplished with the help of a linear "pattern-based" search that is applied in background.

In contrast to this, the concept that has been presented so far in this thesis provides, at first, a way to identify and access available data and systems within a loosely coupled smart environment (not on a single robot), which secondly transforms it into an intermediate representation, and thirdly SelectScript offers a method to express the required type of information, etc. What is left out, however, is a methodology that allows defining reasoning tasks. But in contrast to most systems (also RPSL), where one strategy is applied only, the intended method for SelectScript, which is described within the next part, allows to choose between different strategies and to apply optimizations.

3.3.2.3. Search Programming with Hierarchical Queries

The main idea that was presented in [11] dealt with a language extension that enabled reasoning on the basis of hierarchical queries. Reasoning as it is commonly supported by Prolog applies only one algorithm, namely backtracking, which is a simple depth-first search. In contrast to this and other related approaches, the implemented system allows to apply different algorithms as well as to tweak them as desired, without (or only marginally) changing the problem definition at all.

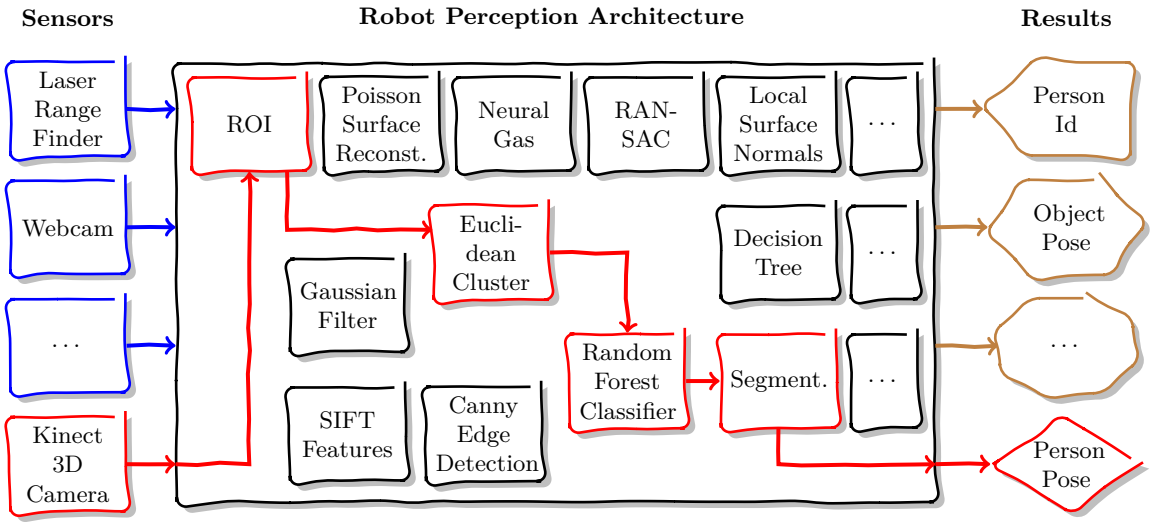


Figure 3.9.: Robot Perception Architecture as in [95], consists of heterogeneous sensor systems, processing components, and output information. The task-relevant perception graph is identified at runtime.

A recursive or hierarchical query in SQL denotes a special type of query, which is used to handle hierarchical structured data. A common example is a company database, which stores all employees' data as well as the ID of the direct supervisor. Thus, reconstructing the leadership structure requires additional recursive capabilities. The Lis. 3.13 depicts an alternative, not standardized, syntax for recursive queries that was implemented in SelectScript. It adopts the `START WITH ... CONNECT BY` construct that was originally introduced by Oracle [173]. The SQL standard [69] defines a syntax that involves the keywords `WITH RECURSIVE` and requires to associate query expressions with a name, which allows to reuse them. However, since we are not dealing with tables of finite length, but rather with functions that might generate endless results, an additional `STOP WITH` construct was introduced, used to define abort conditions. The `START WITH` expression is used, as the name suggests, to define local variables that might be required as well as initial conditions. The presented example in Lis. 3.14 furthermore depicts that the program structure is more compact compared to Lis. 3.12 and allows the generation of results more efficiently (a benchmark is provided in [11]).

Listing 3.13: SelectScript extensions to support hierarchical queries

```
45              ("START" "WITH" <PARAMS>)?
46               "CONNECT" "BY" <X_OP> <PARAMS>
47               "STOP"  "WITH" <STMT>
48  #-------------------------------------------------------------------------------#
49  # additional parameters to optimize or change the applied search strategy       #
50    <X_OP> ::= ("NO" "CYCLE")?          # prevents cycles                         #
51               "UNIQUE"?                # allows only unique select results       #
52               ("MEMORIZE" <STMT>)?     # generates a graph with a certain length #
53               ("COST" <STMT>)?         # cost function (heuristic search)        #
```

All of the work is actually done with the help of the central `CONNECT BY` expressions. It is used to denote which values within the search are affected and how they are changed from iteration to iteration. The listed language elements, namely `SELECT`, `FROM`, `WHERE`, `START`

`WITH`, `CONNECT BY`, and `STOP WITH`, are sufficient to describe the entire search problem but, as previously mentioned, also the possibility to utilize different search strategies was included.

The selection of search and optimization strategies can be manually performed by using the accompanying keywords in the `CONNECT BY` expression. The method that is applied by default is a depth-first search algorithm, which actually generates the same solutions, as they were presented in Fig. 3.8b. This strategy generates also cycles, which might be desired in some situations, but preventing them and therefore reducing the search space, the `NO CYCLE` keyword can also be added to the query. The application of the keyword `UNIQUE` further reduces the space of possible paths, by allowing a node within the search to be traversed only once. The application of `MEMORIZE` automatically generates a connected graph at first, which for the problem definition in Lis. 3.14 is actually equal to the one in Fig. 3.8. This graph can be subsequently traversed with different search strategies, to which a bidirectional search algorithm is applied by default (cf. Sec. 4.3.4). This reduces the search space by half trading memory consumption and prevents multiple visits of a node, but it requires an additional stop parameter that determines the maximal length of the resulting paths. This maximal path length is defined by the associated value. For optimization purposes and to enable heuristic searches, a cost expression was introduced.

Listing 3.14: Solving the Towers of Hanoi with SelectScript — hierarchical query

```
1  moves = [[0,1], [0,2], [1,0], [1,2], [2,0], [2,1]];
2
3      SELECT to(this, "move "+str(count)), showTowers(move(this, tower))
4        FROM moves
5       WHERE [[],[],[5,4,3,2,1]] == move(this, tower)
6  START WITH tower = [[5,4,3,2,1],[],[]], count=1
7  CONNECT BY MEMORIZE 31
8             tower = move(this, tower), count=count+1
9   STOP WITH [] == tower;
10 # result:  [[[{'move 1': [0, 2], 'showTowers': '20000'}],
11 #            [{'move 2': [0, 1], 'showTowers': '21000'}],
12 #            [{'move 3': [2, 1], 'showTowers': '11000'}],
13 #            ....
39 #            [{'move 30': [1, 2], 'showTowers': '02222'}],
40 #            [{'move 31': [0, 2], 'showTowers': '22222'}]]]
```

In the screencast listed below, it is further demonstrated that the same techniques can also be applied to reason about trajectories for a mobile robotic platform within a 3D environment: `https://www.youtube.com/watch?v=EFRV0JSdK3M`

3.3.3. Discussion

The presented concept for the query language SelectScript adopts elements of SQL and adds new features to it. These features cope with the ability of requesting different types of representation, dealing with temporal aspects, the interaction with and the extendibility by common programming languages (services) and their functionality. As it was described in the last part, it also offers a method to define reasoning problems. This feature in combination with the application of procedures, which are used for code substitution, even allows it to reason about the appropriate sequence of applying filter and transformations functions. Hence, the language itself provides a basic syntax for machines and humans to express their desires in CPS. Since the language itself is intended to be an embedded language, it can also be applied to extend their host programming language with declarative aspects.

3.4. Summary & Discussion

The virtual overlay network described in the first part of the concept provides abstract repository up to layer 4, according to the conceptual organization that was introduced in Sec. 2. It provides links to source layer descriptions (metadata), the data and information layer (raw & virtual data) as well as to different types of knowledge representations formats. The complex container thereby defines a hierarchical organization that structures these different elements according to space and time (and producer) in a distributed scene-graph and allows the attachment of additional context information. This structure can be easily queried and traversed, as it is described within the associated implementation section (Sec. 4.1). It can be composed of distributed databases, whereby not every entity is forced to upload all of its data to a cloud. But instead, every entity hosts its own local database for its own purpose and updates its own local data accordingly. The applied cloud-based infrastructure is used to handle the access to all local repositories.

It thus defines the basis for the second part, in which elements from the distributed scene-graph are translated into a 3D rigid-body simulation. This rigid-body simulation provides a more precise replica of the environment for a certain area. It can be updated in real-time with real-time data in order to mimic the behavior of real entities within the virtual world. All related information can be associated to the virtual elements as it is further described within the implementational description in Sec. 4.2. The local environmental model is intended to be used for further analyses and abstractions. It can be either created on purpose for single analysis or used as an application's central world model for continuous analyses.

The main problem here lies in the diversity of facts to extract and abstractions to generate from the local model. To deal with these issues a new type of query language was introduced, which allows defining in a declarative manner what kind of information or what kind of environmental abstraction (map) is required. The syntax of the language was inspired by SQL, but with adaptations that allow the query of simulation environments online and request for different formats (accomplished by the keyword `AS`). It also includes basic scripting capabilities, allows to deal with temporal issues and to define reasoning problems. As it is further described in Sec. 4.3, the implementation of the language and its interpreter are easy to extend, further allowing to adapt it onto new simulation environments and to define new functionalities and new representation formats. As later presented in Chap. 5, the language itself can also be used to cover all parts that deal with the generation of the local environment model and, thus, also with the access to the virtual overlay database. Another benefit of the developed declarative language concept is that it can also be applied as some kind of "lingua franca" between systems with different world models since the language is interpreted and can be applied onto different simulation environments. The only thing required is a common standard for the naming of request formats, functions and filters.

The described concept leaves it free where a query is answered and thus also where the model is created. It can happen locally, but also externally on purpose, in such a way that the response to a query can be generated similar to the delivery of a service. It thus enables simple and "low performance" smart systems to request for a map, some information, etc., but without the need to spend computational power (locally). The only requirement is to locally adapt the string queries.

Another benefit of the described three way concept lies in the reduction of efforts for developing transformation functions. As it is sketched out in Fig. 3.10, only "adaptors" to

the local environment model as well as transformations from the local model to the desired information/abstraction have to be defined. It thus requires only to define $n+m$ adaptors instead of maximum $n! * m$, to be able to react onto all combinations.

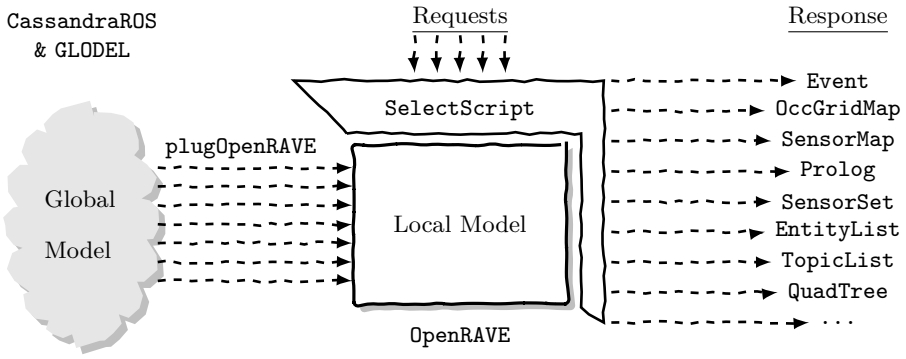


Figure 3.10.: Composed view on the entire system implementation

To conclude with the initially defined frog problem:

1. **Lack of Intelligence:** None of the presented steps requires any form of intelligence; everything can be accomplished with simple recursive algorithms/structures and if more elaborate methods for problem solving are needed, the developed language concepts additionally allow to define reasoning problems and to apply different search and optimization strategies.

2. **Access to Information . . . :** The diversity of data, information, systems, knowledge representations, etc. becomes manageable and accessible due to the application of a local environment model, into which all relevant elements from the underlying virtual overlay database have to be translated.

3. **. . . and Memory Externalization:** The virtual overlay database provides links to all measurement data, data-sheets, etc., within a smart environment, which enables the access to any kind of data on purpose.

4. **Reconfiguration and Representation:** Models can be constructed and reconstructed dynamically and by applying callback mechanisms — it is also possible to react on occurred changes. Here, situations can be defined with the help of the developed language concepts.

4. Implementation

> "If you can't beat them, join them."
>
> —unknown

The following chapter is structured according to the previous one. Every section is used to describe the developed application or library in detail. Thereby, within the first two sections mainly existing systems were applied and adapted, in contrast to the last one where a declarative programming language was adopted and introduced to a new field. All of the developed systems are available under BSD-license and can be downloaded from the following repository (among other things): `https://gitlab.com/groups/OvGU-ESS`

Because most systems were developed either with the help of ROS or for ROS, there are also tutorials available at the ROS website, and I will refer to them if necessary.

4.1. Organization

The following section is divided into three parts. The second one herein deals with the application of Cassandra as a common data repository for any kind of ROS messages. Within the third part, the previously developed `cassandra_ros` library was applied and extended to form a virtual overlay database that links to any kind of data within a smart or intelligent environment. But to start with, a short introduction to Cassandra is given in order to clarify why especially this database system was chosen for the next step.

4.1.1. Why Cassandra?

As it was introduced in [12], Cassandra was initially developed as a so-called NoSQL database for Facebook [132] and received only little attention in the robotics community. It provides a distributed key-value store. Thereby, keys map to rows, which can contain a multitude of columns (values), while rows by themselves are stored in column-families (similar to tables). Further columns can be added dynamically or removed at any time, so that different rows can store different types and different amounts of values (although they are, in principle, stored within one table). That is to say, the structure of the database can change over time and adapt to varying requirements, unlike a classical relational database system.

As described in Sec. 3.1.2, an entity can host and update its own Cassandra instance independently, but instances can also be grouped into larger clusters and keyspaces. Thus, by querying data on local instance, it also queries data from other connected entities that are within the same cluster. Different strategies for replication between instances provide a high availability of data with no single point of failure. An entity/robot/application can thus leave the cluster while its data remains on other nodes (if required). Every value is marked with a time-stamp, which allows defining Time To Live (TTL), so that data can be forgotten after a certain period of time. This is something that is also implemented

for the Distributed Robotic Scene-Graph (cf. Sec. 2.3.4.2.4). The usage of time-stamps enables eventual consistency (see also [215]), which is a weaker consistency level than strict or immediate consistency (commonly used in relational databases), while the first is actually more appropriate for distributed systems. Eventual consistency means that, due to the distribution of data and the availability of nodes, it is guaranteed to receive a valid value which, though, may not be the most recent one. Tunable consistency levels enable application dependent refinements.

Cassandra was also identified in [212] to be an ideal storage for sensor data. This was also underpinned by the results from the benchmark in [12] in terms of storage consumption, write and read performance, as well as query capabilities for arbitrary messages. For more information on applicable database systems in this area, see also the overview on the related work that was presented in [12].

4.1.2. A Holistic Data Store

This part describes the architecture, implementation details, and the general concepts behind the application of Cassandra as a holistic data store and, furthermore, how it is used to define queries on stored ROS messages in a way that it is not possible with the help of any other system.

The architecture was segregated into two basic parts; one is responsible for abstracting the database management, while the other enables data access in a ROS typical manner. That is, in contrast to handling data within a database, an application (or application developer) only has to cope with messages and topics. Because it was not possible to foresee all the desired purposes, the system and the interface were designed to be not only as generic as possible but also easily extendable. A simplified draft of the system architecture and its classes are given in Fig. 4.1.

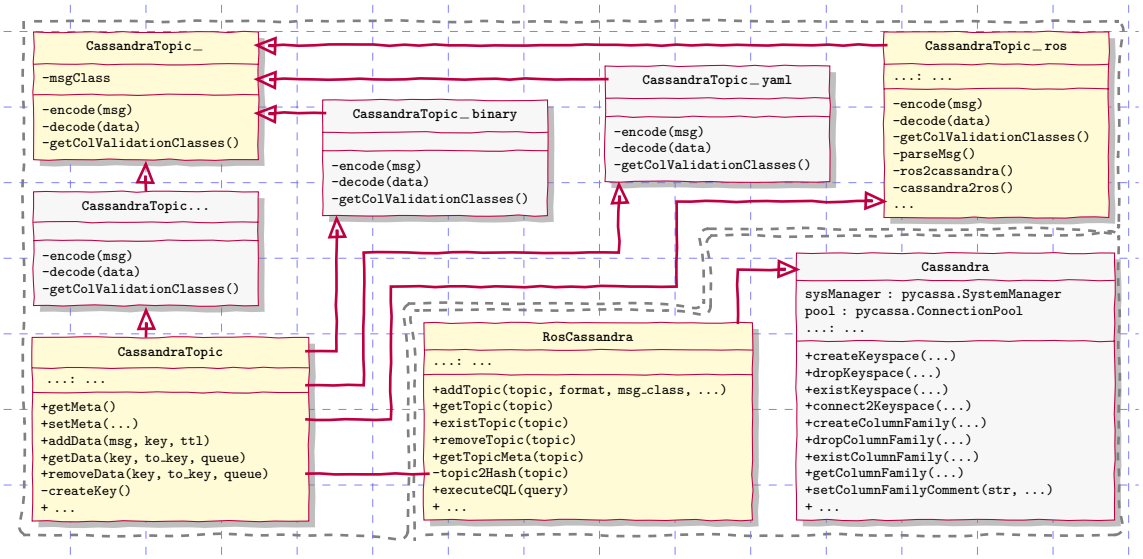


Figure 4.1.: Simplified UML class diagram of the cassandra-ros implementation, it is divided into a topic handling part (left) and database management part (right) classes.

The library is called cassandra-ros, it is implemented in Python and with the help of pycassa (a Cassandra client library). The project is freely available under the BSD-license and through the following link: `http://ros.org/wiki/cassandra_ros`

(Take also a look at the appertaining playlist of YouTube videos that demonstrate the capabilities of this library:

`https://www.youtube.com/playlist?list=PLgJeoIw_8oS55z_xczH_AbJCb48aZQIEU`

and presentation slides at:

`http://eos.cs.ovgu.de/wp-content/uploads/2015/06/sozi-presentation.svg`)

4.1.2.1. System Architecture

`RosCassandra` can be seen as the topic management system, abstracting and hiding all Cassandra-related stuff, with an interface designed to be as close as possible to the common ROS ideology. Thus, topic-containers are created and accessed, messages are stored and queried in a way similar to the ROS publish/subscribe API, without bothering about message conversation or database-related issues. An example is given in the following listing.

Listing 4.1: Minimal source code example of the `cassandra_ros` API

```
# init Cassandra connection, such an initialization allows to maintain
# multiple connection
rosCas = RosCassandra(host, port)
rosCas.connectToKeyspace(keyspace)
# create new topic-container for tf of type geometry_msgs/TransformStamped
rosCas.addTopic(topic='tf', format='ros', msg_package='geometry_msgs',
                msg_class='TransformStamped', key_format= ...)
casTopic = rosCas.getTopic('tf')
# adding messages the topic-container
casTopic.addData(msg)
...
# retrieving directly messages from the topic-container, without the need for
# conversation
msg = casTopic.getData(key)
```

Cassandra column-families are applied as topic-containers, storing one ROS message per row. `RosCassandra` therefore applies instances of class `CassandraTopic`, responsible for converting nested ROS messages into a "scalar" Cassandra format and vice versa; Messages can be stored in various formats, with different advantages and disadvantages regarding speed, size, or methods for requesting and filtering messages (described in more detail within the next sub-section).

Due to the fact that column-family names are limited to a maximum size of only 48 B, whereas ROS topic names easily exceed this limit, hashes of topic names are applied as column-family names. The real topic name as well as other metadata is stored during creation time of a `CassandraTopic` within the comment field of every column-family. This metadata is also used to describe the internal structure of messages and column-families; altering of values is therefore only allowed to a limited extent. Other metadata is, for example:

- `package` and `msg_class`, which define the format of a message,
- `key_format` specifies the primary key (e. g., a time-stamp, a hash value, etc.), but in some cases it is more appropriate to use parts of the message itself as primary keys (e. g., sequence numbers), which can be defined with the help of `key_msg_part`,
- `cassandra_format` defines the conversion format or, in other words, how messages are stored within Cassandra (binary, yaml, encoded ROS, etc.)
- other values include the creation time of topic-containers (`date`) or additional `comments`

Storing metadata within the comment field has several advantages compared to the usage of a separate column-family. It does not affect any column-family structure, it is easy to extend and interpret and it does not require any further replication or synchronization. Furthermore, this kind of information storage is used for topic identification, which allows topics to exist in parallel with other column-families within the same keyspace. Thus, querying for available topics or other metadata is done via retrieving and parsing all column family comment fields.

4.1.2.2. Translation of Messages

As mentioned before, this system allows the storage of messages in various formats, such as `string`, `json`, `ros`, or `binary`. Further formats can be added by implementing new classes that inherit methods from class `CassandraTopic_` and overwrite the three methods, `encode(msg)`, `decode(data)`, and `getColValidationClasses()`. The `ros` format is thereby the actual key feature that automatically translates a ROS message into a congruent Cassandra format, as it is depicted in listings below. The method `getColValidationClasses()` is thereby only called during the creation of new `CassandraTopic` (and thus, during the creation of a new column-family) and is used to define the format of every column. In some cases this is done statically, if the whole message is stored at once by using just one column, like in `binary`- (`BYTES_TYPE`) or in `string`-format (`UTF8_TYPE`). These formats are more appropriate for fast storing of raw data, such as video streams or laser scans, but it also diminishes the possibility for filtering and extended querying. In contrast to this, "virtual" data can be stored in the `ros` format. Therefore, the definition formats of ROS messages are parsed and translated from a nested message structure (see Lis. 4.2) into a list of columns (see Lis. 4.3). Furthermore, primitive ROS types (cf. [258]) are translated into Cassandra data types (cf. [237]) to define column validation classes. Lis. 4.3 presents the resulting translation for the message definition of `geometry_msgs/TransformStamped`, compared with the input in Lis. 4.2. The fact, that the original tree structure is used to define the column names and that column validation classes are generated automatically, allows to query or to filter messages with a similar syntax as it is used to access message objects from a programming language like C++ or Python. Column validation classes are a prerequisite in Cassandra for building secondary indexes on columns. It is therefore the basis for querying the data store afterwards with the CQL (as described shortly within the next sub-section). Class `CassandraTopic_ros` thus enables a full exploitation of Cassandra's querying methods on secondary indexes. Automatic packing and unpacking of messages is afterwards implemented within the methods `encode` and `decode`.

Cassandra supports a maximum column key (and row key) size of 64 kB and column values up to 2 GB (cf. [28]). These values are sufficient to store messages without applying extended conversion methods as they were used to rename topics.

As depicted in Fig. 4.1, the parents of `CassandraTopic` define the storage format, in other words, how messages are translated into a format that can be stored in Cassandra. Inheriting from multiple classes may create problems, such as the "diamond problem"[1]. This problem could be solved by deleting not required parents during the initialization phase. Python as a metaprogramming language allows to change and reorder used base classes. Thus, from a formal point of view, class `Cassandra-Topic` inherits features from only one base class (depending on the conversion format).

[1] Ambiguity problem, raised if a specific feature is implemented in more than one superclass (cf. [149]).

Listing 4.2: Example of a ROS nested message definition

```
1  std_msgs/Header header
2     uint32 seq
3     time stamp
4     string frame_id
5  string child_frame_id
6  geometry_msgs/Transform transform
7     geometry_msgs/Vector3 translation
8        float64 x
9        float64 y
10       float64 z
11    ...
```

Listing 4.3: ROS encoded version into Cassandra columns

```
header.seq: INT_TYPE
header.stamp: DATE_TYPE
header.frame_id: UTF8_TYPE
child_frame_id: UTF8_TYPE
transform.translation.x: DOUBLE_TYPE
transform.translation.y: DOUBLE_TYPE
transform.translation.z: DOUBLE_TYPE
transform.rotation.x: DOUBLE_TYPE
transform.rotation.y: DOUBLE_TYPE
transform.rotation.z: DOUBLE_TYPE
transform.rotation.w: DOUBLE_TYPE
```

The translation of arrays in the `ros` format is not fully resolved yet, if an array is not of primitive type. One solution to this problem might be the decomposition of complex array elements into super columns, but this would only work for one iteration only (further nesting is not allowed). Another problem in using super columns is that a super column cannot be indexed (cf. [28]) and thus, the intended querying is not supported. Furthermore, super columns are not officially deprecated, nor are they recommended for usage. Another solution might be the usage of composite types, as described in [28], but this would require totally different conversion methods. Arrays within arrays would cause the generation of huge composite columns, hard to maintain and hard to query, and with increased storage consumption. Thus, the `binary` format is more appropriate in such cases.

In order to cope with complex arrays and to preserve the column structure of `CassandraTopic_ros`, the array information is included into column names, by using square brackets like "`obj[0].val.a`". In fact, not every value of a message might be equally important and thus, there is possibly no need to decompose every message in such detail. I therefore started to implement another encoding method, which requires a little user interaction. The encoding class simply inherits from all other encoding classes and by defining masks for every message, it should also be possible to circumvent the "diamond problem" and use different encoding formats for different parts of a message.

4.1.2.3. Accessing and Querying

As presented in Lis. 4.1, there are two "simple" methods for directly storing and accessing messages, these are `addData` (line 10) and `getData` (line 14) that are defined in class `CassandraTopic` (see also Fig. 4.1). Method `addData` allows addition of a primary key and a TTL to every message. If these optional parameters are left out, then the default values are used, which were defined at the creation of the topic-container, see line 6 in Lis. 4.1. Requesting single messages or a stream of messages requires one to know the primary keys, or a start and an end key. As already mentioned, in most cases these methods are sufficient especially when dealing with large amounts of raw data, which is stored in a `binary` format. But when dealing with virtual data, with queries that cover multiple topics (column-families), or with applying filters, a method more dedicated than just key-value requests is required.

As already introduced, it is also possible to apply Cassandra's query language CQL to enable complex queries over multiple of topics that are stored within the `ros`-format. But while column names are equal to ROS message attribute names, column-family names and topic names differ, due to the usage of hash values (as described in Sec. 4.1.2.1). As listed below, to deal with this issue class `RosCassandra` offers a method `exequteCQL` that gets

a string (full CQL statement) as input, translates topic names into their column-family names, and executes the query, by using Cassandra's Python driver for CQL. The example below is taken from the evaluation scenario that was presented in [12]. It is part of a longer demonstration in which data gathered by mobile robot was analyzed afterwards not only to identify the area where a box with markers was placed, but also to reconstruct a map for this specific area based on additional sensor measurements taken in the vicinity. `t1` and `t2` denote the time of the first and last appearance of the box within a previously analyzed camera stream, which are used below to identify the location of the robot (`nodeID`) during that time. The previously mentioned YouTube playlist in Sec. 4.1.2 is actually a replay of all intermediate steps of the evaluation presented in [12].

