Technische Universität München
Fakultät für Informatik,
Lehrstuhl 11 (Prof. Dr. Schlichter):
Angewandte Informatik / Kooperative Systeme

Autonomous Sensor Data Processing

Lars Nagel

Diplomarbeit

Themensteller:	Univ.-Prof. Dr. Johann Schlichter
Betreuer:	Dr. Georg Groh
	Dr. Michael Berger
	Michael Pirker
Abgabetermin:	15. Januar 2006

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Januar 2006

.....
(Unterschrift des Kandidaten)

Kurze Zusammenfassung

Die Arbeit beschäftigt sich mit der lokalen Vorverarbeitung von Sensordaten durch autonome Agenten und beschreibt Szenarien, in denen diese eingesetzt werden können. Auf Basis dieser Szenarien werden Anforderungen formuliert. Im Anschluß an die Betrachtung verwandter Ansätze wird die AIP-Architektur (Autonomous Information Processing) entwickelt, die als Grundlage für die Implementierung eines Frameworks dient, mit dem die Agenten erstellt werden. Ein Prototyp für die computergestützte Altenpflege wird beschrieben. Die Arbeit endet mit einer Diskussion über Verbesserungen und Erweiterungen.

Short Abstract

The thesis investigates local preprocessing of sensor data by autonomous agents and presents scenarios for applications. From these scenarios requirements are derived. After the study of related approaches an architecture for autonomous information processing (AIP-Architecture) is developed as a base for implementing a framework for the generation of agents. A prototype for computer aided care of elderly people is explained. The thesis concludes with a discussion on possible improvements and extensions.

Contents

Chapter 0

Introduction

0.1 Basic Motivation and Problem Statement

The idea of a **robot**, as an autonomously acting machine, is rather old. Already in the 18th century a fully automated loom was constructed. In the centuries thereafter further mechanical automats came up, until in 1954 George C. Devol filed a patent application for the first electronic and programmable manipulator. Nowadays robots are widespread. Pushed by influential industries like the automobile industry, mainly industrial robots are developed. Often these robots do not possess any sensors. They rather remember sequences of motions taught through control and input devices. But there is also remarkable progress in the area of sensor guided robots. Well known probably are soccer playing robots, who even compare their skills in world championships. [31, 64]
Modern robots are controlled by integrated software that processes sensor data and triggers actions accordingly. In fact it seemed natural to employ autonomous entities independently of robots. By the end of the 1970s the name **software agent** came up for these entities. Potential properties of software agents are [31, 79]:

- **Knowledge** storing and representing. Agents have **beliefs** about the world and themselves.
- **Learning** aptitude.
- **Reasoning** and **planning**.
- Ability to frame and pursue **goals** or **desires**.
- **Adaptability**.
- **Collaboration** with other agents.

Even if not all properties are necessary, at least a few of them should be present to call it an agent. Opinions differ about the minimum requirements. Usually one does not talk of

an intelligent agent, unless it can adapt itself to the environment on the base of its own sensory perception.
Even if the potentials of (intelligent) agents are by far not yet exhausted, agents are utilized in various fields. E. g. they filter information in the internet, control persons in computer games and even steer vehicles on other planets.
The objective of this thesis is to develop an **agent for the preprocessing of sensor data**. The idea is to install it in major systems that employ a variety of sensors. The advantages of preprocessing are:

- **Filtering.** Irrelevant data can be **filtered** out, e. g. when a value did not change. Data can be **fused** before the filter is applied, as in some cases the categorization of data is easier having a broader view.
  Learning algorithms can be used in cases, where it is difficult to categorize data for filtering. The program can learn in advance or at runtime, which data is to be neglected.

- **Locality.** Data is processed there, where it is measured. Filtering **reduces data traffic** and **relieves the application**. In spite of growing speed and memory size a central processor may still form the system's bottleneck.

- **Modularity.** Preprocessing simplifies the application's logic. The problem splits into the application program and the adjustment of the agents. This principle is called **modularity** or **encapsulation** and supports the separation of concerns.

Because a priori we do not know which software will be applied and because the meaning of agent is somewhat vague we will in the sequel not speak of an agent but rather neutrally of a module. And since it deals with **autonomous (sensor) information processing (AIP)**, we call it an **AIP module**.

0.2 Thesis-Structure and Course of Argumentation

Chapter 1

In chapter 1 the definition of a sensor is stated. Several scenarios for the AIP module are explained and requirements are derived from them. These requirements are taken in chapter 2 to evaluate related work and in chapter 3 for developing the AIP architecture. To prepare for the scenarios some wireless technologies used in them are presented.

Chapter 2

In order to illuminate the requirements of chapter 1 and to illustrate the means of data preprocessing, concepts and processing tools are introduced. Some of these concepts and

tools are employed by related architectures, which are presented afterwards together with an investigation in how far these architectures meet the requirements.

Chapter 3

Chapter 3 is devoted to the AIP architecture. In the design rationale we argue what the overall architecture must look like to satisfy the requirements. After an overview the parts of the architecture are studied in detail. Then the requirements from chapter 1 are checked. Finally the software requirements due to the architecture are formulated.

Chapter 4

Based on the AIP architecture of chapter 3 the implementation of a prototype is examined. The chapter's structure follows the previous one considering the parts one after another. Software tools that were utilized or considered along the way are discussed.

Chapter 5

This chapter presents the implementation of the scenario "Smart Elderly Care Home" taken from 1. The sensors and the environment are simulated by a computer program.

Chapter 6

The thesis is concluded by a summary and a critical discussion of the results.

Chapter 1

Scenarios and Requirements

This chapter describes several scenarios which illustrate the different demands made on a software module for distributed sensor data processing. From these scenarios and from the general idea of sensor data processing we will derive the basic requirements. First we will discuss an extended definition of a sensor and give a short overview of the wireless technologies used in the scenarios.

1.1 Sensors

As sensors are fundamental features in engineering, there are lots of definitions in literature. The most common definition – with slight differences found in various books [31, 65] – is:

> A **sensor** is a system that converts a physical quantity and its changes into appropriate electric signals.

These sensor signals will then be processed by a measuring device that provides the data in a form usable for men or machines. As we are not primarily interested in the sensors themselves, we consider sensor and measuring device as one unit. It provides us with data, either answering a request (pull mode) or automatically at certain intervals (push mode). Therefore we abstract and extend the definition for our purposes:

> A **sensor** is a system that retrieves data, especially measurements of physical quantities, and transfers it over a software interface.

On this basis we will be able to treat any data source as sensor. Hence databases and common sensors can be handled the same way.

1.2 Wireless Technologies

Recently wireless technologies gained importance. During the last ten years several new standards were developed. Since wireless technologies play a major role in the scenarios, we will give a brief survey of standards and their properties. WLAN (Wireless Local Area Network), Bluetooth, ZigBee, NFC (Near Field Communication) and RFID (Radio Frequency Identification) are wireless technologies that answer different purposes.
The **Radio Frequency Identification (RFID)** is used for automatic identification. We distinguish RFID tags and RFID readers. A RFID tag is a transponder (artificial word combining transceiver and responder). That means it receives signals and answers automatically to them. The RFID reader on the other hand sends signals in order to read tags. Often RFID tags send back 32, 64 or 96 bit identification or serial numbers only, because these tags are normally used for mere identification. But they may also have memory up to 2048 byte.
One distinguishes passive, semi-active and active tags.

- **Passive tags** have no internal power supply. The electrical current induced in the antenna by the incoming radio frequency signal must be used to power the IC (integrated circuit) and transmit a response. The range is limited to typically a few centimeters, but can be a few meters. As no battery is needed, the tag is small and its lifespan is unlimited.

- **Semi-active tags** have a battery to power the IC constantly, so that the antenna only has to collect power for the backscattering signal, thus reducing the response time.

- **Active tags** have an internal power supply to power any ICs and to generate the outgoing signal. They can have a larger memory than passive tags and a longer range, up to tens of meters. The smallest devices have the size of a coin.