Listing 4.4: `cassandra_ros` CQL example

```
stmt = "SELECT ... translation.x  FROM tf " + \
       "WHERE  ... child_frame_id=" + nodeID + \
             " and KEY > " + str(t1-5) + " and KEY < " + str(t2+5) ...

x_list = rosCas.exequteCQL(stmt)
x_min = min(x_list)[0]; x_max = max(x_list)[0]
```

4.1.3. A Virtual Overlay Database

This part is used to describe glodel the implementation of the concept that was developed in Sec. 3.1.2 and published in [10]. It defines the general infrastructure to store, access, and organize data in a distributed overlay database. To get a first impression on the capabilities of the system, take a look at the accompanying YouTube screencasts that were used to demonstrate the capabilities of the software and the concept behind:

`https://www.youtube.com/playlist?list=PLgJeoIw_8oS5IF9MixOYv1zEzKLE4cCRN`

4.1.3.1. Implementation of a GLObal and distributed world moDEL

As depicted in Fig. 4.2, the project mirrors the described database structure in Sec. 3.1.2. It is furthermore based on the previously presented cassandra-ros project and extends class `Cassadra` by implementing a specific database interface. This specific (and general) `Base` class is used to derive the specialized classes `BaseRobot`, `BaseSensor`, `BaseObject` and `BaseLocation`, which are used to access the associated columns. `BaseComplex` is used in two ways, to access column `Complex`, but it is also as a central interface that instantiates the other helper classes.

As described in the concept, all data is stored linearly in columns families. But the benefit of applying Cassandra as the primary data store is that the internal structure of columns does not have to be predefined. As the excerpt of column-family objects in Lis. 4.5 shows, each row can have a different set of columns. They can be static, such as all rows contain a comment column (describing an object in more detail in a UTF-8 encoded string), but they can also be dynamic, such as the columns "`wrl`" or "`pcd`" in row `isx34s`, which are furthermore also stored in different formats (i. e., binary and string). These columns, for example, contain a representation of a mug in different formats and qualities, so that the "application" can chose the most appropriate format and also the required quality. The quality is associated with the number within the column name, which is at the moment a manually defined value.

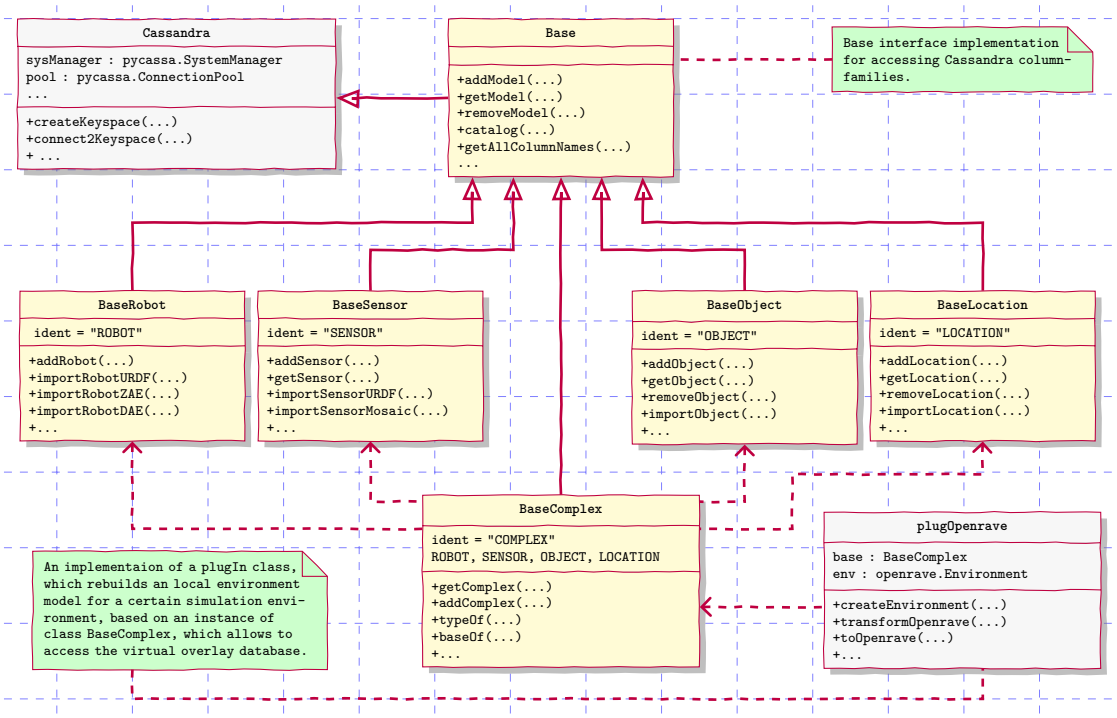


Figure 4.2.: Simplified UML class diagram of the implementation of glodel

Listing 4.5: Extract of column-family objects, row keys (bold), columns (blue), and values in different formats

```
1   objects : {
2     ...
116   1sx34s : {
117     comment : "mug ... ",
118     stl_85 :  00000000  42 69 6e 61 72 79 20 53  54 4c 20 6f 75 74 70 75
119               00000010  74 20 66 72 6f 6d 20 42  6c 65 6e 64 65 72 3a 20
120               00000020  43 3a 5c 55 73 65 72 73  5c 61 64 69 65 74 72 69
121               00000030  63 68 5c 44 6f 63 75 6d  65 6e 74 73 5c 42 6c 65
122               ...
362     wrl_90 :  #VRML V2.0 utf8
363               Transform {
364               translation -1067.343 -1881.088 14.086
365               scale 0.572 0.572 0.572
366               children [DEF GRP_Group1 Group {
367               children [
368               Shape {
369               ...
483   6yfr48 : {
484     comment : "standard phd student table ... ",
485     pcd_85 :  00000000  23 20 2e 50 43 44 20 76  30 2e 37 20 2d 20 50 6f
486               00000010  69 6e 74 20 43 6c 6f 75  64 20 44 61 74 61 20 66
487               00000020  69 6c 65 20 66 6f 72 6d  61 74 0a 56 45 52 53 49
488               00000030  4f 4e 20 30 2e 37 0a 46  49 45 4c 44 53 20 78 20
489               ...
```

Locations are stored in a similar way and in different formats, which is equal for robot descriptions (stored in URDF, COLLADA, or in an OpenRAVE description format). Sensors are currently described only in an OpenRAVE format, but others are also possible. Keep in mind that these are only containers that store basic description, data-sheets, and abstract data for a certain type or class of robot or sensor, they are not used to describe

a certain entity. Context data is associated with the help of the column-family "complex" (see therefore also Lis. 4.6). As it was described in Sec. 3.1.2, this container is used to put all data into a general context. A complex entry thus always represents a certain real object, a single robot, sensor, or a desk, which is present within the environment. As showed in Lis. 4.6, every complex entry has a key and a type, which are used as references to the general description of a robot, a sensor, objects and locations as well as to further complex entries (simple placeholders). As further shown in Lis. 4.6, every complex entity has a certain position, orientation, which might be affected by uncertainties. These spatial relations are defined relative to a base, such as the robot's position is relative to a room, the room's position and orientation are relative to a floor, while the floor's position is relative to a building, etc. Fig. 4.3 depicts this relative positioning, starting with some sensors that are attached to a Katana robot and, thus, are relative to the robot's position. The "base" entry is thus the central element for organization, allowing the storage of complex entities hierarchically.

Listing 4.6: Showing the complex entry `katana_62x` of type robot, whose description is stored in row `katana_450` in column-family robots, whereby position and orientation are defined relative to the complex base entry `room_309`

```
1   complex : {
2     ...
256   katana_62x : {
257     key : "katana_450",
258     type : "robot",
259     base : "309",
260     position : [2.329, 3.232, 1.15],
261     quaternion : [0.926, 0.0, 0.0, 32.2],
262     covariance : [0.04234, 0.25597, 0.0112287, 0.1433849, 0... ],
263     ros-master : "http://moritz:11311",
264     topics : ... },
518   309 : {
519     key : "309",
520     type : "location",
521     base : "4_floor",
522     ... },
```

Lis. 4.7 within the next section demonstrates briefly how the central class `BaseComplex` can be used to query the environment and how the results can be translated into an OpenRAVE rigid-body simulation (see therefore also Fig. 4.3).

The entire database and "complex" entries thus only represent a snapshot of the global environmental configuration, a graph similar to the idea of the distributed robotic scene-graph. Hence, an application is responsible for updating its personal data, such as position, orientation, etc. as well as its performed changes on the environment, such as newly placed objects (cargo, mug, etc.) or newly discovered objects (within a video frame or a Kinect scan) within this database structure. Everything else, like maintaining consistency, redundancy issues, distributing queries, aggregating and transmitting results, is left out to the cloud database system Cassandra.

But it does not make sense and would be far too slow, if column-family complex would be queried continuously to identify rapid changes within the environment. A robot's position, its configuration, or a sensor is thus also associated with real-time data, for example, a ROS-master, different topics, and services. Of course it can also be any other communication paradigm. As long as a link is provided, it should be possible to establish a connection and interpret the required data within a local environment model. The connection to raw and virtual data stored within Cassandra topic-containers is established in the same way,

associated with a complex entity, and thus with a certain sensor, robot or, if required, also a location. How local environment models are created on the basis of this overlay structure and connected to real-time data is described within the next section.

4.2. Idealization

The previously described glodel systems already possess an interface for reconstructing models for OpenRAVE. As depicted in Fig. 4.2 an additional class named `plugOpenrave` is responsible for gathering all the required data from the cloud and for transforming it into an OpenRAVE simulation. Other "plugs" for other simulation environments can be developed similarly.

See also the screencast that was uploaded to YouTube to get a first impression, it demonstrates how the virtual overlay structure is accessed and queried in order to generate local world models for OpenRAVE: `https://www.youtube.com/watch?v=kvoC5yxdzsw`

4.2.1. Why OpenRAVE?

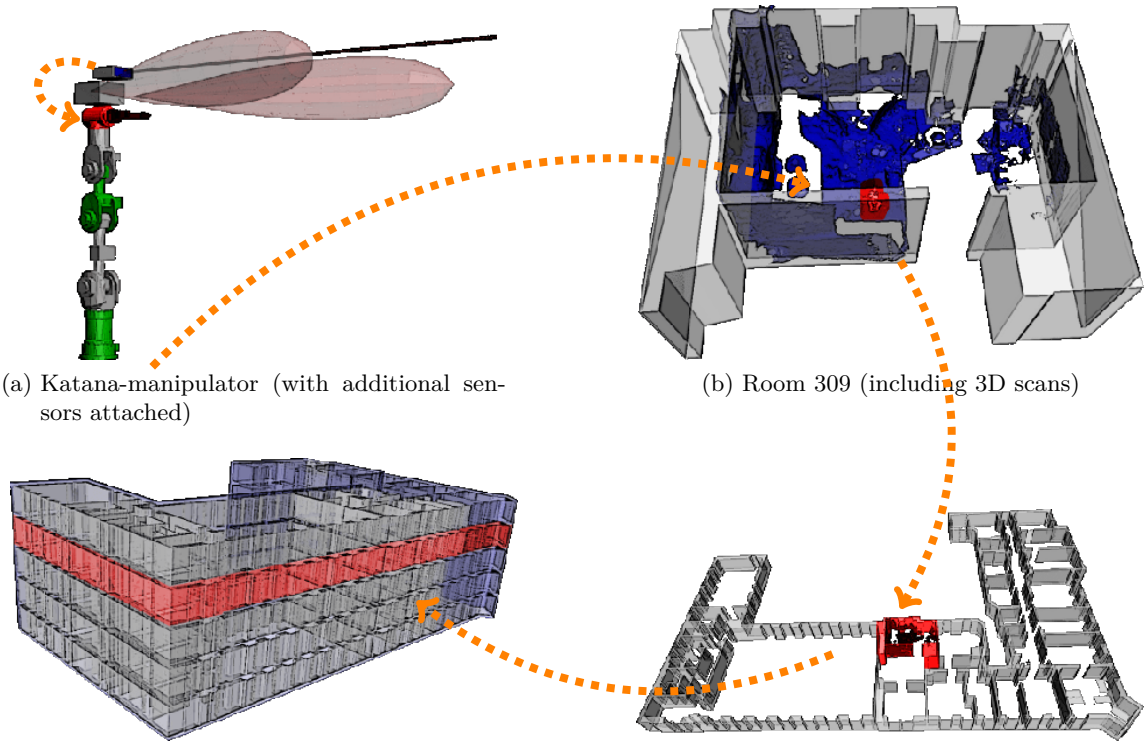
The utilization of this simulator, which stands for OPEN Robotics Automation Virtual Environment, for local world modeling was initially described in [4] for a couple of different application scenarios. As already mentioned, OpenRAVE is an OpenSource rigid-body simulation environment (among others) for robotic applications, which is well-known and possesses a large scientific community. Its main focus lies on simulation and analysis of kinematic and geometric information for testing, developing and deploying motion planning algorithms [62]. It was developed in conjunction the IKFast library [253] that analytically solves robot inverse kinematics equations, based on the robotic model, and generates optimized C++ code for various types of kinematics. But in addition to the kinematic simulation capabilities, it also provides a realistic simulation environment for various different types of sensor systems and also allows the utilization of different physics engines in the background. The modular architecture allows to integrate and develop new functionality (e. g., sensors, planners, controllers, etc.) that can be easily attached to the simulation environment. Different kinds of plugins were thus developed as extension to OpenRAVE for sensing, transforming, establishing a connection to ROS, and querying. The most important ones are briefly described within the following sub-sections. All simulated objects, robots, and sensors can be described with the help of an XML notation that can be further extended (see Sec. 4.3.3). Last but not least, it has native support for the Python programming language and C++. And the supported client-server protocol allows it to easily build interfaces to further programming languages (e. g., Matlab/Octave).

4.2.2. `plugOpenrave`

As depicted in Fig. 4.3 and previously described in Sec. 3.2.1, local environment models are represented as physics rigid-body simulation. All of the sub-figures in Fig. 4.3 show screenshots of OpenRAVE simulations that were generated by querying the virtual overlay structure at different levels. The responsible functionality as well as the connecting interface is implemented in class `plugOpenrave`, which is depicted in the UML diagram in Fig. 4.2. Other classes for other simulation environments can be implemented similarly by utilizing the interface of class `BaseComplex`.

Everything else that is required is a bootstrap, such as the Katana's identifier, a recognized marker at a wall, a room number, etc. All of these explicit entities are represented as complex entries, which makes it possible to query the database for them: for example the Katana manipulator with the id `katana_62x` represents the base for three additional sensor systems, with no further complex entries that have the explicit sensor identifiers defined as their base. Hence, the "downward" query ends. But it is also possible to query "upwards" for the entry that defines the base of the Katana, and so forth. Thus, such a hierarchy can be pretty easily traversed and, in fact, this is what happens within `plugOpenrave` — complex elements are queried recursively (to a predefined depth) and transformations are applied accordingly.

The following code excerpt in Lis. 4.7 shows how local models can be generated in principle with a few lines of code and how the environment can be queried for further structural properties.



(a) Katana-manipulator (with additional sensors attached)

(b) Room 309 (including 3D scans)

(c) Building (global base)

(d) 4th floor (assembly of multiple rooms)

Figure 4.3.: Hierarchy of entities, starting from the sensors whose positions are relative to the robot, while the robot is located within room 309, which is part of the fourth floor in building 29. The appertaining queries are shown in Lis. 4.7.

4.2.2.1. Integrating Robots

In general, robot definitions include a geometric and kinematic description, a visual representation, and probably a simplified collision model. There are multiple description formats

Listing 4.7: Exemplary utilization of `BaseComplex` and `plugOpenrave`, the results are depicted in Fig. 4.3

```
# load and prepare OpenRAVE
import openravepy as rave  # ...
env = rave.Environment()   # create a new and empty environment
env.SetViewer('qtcoin')    # open viewer

# load the GLObal environment moDEL library
import roslib; roslib.load_manifest("glodel") # ...
from baseComplex import *                     # with the complex interface
from plugOpenrave import plugOpenrave         # and OpenRAVE sim. generator

# initialize database connection
base = BaseComplex(host="localhost", port=9160, keyspace="Model", debug=F)
plug = plugOpenrave(base) # as well as an OpenRAVE environment generator ...

# id of the Katana robot used as bootstrap to identify its type and base
print base.typeOf("katana_62x") # result -> robot
print base.baseOf("katana_62x") # result -> 309

# generate the local environment model for katana_62x (search depth = 10)
env = plug.createEnvironment("katana_62x", 10, env)  # cf. Fig. 4.3a

# generate the local environment model of room 309 (search depth = 10)
print base.typeOf("309") # result -> location
print base.baseOf("309") # result -> 4_floor
env = plug.createEnvironment("309", 10, env)         # cf. Fig. 4.3b

# generate the local environment of the fourth floor as well as of the entire
# building (as the global base)
env = plug.createEnvironment("4_floor", 10, env)     # cf. Fig. 4.3d
env = plug.createEnvironment("building_27", 10, env) # cf. Fig. 4.3c
...
```

that allow defining a robotic system in these terms, such as URDF or COLLADA[2], OpenRAVE possesses its own XML definition format for manipulators [244], but it is also possible to load COLLADA descriptions or to translate between them into the required format.

To be able to replicate a robot's behavior, also a piece of code is required, which translates incoming robot's status information and control messages (in this case ROS topics, see Sec. 2.3.2.3) into transitions and rotations of links, wheels, the robot itself, etc. Of course, there are multiple ways to accomplish these translations, but this can be easily solved with the help of functions defined within a programming language capable of reflection. Reflection means that a program written in a certain language is able to modify its structure and behavior at runtime. Examples are Python, LUA, LISP, or even Java, which are able to load and execute additional pieces of source code. Thus, for every robot that is simulated a tiny Python script is loaded, which subscribes in the background to relevant data and which performs the required transformation of the robot model.

4.2.2.2. Integrating Sensors

The integration of a virtual sensor system is a bit more difficult, since their behavior cannot be replicated by simply changing their position and orientation relative to a base. A surveillance camera, for example, has to be modeled as a "robot" to which a sensor is attached. Thus, robots and actors are always applied to perform any kind of movement (translational or rotatory), while sensors are applied to measure a certain physical quantity (within the

[2] An interchange XML-based file format for interactive 3D applications [63].

real and the virtual environment). To provide measurements on both sides, I provide an OpenRAVE plugin that is called the situated-sensor, which is freely available for download, and can be easily extended to support different sensor systems. The general approach is depicted in Fig. 4.4, and class `SituatedSensor` offers a generic interface to the ROS communication framework, which has to be coupled with a virtual sensor implementation.

The `DistanceSensor` class in Fig. 4.4 is part of another OpenRAVE plugin that was developed to model and measure with realistic sensor beams for range sensors. For example, in Lis. 4.8 an infrared distance sensor is defined that utilizes a conical beam that is defined in line 6. Other shapes can be easily generated and included to replicate different types of ultra-sonic or radar sensors as well. Thereby, the shape of the beam is not only used to visualize the measurement, but it is also applied to perform the distance measurements with the simulated environment. See therefore also the plugin demonstration at:

`https://www.youtube.com/watch?v=Ts7Acf70D8U`

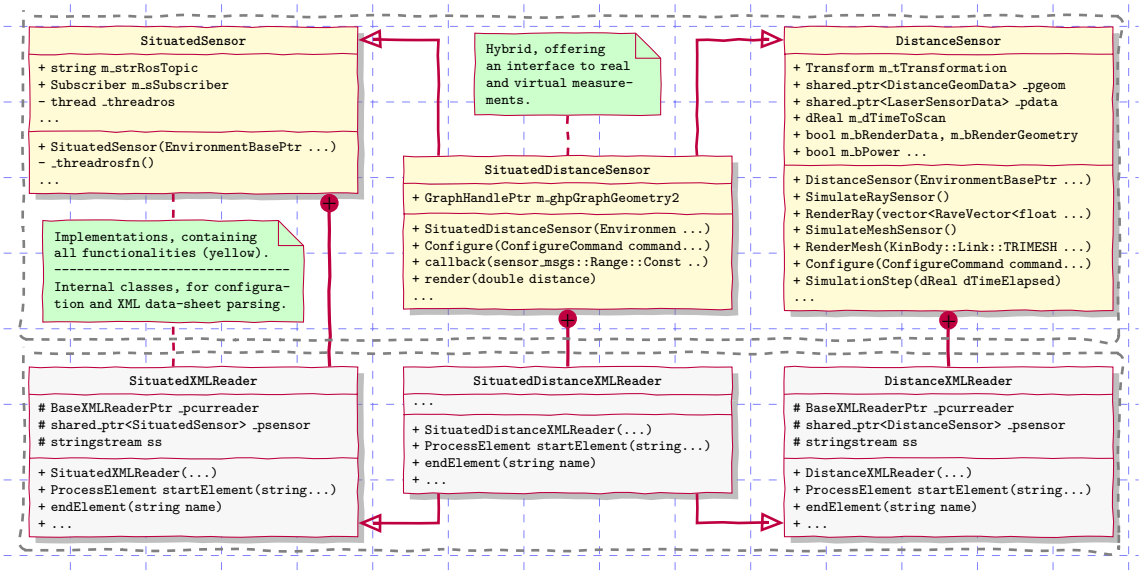


Figure 4.4.: Simplified UML class diagram of the situated-sensor implementation for a sensor of type distance-sensor.

The resulting offspring of both classes, in this case `SituatedDistanceSensor`, is able to measure distances within the virtual world as well as to subscribe for real world ROS messages of type "`sensor_msgs/Range`". The original virtual distance sensor is then applied two times: to measure and visualize the virtual distances and also to visualize real measurements (overlaid, but in another color, see also Lis. 4.8 for configuration). For the sake of simplicity, all configurations are handled by internal classes `*XMLReader`, and in the same way as it was done with the sensor classes, they are also applied to generate a hybrid configurator, which can be applied to change settings during runtime as well as through the parsing config-files (as presented in Lis. 4.8).

Class `SituatedDistanceSensor` thus implements all the required ROS and OpenRAVE functionalities, but it also offers a general interface to all measurements.

Listing 4.8: Configuration example of a situated distance sensor

```
<?xml version="1.0" encoding="UTF-8"?>
<sensor name="sharpGP2D120_2" type="sit_distance">
    <!--################ OpenRAVE DISTANCE SENSOR PARAMETERS ##############-->
    <max_range>0.30</max_range>          <!-- ranges in meter               -->
    <min_range>0.04</min_range>
    <mesh>beams/cone.dae</mesh>          <!-- approximating the real sensor -->
    <color>1 0 0</color>                 <!-- color of the virtual beam     -->
    <scantime>0.039</scantime>           <!-- time in seconds               -->
    <power>1</power>                     <!-- start measuring immediately   -->
    <!--################ ROS-RELATED COMMUNICATION PARAMETERS #############-->
    <master>http://edubot02:3113</master>
    <topic>/infrared2</topic>
    <realcolor>0 1 0</realcolor>         <!-- visualization color of real...-->
</sensor>                                <!-- measurements (same mesh)      -->
```

4.2.2.3. Integrating Arbitrary Information

All reconstructed objects within the simulation environments have the same properties such as identifiers, masses, colors, densities, etc. as defined within column-family complex. Locations (rooms, walls, etc.) do not necessarily require such properties, since they represent immobile and static objects. But as discussed earlier, the amount of additional information that can be attached to a complex entity is arbitrary, such as additional and ambiguous names, possible ownership relations, battery charge and lifetime, etc. To preserve this information that cannot be directly translated into the simulation environment, an additional container is applied. OpenRAVE offers two methods for handling user data that can be attached to any object, robot, or sensor: `SetUserData(key, information)` and `GetUserData(key)`. Additional properties such as ownership are simply stored within that container and can afterwards be queried from within the local environment model, without the necessity of querying the virtual overlay database again. Another simulation environment used for local environment modeling has to implement similar containers.

4.2.3. Summary

The presented components at this section provide an implementation to generate and local world models, based on the data and descriptions stored within the Cassandra-based systems, which was introduced in the previous section. Not all of elements described within the conceptual part were integrated, but it offers an adequate foundation for the following steps. The developed parts can be easily connected to their real world counterparts and are capable of replicating movements and measurements as well within the simulated environment. The OpenRAVE simulation thus offers also an abstract knowledge base, a simplified and idealized version of the real surrounding that can be interfaced to access and extract different kinds of information.

4.3. Extraction & Abstraction

This section is mainly intended to describe the implementation of the SelectScript query language and its extensions to support reasoning. But before that, a short overview of the OpenRAVE module is given, which was developed previously and which enables the abstraction of different kinds or representations from a local (OpenRAVE) world model.