Transponders were invented in the 1920s, used by the military in the mid 1940s and in business in the late 1960s. At present there is done a lot of research, where and how RFID can be used in modern industry and society. [9, 63, 62]
The **Near Field Communication (NFC)** is – according to expectations – also a short range wireless technology, having a range of only a few centimeters. It was jointly developed by Sony and Philips and became an ISO standard in December 2003. Half a year later Nokia, Sony and Philips formed the NFC Forum [50]. The coverage of NFC is arranged for electronic payment and ticketing and for establishing longer range wireless connections by "touching". "Touching" means that two NFC devices are brought together in order to communicate, e. g. to exchange the configuration data and set up a faster connection for longer range. [48, 61]
As NFC is quite new, applications are still developing. In Hessen, Germany, Nokia, Philips and the Rhein-Main-Verkehrsverbund (RMV) started the first NFC-enabled ticketing in public transportation. In buses and trains of the RMV passengers can "touch" the terminals with their mobile phones to sign on, when entering, and to sign off, when leaving.

At the end of the month they get a post-paid-bill, similar to a phone bill [52, 51].
An other application area is starting a longer data transmission by "touching". For example, photos were taken with a built-in camera of a PDA or mobile phone. To display the pictures on the TV set, one only has to "touch" the TV set with the mobile device to start for instance a Bluetooth connection. The photos are transmitted over the connection and shown on the TV screen. [49]
Bluetooth is an industrial specification for Wireless Personal Area Networks (WPAN) that was developed by Ericsson and resumed in 1999 by the Bluetooth Special Interest Group (SIG). Bluetooth provides a way to connect and exchange information between devices like personal digital assistants (PDAs), mobile phones, PCs and printers. The features of Bluetooth involve low power consumption, the use of a low-cost transceiver microchip and a globally available radio frequency for short range connections (0.1, 10, 100 or up to 400 meters). Bluetooth became a Wireless PAN standard (IEEE 802.15.1). [58, 41]
During the boom of mobile devices (mobile phones, PDAs and smart phones), Bluetooth was developed, and lots of applications deal with the data transmission between mobile devices, between mobile devices and PCs and between PCs and peripheral devices like printers and headsets.
ZigBee is a wireless technology which is also meant to build up Wireless Personal Area Networks (WPAN) with low data rates and low power consumption, but is designed to be simpler and cheaper than Bluetooth. Its transmission range is 10 to 75 meters. ZigBee 1.0 was ratified in December 2004. [67]
Wireless Local Area Networks (WLAN) have in contrast to WPANs higher transmission power, range and transfer rates. Depending on the antenna the range is between 30 and 300 meters. In the late 1990s, after several proprietary protocols, a few standards were introduced that all belong to the IEEE 802.11 family. WLAN is used in two different modes [66]:

- The **infrastructure mode** is normally used to connect the wireless network with a wired Ethernet network or with other wireless networks. A base-station, usually a wireless access point, serves as the central WLAN communication station and coordinates the networks nodes. This is typically used by a stand-alone base-station.
- In the **ad-hoc mode** or **peer-to-peer mode** there is no special station for the coordination. All stations are peers. Ad-hoc-networks are easy to build up. It's not intended that packets are forwarded, so that it is possible that not every computer can reach all others.

In general **wireless technologies** are mainly used in office and mobile applications, in situations where sensors and actuators are difficult to connect, also where a later installation would raise problems, for example in moving machines, and with data acquisition in logistics. According to [56] WLAN (for long distances), Bluetooth and ZigBee (for short distances) will be the successful future technologies. For very short distances NFC, pushed by three global players, has a good chance to become established in the next years.

Name	Range	Data Rate	Energy Consumption	Standards	Implementation
Radio Frequency Identification (RFID)	2 mm – 80 m	small	depending	ISO/IEC 15961:2004, ISO/IEC 15962:2004	late 60s
Near Field Communication (NFC)	few cm	106, 212, 424 kbit/s	very low	ISO/IEC 18092:2004 or ECMA-340	Dec 2003
Bluetooth	10 cm – 400 m	2.2 Mbit/s	low	IEEE 802.15.1	1998
ZigBee	10 – 75 m	20, 40, 250 kbit/s	low	IEEE 802.15.4	2004
Wireless Local Area Network (WLAN)	30 – 300 m	up to 54 Mbit/s (540 Mbit/s in 2006)	high	IEEE 802.11g, 802.11b, 802.11h, 802.11a, (802.11n)	Standards since 1997

Table 1.1: Survey of wireless technologies.