4.3.1. Abstracting Different Representations

Fig. 4.5 depicts the filter module that was developed to transform an OpenRAVE simulation in other representations, such as occupancy grid maps, quad trees, etc. As it was mentioned before, nearly all abstractions presented in Fig. 2.5 on page 36 were generated with the help of this module. The system as it is depicted in Fig. 4.5 is freely available for download. To get a second impression on its application and capabilities, take a look screencast at:

`https://www.youtube.com/watch?v=DTX2pXk5Q2Q`

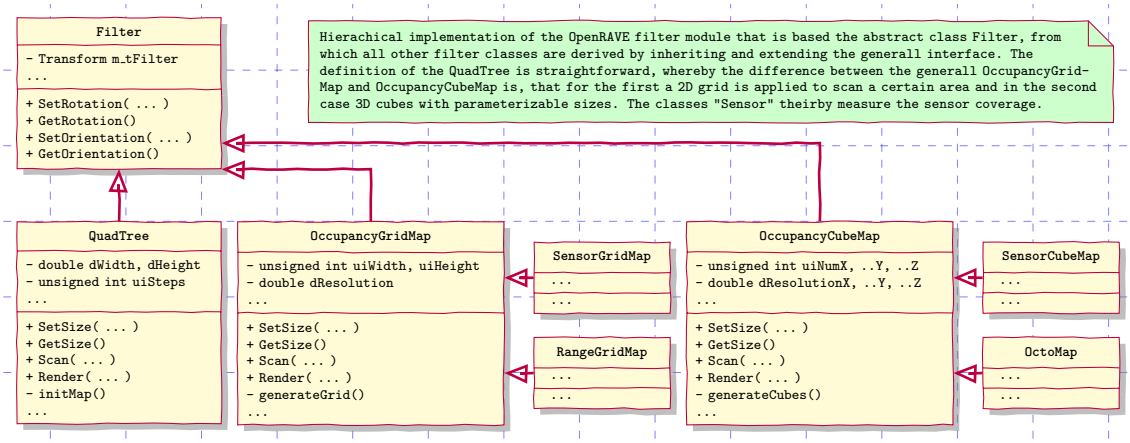


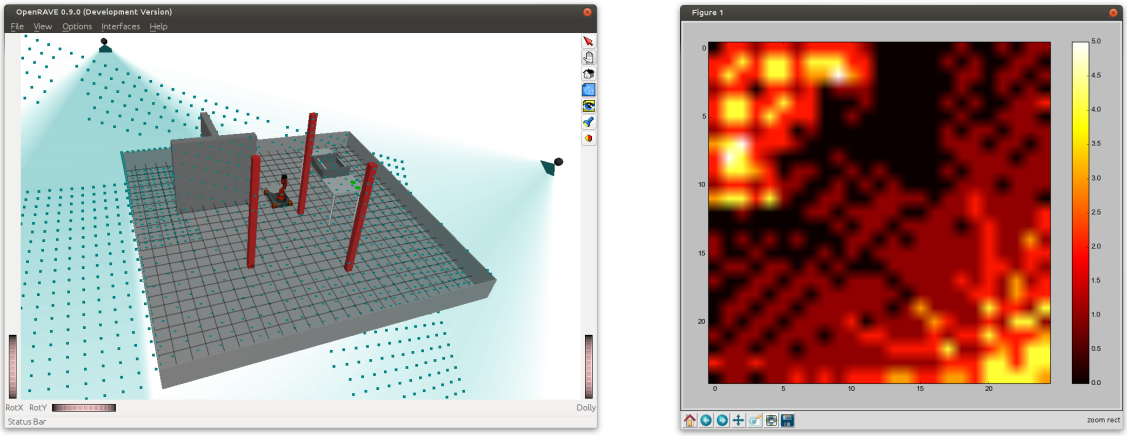
Figure 4.5.: Simplified UML class diagram of the filter plugin implementations for OpenRAVE

The simple architecture depicted in Fig. 4.5 allows to derive new types of filters from existing ones, such as the `SensorGridMap`, as it is depicted in Fig. 4.6 below. The basic interface is thereby defined by the abstract class `Filter`, its usage is shown in the Python example below. Other filters are applied similarly. (See the source code examples that were attached to the module.)

Listing 4.9: Application of the OpenRAVE filter module for abstracting a sensor coverage map

```
1  #initialize the filter module and apply it to the simulated environment
2  Filter = RaveCreateModule(env,'sensorgridmap')
3  Filter.SendCommand('SetTranslation -2.5 -2.5 0.1')
4  Filter.SendCommand('SetSize 25 25 0.2')
5
6  Filter.SendCommand('Render')      # visualize the grid in the scene, Fig. 4.6a
7  scan = Filter.SendCommand('Scan') # perform the sensor coverage scan, Fig. 4.6b
```

As it was already discussed, the appropriate configuration of these and other modules for deriving the required abstraction can be quite complicated — although the example might suggest something else. There are frequent context changes within a dynamically running simulation, which also affect simple parameters. To deal with this and to hide the access to different abstraction, the query language SelectScript, whose implementation is described within the next part, has been developed.



(a) Exemplary OpenRAVE scene

(b) Plotted filter result

Figure 4.6.: Estimating the sensor coverage (b) of a simulated area that is monitored by two external camera systems (a)

4.3.2. Extraction of Information

As depicted in Fig. 4.7, SelectScript is divided into two parts. Thus, code written in SelectScript is not directly executed, but instead translated into an intermediate representation, which is then passed to the interpreter. The upper part, which is generated with the help of the parser generator ANother Tool for Language Recognition (ANTLR) [168], is responsible for translating code into a simplified type of an Abstract Syntax Tree (AST), which is then passed to the interpreter module below.

Decoupling the parsing and execution process has several advantages; the AST can be optimized beforehand and executed multiple times afterwards. The AST thereby consists of a hierarchy of nodes that match with the principal features of a script. The interpreter then only has to traverse the AST from top-down, which can be easily accomplished with recursive functions. Instead of building a virtual machine with complex commands and its own stacking procedures, the stacking implemented by the host programming language can be utilized. A disadvantage, thereby, is that the parsing and interpretation modules must be maintained in precise synchrony.

Therefore, the generated bytecode is described in more detail within the next section, which is followed by a briefly explained interpretation sections, since it represents a simple (recursive) walk through the AST.

In addition to the basic interpreter and the derived interpreter for OpenRAVE, there is currently also an interpreter for ODE available. The following YouTube screencast demonstrates the application of SelectScript for a chaotic particle simulation:

`https://www.youtube.com/watch?v=F1XNch1JC9Y`

4.3.2.1. Intermediate Representation

The example in Lis. 4.10 shows how the ANTLR generated module (`SelectScript`) can be integrated into any Python code and applied to translate a script (string representation) into an AST, which shows a LISP-like representation (see line 39 in Lis. 4.10). Line 4

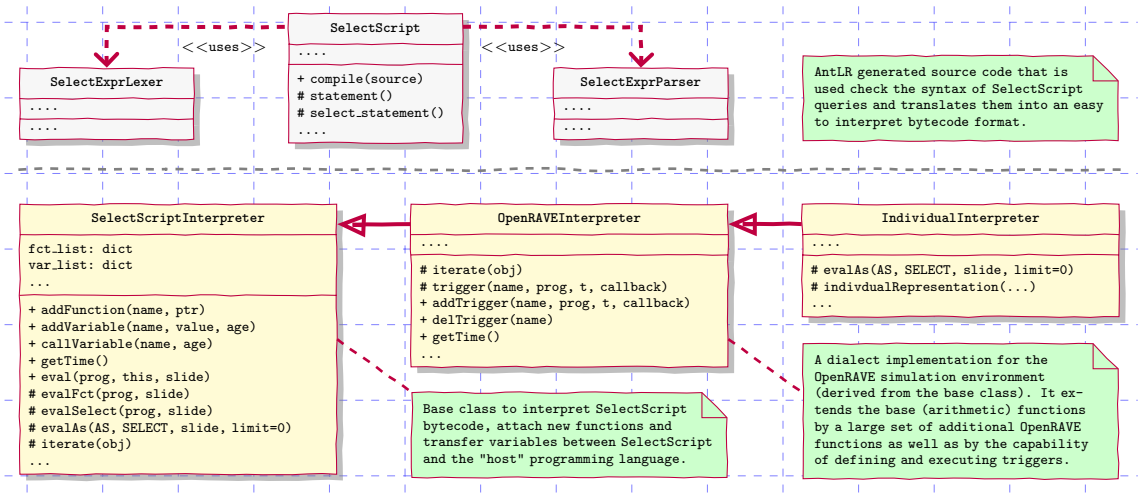


Figure 4.7.: Simplified UML diagram of the two-piece class structure of SelectScript

furthermore depicts how the basic compiler can be manually extended (with a sinus function) to be able to pre-evaluate extended arithmetic expressions (standard operations are already included) and, thus, to generate an optimized AST. The SelectScript in line 32 contains five statements (as mentioned before every statement ends with a semicolon), a logical expression, an arithmetic expression, an arithmetic expression whose result is stored within a variable, a function call, and a definition of a nested list.

The generated representation in line 39 shows the optimized AST, which can be directly evaluated by the interpreter. The compiler already reduces some expressions, but it allows getting a first impression on the general concept of the generated bytecode, which is a recursive structured list of lists. This representation was taken over from the internal representation of different "Computer Algebra Systems" that are either based on LISP-like Maxima or have been inspired by it, such as Maple or MuPad (see also [72]). For example, the expression in line 34 in Lis. 4.10 is represented internally in Maple and Maxima as `[^, [*, 7, 8], 9]`.

Listing 4.10: Translation of SelectScript with five statements into an optimized intermediate representation

```
1  import math
2  from SelectScript import *                    # load the module
3  ssc = SelectScriptCompiler()                  # instantiate the compiler
4  ssc.simplify_ops['sin'] = math.sin            # attach a function pointer
5  ...
32 script= "(0 and True) or (1 xor False);       # logical expression          \
33           2.3 + 4 * 5;                        # arithmetic expression       \
34           var{6} = (7 * 8)^9;                 # variable definition         \
35           sin(var + 10 * 11);                 # function call               \
36           [var{2}, [12 % 13]];                # list as return value        "
37 bcode = ssc.compile(script)
38 print bcode # list below in comments
39 #[[3,1],[3,22.3],[0,'assign',[[3,'var'],[3,5416169448144896],[3,6]]],[0,'sin'
40 #,[[0,'add',[[1,'var',[3,0]],[3,110]]]]],[2,[[1,'var',[3,2]],[2,[[3,12]]]]]]
```

In contrast to this, see also Lis. 4.11 for the not optimized version (by setting the `debug` parameter to `True`). Both versions can be evaluated with the interpreter and would generate exactly the same result. The generated representations consist of 12 atomic elements, whereby the first element of each list is used to indicate its type. These 12 numbers, which

can be related to operation code (opcode) for the interpreter, are described within the following enumeration:

0 Functions: These are represented by lists with three elements, whereby the first value is of course always 0, the second is used to identify the function, and the third contains a list of parameters, which have to be evaluated before the function can be evaluated.

By comparing the expressions in Lis. 4.10 with the not optimized version in Lis. 4.11, it becomes obvious that all arithmetic, logical, or comparison operations, as well as the assignment are internally represented as functions, whereby the operator precedence is represented by their position.

1 Variables: They are used to return already stored values. Their structure is similar to functions, the second list element is a string that identifies the variable and the third is also a list. This last list represents parameters (which have to be evaluated first) and identify the time horizon and, thus, the age of a value (see therefore also Sec. 3.3.1.3).

2 Lists: It is a simple method to define user specific return formats and consists only of two elements, whereby the second represents the list with its elements.

3 Values: Most of the elements within the optimized code in Lis. 4.10 are values, which (similar to the list) consist of two elements, whereby the second represents the actual value. This value can be of type boolean, integer, double, or string (cf. Sec. 3.3.1.3).

Listing 4.11: Unoptimized intermediate representation of the script defined in Lis. 4.10

```
1  bcode = ssc.compile(script, debug=True)
2  print bcode
3  #[  [0, 'or',      [  [0, 'and', [[3, 0], [3, True]]],
4  #                     [0, 'xor', [[3, 1], [3, False]]]],
5  #   [0, 'add',     [  [3, 2.3],
6  #                     [0, 'mul', [[3, 4], [3, 5]]]],
7  #   [0, 'assign', [  [3, 'var'],
8  #                     [0, 'pow', [  [0, 'mul', [[3, 7], [3, 8]]],
9  #                                   [3, 9]]],
10 #                     [3, 6]]],
11 #   [0, 'sin',     [  [0, 'add', [  [1, 'var', [3, 0]],
12 #                                   [0, 'mul', [[3, 10], [3, 11]]]]]]],
13 #   [2,            [  [1, 'var', [3, 2]],
14 #                     [2, [  [0, 'mod', [[3, 12], [3, 13]]]]]]]
```

Lis. 4.12 shows an exemplary select statements. Only the `SELECT` and `FROM` expressions are mandatory in a script, while the others are optional (but nevertheless mandatory for an improved evaluation). The result shows new elements that are required for an appropriate evaluation of query.

4 Select-Statements: As it is visible in the pretty printed representation in Lis. 4.12, a `SELECT` query contains eight further lists, one for every possible sub-expression (marked in the comments). Each of these lists contains the functions and expressions that have to be evaluated, except `HAVING` that is not yet implemented and the last elements that are used for recursive queries. The `SELECT` expression, for example, contains two lists which are equally structured as the previously discussed elements. In most cases these are functions that have to be applied onto the iterated elements, with a nested structure, etc.

Listing 4.12: Intermediate representation of a `SELECT` query

```
code = " SELECT f1(a.this), f2(b.this, f3(a.this))  \
           FROM a = struct, b = struct              \
          WHERE f4(a.this) >= f5(b.this)            \
       GROUP BY f6(a.this), f7(b.this, var)         \
       ORDER BY f8(a.this) DESC                     \
          LIMIT 10                                  \
             AS representation(1, f9(var));         "
bcode = ssc.compile(code)
print bcode
# [  [4,  [  [0, 'f1',  [  [5, 'a']]],                              ## SELECT
#            [0, 'f2',  [  [5, 'b'],
#                          [0, 'f3', [  [5, 'a']]]]],
#         [  [0, 'assign', [  [3, 'a'],                              ## FROM
#                             [1, 'struct', [3, 0]],
#                             [3, 0]]],
#            [0, 'assign', [  [3, 'b'],
#                             [1, 'struct', [3, 0]],
#                             [3, 0]]]],
#         [0, 'ge',  [  [0, 'f4', [[5, 'a']]],                       ## WHERE
#                       [0, 'f5', [[5, 'b']]]]],
#         [  [0, 'f6',  [  [5, 'a']]],                               ## GROUP BY
#            [0, 'f7',  [  [5, 'b'],
#                          [1, 'var', [3, 0]]]]],
#         [ ],                                                       ## HAVING
#         [  [  [0, 'f8', [  [5, 'a']]], 1]],                        ## ORDER BY
#         ['representation', [  [3, 1],                              ## AS
#                               [0, 'f9', [  [1, 'var', [3, 0]]]]],
#         [  [ ],                 ## START WITH
#            [ ],                 ## CONNECT BY
#            [ ],                 ## STOP WITH
#            [0, 0, [], []]]]     ## list for additional optimizations
#         [3, 10]]                                                   ## LIMIT
```

The third last list in this case is used to define the required output format of a select query. The four standard response representations (i. e., `list`, `dictionary`, `value`, and `dummy`) are part of the language definition and can thus be defined in uppercase, lowercase, or mixed, but in the AST they are either represented as `'l'`, `'d'`, `'v'`, `'0'`. For all other representation formats that are user-defined and, thus, are extensions of the basic interpreter, the compiler passes the complete identifier of the requested format. Because optional parameters are allowed (see Lis. 5.12 on page 126), the second list contains additional parameters, which have to be defined within parentheses.

5 This-Pointers: Looking at the resulting `FROM` list in Lis. 4.12, it shows two elements, or two assignment functions that are used to define new variables (`a` and `b`, that are only pointing on variable `struct`) in order to generate the Cartesian product of all elements within `struct`.

To be able to evaluate functions with the correct order of parameters and to be able to choose between the required elements, the `a.this` keyword (or `[5, 'a']` internal representation) is applied. Whereby the string is representing the variable, the Cartesian product of `...FROM env, so2, x=func(...)...` would result in three different elements (i. e., `[5, 'env']`, `[5, 'so2']`, `[5, 'x']`). In contrast to this, in a `FROM` definition with only one element, it is sufficient to use only `this`, resulting in an empty string `[5, '']`.

7 Internal Evaluation: If a string with some SelectScript code has to be evaluated at runtime, the internal `eval` function can be applied. This results in an internal representation [`7, 'string-code'`] and dynamically allows the generation of new code and, thus, reflexive programming.

...: There are of course also further additional elements that are used internally only.

10 The IF Expression: The element to be mentioned at last is the `IF` expression, as it was already presented in Sec. 3.3.2.1 in Lis. 3.12. It is translated into the following structure [`10, [stmt] [stmt], [stmt]`], in which the result of the first statement defines which statement is to be evaluated afterwards; if true then the first, otherwise the second. The evaluated statement also defines the return value of the `IF` expression.

Listing 4.13: Intermediate representation of a procedure and a sequence

```
code = " PROC(a, b) : (a.this; b.this; a.this+b.this;) ; "
print = ssc.compile(code)
# [[11, ['a', 'b'], [[5, 'a'], [5, 'b'], [0, 'add', [[5, 'a'], [5, 'b']]]]]]
```

11 Procedures: As listed above, procedures can be used to simplify queries, as they store code — that is executed when they are called. A procedure therefore has to be stored in a variable, list, etc. The first list thereby defines parameters which can be passed over as pointers to the procedure, when it is evaluated.

Sequences: A script itself is defined as a list of list, which defines a successive chain of instructions that have to be evaluated (the last defines the return). In the same manner, set of instructions can be defined within parentheses, as it is done in the example above. All elements are evaluated and the result of the last is returned as the result of this evaluation. Sequences can be defined everywhere, in such a way that evaluating a `WHERE` expression can also contain sequences, which enables a more complex definition of queries.

The open language definition of SelectScript allows it to place `SELECT` statements nearly everywhere within a script. The equivalent code of the obscure script, which is listed in Lis. 4.14 below, implemented within pure Python or any other language would probably generate a looping monster that is hard to define, hard to maintain, and even harder to change and adapt. Due to the applied LISP-like intermediate representation, the implementation of an interpreter, which is presented in within the next sub-section, is actually quite simple.

4.3.2.2. The Interpreter

The `SelectScript_Interpreter` as depicted in Fig. 4.7 consists of only one class and has to be extended by derived classes in order to be able to access simulation data. Nevertheless, this basic interpreter can operate on simple lists, dictionaries, values, as it already covers basic operations (in fact, all binary and unary operations listed in Lis. 3.3). And as already mentioned, the implementation of the interpreter is "pretty simple" and should furthermore also be possible for other languages than Python. The current proof of concept implementation consists of less than 160 lines of code, if the recursive elements (search programming)

Listing 4.14: Obscure example of a nested SelectScript query

```
SELECT foo(this, max( SELECT bar FROM struct ))
  FROM main_struct = SELECT bar FROM struct2
                      WHERE foo_bar(this, 2) in SELECT foo
                                                  FROM struct3
                                                    AS dict (max(SELECT bar
                                                                   FROM struct
                                                                     AS list))
                         AS list
    AS Map(height = SELECT pos FROM main_struct
                     WHERE test1(this) AND NOT IF( test3(this, 'pi'),
                                                   print("wow"), True,
                                                   eval("fct("+str(this)+")"))
                     ORDER BY pos ASC
                        AS val  );
```

are excluded. The reason for this lies in the recursive implementation of the interpreter, which is directly related to the structure of the intermediate bytecode representation. The following example is therefore written within pseudo code by applying a Pascal-like syntax.

Listing 4.15: Pseudo code implementation of the SelectScript interpreter

```
FUNCTION eval(p_list, slice = NIL) BEGIN
    CASE p_list[0] OF
        is Array : BEGIN    { sub-lists with code (expressions/phrases/...) }
            results := Array[];
            FOR sub_list in p_list DO
                append( results, eval(sub_list, slice) );
            RETURN results;                                              END
        0 : BEGIN { ----------- Functions (name, [parameters]) ------------ }
            RETURN callFunction( p_list[1], eval(p_list[2], slice) );    END
        1 : BEGIN { --------------- Variables (name, time) ---------------- }
            RETURN callVariable( p_list[1], eval(p_list[2], slice) );    END
        2 : BEGIN { ------------------------ Lists ------------------------ }
            RETURN eval( p_list[1], slice );                             END
        3 : BEGIN { ------------------------ Values ----------------------- }
            RETURN p_list[1];                                            END
        4 : BEGIN { ------- Select-Statements ([select][from][...]) ------- }
            RETURN evalSelect( p_list[1..8], slice );                    END
        5 : BEGIN { --------------------- This-Pointer -------------------- }
            RETURN slice[p_list[1]];                                     END
        ..:       { -------------------- Miscellaneous -------------------- }
        7 : BEGIN { ----------------- Internal Evaluation ----------------- }
            RETURN eval( compile( p_list[1] ), slice );                  END
        ..:       { -------------------- Miscellaneous -------------------- }
        10: BEGIN { --------------------- If Then Else --------------------- }
            IF eval(p_list[1], this) THEN: RETURN eval( p_list[2], slice );
            ELSE: RETURN eval( p_list[3], slice );                       END
        11: BEGIN { ---------------------- Procedure ---------------------- }
            RETURN Procedure(p_list[1], p_list[2])
    END { CASE }
END { FUNCTION }
```

The program itself and defined sequences, which define successive commands or expressions terminated by a semicolon, are organized as successive lists covered by the first case in the central `eval` function. Other elements are identified with a leading number: for example, `0` identifies a function call followed by a unique function name and by a list of parameters that have to be evaluated beforehand. These elements form the input parameter of the helper function `callFunction`, which executes the appropriate function with the given parameters and returns its result. New functions can be added quite easily, as it is described within the next sub-section.

Calling the content of a variable works similarly (but operates on different structures in the background); the second list element contains the name and the third comprises time parameters. The assignment of variables is handled with the help of a function call associated to "`assign`" (to which the "`=`" operator is transformed) — the parameters are the name of the variable and its time definition. That is it, and the evaluation of lists along with values is straightforward. The cases 4 and 5 are required to evaluate select queries correctly and `evalSelect`, as listed below, is used as another helper function.

Listing 4.16: Pseudo code for evaluating `SELECT` expressions

```
1  FUNCTION evalSelect(p_list, slice) BEGIN
2      Select  := p_list[0];   { list of expression to apply to the result set }
3      Result1 := evalFrom( p_list[1], slice );
4      IF recursive elements have been defined THEN:
5          Result2 := evalRecursion( p_list, Result1 );        { see Sec. 4.3.4 }
6      ELSE:
7          Result2 := evalWhere( p_list[2], Result1 );
8      Result3 := evalOrder( p_list[5], Result2 );
9      Result4 := evalGroup( p_list[3], Result3 );
10   { Result5 := evalHaving(p_list[4], Result4 ) }
11     RETURN evalAs( p_list[6][1],  { string, identifying the response format }
12               eval( p_list[6][1], slice ),   { list of additional parameters }
13                     Select,        { list of expressions to apply to Result4 }
14                     Result4,     { identified, ordered, and grouped elements }
15                     eval(p_list[8]));            { evaluate LIMIT expression }
16 END { FUNCTION }
```

It is used to call all required "filter" steps in the correct order. As mentioned before, only the select- and from parts are mandatory, defining the source and the result. The default result output format is a table-like structure ("dictionary"), generated with the help of `evalAs`. The other steps are used to refine the result set. Also all of these functions make use of the central `eval` function. `evalFrom` is used to create an iterator that either iterates through a list, traverses all nodes of a tree, or steps through any other structure. If there are multiple (comma separated) structures to iterate through, then they are iterated repeatedly and form the Cartesian product. Recursion is left out in this consideration, but described in more detail in Sec. 4.3.4 on page 113.

The "from" generated iterator is also the source of the second "`slice`" input parameter of function `eval`. It represents the current element/row of the result list/matrix to be evaluated. It is passed to all "where", "order", "group", or "as" evaluation steps. Case 5 in the central `eval` function determines the correct element by evaluating the `this` expression and returns its value. Because queries might contain further (nested) queries, the `slice` parameter is also extended recursively (see, for example, Lis. 4.28 on 115).

The resulting format is generated by the `evalAs` function, which takes a string as input (that identifies the requested output format, e. g., list, map, etc.), the second parameter contains a list of input parameters (which probably also have to be evaluated, because they might also contain expressions and further select statements), the third parameter simply contains a list of expressions that have to be applied on the identified result set (parameter four).

As described within the next sub-section, all of these functions can be redefined or overloaded in order derive new interpreters for new simulation environments or to integrate and generate more and different kinds of abstractions.

4.3.3. Building Extensions

There are already complex dialects available for OpenRAVE and ODE, but the following example is intended to present insights on the extensions and adaptation process of the interpreter. Therefore, the ten following listings are used to give a rather abstract example, by developing a prototypical extension for strings. For further information, see also the notebooks that are available at: `https://gitlab.com/OvGU-ESS/SelectScript_demos`

As already mentioned, new functions can be attached without the need for deriving a new interpreter class. As it is listed in Fig. 4.7 and indicated in Lis. 4.17, the basic interpreter `SelectScriptInterpreter` already contains a method (`addFunction(name, ptr, ...)`) that allows it to add new functions or methods. These are stored internally in an associative array. Thus, the used function name within a script has to match this key within the "function repository" that is to be called.

Listing 4.17: Adding new functionality to the basic SelectScript interpreter

```
1  import SelectScript
2  import SelectScript.Interpreter
3
4  ssi = SelectScript.Interpreter()
5  ssi.addFunction("ord", ord, "Returns the int "+  # attaching a new function,
6  "ordinal of a character.\nUsage: ord('a')->97")  # with an optional info string
```

As it is demonstrated below, the SelectScript internal help system, which is common for many scripting languages, allows it to explore the available functionality.

Listing 4.18: Using the internal help system

```
7  prog = SelectScript.compile("help();")         # requesting help results in a
8  print ssi.eval(prog)                           # list of all available func-
9  # ['help','clear','print','to','var','ord']    # tions, operators not included
10
11 prog = SelectScript.compile("help('ord');")    # getting help for a certain
12 print ssi.eval(prog)                           # function # ord #
13 # Returns the int ordinal of a character.
14 # Usage: ord('a')->97
```

Operators are not listed, but their behavior can also be changed if required. Since operators (see line 18 in the listing on page 79) are handled internally as functions, they can be adapted accordingly. The simple example in Lis. 4.19 below shows how the modulo operator can be made applicable onto the string data type as well. The result is a list of sub-strings with a length that is defined by the second operator.

Listing 4.19: Adapting the behavior of SelectScript operators

```
15 prog = SelectScript.compile("'abCdeFghI' %3;")
16 print ssi.eval(prog)
17 # ---------------------------------------------------------------------------
18 # TypeError                                 Traceback (most recent call last)
19 # ...
48 def newModulo(op_1, op_2):
49     if isinstance(op_1, str):
50         return [ op_1[i:i+op_2] for i in range(0, len(op_1), op_2) ]
51     return op_1 % op_2 # else part
52
53 ssi.addFunction("mod", newModulo)
54 print ssi.eval(prog)
55 # ['abC', 'deF', 'ghI']
```

For exchanging data between the SelectScript interpreter and the host programming language, the internal method `addVariable` can be used. It requires some data and a new

name. Similarly, complete scene-graphs and simulation environments can be attached to the interpreter. All available variables, including those that have been created internally for storing intermediate results, can also be listed and explored with the internal help function `mem()` (see the listing below).

Listing 4.20: Porting data to the SelectScript interpreter

```
56 data = "Xabcdefghabc543201AVBX"              # attaching new data to the inter-
57 ssi.addVariable("abc", data)                 # preter is similar to the attach-
58                                              # ment of new functions ...
59 prog = SelectScript.compile("mem( );")       # showing all stored elements is
60 print ssi.eval(prog)                         # similar ... procedures are stored
61 # ['abc']                                    # in the local memory too
62
63 prog = SelectScript.compile("mem('abc');") # and also getting more informa-
64 print ssi.eval(prog)                         # tion for debug purposes; proce-
65 # age:  0                                    # dures generate other outputs
66 # val:  'Xabcdefghabc543201AVBX'
```

These variables/data can be accessed by the host programming language in two ways: either via directly calling the variable with a script — the last statement of a script also defines its return value. Or, in order to simplify this process, without the need for generating an intermediate representation, variables can also be accessed directly by calling method `callVariable`.

Listing 4.21: Getting data from SelectScript back to the host programming language

```
67 prog = SelectScript.compile("abc;")          # a script can also be used to re-
68 print ssi.eval(prog)                         # turn the value of a variable ...
69 # 'Xabcdefghabc543201AVBX'
70 print ssi.callVariable('abc')                # but it is also possible to re-
71 # 'Xabcdefghabc543201AVBX'                   # quest them via "callVariable"
```

As it is visible within the example below, trying to query strings does not work as expected. The basic SelectScript interpreter interprets strings as a whole and, thus, returns only one element, dividing a string into its elements, namely characters requires either to apply the previously adapted modulo operator or to derive an new class that overwrites method `getListFrom`.

Listing 4.22: Trying to apply queries to strings, step 1 & 2

```
72 prog = SelectScript.compile("SELECT this FROM abc AS list;")
73 print ssi.eval(prog)
74 # ['Xabcdefghabc543201AVBX']
75
76 prog = SelectScript.compile("SELECT this FROM abc%1 AS list;") print
77 ssi.eval(prog) #
78 ['X','a','b','c','d','e','f','g','h','a','b','c','5','4','3','2','0', ...
```

The example in the listing below depicts a little Python hack, which allows changing the method of a certain object without the need for deriving a new class. The new `iterString` method is used to replace the method `iterateThrough`. It checks the data types of elements that are called within a `FROM` part, all data is automatically passed to this method before it is evaluated, and if it is a string it will return all characters separately; otherwise, the object is passed back to the super class method.

Thus, repeating the query now does not require the usage of an additional modulo operator or another function, and it allows to use of all the common `SelectScript` functionalities without further changes.

Listing 4.23: Integrate the ability to iterate over characters within a string

```
import types
def iterString(self, obj):
    if isinstance(obj, str):
        for c in obj: yield c
    else:
        yield SelectScript.Interpreter.iterate(self, obj)
ssi.iterate = types.MethodType(iterString, ssi, SelectScript.Interpreter)
```

Listing 4.24: Trying to apply queries to strings, step 3

```
prog = SelectScript.compile("SELECT to(this, 'char'), ord(this) \
                             FROM abc WHERE ord(this) < 60 \
                             AS dict;")
print ssi.eval(prog)
# [{'char': '5', 'ord': 53},
#  {'char': '4', 'ord': 52},
#  {'char': '3', 'ord': 51},
#  {'char': '2', 'ord': 50},
#  {'char': '0', 'ord': 48},
#  {'char': '1', 'ord': 49}]
```

The last element that has not been touched yet is the ability to define different response formats, so far only `lists` and `dicts` have been requested. The process for this is similar to the previous extension, and it also requires deriving a new class and overwriting method `evalAs`, or, as it is presented below in Lis. 4.25, to directly redefine a method of an object. The requested format is automatically passed on in parameter `AS` and has to be checked at first. If the result of a query is requested `AS string`, all parameters are passed to the super class method `evalAs_list` for simplicity and then joined into a string. More complex representations can also be derived; therefore, all additional `SELECT` expressions and results of the previous evaluations (`WHERE` ...) are passed on as additional parameters.

Listing 4.25: Defining new response formats for SelectScript

```
def evalAs_string(self, AS, PARAMS, SELECT, RESULTS, LIMIT):
    if AS == 'string':
        return "".join(self.evalAs_list(PARAMS, SELECT, RESULTS, LIMIT))
    return SelectScript.Interpreter.evalAs(self, AS, PARAMS, SELECT,
                                                     RESULTS, LIMIT)
ssi.evalAs = types.MethodType(evalAs_string, ssi, SelectScript.Interpreter)
```

Lis. 4.26 shows the final example in which the result is requested as a string and the modulo operator is applied onto an integer value (and it works properly). Only characters with an even ordinal number are returned in order.

Listing 4.26: Requesting the result of a query `AS string`

```
prog = SelectScript.compile("SELECT this FROM abc     \
                             WHERE ord(this) % 2 == 0 \
                             ORDER BY ord(this)       \
                             AS string;")
print ssi.eval(prog)
# 024BVXXbbdfh
```

All other `SelectScript` dialects have been developed similarly, but with two additional features. The derived class has to overwrite the method `getTime`, which is used as a connection to the internal simulation time and allows it to add time-stamps to variables. Time-stamping and dealing with temporal variables is afterwards also handled in the background of the language interpreter. Furthermore, the ability to apply triggers, similar to the idea in active databases, is currently defined within the derived classes (see, therefore,

the application examples in Lis. 5.14, Lis. 5.15, or Lis. 5.16). It is currently not a part of the basic interpreter, since every simulation environment has its own way of integrating continuously executed functionalities.

4.3.4. Recursive Evaluation

As already introduced in Sec. 3.3.2.3, recursive queries offer a convenient way of defining search problems. Different search algorithms can be applied and tweaked without bothering about their details, while the definition of the actual problem remains fixed. (See, therefore, the example below in Lis. 4.27, as it depicts the query that is used to solve the Towers of Hanoi problem, but in a reduced version by selecting only the plain tower configuration with no additional and pretty printed steps.)

Listing 4.27: Recursive query with generated bytecode example

```
script = " moves = [[0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1]];          \
    SELECT move(this,tower) FROM moves                                         \
     WHERE move(this,tower) == [[],[],[6,5,4,3,2,1]]                           \
START WITH tower = [[6,5,4,3,2,1],[],[]]                                       \
CONNECT BY MEMORIZE 63                                                         \
           tower = move(this, tower)                                           \
 STOP WITH tower == []                                                         \
     LIMIT 123;                                                                "
bcode = SelectScript.compile(script, debug=True)
pp.print(bcode)
# [  ... initialization of variable moves ...
#    [4, [[0, 'move', [[5, ''], [1, 'tower', [3, 0]]]]],       ## SELECT
#        [[1, 'moves', [3, 0]]],                               ## FROM
#        [0, 'eq', [[0, 'move', [[5, ''], [1, 'tower', [3, ... ## WHERE
#        [], [], [],     ## GROUP BY, HAVING, ORDER BY
#        ['d', []],      ## AS dictionary (default)
#        [ [[0, 'assign', [[3, 'tower'], [2, [[2, [[3, 6], ... ## START WITH
#          [[0, 'assign', [[3, 'tower'], [0, 'move', [[5,  ... ## CONNECT BY
#          [0, 'eq', [[1, 'tower', [3, 0]], [2, []]]],         ## STOP WITH
#          [0, 0,          ## NO CYCLES (0/1) | UNIQUE (0/1)
#          [3, 63], []]],  ## MEMORIZE (STMT) | COST (STMT)
#        [3, 123]]]                                            ## LIMIT
```

The pretty printed bytecode within comments actually reveals nothing new; the only difference from previous examples is that the last list, which is used to define the hierarchical aspects, is not empty anymore. This recursive part consists of four sub-lists:

[[`START WITH`], ...] It contains a couple of (originally comma separated) statements that define starting conditions. These are executed only once at the beginning of a search.

[[`CONNECT BY`], ...] The second list defines the recursive structure, what elements are affected and how, it therefore contains a couple of (comma separated) statements that are executed at every recursion step.

[`STOP WITH`] In contrast to these, a stop condition is defined only with one expression, which is evaluated after every recursion step. If and only if it evaluates to `False` the evaluation of this "branch" is continued by newly interpreting the `FROM` and `WHERE` expressions.

That is it, these definitions — in combination with the primary `SELECT` ... `FROM` ... `WHERE` statements — are sufficient to let the abstract machine perform a search. But which search strategy gets applied is determined by the additional keywords that are listed below (see

also Sec. 3.3.2.3). Every keyword changes a value within the last sub-list (state vector) in this enumeration and, therefore, triggers the application or adaptation of a certain search algorithm.

[**`NO CYCLE,`** it is equal to the original definition (see [173]) and prevents cycling of repetitive results. The definition of this keyword simply sets the first list element to 1 otherwise (and by default) it is set to 0. The results are defined by the `SELECT` expression and repetitive results are compared at every hierarchical step. Thus, if the result of a previous `SELECT` expression is equal to the current, to be evaluated, this "path" is not followed anymore. It thus defines some kind of additional stop criterion.

`UNIQUE,` this keyword sets the second list value to either 1 or, if specified otherwise, to 0. It triggers another, stricter optimization strategy that backups all intermediate results (at least their hashes) and allows it, only to follow search paths with intermediate results that have not been visited yet. It thus prevents cycling on a global scale, while the previous one only keeps track of the current search path.

`MEMORIZE,` while the last keywords have been optional optimization to the depth-first search strategy, which is applied by default, this keyword changes the entire search algorithm and performs a bidirectional search on the basis of a directed graph. This graph is constructed with the help of the previously described recursive statements, where `SELECT` expressions represent nodes, and edges are defined by the `CONNECT BY` expression. To prevent the endlessly growth of this graph, the keyword `MEMORIZE` is accompanied by an additional expression that defines the maximum graph depth, thus the amount of allowed recursive steps. Since the applied search algorithm only generates simple paths (without cycles) and the graph itself contains only unique nodes, setting of previous parameters is currently obsolete. But it can also be replaced by other algorithms...

`COST`] And at last, an element for defining a cost "expression" is available. It is evaluated prior to each and every recursive in such a way that in the depth-first search algorithm, the elements from the `FROM` expression are ordered accordingly. The element with the lowest cost is then evaluated within the next step and so on. Within the creation process of the directed graph (`MEMORIZE`), this element defines the weight for every generated edge. After the graph is created, these edge weights are used to calculate the "length" of a simple path and to return identified paths ordered by the sum of their edges, starting with the smallest one.

The following two parts briefly describe how the generated intermediate code, which has been described so far, is used to perform different search strategies. For the sake of simplicity, only two strategies are described, without the application of optimizations.

4.3.4.1. Recursion: Depth-First Search

The code example in the listing below depicts the general evaluation of a recursive query, if only `START WITH`, `CONNECT BY`, and `STOP WITH` expressions are defined. For the sake of clarity, these sub-list containing the relevant expressions are copied into local variables (line 3). The `START WITH` expressions are evaluated only once (line 8), and since they are only used to define the local variables and the initial settings, there is no need to store the results of this evaluations, nor to react to them. The only thing that has to be stored is

the current state of defined (global) variables, which includes identifiers and their values (line 10). These are restored at every iteration of the loop (line 13). Within the loop, each element/row of the list/matrix generated originally by the `evalFrom` function is evaluated (line 15) and tested to determine if it fulfills the goal condition (line 17) or the stop condition (line 20). Only if the stop condition is not met, the variables are changed by evaluating the `CONNECT BY` expressions (line 21) and the `evalRecursionDepthFirst` is called recursively, returning new sequences that lead to a goal condition.

Listing 4.28: Pseudo code for evaluating simple recursive queries with the depth-first search algorithm

```
1  FUNCTION evalRecursionDepthFirst(p_list, slice, From) BEGIN
2      { copy recursive expressions }
3      Start := p_list[7][0];   Connect := p_list[7][1];   Stop := p_list[7][2];
4
5      p_list[7][0] := Array[];  { clear start expression, to prevent from ... }
6      Results := Array[];               { ... being evaluated multiple times }
7
8      FOR expr IN Start DO
9          eval(expr, slice);                 { < generate local variables and }
10     backup := variable_repository;         { < ...   make a back up of them }
11
12     FOR row IN From DO      { go through all rows }
13         variable_repository := backup;            { restore local variables }
14         { generate a new intermediate result }
15         current := evalAs(p_list[6][0],p_list[6][1],p_list[0],[row,slice]);
16
17         IF evalWhere(p_list[2], [row, slice]) THEN
18             append( Results, current );
19
20         IF not eval(Stop, [row, slice]) THEN
21             FOR expr IN Connect DO             { change local variables ... }
22                 eval(expr, [row, slice]);              { ... in repository }
23
24             FOR next IN evalRecursionDepthFirst(p_list, slice, From) DO
25                 append(Results, concatenate(current, next));
26
27     RETURN Results;
28 END { FUNCTION }
```

There is no such value that defines something like a maximum search depth, in fact this has to be defined explicitly (if required) within the script, which can be achieved with the following code:

```
1  ...
2  START WITH ...., depth = 0
3  CONNECT BY ...., depth = depth+1
4  STOP WITH .. AND depth > 30
5  ...
```

As mentioned before, optimization strategies have not been considered as well as stacking, as it is handled by the host programming language, whereby other implementations without recursive function calls and local stacking are imaginable.

4.3.4.2. Recursion: Bidirectional Search

If keyword `MEMORIZE <STMT>` is defined, another search strategy is executed. In contrast to the previous version, the definition of the maximum search depth as an additional value here is mandatory. Here, it does not directly define the maximum search depth but length of the generated graph (line 16), which is searched afterwards (line 43). Thus, the function

below does not perform the actual search; it is used to transform the problem (query) into an equivalent graph representation. This construction process consumes relatively more memory and time, but it allows applying different and more efficient search algorithms. The beginning of the function is thus similar to the previous one, whereby the graph is initialized with two nodes (the source and the target node). Afterwards, new nodes that are created recursively are added to the graph, together with an additional edge. If a newly generated node already exists within the graph, then only the new edge is attached. For all nodes that also fulfill the goal condition (line 39), a new edge is added pointing from that node to the target node.

Stored directly within the node (or another structure, such as hash map, associative array, etc.) are all intermediate results as well as the configuration of variables, which have to be restored at every iteration step. The benefit of this approach is that cycles are prevented and every node is unique. Thus, the optimization strategies `NO CYCLE` and `UNIQUE` are not required here.

Listing 4.29: Pseudo code for generating a graph on the basis of the recursive definitions, which is searched afterwards for all simple paths

```
1  FUNCTION evalRecursionGraph(p_list, slice, From):
2      { copy recursive expressions }
3      Start := p_list[7][0];   Connect := p_list[7][1];   Stop := p_list[7][2];
4      p_list[7][0] := Array[];                    { delete start expressions }
5
6      G := Graph();  { new graph to store the entire structure }
7
8      FOR expr IN Start DO              { < generate local variables and ... }
9          eval(expr, slice);            { v store them within the first node }
10     source = Node(level=0,   data=NIL, memory=variable_repository);
11     target = Node(level=INF, data=NIL, memory=NIL);
12
13     add_node(G, id='source', node=source);
14     add_node(G, id='target', node=target);
15
16     FOR L in 0..p_list[7][3][2] DO            { defined by MEMORIZE <STMT> }
17         FOR node in G with level equal to L DO
18             FOR row in From DO
19                 variable_repository := node.memory; { restore var. settings }
20                 { generate intermediate results ... }
21                 data := evalAs( p_list[6][0] ,p_list[6][1],
22                                 p_list[0], [row, slice] );
23                 { ..., which can be used to generate a node identifier ...  }
24                 id := hash(data);
25
26                 IF id not in G THEN { create & insert a new node into graph }
27                     IF eval(Stop, [row, slice]) THEN
28                         memory := NIL; level := NIL;
29                     ELSE
30                         FOR expr IN Connect DO
31                             eval(expr, [row, slice])
32                         memory := varible_repository; level := L+1;
33
34                     new_node := Node(level, data, memory);
35                     add_node(G, id, new_node);
36
37                 add_edge(G, node.id, id); { insert new edge }
38
39                 IF evalWhere(prog[2], [row, slice]) THEN
40                     add_edge(G, id, 'target'); { new target node identified }
41
42     { now, search G for paths from source to target }
43     RETURN bidirectional_all_simple_paths(G, 'source', 'target');
44 END { FUNCTION }
```

As listed in line 43 above, the graph is searched for simple paths at the end. A simple path is a path with no repeating nodes. The currently implemented strategy is a bidirectional search algorithm that runs in two direction: one from the source to the target and another from the target to the source. Thereby only matching leaves have to be identified, it thus minimizes the search space by half. In addition and as a further benefit, paths are generated in order, starting from the shortest paths and then increasing stepwise in length. More information on the algorithm and a benchmark against common search algorithms is presented in [245].

4.3.5. Discussion

The developed programming language is not only a DSL for one specific purpose, which allows abstracting a simulation environment as if it was a database. Instead, it is extendable so that it can be used in various ways that even go beyond the notion of a simple query language. As it is shown in the next chapter, every kind of query is handled with simple SelectScript requests, thereby it is possible either to apply the declarative paradigm, or the imperative — or a combination of both. And as it is demonstrated in the Appendix on page 141, the language can also be used to reason about the application and order of filter, fusion, and transformation functions, which allows it to identify a sequence of such functions, depending on the given input and the desired output.

The current version of the interpreter is developed in Python, which is a drawback in terms of speed and memory efficiency; besides this, Python offers a convenient way for rapid prototyping, experimenting with new language elements, and an object-oriented possibility for adaptations. However, the implementation of the language and the interpreter is so minimalistic that it can also be implemented in other languages. A first prototype is currently implemented in C, which is so small that it can even be applied on microcontrollers, allowing the query of not only the environment with this language, but also every system, embedded or not. Having the same capabilities in terms of dynamic typing, it is required to develop an own type system. In terms of speed, first tests reveal that the pure C implementation is approx. 500 times faster than the current Python prototype.

In addition to the interpretation of the language, it is also imaginable to directly translate a `SelectScript` into optimized Python bytecode, which could be directly executed on the Python VM (without the need for an additional interpreter). In the same way, it should therefore also be possible to apply the LLVM (Low Level Virtual Machine) infrastructure in order to produce just-in-time compiled code for different queries.

5. Evaluation

The following evaluation is thought to present a typical delivery scenario in a smart factory environment (see Fig. 5.1 on page 123). Therefore, a mobile robotic platform at first has to reach the current location of a packet and deliver it to a certain destination. This "pretty" simple task includes several sub-tasks to be solved, such as:

- identifying and selecting the appropriate mobile platform,
- identifying the relative positions of the packet to deliver and the mobile platform,
- generating a local model that includes both entities as well as others,
- abstracting different types of maps, which are used as a basis for different path planning strategies as well as for the robot's localization,
- defining situations onto which the mobile platform has to react,
- online identification of robots that are in the coverage area of different sensors, to be used for better localization and to enable extended environmental perception,
- and abstracting logical/formal models that are used to reason about complex action sequences (grasping the packet) ...

As it was mentioned before, all of these sub-tasks will be solved with the help of SelectScript — or SelectScript will at least provide the basis for a solution. Furthermore, all of the three steps that were defined within the concept (i. e., access to the global model, generation of a local model, and extraction of further representations and information) are abstracted by the developed query language itself. Therefore, a special "dialect" has been developed (similar to the one presented in Sec. 4.3.3) that combines all previously developed components. It is described briefly within the next section. The following sections introduce the access to the global world model and the generation of local world models, which is then combined within the fourth section to solve each of the intermediate steps that are listed above and that are necessary to accomplish the packet delivering task.

5.1. Initialization

The developed language interpreter in module `SelectScript_Evaluation` encapsulates all parts of the described concept and can be downloaded at:

`https://gitlab.com/OvGU-ESS/SelectScript_OpenRAVE`

It, therefore, inherits aspects from all software solutions that have been developed, these are glodel (for accessing the global model, see Sec. 4.1.3), `plugOpenrave` (for generating local models, Sec. 4.2.2), and the existing SelectScirpt dialect for OpenRAVE (for extracting information from an OpenRAVE simulation and abstracting further environmental

representation, as it was described in [6, 11]), which has also been combined with the developed `filter` plugins (see Sec. 4.3.1).

Lis. 5.1 shows a Python snippet that is sufficient to generate a local OpenRAVE instance as well as an instance to the new language extension for SelectScript. The connection to a Cassandra instance and thus to the global model is generated automatically. Compiling queries to the intermediate representation is, for simplicity reasons, also kept in the background. Thus, as it is shown in the last line in Lis. 5.1, the response to a query is directly generated.

Listing 5.1: Python initialization of the evaluation scenario

```
from openravepy import *        # load OpenRAVE lib for Python
env = Environment()             # create empty OpenRAVE environment
env.SetViewer("qtcoin")         # set up visualization

# load interface to new SelectScript dialect
from SelectScript_Evaluation import Interpreter

base = Interpreter(environment = env, # attaching an OpenRAVE environment
                   host = "localhost",      # global model access points
                   port = 9160,             # with host, port, and key-
                   keyspace = "Evaluation", # space ...
                   debug = False)     # not running in debug mode

base.query("help();")    # first query, gets directly compiled and executed #
# ["sensingEnvIDs", "help", "get_time", "get_location", "within",
#  "environmentID", "free_model", "get_sensor", "get_robot", "id", ... ]
```

The listing above is the only Python example; all of the following queries and scripts are represented as pure SelectScript snippets, which are directly passed to `base.query`. The generated results are then either presented as screenshots or as comments at the end of a listing.

5.2. Accessing the Global World Model

The SelectScript interface to the data stored in the virtual overlay database was defined in accordance with the main column-families that were described in Sec. 3.1. There are four functions used per column-family, for accessing (`get`), for adding (`add`), for updating (`set`), and for removing (`rm`). Handling of raw and virtual data is currently not integrated and requires applying the original interface as it was described in [12].

As shown in the listing below for column-family robot, a robot can be described in many ways (e. g., zipped COLLADA, XML, URDF, CAD, etc.), which enables an application or, in this case, the described `plugOpenrave` module to select the most appropriate format.

Listing 5.2: Accessing column-family robots

```
get_robot();
# ["barrett-wam", "katana", "kawanda-hirox", "kuka-youbot", "puma"]
get_robot("katana");
# OrderedDict([("zae", "PK\x03\x04\x14\x00\x00\x00\x00\x00\x08\x00\xa9\x8d\x
#                       d7>\xb8p\xd4\x15:\xa3\x0c\x00\xfaQP\x00\x14\x00\x1c\
#                       x00neuronics-katana.daeUT\t\x00\x03\x1e\xfd\x02N\x04
#                       \xfc\x02Nux\x0b\x00\x01\x04\xe8\x03\x00\x00\x04\xe8\
#                       x03\x00\x00\xcc\xfd]\xb347\x92\xa0\x07^\xf7\xfc\x8ac
#                       ....
set_robot("katana", "xml", "<Robot name= ... ")
```

Sensors are actually organized in the same way as robots; hence, for demonstration and for the sake of simplicity the standard OpenRAVE sensor description format was utilized to model and, thus, to describe the capabilities of real sensor systems. The sensor description in Lis. 5.3, for example, is equivalent to the description of Hokuyo URG-04LX-UG01 laser scanner, in the data-sheet [254].

Listing 5.3: Accessing column-family sensors

```
get_sensor();
# ["geovision:GV-FER5302", "hokuyo:URG_04LX_UG01", "sharp:GP2Y0A41SK0F", ...
get_sensor("hokuyo:URG_04LX_UG01");
# OrderedDict([("xml", "<sensor type='BaseLaser2D'>
#                           <minangle>-120</minangle>
#                           <maxangle> 120</maxangle>
#                           <maxrange>   4</maxrange>
#                           <resolution> 0.36</resolution>
#                           <scantime>   0.10</scantime>
#                       </sensor>")])
```

As it is furthermore visible, the column-families for robots and sensors contain only a little amount of data if it is compared with the amount of sensors that are actually used within a local environment model (see the next section). The reason for this was already described within the concept in Sec. 3.1. Sensors, robots, locations, and objects are only placeholders for a certain type or class of entity, which is put into context and associated with additional information by the hierarchical organized column-family complex (see Sec. 3.1.2). As it is shown in the listing below, there might be multiple different Youbots or Hokuyo laser scanners, but each of them points to a certain sensor or robot description, which is identified by the entries `type` and `key`.

Listing 5.4: Accessing column-family complex

```
get_complex();
# ["*", ..., "cam_g28d", ..., "hall1", ... , "hokuyo_8502", .. "plant", ...,
# "youbot_1kfe", "youbot_29tg", "youbot_8502", "youbot_x7e1", "youbot_xk99"]
get_complex("hokuyo_8502");
# OrderedDict([("base",        "youbot_8502"),
#              ("key",         "hokuyo:URG_04LX_UG01"),
#              ("link",        "base"),        # == part of the robot
#              ("master",      "http://youbot_8502:13675"),
#              ("orientation", "0.9 0 0 0.9"),
#              ("topics",      ["/scan", "/scaninfo"])),
#              ("translation", "0.3 0 0.1"),
#              ("type",        "SENSOR")])
```

As it is also visible, every complex entry also has, in addition to supplementary communication parameters, a relative position to another complex base. For instance, the sensor's position in the listing above is attached with a certain translation and orientation to the base of complex entry with key `youbot_8502`.

In order to deal with these hierarchies, there are two functions defined: `pred` and `succ` (see Lis. 5.5), which stand for predecessor and successor. While there is currently only one predecessor allowed, which is defined by the complex entry `base`, an entity can have multiple successors. Thus, different complex entries can point to the same origin, such as multiple sensors can be attached to one robot.

The listing below emphasizes this situation with multiple different successors. The complex entry for `*` thereby denotes the root of the hierarchy (see, therefore, also the example Fig. 5.1c). It thus can be used as a convention for a common entry point, if there is no further information on other entries that can be used to build a local environment model and it is also the only complex entry with no predecessor.