For RFID several business concepts will be developed, but the main application area will probably be logistics. [56, 57]

1.3 Scenarios

The sensor definition and the information about wireless technologies will help us, while considering the scenarios. This section deals with possible application areas for autonomous sensor information processing (AIP). The outlined agent will be called AIP agent or AIP module.

1.3.1 Automotive Applications

"Today in high-class cars there is more computing power than needed for sending the first men to the moon." [94] In modern cars several computers process input from numerous sensors that measure great amounts of data to provide safety and comfort. Most of the tasks are independent of each other and located in different places. That makes them well suited for data processing in local AIP modules. If data is needed by other AIP moudles, it can be communicated. Examples of such applications in the automobile fields are:

- **Automatic air conditioning system.** Self-regulating air conditioning systems have become quite common in middle class cars. The necessary data is provided by sensors that could be read out by an AIP module. The AIP module would observe the airing, heating, humidity and amount of dust in the air. Typical sensors for this application are solar sensors, measuring the direction of incidence and the radiation intensity of the sun, and sensors for temperature and humidity. Moreover the passenger's settings are part of the input. To guarantee an optimized climate for every passenger, the sun's radiation intensity will be determined for the driver's side and the passenger's side as well. If the sun is so low that the driver could be blinded, sun shades will be automatically lowered.

- **Automatic windscreen wiper and lights.** Rain sensors and solar sensors help to control windshield wipers and lights automatically, allowing the driver to concentrate on the traffic. Thus comfort and safety of passengers and driver are increased.

- **Swiveling headlight.** Sensor data of speed, yaw angle and steering angle are processed to anticipate the crucial part of the road ahead. The headlights are turned in that direction [36].

- **Display of tyre pressure.** The driver is warned by an optic or acoustic signal, when the tyre pressure has declined. The decrease of pressure can be detected by measuring the tyre pressure directly and transmitting the value via radio or by monitoring the wheel speed that changes with the radius of the tyre [42].

- **Electronic Stability Program (ESP).** The ESP was developed for holding the car stable in situations of quick steering, for instance to avoid an obstacle. It adapts the actual behaviour of the car to the drive input (up to a 150 times per second) and prevents oversteering and understeering by decelerating single wheels and controlling the overall speed. The car's bearing is evaluated from the wheel speed, the motor values and the cross acceleration (yaw angle). The steering-angle sensor indicates the drive input. Beside the need of filtering and fusing the data, another requirement turns out: The yaw angle is needed for the swiveling headlight and for the ESP, and hence, if only one sensor is to be used, an exchange of sensor data must be implemented. [34, 60, 36]

- **Detection of Overfatigue.** Many car accidents happen, when drivers are so tired that their reactivity is heavily reduced. Often drivers do not realize or do not want to realize, when their attention weakens. Hence, an unbiased computer system could help monitoring the driver's behaviour and alerting him by sound, when his driving abilities are decreasing. Appropriate means could be a camera monitoring his eyes, pressure sensors in the seat, the journey time, the course of speed, the steering behaviour and maybe even body sensors taking pulse, blood pressure or breathing rate. As normally a single one of these values is not significant and as false alarms should be avoided, a fusion of sensor data is self-evident.