Listing 5.5: Application of `pred` and `succ` for dealing with the hierarchy in column-family complex

```
pred("hokuyo_8502");       # equal to the entry base, see Lis. 5.4
# youbot_8520

succ("*");
# ["plant", "outside_terrain"]

succ("plant", full=True);
# [("hall1",
#    OrderedDict([("base", "plant"), ("translation", "8 1 0"),
#                 ("type", "COMPLEX"), ("info", ...), ... ])),
#  ("hall2",
#    OrderedDict([("base", "plant"), ("translation", "4 13 0"),
#                 ("type", "COMPLEX"), ("info", ...), ... ])),
#  ("hall3",
#    OrderedDict([("base", "plant"), ("translation", "23 9 0"),
#                 ("type", "COMPLEX"), ("info", ...), ... ]))]
```

5.3. Generation of Local World Models

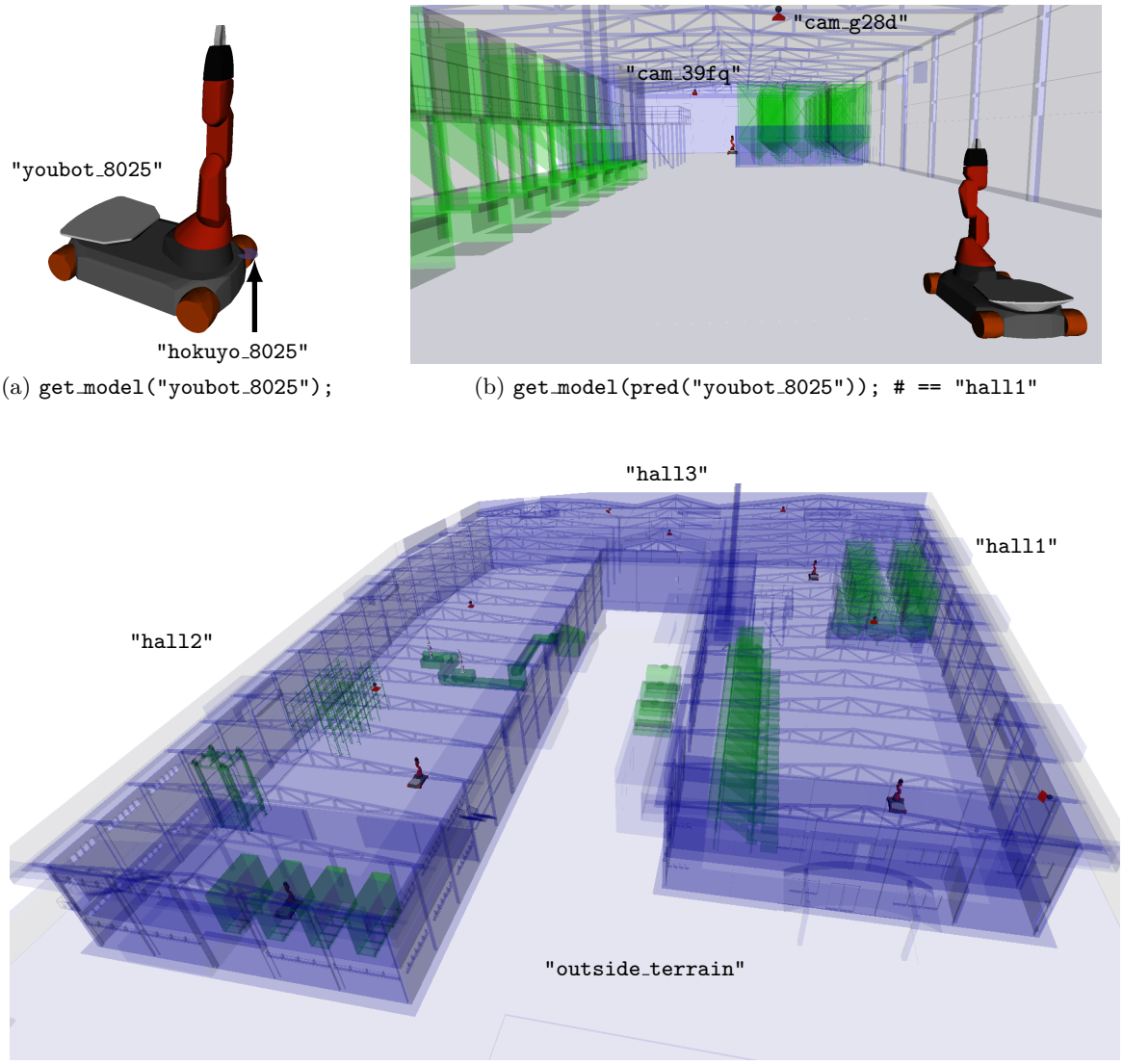
The generation of local world models on the basis of complex entries, as it was described in Sec. 3.2 and implemented in module `plugOpenrave` (see Sec. 4.2.2), is handled by one central function `get_model`. As it is depicted in the listing below, it requires two parameters. Here, the second parameter is optional as it only defines the recursion depth, or in other words, the depth of successors is inserted into the model. If no second parameter is passed, then all successors are integrated into the local model.

Listing 5.6: Generation of local world models

```
help("get_model");
# get_model:
# Generates a OpenRAVE model from "glodel", based on the passed identifier,
# the int value for recursion_depth is optional ...
# Usage: get_model(string:identifier, int:recursion_depth) -> OpenRAVE model
```

Fig. 5.1 shows three examples for reconstructed local world models as OpenRAVE simulations in different complexity. Every location-specific entry is marked by the blue color, while entities from column-family objects are highlighted by the green color. The whole environment as depicted in Fig. 5.1c consists of three halls and an outside terrain, these complex entries are further segregated into quadrants of equal size, which allows generating even more fine-grained local world models (see, for example, Fig. 5.4 on page 129).

The reconstructed local world model represents the latest state of the environment that is stored within the global model. To update the local model, it is therefore required to subscribe for topics of included robots that are used to publish the current robotic state. As it was described in Sec. 3.2.1, every real action can thus be replicated within the virtual world allowing the observation of the effects an action has onto the environment. As it is shown within the next section, it is more appropriate to subscribe for real sensor data only if it is required for a certain task, or to minimize communication overhead: yet SelectScript helps to identify which sensor data is relevant or not.



(a) `get_model("youbot_8025");`

(b) `get_model(pred("youbot_8025")); # == "hall1"`

(c) `get_model("*"); # == get_model( pred(pred("hall1")) )`

Figure 5.1.: Three screenshots of reconstructed local world models. (a) shows the model of the Youbot with the identifier `youbot_8025`, which also includes the associated Hokuyo laser scanner. (b) depicts the reconstructed model of production `hall1`, the current predecessor of the shown Youbot in (a). The entire `plant` along with its outside terrain and all assembly lines, robots, external sensor systems, etc. are depicted in (c).

5.4. Application of SelectScript

While the previous sections more or less dealt only with functions that were attached to SelectScript, the following one is intended to demonstrate the complete capabilities of the language and how it can be used for information extraction and representation abstraction.

As it was introduced before, the first sub-task covers the identification of a mobile robotic platform that can be used to accomplish the task of delivering the packet, which is registered

under the complex identifier `cargo_a32s` within the virtual overlay database. Lis. 5.7 is therefore used to query for the identifier of a mobile robotic platform. Since complex entries only possess relative position definitions, at first a local model of the whole environment is created with the command `get_model` (see also the appertaining screenshot in Fig. 5.1c). All robots within the generated model are checked if they fulfill the characteristics of being a mobile robot in idle state. These are ordered according to their Euclidean distance to the packet that has to be delivered, and the identifier of the first robot is directly returned. If required, also other parameters such as battery status, size, max load parameters could be added to the query; or if the robotic platform possesses a manipulator, this manipulator can be used to grasp the object (check within the simulation), etc.

Listing 5.7: Query for identifying the closes mobile robot in idle state to the target

```
  SELECT id(this)
    FROM get_model( "*" )
   WHERE isRobot(this) AND isMobile(this) AND state(this) == "idle"
ORDER BY distance(this, "cargo_a32s") ASC
      AS value;
#result: youbot_8502
```

The next step could target the identification of the minimal model containing both entities: the robot and the packet, namely the complex identifier of the nearest predecessor of both entities in the hierarchy. Although there are multiple solutions imaginable as well as a specialized function defined within the host programming language, the listing below shows a simple way through which this recurring problem can be solved with the possibilities of SelectScript. A procedure with two input parameters that is based on the previously described predecessor function was therefore defined. It applies the basic recursion strategy that was described in Sec. 4.3.4.1 on page 114 to search for a common predecessor of both entities that is not equal to the global source of the complex hierarchy. Within the `IF` expression, it is evaluated whether the search was successful or not; if not, then the global source identifier `*` is returned as the result. Remember, the result of the last statement defines the return value of a script or procedure.

Listing 5.8: Search for the closest common predecessor identifier

```
MIN_MODEL_ID = PROC(source, target) : (
    ID = SELECT parent[this] FROM [0,1]
          WHERE parent[0] == parent[1]

     START WITH parent = [source.this, target.this]
     CONNECT BY parent[this] = pred(parent[this])
      STOP WITH pred(parent[this]) == "*"

           LIMIT 1       # limit of one, because one result is sufficient
              AS list;   # result as a list

    IF ( ID != [],  # if:   the result is not empty
         ID[0,-1],  # then: return the last element within the list
         "*");      # else: return the main entry id (entire model)
); # end of PROCEDURE
```

The procedure above is used within the following listing to identify the closest predecessor of the robot and its target, and its result is used to reconstruct the smallest local environment that includes both entities. The positions of the robot and the target objects within the newly created model are printed out by the last two function-calls.

Listing 5.9: Generating the minimal local model with the robot and its cargo

```
robot_id  = "youbot_8502";  # id of the previously identified robot
target_id = "cargo_a32s";   # id of the target for the transport to perform

model_id  = MIN_MODEL_ID( robot_id, target_id);  # plant, the successor of
                                                 # hall1, hall2, and hall3

get_model( model_id );      # generate a new local environment model

print( "source position:", position(robot_id)    /*[x, y, z]*/ );
print( "target position:", position(target_id)   /*[x, y, z]*/ );
# source position: [12.0,  3.5, 0.65]
# target position: [11.5, 20.5, 0.0 ]
```

The following expression demonstrates how a local environment model can be simplified or cleaned up, if necessary. The resulting model only contains elements associated with location and the previously identified Youbot. All of the other previously included objects, sensors, and robots are filtered out. The variable `environment` is thereby associated with the OpenRAVE simulation environment. Whenever `environment` is applied within a script, it is used to access the current simulation environment. By assigning the result of a query to this variable, the current state of the simulation environment itself and, thus, its visualization is changed as well. But it is also possible to create other OpenRAVE environments by assigning them to other variable identifiers than `environment`. In this case, clones of the existing environments are generated, which can be used in parallel for different purposes. The generation of new OpenRAVE simulation environments is defined by the keywords `AS environment` (see the last line in the listing below).

Listing 5.10: Filtering the local environment model

```
environment = SELECT this
                FROM environment
               WHERE isLocation(this) OR id(this) == robot_id
                  AS environment;
```

The next procedure demonstrates how recursive queries can be used to identify a path from the current position of the robot to the position of the target object or, in other words, to check if a path to the location of the target object exists. The program below represents a simplified version to the program that was published in [11]. The first `SELECT` statement is used to generate a list that defines the directions into which a robot can move, based on the passed value for the parameter `step_width`.

The second `SELECT` searches a path from the source position to the position of the target object. It therefore literally moves the virtual robot within the virtual environment into all directions, which were generated by the first `SELECT` statement. To speed up the search, an optimization method that was introduced in Sec. 3.3.2.3 on page 84 and that is defined by the keyword `UNIQUE` is used. The search process is disrupted if a correct path is identified, if the target position cannot be reached anymore with the remaining number of steps, or if a collision between the robot and its environment occur. The result of this procedure is a list of x,y coordinates. (See also Fig. 5.2a, which depicts a screenshot of the cleaned up minimal local environment model (top-view) along with the traced positions that were visited during the search (red) and the identified path (black)).

The identified trajectory has a length of 150 steps. This kind of reasoning can surely be used for path planning, but there are various different kinds of algorithms that are much more elaborate and can solve this problem in a more efficient way. As mentioned earlier,

Listing 5.11: Test for reachability

```
TEST_REACHABILITY = PROC(robot, target_pos, steps, step_width) : (
    # generate a list with directions as [x,y] coordinates
    directions = SELECT [x.this, y.this]
                   FROM x=[step_width.this, -step_width.this,0], y=x
                  WHERE [x.this, y.this] != [0,0]
                     AS list;

        SELECT (this+cur_pos)
          FROM directions
         WHERE target_pos.this == move(robot, this+cur_pos)

    START WITH cur_pos = position(robot,0,2), level = steps.this
    CONNECT BY UNIQUE
                  level = level-1, cur_pos = move(robot,cur_pos+this)
     STOP WITH distance(target_pos,this+cur_pos) > step_width.this*level
                  OR checkCollision(robot)
          LIMIT 1    # one identified path is sufficient
             AS list;
); # end of PROCEDURE

path = TEST_REACHABILITY( robot_id, position(target_id, 0, 2), 150, 0.25 );
#[[12.25, 3.75], [12.50, 4.00], [12.75, 4.25], [13.00, 4.50], [13.25, 4.75],
# [13.50, 5.00], [13.75, 5.25], [14.00, 5.50], [14.25, 5.75], [14.50, 6.00],
# [14.75, 6.25], [15.00, 6.50], [15.25, 6.75], [15.50, 7.00], [15.75, 7.25],
# [15.50, 7.25], [15.25, 7.00], [15.25, 7.25], [15.00, 7.00], [15.00, 6.75],
# [15.25, 6.50], [15.50, 6.75], [15.75, 6.50], [16.00, 6.25], [16.25, 6.00],
# [16.50, 5.75], [16.75, 5.50], [17.00, 5.75], [17.25, 5.50], [17.50, 5.75],
# [17.75, 5.50], [18.00, 5.75], [18.25, 6.00], [18.50, 5.75], [18.75, 6.00],
# [19.00, 6.25], [19.25, 6.50], [19.50, 6.75], [19.75, 6.50], [20.00, 6.75],
# ......
```

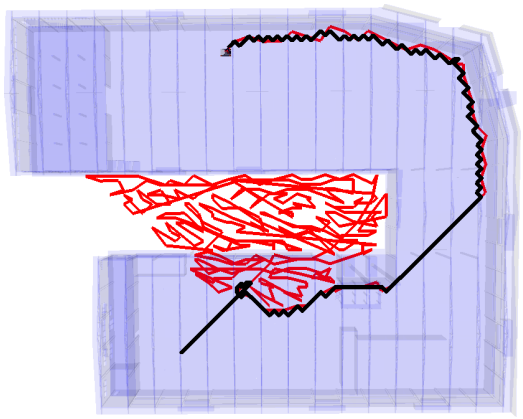
scripts can also be used to request the needed input to an algorithm or system in a desired format. The following query in Lis. 5.12, for example, generates an occupancy grid map at a certain height and with a certain resolution, which contains only task-relevant entities and that can be either used for later localization or used as an input for a path planning algorithm. (See the resulting map in Fig. 5.2b.)

Listing 5.12: Abstractions of further environmental representations

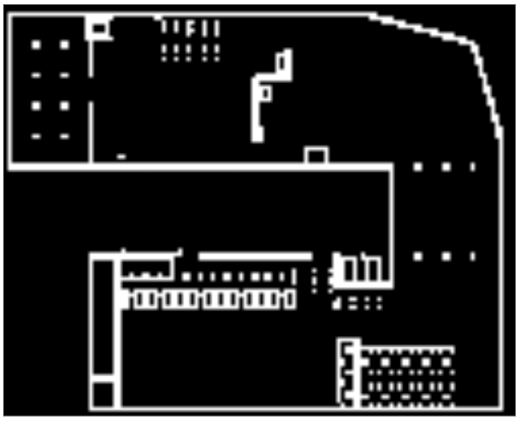
```
get_model(model_id); # restore minimal local model

SELECT this FROM environment
 WHERE isLocation(this) OR isObject(this)
    AS occupancygrid( resolution("hokuyo_8502"),  # get scanner resolution
                      position("hokuyo_8502", 2, 3) );  # get z position
```

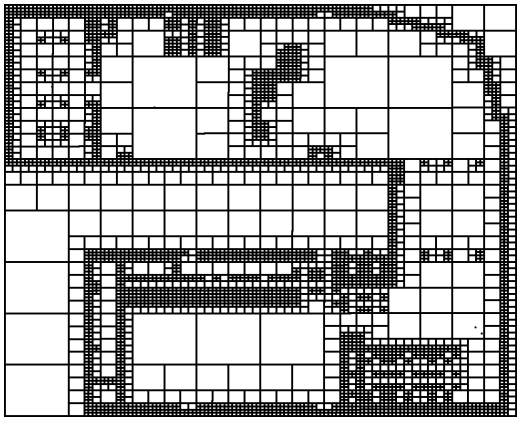
A simple change of the desired format, followed by the keyword `AS`, results in completely different environmental abstractions. In addition to a simple textual representation as list, table-like dictionaries, etc., the result of a query can also be requested as a quadtree Fig. 5.2c, as an octomap Fig. 5.2e with the height of the Youbot. But it is also possible to generate something totally different, which was actually hard to grasp, such as a map showing the sensor coverage of an area Fig. 5.2d. Such a combination of a sensor coverage map and an occupancy grid map could be used to identify optimal paths in terms of external surveillance. Another method could apply to historical positions of robots from previous trajectories to derive histograms showing highly frequented areas that could be avoided by the path planning algorithm.



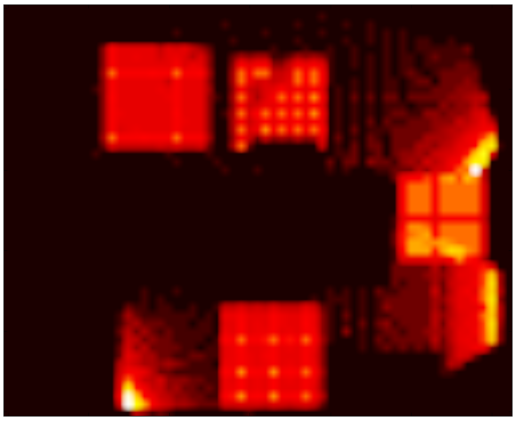
(a) Traced reachability test



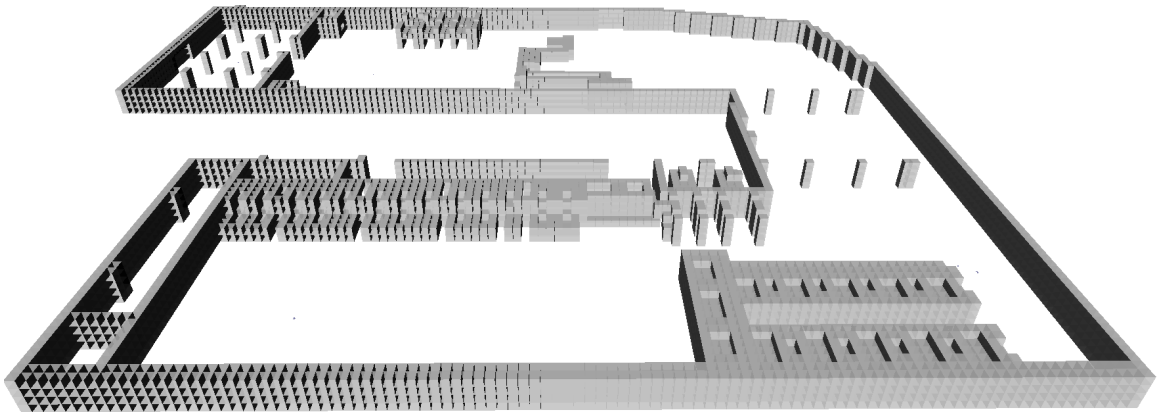
(b) `AS occupancygrid`



(c) `AS quadtree`



(d) `AS sensor_map`



(e) `AS octomap`

Figure 5.2.: Generated results of SelectScript queries, while (a) shows the result of Lis. 5.11, all the other figures depict results of Lis. 5.12 in different request formats.

Increasing localization accuracy by applying external sensor systems could be accomplished by the simple method that is depicted below in Lis. 5.13. It is used to identify all robots that are currently in the detection area of a sensor, the result is grouped by the robot identifiers themselves, as printed out in the last lines of the snippet.

Listing 5.13: Identifying combinations of robots and their observing sensor systems, the result is grouped by the robot identifiers

```
  SELECT id(sensor.this)
    FROM robot = environment, sensor = environment
   WHERE isRobot(robot.this) AND isSensor(sensor.this) AND
         isSensing(sensor.this, robot.this)
GROUP BY id(robot.this)
      AS list;
# { "katana_12e4": ["cam_q17b"], "youbot_4567": ["cam_8502", "cam_4389"],
#   "youbot_2389": ["cam_p15c", "hokuyo_b9f5"], "katana_96aa": ["cam_q17b"],
#   "youbot_8502": ["cam_g28d"] }
```

But the same can also be accomplished online for one robot, which requires knowledge about its observers so that external sensor systems can be used for a better localization or to monitor the space that will be entered by a robot in the future. The example in Lis. 5.14 therefore contains four statements: The first `SELECT` statement is only used to store all sensor identifiers of the local environment model within a list, which is then used in the second procedure to identify those sensors that are currently monitoring the robot with the identifier that is stored in `robot_id`. This procedure is then registered with a unique identifier, a repetition time for checking of 0.5 s, and a procedure that is called, if the result of the query changes. This is also the reason for the different time-stamps that are printed out at the end of Lis. 5.14, as it was described previously in Sec. 3.3.1.2.4; the callback procedure is only called if the result changes. Thus, the last state of a query is persistent as long as the callback procedure is not called (which reduces the communication effort tremendously). This process is better visualized in Fig. 5.3 from two perspectives, where only those sensors are rendered that are monitoring the certain Youbot at the moment.

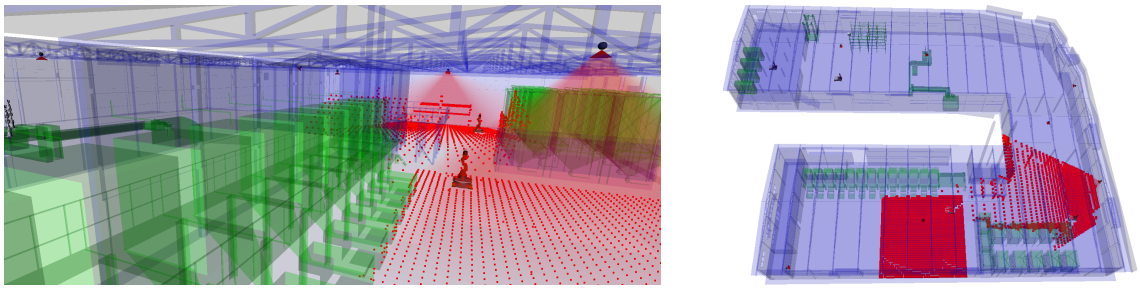
Listing 5.14: Defining a simple situation in SelectScript and reacting onto occurred changes by applying a callback mechanism, see the rendered result in Fig. 5.3

```
sensors = SELECT this
            FROM environment
           WHERE isSensor(this)
              AS list;

SENSOR_PROC = PROC: SELECT id(this), master(this), topics(this)
                      FROM sensors
                     WHERE isSensing(this, robot_id)
                        AS dict;

CALLBACK = PROC(msg): print("time:", get_time(), "msg:", msg.this);

# unregister_callback("robot_surveillance");
register_callback("robot_surveillance", SENSOR_PROC, 0.5, CALLBACK);
# time: 2.0  msg: [{"id": "cam_8502", "master": "http://cam_8502:13675",
#                   "topics": ["/camera/info", "/camera/image_rgb", "/cam...
# time: 5.5  msg: [{"id": "cam_4398", "master": "http://cam_4398:13223",
#                   "topics": ["/camera/info", "/camera/image_rgb", "/cam...
# ...
# time: 39.5 msg: []
# ...
```



(a) View from inside hall 1 (b) Top-view

Figure 5.3.: Snapshot of the result of Lis. 5.14 from two perspectives, rendering only sensor systems that have Youbot with identifier `youbot_8502` in their detection range

The same method can also be used to abstract more specific and local models, such as defined Lis. 5.15. It is used to dynamically generate smaller environment models that only contain elements that are in direct vicinity to the moving robot. The result for different points in time is shown in Fig. 5.4. In the same way, smaller maps can be generated as well, which change according to the dynamic changes of the environment.

Listing 5.15: Another method for dynamically generating local environment models, see the result in Fig. 5.4

```
1 MODEL_PROC = PROC: SELECT this
2                      FROM environment
3                     WHERE distance(this, robot_id) < 4
4                        AS environment;  # or AS occupancygrid ... or AS ...
5 register_callback("model_change", MODEL_PROC, ...
```

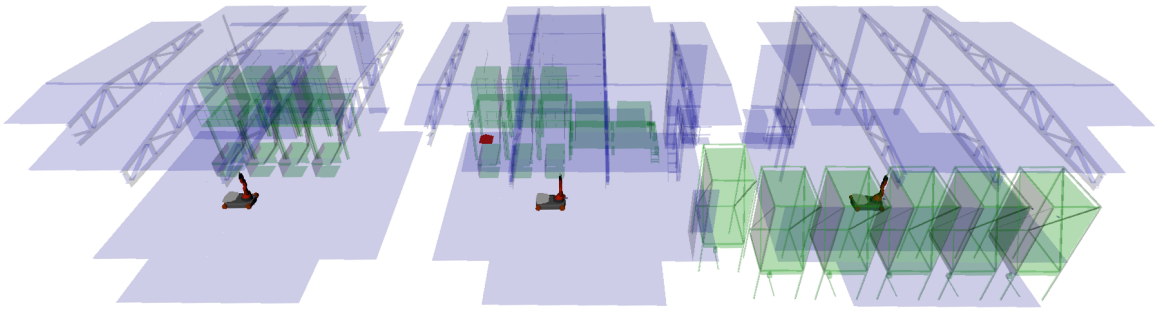


Figure 5.4.: Dynamically generated local environment models for different points in time, containing only elements that are in direct vicinity to the moving robot (see script in Lis. 5.15)

The more complex example in Lis. 5.16 below shows the application of temporal variables and, as previously mentioned in Sec. 4.3.2.1 on page 107, demonstrates that nesting of queries is allowed nearly everywhere. This script is intended to identify mobile robots that are for 2 s within the safety margin of the moving Youbot. Thus, only if a foreign robot was for the entire 2 s within the safety margin, the callback is triggered. The script is also stored as a procedure and fires its results via the previously presented mechanism. The first variable is a temporal variable, which stores the result of the query (all mobile robots within

a certain range to the current position of the Youbot) for 5 s. Within the second query, how the values of the last 2 s are accessed is shown. As it was described in Sec. 3.3.1.3 on page 80, getting the value for a certain point in time requires a positive value to be included into curly braces, for all values the braces have to be empty; and without braces, only the last recent value is returned (similar to an ordinary variable). The result of the second query is a dictionary (default representation), with the identifiers and current positions of robots that have been within the safety region of the mobile robot since 2 s in total.

Listing 5.16: Application of temporal variables to identify other robots within the safety area to the moving Youbot

```
NEAR_ROBOTS = PROC: (
    ROBOTS{5.0} = SELECT this
                    FROM ( SELECT this
                             FROM environment
                            WHERE isRobot(this) AND isMobile(this) AND
                                  id(this) != robot_id
                               AS list )
                   WHERE distance(this, robot_id) < 4
                      AS list;

    SELECT id(robot.this), position(robot.this)
      FROM robot = UNION( NEAR_ROBOTS{-2.0} )
     WHERE NEAR_ROBOTS{-2.0} == ( SELECT this
                                    FROM NEAR_ROBOTS{-2.0}
                                   WHERE robot.this in this
                                      AS list );
);
register_callback("near_robots", NEAR_ROBOTS, 0.1 ...
# time:  0.0 msg: []
# time: 43.7 msg: [{"id": "katana_86aa", "position": [17.64, 18.30, 0.32]}]
# time: 44.5 msg: [{"id": "katana_12e4", "position": [16.81, 16.85, 0.32]},
#                  {"id": "katana_86aa", "position": [17.64, 18.30, 0.32]}]
# ...
```

Finally, the robot arrived at the target position near the packet that has to be transported. The code in Lis. 5.17 generates a local model that contains a rack, which is the predecessor of the packet and also contains other packets stacked within.

Listing 5.17: Generating a local model for the cargo environment

```
get_model( pred("cargo_6h20") );      # creating a local model
set_color( "cargo_6h20", [1,0,0] );   # change color of object to red
```

It is easy to see that the red object cannot be directly grasped; instead, it requires identifying an action sequence for removing objects from above and placing them elsewhere before the actual target object can be grasped and placed onto the Youbot's transport platform (and probably putting the other objects back into the rack). As it was described in Sec. 2.3.4.1.10 and Sec. 2.3.4.1.11, such a type of problem can be easily solved by logical reasoning. It is thus often presented as an example for Prolog [248] or GOLOG, but it requires a logical representation of the environment. As it was also discussed earlier in Sec. 3.3.1.2.6, the ability of deriving such representations is mandatory as it is already encoded within the virtual world, and extracting the required facts is actually as easy as abstracting different types of maps or 3D representations. The only requirement is to request the result as a logical representation, by applying the keywords `AS prolog`, as it is done within the example in Lis. 5.18.

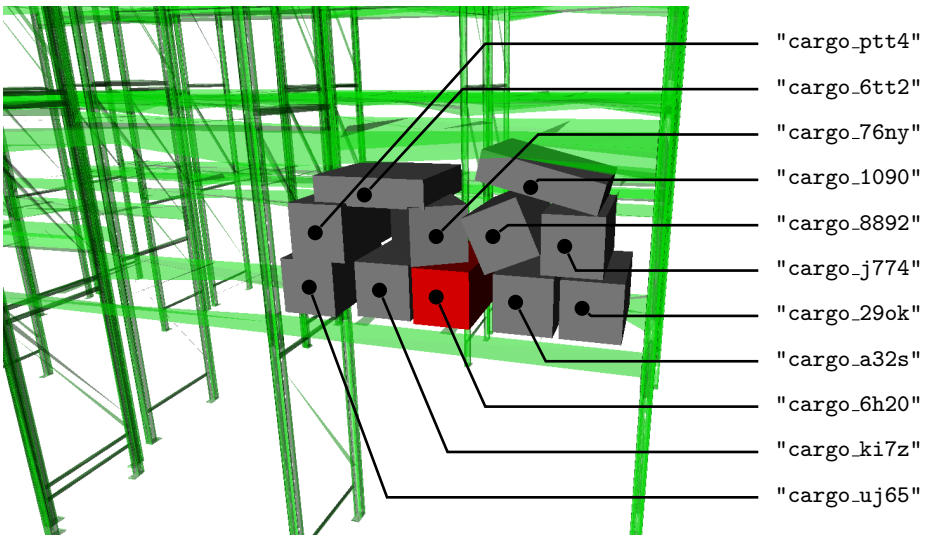


Figure 5.5.: Screenshot of the generated local model from Lis. 5.17, which contains the predecessor of the packet with identifier `cargo_6h20` as well as all of its successors

Listing 5.18: Deriving logical environmental representations (`AS prolog`)

```
1  SELECT above   (a.this, b.this),      below(a.this, b.this),
2     to( position(a.this), "pos"),
3     to( volume  (a.this), "vol"), to( within(a.this, b.this), "in")
4     #   left_of (a.this, b.this),   right_of(a.this, b.this),
5    FROM a=environment, b=environment
6   WHERE within("rack", a.this)
7      AS prolog;
```

The result of this query is depicted below in Lis. 5.19, and it shows a list of facts that can directly be fed into Prolog knowledge base, as it is done in Lis. 5.20. The result set in the listing below shows clearly that not everything is converted into a logical expression. Rather, clauses are generated only if the result of a function called within the `SELECT` expression does not evaluate to false or an empty list. The built-in function `"to"` is thereby only used to rename some of the result expressions. Facts about the left and right relation are commented out for the sake of readability, but it is possible to generate these facts also based on the current viewport (allowing to take over somebody else's perspective) or based on global coordinates.

The Python example in Lis. 5.20 below shows a simple method of how the facts, which were extracted from the local environment model, can be inserted into a Prolog knowledge base and the last line shows how this knowledge base can be queried in the Prolog typical manner.

Listing 5.19: Generated Prolog clauses as the result of the query in Lis. 5.18

```
result = [ "above(cargo_29ok, cargo_1090)",    "above(cargo_29ok, cargo_j774)",
           "above(cargo_6h20, cargo_6tt2)",    "above(cargo_6h20, cargo_76ny)",
           "above(cargo_6h20, cargo_8892)",    "above(cargo_76ny, cargo_6tt2)",
           "above(cargo_8892, cargo_1090)",    "above(cargo_a32s, cargo_1090)",
           "above(cargo_a32s, cargo_8892)",    "above(cargo_a32s, cargo_j774)",
           "above(cargo_j774, cargo_1090)",    "above(cargo_ki7z, cargo_6tt2)",
           "above(cargo_ki7z, cargo_76ny)",    "above(cargo_ptt4, cargo_6tt2)",
           "above(cargo_uj65, cargo_6tt2)",    "above(cargo_uj65, cargo_ptt4)",

           "below(cargo_1090, cargo_29ok)",    "below(cargo_1090, cargo_8892)",
           "below(cargo_1090, cargo_a32s)",    "below(cargo_1090, cargo_j774)",
           "below(cargo_6tt2, cargo_6h20)",    "below(cargo_6tt2, cargo_76ny)",
           "below(cargo_6tt2, cargo_ki7z)",    "below(cargo_6tt2, cargo_ptt4)",
           "below(cargo_6tt2, cargo_uj65)",    "below(cargo_76ny, cargo_6h20)",
           "below(cargo_76ny, cargo_ki7z)",    "below(cargo_8892, cargo_6h20)",
           "below(cargo_8892, cargo_a32s)",    "below(cargo_j774, cargo_29ok)",
           "below(cargo_j774, cargo_a32s)",    "below(cargo_ptt4, cargo_uj65)",

           "in(rack, cargo_1090)",             "in(rack, cargo_29ok)",
           "in(rack, cargo_6h20)",             "in(rack, cargo_6tt2)",
           "in(rack, cargo_76ny)",             "in(rack, cargo_8892)",
           "in(rack, cargo_a32s)",             "in(rack, cargo_j774)",
           "in(rack, cargo_ki7z)",             "in(rack, cargo_ptt4)",
           "in(rack, cargo_uj65)",

           "pos(cargo_1090, [-2.33, 1.25, 1.77])",
           "pos(cargo_29ok, [-2.52, 1.28, 1.38])",
           "pos(cargo_6h20, [-2.02, 1.28, 1.37])",
           "pos(cargo_6tt2, [-1.76, 1.25, 1.72])",
           "pos(cargo_76ny, [-1.88, 1.25, 1.58])",
           "pos(cargo_8892, [-2.21, 1.26, 1.59])",
           "pos(cargo_a32s, [-2.29, 1.28, 1.37])",
           "pos(cargo_j774, [-2.45, 1.25, 1.58])",
           "pos(cargo_ki7z, [-1.80, 1.26, 1.37])",
           "pos(cargo_ptt4, [-1.58, 1.29, 1.58])",
           "pos(cargo_uj65, [-1.53, 1.29, 1.37])",

           "vol(cargo_1090, 0.004)",           "vol(cargo_29ok, 0.002)",
           "vol(cargo_6h20, 0.002)",           "vol(cargo_6tt2, 0.002)",
           "vol(cargo_76ny, 0.004)",           "vol(cargo_8892, 0.004)",
           "vol(cargo_a32s, 0.002)",           "vol(cargo_j774, 0.002)",
           "vol(cargo_ki7z, 0.002)",           "vol(cargo_ptt4, 0.002)",
           "vol(cargo_uj65, 0.002)" ]
```

Listing 5.20: Proof of concept, querying the Prolog knowledge base

```
from pyswip import Prolog  # interface to SWI Prolog

prolog = Prolog()          # generate a new instance and
for fact in result:        # feed in the facts stored in variable result
    prolog.assertz(fact)

print list(prolog.query("above(cargo_6h20, X)"))
# [ "cargo_6tt2", "cargo_76ny", "cargo_8892" ]
```

5.5. Discussion

Although all entities within the evaluation were simulated and, thus, only synthetical data was applied, it was sufficient to highlight integral parts of the developed concept and its implementation. Nevertheless, it indicates that it is indeed possible to treat a smart environment as if it would be a database system, but it requires from all entities within the environment to store their data accordingly. Other reoccurring tasks can be solved with the help of simple SelectScript queries. The environment and its inhabitants as well as data might change, but the queries remain and will generate other results, depending on the environmental configuration. This enables an entity to operate in new surroundings, whereby it only has to keep its bunch of predefined queries — the results are generated automatically without further knowledge or intelligence. It thus contrasts the imperative way, in which a system would have to discover and identify relevant data and information manually and try to transform them into the desired representation.

Sensor data access, the earlier mentioned fusion and service capabilities have been left out, but the system so far can be used as a basis for various further extensions. Sensor measurements can only be interpreted and evaluated with additional context information and the system allows not only the access to this information freely but also theaddition of new ones. For example, the "service" of identifying and localizing a person with the help of existing cameras and laser scanners (mounted onto mobile robotic platforms) could be easily achieved, if current positions and orientations of sensor systems are known and if appropriate algorithms are applied:

```
1  SELECT position(this)
2    FROM environment
3   WHERE isPerson(this) and id(this) == "person of interest"
4     AS value;
```

Such extensions could also be hidden behind the query language too, similar to the abstraction of new representations. For example, the generation of a map required three intermediate steps: first the abstraction of a list of objects, which is then translated into a new "clone" environment, onto which the filters are applied.

Furthermore, any of the implementations can be replaced without affecting the entire system at all, such as the underlying database infrastructure, the communication interfaces (ROS), the simulation environment, applied filters, etc. The only thing that has to remain constant is the language that is used to assemble all of the underlying mechanism in order to extract the required information.

Another aspect, also mentioned earlier, is that the language itself can be used as some kind of lingua franca to translate between different world models the robotic systems might work with, since robots might possess their own and exclusive internal "simulation" of the outside world. The only thing required is a SelectScript interpreter on top of that simulation environment. The additional overhead caused by the interpreter in this case can be neglected, the Python interpreter is constructed with less than 500 lines of code, or the next version interpreter developed in C (and discussed within the next chapter) requires less than 32 kB program memory and can therefore also run on micro-controllers.

A generated environment model can be built and maintained locally, but the concept allows it also to perform all steps remotely, in such a way that an entity only has to send its request script to a certain instance that generates all results for it and sends them back in the desired format defined within the query. This allows conserving resources and executing computationally heavy tasks on a "proxy", while the actual robotic systems can get along

with only little computational power (and only with its required maps and information). Furthermore, this is also possible with registered callbacks, which allows to evaluate "situation" procedures externally and to retrieve a message, if a certain environmental state has changed.

6. Summary & Outlook

Concerning the main objectives of this thesis, which were stated within the first chapter:

*. . . develop a concept and mechanisms for declarative information gathering, which enable an entity to access any kind of information (**2.** & **3.**) in any kind of desired format (**4.**) from an instrumented environment, without the need for extraordinary intelligence (**1.**).*

The solution for this can be summarized in two sentences. A first, by making all data accessible by applying a cloud-based approach with a common hierarchy, from which required parts are translated into precise 3D rigid-body simulation of the environment. Secondly, from this simulation all required information is extracted or abstracted, whereby the required kind of information and desired formats are defined with the help of declarative query language.

Concerning the four sub-problems that were described in Sec. 1.4:

1. **Lack of Intelligence:** As it was mentioned earlier, none of the three conceptual layers requires some kind of intelligence nor does any system within a SmE. It is thus a solution for the problems of "Robotics Revolution" (cf. Sec. 1.3) and for the near future, which would even enable a robotic vacuum cleaner to access the information of its surroundings and does, therefore, not require systems, which were described within Sec. 1.2 ("Robotic Evolution").

2. **Holistic Access to Information . . . :** The declarative aspects of the developed query language and the systems below, which encapsulate system descriptions, online and historic data, as well as context enable a system to access the first four layers that were introduced in Chap. 2 (i. e., sources, data, information, and knowledge).

3. **. . . and Memory Externalization:** Instead of storing possibly relevant data locally on a system with its own format, the concept of the global world model allows the access to all kind of historic data, which different systems (producers) store for their own purposes. The only difficulty thereby is to enforce all systems to store their data according to a common standard and according to a global hierarchy.

4. **Reconfiguration and Representation:** The declarative solution allows defining what kind of information is required and also the desired format. The entire environment might change continuously and, thus, also affect the generation process of information itself, but all this is hidden behind an abstract automate that is responsible for generating the information.

All of this was accomplished by separating the problem solution into three distinct layers/steps: a global world model, a local environment model, and a declarative query language. Each of these layers provide a sub-concept and the developed implementations can be used by their own, whereby these prototypes provide interchangeable solutions. Interchangeable

in this context means that the applied database system or the simulation environment can be replaced by others.

Although Cassandra has been mentioned in the literature to be an ideal storage for sensor data and for future robotics memory management (cf. [214, 213, 212]), it was firstly introduced to robotics and SmEs with the developed systems in [12] and [10]. These systems allow to store and query any kind of data, whereby messages are dynamically translated from the ROS message format into a decomposed Cassandra format. Furthermore, the decentralized system was applied to implement a hierarchy with a graph-like structure to represent the global state of an environment, which also allows the addition of context (e. g., communication, location & time, quantities, etc.) to every "complex" element. It thus represents the global state of the environment. But this kind of representation is only useful as a global container; it cannot be used to react onto fast and occurring changes in the environment, nor does it allow the interpretation of different facts in different contexts.

Therefore, the concept of a local world model has been introduced, and defined as a precise co-simulation of a smaller part of the surrounding. It is a new idea to translate parts of the global world model into a scene-graph for a simulation environment, which can be updated in real-time and that is used to replay real behavior of real entities within a VM environment model. It enables different kinds of analyses and, thus, also querying and transformations, which goes far beyond the capabilities a database system (the global world model). These local environment models were set up on the basis of OpenRAVE, whereby different extensions have been provided to support various kinds of filtering, querying, and sensor integration. Of course, however, there is still a lot of work to do, which requires additional metrics, for completeness, confidence, quality, etc. (as it was discussed in Sec. 3.2.2.2).

All of this is enclosed by a newly developed multi-paradigm programming language called SelectScript. It does not only support declarative programming, but also structured programming (in an imperative way) and, if required, by means of support for functions, loops, and conditional jumps, and various different data types and structures. It is an embedded and extendable language that can be adapted to become applicable in various different problem domains. This language on top of the previously developed conceptual layers is the key element that allows the abstraction of a smart environment with all included and interconnected entities, similar to an ordinary database system, by applying the same metaphors. But also in this case, it goes beyond standard SQL-like approach, since it does not only support to access and query data in standard data types (i. e., boolean, numeric, string, arrays, tables), but also entities of a running simulation, for 3D models, or maps. And, as it was demonstrated in Sec. 4.3.3, it is also extendable in this case, allowing the definition of new request types for information. It has to be mentioned that the idea of accessing running simulations with the help of a database query language seems to be something that is totally new and has not been applied before. Furthermore, the implementation of recursive queries supports the handling of reasoning problems, which, due to the implementation of different strategies that are also extendable and tweakable, allows to solve such problems more efficiently than other declarative programming languages, such as Prolog that only implements one back-propagation algorithm.

The development of one future aspect has already started. In order to make a "frog" able to access and query any kind of "prey", the idea of a holistic query and programming language was extended onto the field of smaller embedded devices, allowing the treatment of any component of the environment as if it would be a database. Therefore, some aspects of the language have been rethought, such that the newest version of the language has also support for new data types such as sets, inline and list comprehension, exception handling,

loops, and it also includes functional aspects. The idea is not to bring another scripting language to this area of embedded devices; there already exist projects such as eLUA[1], NanoVM[2], PyMite[3], or Espurino[4], but all of them share similar programming principles and paradigms, in contrast to SelectScript. The new implementation of the compiler translates SelectScript code into a bytecode that can be executed on a SelectScript Virtual Machine (VM), which is implemented in C and also comprises a dynamic type system.

The implementation of a VM has various benefits in contrast to the previous directly interpreted version. A faulty script does not affect other programs running on other VMs. It enables a parallelization by running multiple VMs and it also enables a restricted assess to the underlying functionality for different users (roles). Functions are still developed in the host programming language and only linked to the VM, similarly as it was done in Python, but different functions can now also be added to different VMs.

Since SelectScript also has support for structured programming, entire programs can also be written, which can be used to control an embedded device. It enables us not only to update programs without flashing — which is accomplished simply by replacing scripts — but also to configure a system. Callbacks can be applied to support event handling, and different scheduling mechanisms can be applied in order to organize the execution of different scripts on different VMs depending on priority settings.

If it is applicable to embedded systems, it can also be applied to other, larger systems and, thus, really be used to implement some kind of lingua franca for a distributed smart system. And instead of subscribing to certain kinds of data/topics, SelectScripts could be used to define complex events, similar to the introduced callback mechanisms. By applying the global and local world model into the compilation and distribution of scripts, it should thus be possible to identify relevant nodes/entities and to shift to them the execution of parts of a script.

[1] embedded LUA: `http://www.eluaproject.net`

[2] embedded Java: `http://www.harbaum.org/till/nanovm/index.shtml`

[3] Python on a Chip: `https://wiki.python.org/moin/PyMite`

[4] JavaScript for Micro-controllers: `http://www.espruino.com`

A. Appendix

A.1. An Essay on Programming Paradigms

If we think of programming as a general concept for expressing problems in terms of an *Input*/*Output* system and by what is given, then three general paradigms can be differentiated. Interestingly, these three "methods of resolution" directly mirror the three concepts of logical reasoning, namely deduction, induction, and abduction (cf. [77]).

Imperative Programming & Deduction This first (and mostly applied) paradigm can be defined as:

$$Program(Input) \Rightarrow ?$$

Thereby, the *Input* data is known or given and the *Program* is defined in terms of transformation steps in order to generate the "unknown" *Output* data. Other paradigms such as structured, procedural, aspect-, feature-, and object-oriented programming can be associated primarily to this paradigm, since its main concern lies in the definition of the control flow. In other words, we are deducing the result, based on a set of rules and a given case.

Declarative Programming & Induction It is mostly assumed to be the opposite of the previous paradigm, as it can be defined as follows:

$$?(Input) \Rightarrow Output$$

It provides some form of notation or description of the *Input* and *Output*, whereby the question mark does not represent the abstract concept of a *Program* in terms of a control flow. But, instead, an abstract machine, reasoner, a (search) algorithm, etc. is applied that generates a result on the basis of the formal description of the *Input* and *Output*. It is thereby also wrong to consider *Input* & *Output* as some form of data only, as it rather defines both values and transformations, depending on the applied "concept". For example, in functional programming the *IO* is expressed in terms of mathematical notion of functions and parameters, in logical programmming as facts and rules, in "relational" programming as entities and relations, or in symbolic programing as symbols and formulas, etc. Thus, both of them can be generated, just as in Prolog it is possible to resolve on the basis of facts and rules, if a certain fact is true/false or under which circumstances it becomes true/false, but it is also possible to query for a set of successive rules or to derive new ones.

Thus, the same is true for induction, where a case and the results are known and the goal lies in the identification of rules, which are required to generate the result on the basis of the given case.

Probabilistic Programming & Abduction For the sake of completeness, the last combination that has been left out so far is the following, which has recently been denoted as the probabilistic programming paradigm:

$$Program(?) \Rightarrow Output$$

In this case, the *Program* cannot be considered as a strict set of transformation steps either, which are simply applied backwards in reverse order to get from a known *Output* to its *Input*. Instead, the *Output* here has to be considered as some form of observation and the *Program* as some kind of probabilistic model. This model can be a vague description of a system/calculation/control-flow by using background knowledge, beliefs and assumptions, possible interactions between elements or transitions between states, etc. Here, each of these aspects is described in terms of distributions and probabilities, which do not have to be defined in particular. Literally spoken, the *Program* itself is a huge sample generator for probability distributions that cannot be described mathematically. Hence, having a model and some observations allows drawing inferences about the *Input*. Thereby, *Input* does not necessarily define input data, but also model parameters, probability functions, etc.

The most prominent example is the language "Picture" [129], which was developed in 2015 at the Massachusetts Institute of Technology. 50 lines of Picture code that describe believes about human faces and projections were sufficient to solve the problem of inverse graphics, in which the most likely 3D models of human faces were inferred on the basis of given 2D photos (thus, the actual *Output*).

Surprisingly, this kind of problem solving directly matches abduction, where the result is known as well as the rules that might lead to it and the goal is to identify the original case. Compared to the other concepts, the vaguest "results" are generated with this method.

A.2. Reasoning about Filter-Sequences

Another aspect that has not been mentioned so far is concerned with the dynamic assess and combination filters and fusion functions of sensor data as it is described in the work of Tino Brade [41, 42]. He proposes a failure semantic for distributed sensing in order to encapsulate the individual failure characteristic. Based on the thesis of Sebastian Zug [231], which identified a set of 14 failure classes (outliers, offset, saturation, etc.), Brade had derived a mathematical model describing the effect of a component (sensor, filter, failure detector) on amplitude and occurrence probability related to each failure class. Similarly, the fault tolerance level of the application is individually defined. It enables a developer (at design-time) or a monitoring component (at runtime) to evaluate the applicability of a processing chain (i. e., filters, detectors) automatically. Based on the defined algebra, such processing changes could also be generated on demand with the help of simple SelectScript programs.

The following two snippets are only intended as examples to demonstrate the principle of how such sequences can be generated; it is not represented in the current research by Tino Brade and Sebastian Zug. The first assignment in Lis. A.1 defines a list with simple arithmetic expressions, which are used in the following `SELECT` query. The goal is to identify a sequence of applicable functions that transform start value `x`, defined with the `START WITH` expression along with the number of operations allowed, into a value as it is defined within the `GOAL` procedure. The `OPS` procedure list defines the applicable set of transformations, whereby the `x` value is substituted by the numerical values of `x` are calculated during the evaluation of the script. The first five results of that query are depicted within the comments below, with all intermediate operations and results.

Listing A.1: Dynamic generation of operation sequences, part 1

```
1  OPS = [[PROC: x+1.,   "+ 1"], [PROC: x-1.,   "- 1"], [PROC: x+x,     "+ x" ],
2         [PROC: -x,     "neg"], [PROC: x/2.,   "/ 2"], [PROC: sqrt(x), "sqrt"],
3         [PROC: sin(x), "sin"], [PROC: cos(x), "cos"], [PROC: tan(x),  "tan"]];
4
5  GOAL = PROC: this[0] < 0.5 AND this[0] > 0.4999;
6
7      SELECT this[1], this[0]
8        FROM OPS
9       WHERE GOAL
10
11 START WITH x = 22, step = 5
12 CONNECT BY NO CYCLE    # COST this[2]
13            x= this[0], step=step-1
14  STOP WITH step == 0   # get_time() - start_time >= .1
15
16      ORDER BY len(this) DESC
17      LIMIT 5
18         AS list;
19 #[['+ 1', 23.0, 'tan', 1.588153083, 'sin', 0.9998493752, '/ 2', 0.4999246876]
20 # ['+ x', 44, 'neg', -44, 'cos', 0.999843308647691, '/ 2', 0.499921654323845]
21 # ['+ x', 44, 'sin', 0.0177019251, 'cos', 0.9998433250, '/ 2', 0.49992166250]
22 # ['+ x', 44, 'cos', 0.9998433086, 'sqrt', 0.9999216512, '/ 2', 0.4999608256]
23 # ['+ x', 44, 'cos', 0.9998433086476912, '/ 2', 0.4999216543238456]]
```

As it is also depicted in the comments above, the script could be further tweaked in such a way that the `STOP WITH` expression could contain another condition that allows meeting real-time requirements. It stops the search, if too much time is consumed. As an additional element, cost values could be added as a third element to the operations list, indicating the computational effort. This value could be directly applied in the `COST` expression, which would search for cheap sequences first.

The goal condition was defined by a boundary, which is easier to match than an exact value. As depicted in the listing below, it can also be used to search for any kind of goal value. And as the result in the comments below reveals, there exists only one sequence of at max. 6 operations that can be used to generate a required result of 1.2345...

Listing A.2: Dynamic generation of operation sequences, part 2

```
...
GOAL = PROC: this[0] < 1.23456 AND this[0] >= 1.2345;
...
#[['- 1', 21.0, 'sqrt', 4.58257569495584, 'sin', -0.9915860810139135,
#  '- 1', -1.9915860810139, 'tan', 2.234536338627, '- 1', 1.234536338627]]
```

A.3. Publications

Fig. A.1 depicts all relevant publications I have written or co-authored. It can be read as a timeline, which shows two main areas of my research: Early projects were focused onto the field of middlewares and architectures for distributed systems as well as on fault/error/failure detection, which cover the lower layers of the hierarchy that was identified in Sec. 2.2.3. As it is further visible, there were other contrasting publications that dealt mostly with the representation of different kind of information and world models. Most of the research that was referenced in this thesis is depicted on the right part in Fig. A.1.

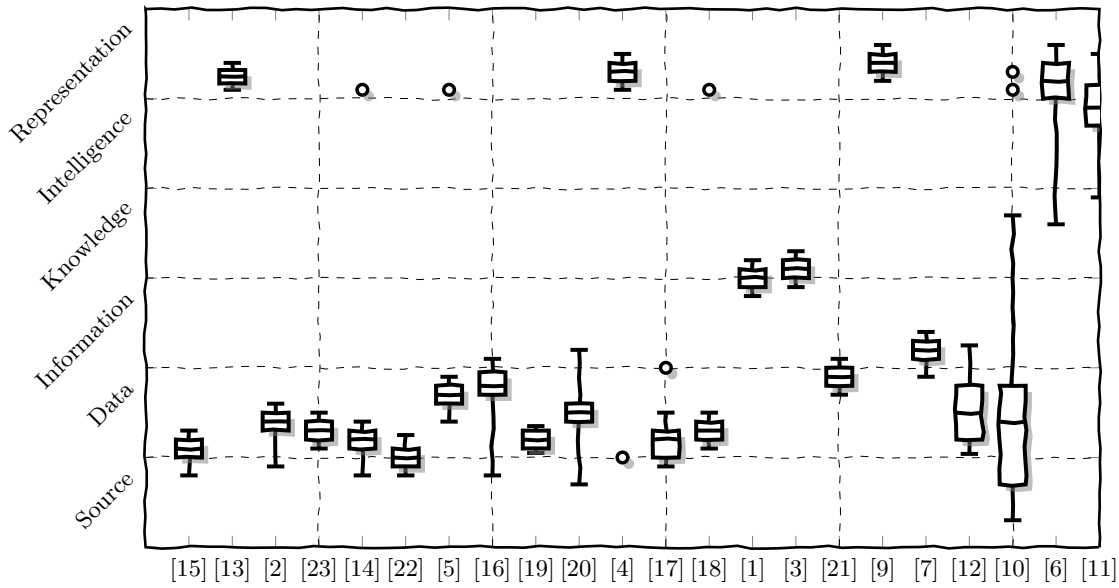


Figure A.1.: Synopsis of my own publications according to the presented literature review in Fig. 2.15 on page 60, the values of the x labels correspond to the reference identifier of the publication.

Bibliography

Publications

[1] André Dietrich. "Nutzung geometrischer Modelle zur Verbesserung der Umgebungswahrnehmung." In: *Tagungsband der 1. Doktorandentagung Magdeburger-Informatik Tage 2012 (MIT 2012)*. Ed. by Sebastian Zug Claudia Krull Eike Schallen. July 17, 2012, pp. 1–8.

[2] André Dietrich, Sebastian Zug, and Jörg Kaiser. "Detecting External Measurement Disturbances Based on Statistical Analysis for Smart Sensors." In: *Procedings of the IEEE International Symposium on Industrial Electronics (ISIE)*. Bari, Italy, July 1, 2010, pp. 2067–2072.

[3] André Dietrich, Sebastian Zug, and Jörg Kaiser. "Geometric Environment Modeling System." In: *7th IFAC Conference on Manufacturing Modelling, Management and Control*. International Federation of Automatic Control. Saint Petersburg, Russia, June 19, 2013, pp. 1429–1434.

[4] André Dietrich, Sebastian Zug, and Jörg Kaiser. "Model based Decoupling of Perception and Processing." In: *ERCIM/EWICS/Cyberphysical Systems Workshop, Resilient Systems, Robotics, Systems-of-Systems Challenges in Design, Validation & Verification and Certification*. Naples, Italy, Sept. 1, 2011.

[5] André Dietrich, Sebastian Zug, and Jörg Kaiser. "Modelbasierte Fehlerdetektion in verteilten Sensor-Aktor-Systemen." In: *11./12. Forschungskolloquium am Fraunhofer IFF*. Fraunhofer Institut für Fabrikbetrieb und Automatisierung (IFF), Jan. 1, 2011.

[6] André Dietrich, Sebastian Zug, and Jörg Kaiser. "SelectScript: A Query Language for Robotic World Models and Simulations." In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Seattle, Washington, May 26, 2015.

[7] André Dietrich, Sebastian Zug, and Jörg Kaiser. "The R in Robotics – rosR: A new Language Extension for the Robot Operating System." In: *The R Journal* 5.2 (Dec. 5, 2013), pp. 117–128. ISSN: 2073-4859.

[8] André Dietrich, Sebastian Zug, and Jörg Kaiser. "Towards Artificial Perception." In: *Proceddings of the SAFECOMP 2012, Third International Workshop on Digital Engineering (IWDE)*. Ed. by Springer-Verlag Berlin. Sept. 26, 2012, pp. 466–476.

[9] André Dietrich et al. "Application Driven Environment Representation." In: *The Seventh International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. Oct. 3, 2013.

[10] André Dietrich et al. "Distributed Management and Representation of Data and Context in Robotic Applications." In: *Proceedings of the IEEE/RSI International Conference on Intelligent Robobts and Systems (IROS)*. Chicago, Illinois, Sept. 14, 2014.

[11] André Dietrich et al. “Reasoning in complex environments with the SelectScript declarative language.” In: *IROS Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-15)*. Hamburg, Germany, 2015.

[12] André Dietrich et al. “ROS Meets Cassandra: Data Management in Smart Environments with NoSQL.” In: *Proc. of the 11th International Baltic Conference (Baltic DB&IS 2014)*. Tallinn, Estonia, June 8, 2014, pp. 43–54.

[13] André Dietrich et al. “Visualization of Robot’s Awareness and Perception.” In: *First International Workshop on Digital Engineering (IWDE)*. Magdeburg, Germany: ACM Press New York, NY, USA, June 14, 2010.

[14] Thomas Kiebel et al. “Identifying patients and visualize their vitality data through Augmented Reality.” In: *The 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS ’10)*. San Francisco, CA, USA, Aug. 12, 2010.

[15] Sebastian Zug and André Dietrich. “Examination of Fusion Result Feedback for Fault-Tolerant and Distributed Sensor Systems.” In: *IEEE International Workshop on Robotic and Sensors Environments (ROSE 2010)*. Phoenix, AZ, USA, Jan. 1, 2010.

[16] Sebastian Zug, André Dietrich, and Jörg Kaiser. “An Architecture for a Dependable Distributed Sensor System.” In: *IEEE Transactions on Instrumentation and Measurement* 60 Issue 2 (Feb. 1, 2011), pp. 408–419.

[17] Sebastian Zug, André Dietrich, and Jörg Kaiser. “Fault-Handling in Networked Sensor Systems.” In: *Fault Diagnosis in Robotic and Industrial Systems*. Ed. by Gerasimos Rigatos. St. Franklin, Australia: Concept Press Ltd., Jan. 1, 2012.

[18] Sebastian Zug et al. “Are laser scanners replaceable by Kinect sensors in robotic applications?” In: *2012 IEEE International Symposium on Robotic and Sensors Environments (ROSE 2012)*. IEEE. Magdeburg, Germany, Nov. 16, 2012, pp. 144–149.

[19] Sebastian Zug et al. “Design and Implementation of a Small Size Robot Localization System.” In: *IEEE International Symposium on Robotic and Sensors Environments (ROSE 2011)*. Montreal, Quebec, Canada, Sept. 1, 2011.

[20] Sebastian Zug et al. “Flexible Daten-Akquisition & Interpretation für verteilte Sensor-Aktor-Systeme im Produktionsumfeld.” In: *10. Magdeburger Maschinentage*. Sept. 1, 2011.

[21] Sebastian Zug et al. “Phase optimization for control/fusion applications in dynamically composed sensor networks.” In: *IEEE International Symposium on Robotic and Sensors Environments (ROSE 2013)*. Washington D.C., USA, USA, Aug. 1, 2013.

[22] Sebastian Zug et al. “Programming abstractions and middleware for building control systems as networks of smart sensors and actuators.” In: *Proceedings of Emerging Technologies in Factory Automation (ETFA ’10)*. Bilbao, Spain, Sept. 13, 2010.

[23] Sebastian Zug et al. “Reliable Fault-Tolerant Sensors for Distributed Systems.” In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS ’10)*. Cambridge, United Kingdom: ACM Press New York, NY, USA, July 12, 2010, pp. 105–106.

Bibliography

[24] Gregory D Abowd et al. “Towards a Better Understanding of Context and Context-Awareness.” In: *Handheld and ubiquitous computing*. Springer. 1999, pp. 304–307.

[25] Franz Andert and Florian Adolf. “Online world modeling and path planning for an unmanned helicopter.” In: *Autonomous Robots* 27.3 (2009), pp. 147–164.

[26] Torsten Andre, Daniel Neuhold, and Christian Bettstetter. “Coordinated Multi-Robot Exploration: Out of the Box Packages for ROS.” In: *Proc. Intern. Workshop on Wireless Networking and Control for Unmanned Autonomous Vehicles (Wi-UAV)*. 2014.

[27] Elli Angelopoulou, Tsai-Hong Hong, and Angela Y Wu. “World Model Representations for Mobile Robots.” In: *Proc. of the Intelligent Vehicles Symposium (IV)*. IEEE. 1992, pp. 293–297.

[28] *Apache CassandraTM 1.1 Documentation - DataStax*. Online available at `http://www.datastax.com/doc-source/pdf/cassandra11.pdf`. Apache.

[29] Ofer Arazy and Rick Kopak. “On the measurability of information quality.” In: *Journal of the American Society for Information Science and Technology* 62.1 (2011), pp. 89–99.

[30] Rajesh Arumugam et al. “DAvinCi: A cloud computing framework for service robots.” In: *International Conference on Robotics and Automation (ICRA)*. IEEE. 2010, pp. 3084–3089.

[31] L. Atzori, A. Iera, and G. Morabito. “The Internet of Things: A survey.” In: *Computer Networks* 54.15 (2010), pp. 2787–2805.

[32] Juan C Augusto et al. “Intelligent environments: a manifesto.” In: *Human-Centric Computing and Information Sciences* 3.1 (2013), pp. 1–18.

[33] Max Bajracharya, Mark W Maimone, and Daniel Helmick. “Autonomy for Mars Rovers: Past, Present, and Future.” In: *Computer* 41.12 (2008), pp. 44–50.

[34] Carlo Batini and Monica Scannapieco, eds. *Data Quality: Concepts, Methodologies and Techniques*. Springer Berlin, 2006.

[35] Andrey Belkin et al. “World Modeling for Autonomous Systems.” In: *Innovative Information Systems Modelling Techniques* 1 (2012), pp. 135–158.

[36] Maren Bennewitz et al. “Fritz-A humanoid communication robot.” In: *Proc. of the 16th International Symposium on Robot and Human interactive Communication*. IEEE. 2007, pp. 1072–1077.

[37] Sebastian Blumenthal et al. “A Scene Graph Based Shared 3D World Model for Robotic Applications.” In: *Proc. of the International Conference on Robotics and Automation (ICRA)*. IEEE. 2013, pp. 453–460.

[38] Sebastian Blumenthal et al. “An approach for a distributed world model with QoS-based perception algorithm adaptation.” In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2015, pp. 1806–1811.

[39] Robert Bogue. “The role of robotics in non-destructive testing.” In: *Industrial Robot: An International Journal* 37.5 (2010), pp. 421–426.

[40] Harold Borko. “Information science: what is it?” In: *American documentation* 19.1 (1968), pp. 3–5.

[41] Tino Brade, Jörg Kaiser, and Sebastian Zug. “Expressing validity estimates in smart sensor applications.” In: *ARCS 2013 - 26th International Conference on Architecture of Computing Systems 2013, Workshop Proceedings, Feburary 19-22, 2013, Prague, Czech Republic*. 2013.

[42] Tino Brade et al. “Sensor- and Environment Dependent Performance Adaptation for Maintaining Safety Requirements.” In: *Computer Safety, Reliability, and Security - SAFECOMP 2014 Workshops: ASCoMS, DECSoS, DEVVARTS, ISSE, ReSA4CI, SASSUR. Florence, Italy, September 8-9, 2014. Proceedings*. 2014, pp. 46–54.

[43] Mathias Broxvall, Beom-Su Seo, and WooYoung Kwon. “The PEIS Kernel: A Middleware for Ubiquitous Robotics.” In: *In Proc. of the IROS-07 Workshop on Ubiquitous Robotic Space Design and Applications*. 2007, pp. 212–218.

[44] M. Broxvall et al. “PEIS ecology: Integrating robots into smart environments.” In: *Proc. of the International Conference on Robotics and Automation (ICRA)*. IEEE. 2006, pp. 212–218.

[45] John A Campbell, ed. *Implementation of PROLOG*. Chichester, UK: Ellis Horwood, 1984.

[46] Stefano Ceri, Georg Gottlob, and Letizia Tanca. “What you always wanted to know about Datalog (and never dared to ask).” In: *IEEE Transactions on Knowledge and Data Engineering* 1.1 (1989), pp. 146–166.

[47] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A STRUCTURED ENGLISH QUERY LANGUAGE.” In: *Proc. of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. 1974, pp. 249–264.

[48] Veronique Cherfaoui and Franck Davoine. *Project PREDIMAP - vehicle Perception and Reasoning Enhanced with Digital Maps*. Tech. rep. ICT-ASIA PREDiMap project, 2013.

[49] Abdelghani Chibani et al. “Ubiquitous robotics: Recent challenges and future trends.” In: *Robotics and Autonomous Systems* 61.11 (2013), pp. 1162–1172.

[50] Eun-Sun Cho and Kang-Woo Lee. “Security Checks in Programming Languages for Ubiquitous Environments.” In: *Proc. of Workshop on Pervasive, Security, Privacy and Trust*. 2004.

[51] Eun-Sun Cho et al. “Scenario-based Programming for Ubiquitous Applications.” In: *Ubiquitous Computing Systems*. Springer, 2006, pp. 286–299.

[52] Keith L. Clark and Frank G. McCabe. *PROLOG: a language for implementing expert systems*. Tech. rep. Imperial College of Science and Technology Department of Computing, 1980.

[53] Diane J Cook et al. “MavHome: An Agent-Based Smart Home.” In: *Proc. of the 1. Annual Conference on Pervasive Computing and Communications (PerCom)*. IEEE. 2003, pp. 521–524.

[54] D.J. Cook and S.K. Das. “How smart are our environments? An updated look at the state of the art.” In: *Pervasive and Mobile Computing* 3.2 (2007), pp. 53–73.

[55] Michael A. Covington. *Natural Language Processing for Prolog Programmers*. Prentice Hall, 1994.

[56] Sajal K Das et al. "The role of prediction algorithms in the MavHome smart home architecture." In: *Wireless Communications, IEEE* 9.6 (2002), pp. 77–84.

[57] G. De Giacomo, Y. Lespérance, and H.J. Levesque. "ConGolog, a concurrent programming language based on the situation calculus." In: *Artificial Intelligence* 121.1 (2000), pp. 109–169.

[58] Giuseppe De Giacomo et al. "IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents." In: *Multi-Agent Programming:* Springer, 2009, pp. 31–72.

[59] S. R. Deb. "Robotics Technology and Flexible Automation." In: Tata McGraw-Hill Publishing Company Limited, 1994. Chap. Robot Languages and Programming.

[60] A.K. Dey, D. Salber, and G.D. Abowd. *A Context-based Infrastructure for Smart Environments*. Tech. rep. Georgia Institute of Technology, 1999.

[61] Anind K Dey, Gregory D Abowd, and Daniel Salber. "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications." In: *Human-computer interaction* 16.2 (2001), pp. 97–166.

[62] Rosen Diankov. "Automated Construction of Robotic Manipulation Programs." PhD thesis. Carnegie Mellon University, Robotics Institute, Aug. 2010.

[63] Rosen Diankov, Ryohei Ueda, and Kei Okada. "COLLADA: An Open Standard for Robot File Formats." In: *Proceedings of the 29th Annual Conference of the Robotics Society of Japan, AC2Q1–5*. 2011.

[64] Sergey Dmitriev. "Language Oriented Programming: The Next Programming Paradigm." In: *onBoard - Electronic Magazine* (2004), pp. 1–14.

[65] Marco Dorigo et al. "Swarmanoid." In: *IEEE Robotics & Automation Magazine* 1070.9932/13 (2013).

[66] Zhihui Du et al. "Design of a robot cloud center." In: *10th International Symposium on Autonomous Decentralized Systems (ISADS)*. IEEE. 2011, pp. 269–275.

[67] Christof Ebert and Capers Jones. "Embedded Software: Facts, Figures, and Future." In: *IEEE Computer* 42.4 (2009), pp. 42–52.

[68] Richard Edwards, Lynne E Parker, and David R Resseguie. "Robopedia: Leveraging Sensorpedia for Web-Enabled Robot Control." In: *International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. IEEE. 2010, pp. 183–188.

[69] Andrew Eisenberg and Jim Melton. "SQL: 1999, formerly known as SQL3." In: *ACM Sigmod record* 28.1 (1999), pp. 131–138.

[70] Alberto Elfes. "Occupancy Grids: A Stochastic Spatial Representation for Active Robot Perception." In: *Proc. 6th Conference on Uncertainty in AI*. 1990, pp. 60–70.

[71] *ETSI TR 102 863 V1.1.1: Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Local Dynamic Map (LDM); Rationale for and guidance on standardization*. Tech. rep. ETSI - European Telecomunication Standards Institute, 2011.

[72] Richard Fateman. "Haddock's eyes and computer algebra systems Some essays." In: *University of California, Berkeley* (2005).

[73] Alexander Ferrein and Gerhard Lakemeyer. "Logic-based Robot Control in Highly Dynamic Domains." In: *Robotics and Autonomous Systems* 56.11 (2008), pp. 980–991.

[74] Gabriele Ferri et al. "DustCart, a Mobile Robot for Urban Environments: Experiments of Pollution Monitoring and Mapping during Autonomous Navigation in Urban Scenarios." In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2010) Workshop on Networked and Mobile Robot Olfaction in Natural, Dynamic Environments*. 2010.

[75] Richard E Fikes and Nils J Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving." In: *Artificial intelligence* 2.3 (1972), pp. 189–208.

[76] Joseph Fisher, Paolo Faraboschi, and Clifford Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[77] Peter A Flach and Antonis C Kakas. "Abductive and inductive reasoning: background and issues." In: *Abduction and Induction*. Springer, 2000, pp. 1–27.

[78] Anita M. Flynn. *Redundant Sensors for Mobile Robot Navigation*. Tech. rep. Massachusetts Institute of Technology, 1985.

[79] Simon Forge and Colin Blackman. *A Helping Hand for Europe: The Competitive Outlook for the EU Robotics Industry*. Tech. rep. European Commission, Joint Research Centre, Institute for Prospective Technological Studies, 2010.

[80] Eckhard Freund and Dirk H Pensky. "COSIMIR® factory: extending the use of manufacturing simulations." In: *Proc. of the International Conference on Robotics and Automation (ICRA)*. Vol. 3. IEEE. 2002, pp. 2805–2810.

[81] Chiara Fulgenzi, Anne Spalanzani, and Christian Laugier. "Dynamic obstacle avoidance in uncertain environment combining PVOs and occupancy grid." In: *Proc. of the International Conference on Robotics and Automation (ICRA)*. IEEE. 2007, pp. 1610–1616.

[82] Andrei Furda and Ljubo Vlacic. "An object-oriented design of a world model for autonomous city vehicles." In: *Proc. of the Intelligent Vehicles Symposium (IV)*. IEEE. 2010.

[83] Shuzhi Sam Ge and Yan Juan Cui. "New potential functions for mobile robot path planning." In: *IEEE Transactions on robotics and automation* 16.5 (2000), pp. 615–620.

[84] Brian Gerkey, Richard T Vaughan, and Andrew Howard. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems." In: *Proceedings of the 11th international conference on advanced robotics*. Vol. 1. 2003, pp. 317–323.

[85] Seok-Jo Go, Min-Kyu Park, and Young-Jin Lee. "Development of Smart Actuator and Its Application." In: *Intelligent Robotics and Applications*. Springer, 2013, pp. 442–449.

[86] Bryan L Gorman, DR Resseguie, and Christopher Tomkins-Tinch. "Sensorpedia: Information sharing across incompatible sensor systems." In: *International Symposium on Collaborative Technologies and Systems (CTS'09)*. IEEE. 2009, pp. 448–454.

[87] Bryan Gorman and David Resseguie. *Final Report: Sensorpedia Phase 3*. Tech. rep. Oak Ridge National Laboratory, 2010.

[88] Birgit Graf, Rolf Dieter Schraft, and Jens Neugebauer. "A Mobile Robot Platform for Assistance and Entertainment." In: *International Symposium on Robotics*. Citeseer. Montreal, Canada, 2000.

[89] David L Hall and James Llinas. "An introduction to multisensor data fusion." In: *Proceedings of the IEEE* 85.1 (1997), pp. 6–23.

[90] K. Hauser. "Robust Contact Generation for Robot Simulation with Unstructured Meshes." In: *International Symposium on Robotics Research*. 2013.

[91] Sumi Helal et al. "Experience of enhancing the space sensing of networked robots using atlas service-oriented architecture." In: *Computer-Human Interaction*. Springer. 2008, pp. 1–10.

[92] Mario Hermann, Tobias Pentek, and Boris Otto. *Design Principles for Industrie 4.0 Scenarios: A Literature Review*. Tech. rep. Technical University Dortmund, Fakulty of Mechanical Engineering, 2015.

[93] André Herms, Georg Lukas, and Svilen Ivanov. "Realism in Design and Evaluation of Wireless Routing Protocols." In: *MSPE*. Citeseer. 2006, pp. 57–70.

[94] Nico Hochgeschwender et al. "Context-based Selection and Execution of Robot Perception Graphs." In: *In Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2015.

[95] Nico Hochgeschwender et al. "Declarative Specification of Robot Perception Architectures." In: *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 291–302.

[96] Nico Hochgeschwender et al. "Towards a robot perception specification language." In: *IROS Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-13)*. 2013.

[97] Owen Holland. "Exploration and high adventure: the legacy of Grey Walter." In: *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 361.1811 (2003), pp. 2085–2121.

[98] Chung-Seong Hong et al. "Context Modeling and Reasoning Approach in Context-Aware Middleware for URC System." In: *International Journal of Mathematical, Physical and Engineering Sciences* 1.4 (2007), pp. 208–212.

[99] Olivier Houdé et al. *Dictionary of cognitive science: neuroscience, psychology, artificial intelligence, linguistics, and philosophy*. Routledge, 2004.

[100] Javier Ibanez-Guzman et al. "Vehicle to Vehicle communications applied to Road Intersection Safety, Field Results." In: *13th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2010, pp. 192–197.

[101] *IEEE Standard for a Smart Transducer Interface for Sensors and Actuators (IEEE 1451.2)*. 2007.

[102] Vikramaditya R Jakkula, Aaron S Crandall, and Diane J Cook. "Enhancing anomaly detection using temporal pattern discovery." In: *Advanced intelligent environments*. Springer, 2009, pp. 175–194.

[103] Vikramaditya Jakkula and Diane J Cook. "Using temporal relations in smart environment data for activity prediction." In: *Proceedings of the 24th International conference on machine learning*. 2007, pp. 20–24.

[104] Nory Afzan Mohd Johari, Habibollah Haron, and Abdul Syukor Mohamad Jaya. "Robotic modeling and simulation of palletizer robot using Workspace5." In: *Computer Graphics, Imaging and Visualisation (CGIV)*. IEEE. 2007, pp. 217–222.

[105] Sanem Kabadayi, Adam Pridgen, and Christine Julien. "Virtual Sensors: Abstracting Data from Physical Sensors." In: *Proc. of the International Symposium on World of Wireless, Mobile and Multimedia Networks*. IEEE Computer Society. 2006, pp. 587–592.

[106] Sanem Kabadayi et al. "Virtual sensors: a demonstration." In: *The 26th international conference on computer communications: demonstrations track (Infocom)*. 2007, pp. 10–12.

[107] Nidhi Kalra, Dave Ferguson, and Anthony Stentz. "Incremental Reconstruction of Generalized Voronoi Diagrams on Grids." In: *Proceedings of the International Conference on Intelligent Autonomous Systems*. Vol. 57. 2. Elsevier, 2006, pp. 123–128.

[108] Koji Kamei et al. "Cloud networked robotics." In: *IEEE Network* 26.3 (2012), pp. 28–34.

[109] Lars Karlsson. "Conditional progressive planning under uncertainty." In: *International Joint Conference on Artificial Intelligence (IJCAI)*. Citeseer. 2001, pp. 431–438.

[110] Hyun Kim et al. "Context-aware server framework for network-based service robots." In: *International Joint Conference SICE-ICASE*. IEEE. 2006, pp. 2084–2089.

[111] Jong-Hwan Kim. "Ubiquitous Robot." In: *Computational Intelligence, Theory and Applications*. Springer, 2005, pp. 451–459.

[112] Jong-Hwan Kim. "Ubiquitous Robot: Recent Progress and Development." In: *International Joint Conference SICE-ICASE*. IEEE. 2006, pp. I-25–I-30.

[113] Jong-Hwan Kim, Yong-Duk Kim, and Kang-Hee Lee. "The Third Generation of Robotics: Ubiquitous Robot." In: *Proc. of the 2nd International Conference on Autonomous Robots and Agents*. 2004.

[114] Jong-Hwan Kim et al. "Multi-Layer Architecture of Ubiquitous Robot System for Integrated Services." In: *International Journal of Social Robotics* 1.1 (2009), pp. 19–28.

[115] Jong-Hwan Kim et al. "Ubiquitous Robot: A New Paradigm for Integrated Services." In: *International Conference on Robotics and Automation*. IEEE. 2007, pp. 2853–2858.

[116] Jeffrey King et al. "Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces." In: *31st Conference on local computer networks*. IEEE. 2006, pp. 630–638.

[117] NV Kirianaki et al. "Data Acquisition and Signal Processing for Smart Sensors." In: *Measurement Science and Technology* 13.9 (2002), p. 1501.

[118] Michael Knappmeyer et al. "ContextML: a light-weight context representation and context management schema." In: *5th International Symposium on Wireless Pervasive Computing (ISWPC)*. IEEE. 2010, pp. 367–372.

[119] Nathan Koenig and Andrew Howard. "Design and use paradigms for gazebo, an open-source multi-robot simulator." In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. Vol. 3. IEEE. 2004, pp. 2149–2154.

[120] Teun Koetsier. "On the prehistory of programmable machines: musical automata, looms, calculators." In: *Mechanism and Machine theory* 36.5 (2001), pp. 589–603.

[121] Kurt Konolige, Eitan Marder-Eppstein, and Bhaskara Marthi. "Navigation in Hybrid Metric–Topological Maps." In: *Proc. of the International Conference on Robotics and Automation (ICRA)*. IEEE. 2011, pp. 3041–3047.

[122] Gerd Kortuem et al. "Smart Objects as Building Blocks for the Internet of Things." In: *Internet Computing, IEEE* 14.1 (2010), pp. 44–51.

[123] Gerhard K Kraetzschmar, Guillem Pages Gassull, and Klaus Uhl. "Probabilistic quadtrees for variable-resolution mapping of large environments." In: *Proceedings of the 5th IFAC/EURON symposium on intelligent autonomous vehicles*. Citeseer. 2004.

[124] J. Kramer and M. Scheutz. "Development Environments for Autonomous Mobile Robots: A Survey." In: *Autonomous Robots* 22 (2 Feb. 2007), pp. 101–132.

[125] Mark Kranz and Rachel Cardell-Oliver. "SENSID: a situation detector for sensor networks." In: *ACM Journal Name* 3.3 (2007), pp. 1–21.

[126] Matthias Kranz et al. "A Player/Stage System for Context-Aware Intelligent Environments." In: *Proceedings of UbiSys* 6 (2006), pp. 17–21.

[127] Benjamin Kuipers and Yung-Tai Byun. "A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations." In: *Robotics and Autonomous Systems* 8 (1991), pp. 47–63.

[128] Benjamin Kuipers et al. "Local Metrical and Global Topological Maps in the Hybrid Spatial Semantic Hierarchy." In: *Proc. of the International Conference on Robotics and Automation (ICRA)*. Vol. 5. IEEE. 2004, pp. 4845–4851.

[129] Tejas Dattatraya Kulkarni et al. "Picture: A probabilistic programming language for scene perception." In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2015, pp. 4390–4399.

[130] Achim Kuwertz. *Towards Adaptive Open-World Modeling*. Tech. rep. Vision and Fusion Laboratory, Institute for Anthropomatics, Karlsruhe Institute of Technology (KIT), 2012, pp. 139–161.

[131] Chin-Feng Lai et al. "OSGi-based services architecture for Cyber-Physical Home Control Systems." In: *Computer Communications* 34.2 (2011), pp. 184–191.

[132] Avinash Lakshman and Prashant Malik. "Cassandra - A decentralized structured storage system." In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[133] Boris Lau, Christoph Sprunk, and Wolfram Burgard. "Improved updating of Euclidean distance maps and Voronoi diagrams." In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ. 2010, pp. 281–286.

[134] Kevin LeBlanc and Alessandro Saffiotti. "Issues of Perceptual Anchoring in Ubiquitous Robotic Systems." In: *Proc of the ICRA-07 Workshop on Omniscient Space*. IEEE. 2007.

[135] Byung S Lee and Ron Musick. "MeshSQL: the query language for simulation mesh data." In: *Information Sciences* 159.3 (2004), pp. 177–202.

[136] Choonhwa Lee, David Nordstedt, and Sumi Helal. "Enabling Smart Spaces with OSGi." In: *Pervasive Computing, IEEE* 2.3 (2003), pp. 89–94.

[137] Edward A Lee. "Cyber physical systems: Design challenges." In: *11th International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*. IEEE. 2008, pp. 363–369.

[138] Kang Lee. "IEEE 1451: A standard in support of smart transducer networking." In: *Proc. 17th IEEE Instrumentation and Measurement Technology Conference IMTC 2000*. Vol. 2. 2000, 525–528 vol.2.

[139] Jacques C Leedekerken, Maurice F Fallon, and John J Leonard. "Mapping complex marine environments with autonomous surface craft." In: *Experimental Robotics*. Springer. 2014, pp. 525–539.

[140] Jerry Y Lettvin et al. "What the Frog's Eye tells the Frog's Brain." In: *Proceedings of the IRE* 47.11 (1959), pp. 1940–1951.

[141] Hector J Levesque and Maurice Pagnucco. "Legolog: Inexpensive Experiments in Cognitive Robotics." In: *Cognitive Robotics Workshop at ECAI*. 2000, pp. 104–109.

[142] H.J. Levesque et al. "GOLOG: A logic programming language for dynamic domains." In: *The Journal of Logic Programming* 19.1-3 (1994), pp. 59–83.

[143] Deyi Li and Yi Du. "Artificial Intelligence with Uncertainty." In: Chapman & Hall/CRC, 2008. Chap. The 50-year History of Artificial Intelligence, pp. 1–20.

[144] Florian Lindörfer. *Semantic Web Frameworks*. Tech. rep. Distributed Information Systems - University of Basel, 2010.

[145] James Llinas et al. *Revisiting the JDL data fusion model II*. Tech. rep. DTIC Document, 2004.

[146] Robert Lundh, Lars Karlsson, and Alessandro Saffiotti. "Dynamic self-configuration of an ecology of robots." In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ. 2007, pp. 3403–3409.

[147] Sam Madden, Joe Hellerstein, and Wei Hong. "TinyDB: In-Network Query Processing in TinyOS." In: *Intel Research, IRB-TR-02-2014* (2003).

[148] S.R. Madden et al. "TinyDB: An acquisitional query processing system for sensor networks." In: *ACM Transactions on Database Systems (TODS)* 30.1 (2005), pp. 122–173.

[149] D. Malayeri and J. Aldrich. *CZ: Multiple Inheritance Without Diamonds*. Tech. rep. DTIC Document, 2008.

[150] Fulvio Mastrogiovanni, Antonio Sgorbissa, and Renato Zaccaria. "Extending the Capabilities of Mobile Robots through Knowledge Ecosystems." In: *International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*. IEEE. 2007, pp. 155–160.

[151] Fulvio Mastrogiovanni, Antonio Sgorbissa, and Renato Zaccaria. "From Autonomous Robots to Artificial Ecosystems." In: *Handbook of Ambient Intelligence and Smart Environments*. Ed. by Hideyuki Nakashima, Hamid Aghajan, and Juan Carlos Augusto. Springer, 2010. Chap. From Autonomous Robots to Artificial Ecosystems, pp. 635–668.

[152] John McCarthy. *Situations, Actions, and Causal Laws*. Tech. rep. Stanford University Artificial Intelligence Project, 1963.

[153] Gerard CM Meijer, Cornelis Maria Meijer, and Cornelis Maria Meijer. *Smart sensor systems*. Wiley Online Library, 2008.

[154] Gajamohan Mohanarajah et al. "Rapyuta: A Cloud Robotics Platform." In: *IEEE Transactions on Automation Science and Engineering* 12.2 (2015), pp. 481–493.

[155] Aekyung Moon et al. "Context-Aware Active Services in Ubiquitous Computing Environments." In: *ETRI journal* 29.2 (2007), pp. 169–178.

[156] Hans P Moravec and Alberto Elfes. *Autonomous Mobile Robots Annual Report - Sonar Mapping, Imaging and Navigation*. Tech. rep. Pittsburgh, Pennsylvania 15213: The Robotics Institute - Carnegie Mellon University, 1985, pp. 116–121.

[157] Hideyuki Nakashima, Hamid Aghajan, and Juan Carlos Augusto, eds. *Handbook of ambient intelligence and smart environments*. Springer Science & Business Media, 2009.

[158] Alwar Narayanan. "Next generation map making: automation from mobile data collection." In: *Proc. of the international conference on Multimedia information retrieval*. ACM. 2010, pp. 7–8.

[159] Nils J. Nilsson. *A Mobile Automaton: An Application of Artificial Intelligence Techniques*. Tech. rep. SRI International - Artificial Intelligence Center, 1969.

[160] Nils J. Nilsson. *SHAKEY THE ROBOT*. Tech. rep. SRI International - Artificial Intelligence Center, 1984.

[161] Shuichi Nishio, Koji Kamei, and Norihiro Hagita. "Ubiquitous Network Robot Platform for Realizing Integrated Robotic Applications." In: *Intelligent Autonomous Systems 12*. Springer, 2013, pp. 477–484.

[162] Andress Nüchter et al. "6D SLAM with an Application in Autonomous Mine Mapping." In: *Proc. of the International Conference on Robotics and Automation (ICRA)*. Vol. 2. IEEE. 2004, pp. 1998–2003.

[163] W O'Brien et al. "An architecture for decision support in ad hoc sensor networks." In: *Electronic Journal of Information Technology in Construction* 14 (2009), pp. 309–327.

[164] *OGC SensorML: Model and XML Encoding Standard*. Open Geospatial Consortium Inc.

[165] *OpenGIS Sensor Model Language (SensorML) Implementation Specification (Version 1.0.0)*. Open Geospatial Consortium Inc.

[166] *OpenGIS Transducer Markup Language Implementation Specification*. Open Geospatial Consortium Inc.

[167] Dae-Hyung Park, Peter Pastor, Stefan Schaal, et al. "Movement reproduction and obstacle avoidance with dynamic movement primitives and potential fields." In: *8th International Conference on Humanoid Robots*. IEEE. 2008, pp. 91–98.

[168] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[169] Maurizio Piaggio, Antonio Sgorbissa, and Renato Zaccaria. "Programming real time distributed multiple robotic systems." In: *RoboCup-99: Robot Soccer World Cup III*. Springer, 2000, pp. 412–423.

[170] Paolo Pialorsi and Marco Russo. *Introducing Microsoft® LINQ*. Microsoft Press, 2007.

[171] Carlo Pinciroli et al. "ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems." In: *Swarm Intelligence* 6.4 (2012), pp. 271–295.

[172] George Pór. "Designing Knowledge Ecosystems for Communities of Practice." In: *Conference on Advancing Organizational Capability Via Knowledge Management*. Los Angeles, USA, 1997.

[173] Piotr Przymus et al. "Recursive query facilities in relational databases: a survey." In: *Database Theory and Application, Bio-Science and Bio-Technology*. Springer, 2010, pp. 89–99.

[174] Hung Keng Pung et al. "Context-aware middleware for pervasive elderly homecare." In: *IEEE Journal on Selected Areas in Communications* 27.4 (2009), pp. 510–524.

[175] M. Quigley et al. "ROS: an open-source Robot Operating System." In: *ICRA Workshop on Open Source Software*. Vol. 3. 3.2. 2009.

[176] J Quintas, P Menezes, and J Dias. "Cloud Robotics: Towards Context Aware Robotic Networks." In: *Proc. of the 16th International Conference on Robotics*. Pittsburgh, USA, 2011, pp. 420–427.

[177] Robert G Raskin and Michael J Pan. "Knowledge representation in the semantic web for Earth and environmental terminology (SWEET)." In: *Computers & geosciences* 31.9 (2005), pp. 1119–1125.

[178] Matteo Reggente et al. "The DustBot System: Using Mobile Robots to Monitor Pollution in Pedestrian Area." In: *CHEMICAL ENGINEERING TRANSACTIONS* 23 (2010).

[179] Gitta Rohling. "Facts and Forecasts: Boom for Learning Systems." In: *Pictures of the Future: The Magazine for Research and Innovation* (2014).

[180] Eric Rohmer, Surya PN Singh, and Marc Freese. "V-REP: a Versatile and Scalable Robot Simulation Framework." In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ. 2013, pp. 1321–1326.

[181] Maayan Roth, Douglas Vail, and Manuela Veloso. "A World Model for Multi-Robot Teams with Communication." In: *Proc. of International Conference on Intelligent Robots and Systems (IROS)*. Vol. 3. IEEE. 2003, pp. 2494–2499.

[182] D. Roy. "Grounding words in perception and action: computational insights." In: *Trends in Cognitive Sciences* 9.8 (2005), pp. 389–396.

[183] D. Roy, K.Y. Hsiao, and N. Mavridis. "Mental Imagery for a Conversational Robot." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 34.3 (2004), pp. 1374–1383.

[184] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)." In: *International Conference on Robotics and Automation (ICRA)*. Shanghai, China, 2011 2011.

[185] R.B. Rusu et al. "Model-based and learned semantic object labeling in 3D point cloud maps of kitchen environments." In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ. 2009, pp. 3601–3608.

[186] R.B. Rusu et al. "Towards 3D Point cloud based object maps for household environments." In: *Robotics and Autonomous Systems* 56.11 (2008), pp. 927–941.

[187] A. Saffiotti et al. "The PEIS-ecology project: a progress report." In: *Proc. of the ICRA-07 Workshop on Network Robot Systems*. IEEE. Rome, Italy, 2007, pp. 16–22.

[188] Daniel Salber, Anind K Dey, and Gregory D Abowd. "The Context Toolkit: Aiding the Development of Context-Enabled Applications." In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM. 1999, pp. 434–441.

[189] Peter H. Salus. *A Quarter Century of Unix*. Addison-Wesley Longman, 1994.

[190] Pericle Salvini et al. "The Robot DustCart." In: *Robotics & Automation Magazine* 18.1 (2011), pp. 59–67.

[191] Alberto Sanfeliu, Norihiro Hagita, and Alessandro Saffiotti. "Network Robot Systems." In: *Robotics and Autonomous Systems* 56.10 (2008), pp. 793–797.

[192] Erik Schulenburg et al. "LiSA: A Robot Assistant for Life Sciences." In: *KI 2007: Advances in Artificial Intelligence*. Springer, 2007, pp. 502–505.

[193] Michael Schulze. "Adaptierbare ereignisbasierte Middleware für ressourcenbeschränkte Systeme." PhD thesis. Otto-von-Guericke Universität Magdeburg, 2011.

[194] Anjum Shehzad et al. "Formal Modeling in Context Aware Systems." In: *Proceedings of the First International Workshop on Modeling and Retrieval of Context*. Citeseer. 2004.

[195] Hideki Shimada et al. "Implementation and Evaluation of Local Dynamic Map in Safety Driving Systems." In: *Journal of Transportation Technologies* 5.02 (2015).

[196] Balkeshwar Singh, N. Sellappan, and P. Kumaradhas. "Evolution of Industrial Robots and their Applications." In: *International Journal of Emerging Technology and Advanced Engineering* 3 (2013), pp. 763–768.

[197] Bruno Sinopoli et al. "Vision based navigation for an unmanned aerial vehicle." In: *Proc. of the International Conference on Robotics and Automation (ICRA)*. Vol. 2. IEEE. 2001, pp. 1757–1764.

[198] Alan N Steinberg, Christopher L Bowman, and Franklin E White. "Revisions to the JDL data fusion model." In: *AeroSense'99*. International Society for Optics and Photonics. 1999, pp. 430–441.

[199] Mark E Stickel. "A Prolog technology theorem prover." In: *New generation computing* 2.4 (1984), pp. 371–383.

[200] Wolfgang G Stock and Mechtild Stock. *Handbook of Information Science*. de Gruyter Saur, 2013.

[201] Thomas Strang and Claudia Linnhoff-Popien. "A context modeling survey." In: *First International Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp*. 2004.

[202] Robert Szewczyk et al. "An Analysis of a Large Scale Habitat Monitoring Application." In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM. 2004, pp. 214–226.

[203] Karim Tari et al. "Context-Aware Dynamic Service Composition in Ubiquitous Environment." In: *International Conference on Communications (ICC)*. IEEE. 2010, pp. 1–6.

[204] M. Tenorth and M. Beetz. "KnowRob – Knowledge Processing for Autonomous Personal Robots." In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ. 2009, pp. 4261–4266.

[205] Moritz Tenorth et al. "Building Knowledge-enabled Cloud Robotics Applications using the Ubiquitous Network Robot Platform." In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2013, pp. 5716–5721.

[206] Moritz Tenorth et al. "Web-enabled robots." In: *Robotics & Automation Magazine, IEEE* 18.2 (2011), pp. 58–68.

[207] Sebastian Thrun. "Robotic mapping: A Survey." In: *Exploring artificial intelligence in the new millennium* 2 (2002), pp. 1–35.

[208] Gilman Tolle et al. "A Macroscope in the Redwoods." In: *Proceedings of the 3rd international conference on Embedded networked sensor systems*. ACM. 2005, pp. 51–63.

[209] Stefano Tonello et al. "WorkCellSimulator: A 3D Simulator for Intelligent Manufacturing." In: *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2012, pp. 311–322.

[210] I. Tuomi. "Data is more than knowledge: Implications of the reversed knowledge hierarchy for knowledge management and organizational memory." In: *Proc. of the 32nd Annual Hawaii International Conference on System Sciences (HICSS)*. IEEE. 1999, pp. 12–34.

[211] Mathieu Vallée, Fano Ramparany, and Laurent Vercouter. *A multi-agent system for dynamic service composition in ambient intelligence environments*. Citeseer, 2005.

[212] J.S. van der Veen, B. van der Waaij, and R.J. Meijer. "Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual." In: *Proc. of the 5th International Conference on Cloud Computing (CLOUD)*. IEEE. 2012, pp. 431–438.

[213] S. Vijaykumar and SG Saravanakumar. "Future Robotics Database Management System along with Cloud TPS." In: *International Journal on Cloud Computing: Services and Architecture (IJCCSA)* (2011), pp. 103–114.

[214] S. Vijaykumar and SG Saravanakumar. "Future Robotics Memory Management." In: *Advances in Digital Image Processing and Information Technology* (2011), pp. 315–325.

[215] Werner Vogels. "Eventually consistent." In: *Communications of the ACM* 52.1 (2009), pp. 40–44.

[216] Wolfgang Wahlster et al. "Sharing Memories of Smart Products and their Consumers in Instrumented Environments." In: *it-Information Technology (vormals it+ ti)* 50.1 (2008), pp. 45–50.

[217] M. Waibel et al. "RoboEarth." In: *IEEE Robotics & Automation Magazine* 18 (2011), pp. 69–82.

[218] Jim Waldo. "The JINI Architecture for Network-Centric Computing." In: *Communications of the ACM* 42.7 (1999), pp. 76–82.

[219] M. P. Ward. "Language Oriented Programming." In: *Software—Concepts and Tools* 15 (1995), pp. 147–161.

[220] Mark Weiser. "The computer for the 21st century." In: *Scientific american* 265.3 (1991), pp. 94–104.

[221] Mark Weiser. "Ubiquitous computing." In: *Computer* 26.10 (1993), pp. 71–72.

[222] Thorsten Weiss, Bruno Schiele, and Klaus Dietmayer. "Robust Driving Path Detection in Urban and Highway Scenarios using a Laser Scanner and Online Occupancy Grids." In: *Proc. of the Intelligent Vehicles Symposium (IV)*. IEEE. 2007, pp. 184–189.

[223] Wikipedia. *OSI model — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-May-2016]. 2016.

[224] Darren Willis, David J Pearce, and James Noble. "Efficient object querying for java." In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2006, pp. 28–49.

[225] Kai M Wurm et al. "OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems." In: *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*. Vol. 2. IEEE. 2010.

[226] G Michael Youngblood, Diane J Cook, and Lawrence B Holder. *The MavHome Architecture*. Tech. rep. Department of Computer Science and Engineering University of Texas at Arlington, 2004.

[227] G Michael Youngblood et al. "Automation Intelligence for the Smart Environment." In: *International Joint Conference On Artificial Intelligence*. Vol. 19. Citeseer. 2005.

[228] Sergey Y. Yurish. "Sensors: Smart vs. Intelligent." In: *Sensors & Transducers* 114 (2010), pp. I–VI.

[229] Juan Cristóbal Zagal and Javier Ruiz-del-Solar. "UCHILSIM: A Dynamically and Visually Realistic Simulator for the RoboCup Four Legged League." In: *RoboCup 2004: Robot Soccer World Cup VIII*. Springer, 2005, pp. 34–45.

[230] Chaim Zins. "Conceptual approaches for defining data, information, and knowledge." In: *Journal of the american society for information science and technology* 58.4 (2007), pp. 479–493.

[231] Sebastian Zug. "Konzeption einer fehlertoleranten Programmierabstraktion für verteilte Sensor-Aktor-Systeme." PhD thesis. Otto-von-Guericke Universität Magdeburg, 2012.

Webliography

[232] URL: http://www.mhi.co.jp/en/products/detail/wakamaru.html (visited on 06/09/2013).

[233] URL: http://www.irobot.com/us/learn/home/roomba/Coverage_Technology/Exploratory_Behavior (visited on 06/12/2013).

[234] Apache. *Apache Hadoop*. URL: https://hadoop.apache.org/ (visited on 07/07/2015).

[235] Apache. *Apache MapReduce*. URL: http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html (visited on 07/07/2015).

[236] Apache. *HDFS Architecture Guide*. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (visited on 07/07/2015).

[237] *Apache Cassandra 0.8 Documentation: About Column Families*. URL: http://www.datastax.com/docs/0.8/ddl/column_family (visited on 08/10/2014).

[238] Gary Bentley. *JoSQL (SQL for Java Objects)*. URL: http://josql.sourceforge.net (visited on 02/02/2015).

[239] S. Blumenthal. *BRICS 3D Perception and Modeling Library*. URL: http://www.best-of-robotics.org/brics_3d/ (visited on 07/07/2015).

[240] *BostonDynamics - Legged Squad Support Systems (LS3)*. URL: http://www.bostondynamics.com/robot_ls3.html (visited on 06/09/2013).

[241] Victor Bret. *Talk: The Future of Programming*. URL: http://worrydream.com/dbx/ (visited on 06/09/2015).

[242] DataStax. *CQL 3 Language Reference*. URL: http://www.datastax.com/docs/1.1/references/cql/index (visited on 10/09/2013).

[243] Rosen Diankov. *Controlling Robots with ROS/OpenRAVE*. URL: http://openrave.programmingvision.com/wiki/index.php/ROS:ControllingRobots (visited on 03/25/2015).

[244] Rosen Diankov. *OpenRAVE Custom XML Format*. URL: http://openrave.programmingvision.com/wiki/index.php/Format:XML (visited on 10/10/2914).

[245] André Dietrich. *A Bidirectional Graph-Search Algorithm (benchmark)*. URL: http://nbviewer.ipython.org/url/gitlab.com/OvGU-ESS/Jupyter-Notebooks/raw/master/bidirectional%20graph%20search%20benchmark.ipynb.

[246] Stefan Edlich. *The ultimate reference for NOSQL Databases*. URL: http://nosql-database.org (visited on 10/09/2014).

[247] Tim Field and Jeremy Leibs an James Bowman. *ROS rosbag package documentation*. URL: http://wiki.ros.org/rosbag (visited on 07/08/2015).

[248] John R. Fisher. *Prolog: Tutorial - 2.19 Actions and plans*. URL: https://www.cpp.edu/~jrfisher/www/prolog_tutorial/2_19.html.

[249] Tully Foote, Eitan Marder-Eppstein, and Wim Meeussen. *ROS transformation package documentation*. URL: http://wiki.ros.org/tf (visited on 07/08/2015).

[250] Tully Foote and Radu Bogdan Rusu. *ROS nodelet package documentation*. URL: http://wiki.ros.org/nodelet (visited on 07/08/2015).

[251] *Golem Krang - Humanoid Robotics at GeorgiaTech.* URL: http://golems.org/projects/krang.html (visited on 12/06/2013).

[252] *JAHIR - Joint-Action for Humans and Industrial Robots.* URL: http://www6.in.tum.de/Main/ResearchJahir (visited on 06/09/2013).

[253] *Kinematics/IKFast - Moveit.* URL: http://moveit.ros.org/wiki/Kinematics/IKFast (visited on 07/07/2015).

[254] Hokuyo Automatic Co. Ltd. *Specifications: Scanning Laser Range Finder URG-04LX-UG01.* URL: http://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG-04LX_UG01_spec_en.pdf.

[255] Eitan Marder-Eppstein and Vijay Pradeep. *ROS actionlib package documentation.* URL: http://www.%20ros.%20org/wiki/actionlib (visited on 07/08/2015).

[256] *RoboEarth at Github.* URL: https://github.com/rapyuta/roboearthDB (visited on 07/07/2015).

[257] *RoboEarth: Documentation.* URL: http://api.roboearth.org/documentation/dev (visited on 07/07/2015).

[258] *ROS messages description language.* URL: http://www.ros.org/wiki/msg (visited on 08/10/2014).

[259] *ROS Service concept.* URL: http://wiki.ros.org/Services (visited on 07/08/2015).

[260] Salesforce.com. *Phoenix - We put the SQL back in NoSQL.* URL: https://github.com/forcedotcom/phoenix (visited on 10/09/2013).

[261] *Sesame.* URL: http://rdf4j.org/ (visited on 07/07/2015).

[262] Service Robotics Ulm. *Gateway Components between SmartSoft and ROS.* URL: http://www.servicerobotik-ulm.de/drupal/?q=node/52.

[263] *UNR Platform - A reference implementation of UNR (Ubiquitous Network Robot) Platform.* URL: http://sourceforge.net/projects/unrpf/ (visited on 10/11/2015).

Software

The following enumeration lists all applied and referenced software. Tools and libraries that are marked with a star (*) were developed by the author itself. Further projects that were not mentioned here can also be found at `https://gitlab.com/groups/OvGU-ESS`.

ANTLR Terence Parr. *ANother Tool for Language Recognition: A parser generator for reading, processing, executing, or translating structured text or binary files.* URL: `http://www.antlr.org`

***AR-ICE** André Dietrich. *Augmented Reality InterfaCE: Simplified interface to ARToolKit for Python and other programming languages.* URL: `http://sourceforge.net/projects/ar-ice`

ARToolKit Philip Lamb. *A software library for building Augmented Reality (AR) applications.* URL: `http://www.hitl.washington.edu/artoolkit`

Cassandra Apache Software Foundation. *A scalable multi-master NoSQL database system.* URL: `http://cassandra.apache.org`

***cassandra-ros** André Dietrich and Sebastian Zug. *Cassandra database layer for ROS, that allows ROS messages to be stored, indexed, and queried according to the ROS publish/subscribe paradigm.* URL: `http://wiki.ros.org/cassandra_ros`

CQL Apache Software Foundation. *Cassandra Query Language: SQL-like declarative query language for Cassandra.* URL: `https://cassandra.apache.org/doc/cql3/CQL.html`

***Openrave-plugins** André Dietrich. *A set of OpenRAVE extensions.*

- ***distance-sensor** *Realistic distance sensors with different shapes for beams, which can be extended by integrating additional meshes.*
- ***filter** *Extendable plugin for scenario filtering, with support for OccupancyGridMaps, QuadTrees, Voxels, SensorCoverageMaps, etc.*
- ***situated-sensor** *A sensor plugin that connects a virtual sensor to real sensor system via ROS, allowing to compare virtual and real measurements.*
- ***trace** *Tracking plugin for objects and their trajectory during a simulation.*

URL: `https://gitlab.com/OvGU-ESS/OpenRAVE-Plugins`

***glodel** André Dietrich. *GLObal world moDEL: Interface to the distributed environment model stored within a virtual overlay database structure in Cassandra.* URL: `https://gitlab.com/OvGU-ESS/glodel`

Hadoop Apache Software Foundation. *A software framework for distributed processing of large data sets across clusters of computers.* URL: `http://hadoop.apache.org`

MongoDB MongoDB, Inc. *Document oriented distributed database system.* URL: `http://www.mongodb.org`

ODE Russell Smith. *Open Dynamics Engine: High performance library for simulating rigid body dynamics.* URL: `http://www.ode.org`

***odeViz** André Dietrich. *A VTK-visualization library in Python for ODE simulations.* URL: `https://pypi.python.org/pypi/odeViz`

OpenRAVE Rosen Diankov. *Open Robotics Automation Virtual Environment: Robotic simulation environment for testing, developing, and deploying motion planning algorithms.* URL: `http://openrave.org/docs/latest_stable`

pycassa Jonathan Hseu, Daniel Lundin, David King et al. *Python client library for Cassandra.* URL: `http://pycassa.github.io/pycassa`

ROS Willow Garage. *Robot Operating System: A set of software libraries and tools for building robotic applications.* URL: `http://www.ros.org`

rosbag Tim Field, Jeremy Leibs, and James Bowman. *Tool and library for working with binary ROS message containers.* URL: `http://wiki.ros.org/rosbag`

***rosR** André Dietrich and Sebastian Zug. *A ROS integration for the statistical programming language R.* URL: `http://wiki.ros.org/rosR`

***rosshell** André Dietrich. *ROS mediator for shell commands.* URL: `http://wiki.ros.org/rosshell`

***SelectScript** André Dietrich. *Implementation of the scripting language stub for select-statements.* URL: `https://gitlab.com/OvGU-ESS/SelectScript`

***SelectScript-ODE** André Dietrich. *Derived SelectScript dialect for ODE.* URL: `https://gitlab.com/OvGU-ESS/SelectScript_ODE`

***SelectScript-OpenRAVE** André Dietrich. *Derived SelectScript dialect for OpenRAVE.* URL: `https://gitlab.com/OvGU-ESS/SelectScript_OpenRAVE`

VTK Kitware. *Visualization Toolkit: 3D computer graphics library for modeling, image processing, volume rendering, scientific visualization and information visualization.* URL: `http://www.vtk.org`

warehousewg Bhaskara Marthi and Nate Koenig. *MongoDB database layer for ROS, that allows ROS messages and associated metadata to be stored, indexed, and queried.* URL: `http://wiki.ros.org/warehouse_ros`

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.“

Magdeburg, den 15.06.2016

André Dietrich