1.3.2 Info Points: Orientation and Information

For the automobilist there are a lot of means for orientation and navigation in big cities, namely road signs, road maps or navigation systems. It is more difficult for pedestrians. Normally the support for pedestrians is rather poor, and especially tourists or disabled people need to ask their way through. An answer to this problem could be info points set up in the streets as part of the infrastructure to provide information to people walking by. A mobile device, for instance a mobile phone, would receive the information. From the wireless technologies, stated in section 1.2, RFID or NFC could be used, but then the user had to "touch" the info point with his device, and in case of RFID the data would be static. Better suited are Bluetooth or ZigBee with lower power consumption than WLAN and with a transmission range of up to 30 m. The user then will get the information, when he approaches the info point, and he does not even have to be aware of the info point at all.
Beside orientation advice, the info points can also send information of special interest, for instance about infrastructure, monuments and museums, history, city planning or public transport. With the amount of information available, the necessity to filter the information will grow and could be met by a preprocessing AIP module. The AIP module inside the mobile device is connected with the interface of the chosen wireless technology. This would send the filtered data to the main application for display.
For example if a tourist wants to see all famous monuments, he will be able to reduce the incoming information to his particular interest by setting his device. Moreover he could have special programs for more sophisticated wishes, that for example contain the information to non-Catholic churches built before 1900. So depending on the chosen program the info points could act like a tourist guide or like an orientation guide for the user. Data that does not correspond with the presetting is filtered out.
Considering the orientation part, it could be interesting to network the info points, so that the info points could arrange how the user is to be guided through the city. Moreover it would be easier to update the info point information over a network.

1.3.3 Logistics

In times of hotly contested markets and shortening product life cycles, the competitor, who is able to react quickly and flexibly to changes on the market will achieve a benefit. Therefore in the future the factory production will have to be more flexible regarding to individual customer preferences. This will increase the complexity of control and material flow and would, if no automatic solution could be found, also mean an increase of logistic and production costs, because today the automatic serial production is limited to inflexible, standardized processes. Changeable material flow structures still need the expensive services of men and manual feed systems like lift trucks.
In [23, 95] these prospects are described along with an automated changeable material flow structure using the RFID technology. We will alter the scenario to include sensor data processing modules.

In a factory along an assembly line or here an assembly network there are branching points called waypoints and different points of action, at which parts of the products are assembled, welded, drilled or checked. The parts are transported in containers that are tagged with RFID tags. To ensure that a container reaches the predetermined point, it has to be navigated through the assembly network. For this purpose the waypoints have RFID readers that read out the IDs, for example Electronic Product Codes (EPC), stored in the RFID tags of the containers passing by. Due to this information giving an overview of the whereabouts of the containers, the containers are routed to the next branches or to the next points of action.
Although information about all containers could be considered when a container is to be routed through a waypoint, local information that shows, whether following line sections are busy, should be sufficient. Hence, the assembly network can be divided into physical or logical areas, so that each area could be supervised by an AIP module. When a container is about to change the area, adjacent sensor data processing modules will have to exchange information, but most of the decisions can be made locally.
The advantage of this decentral solution is the separation of concerns. Processing on one central computer would lead to a more complex logic, increased communication and would require higher computing power. These disadvantages are avoided by the use of distributed AIP modules, which, by communicating with their neighbours only, guarantee a fast routing through the network. The reliability of the distributed system is higher and changes or extensions are easier implemented, because only AIP modules must be added or exchanged locally.
Additionally the system could improve the routing by using learning algorithms. When a container reaches its destination the time spent is taken and sent as feedback to the different AIP modules that participated in the routing procedure. The modules test different strategies and compare the results.
Moreover the sensor data processing module could be connected with other types of sensors, for example monitoring cameras recognizing accidents. So in dangerous situations the whole system or parts of the system could be stopped automatically.

1.3.4 Smart Elderly Care Home

Worldwide the number of people older than 65 years will double from 1990 to 2025. In Germany the proportion of people older than 60 years grew from 14% in 1990 to 24.1% in 2001 and will reach 34.4% by 2030. The number of people needing care will grow from 2 millions today to 2.8 millions in 2020. [54, 55]
In a population that steadily ages, the care for the elderly and sick becomes more important, more expensive and some day unaffordable. Hence, new and financially feasible concepts for the care are needed.
Considering these arguments computer-supported health monitoring of elderly or sick people might be a solution. Naturally nobody likes to be watched, but the alternative would not be better: Perpetuation of autonomy vs. intrusion into privacy. Such a com-

puter system would be fed by sensor data that reports the behavior of the monitored person, measures bodily functions or retrieves ambient conditions. Means of acquiring the person's behaviour are motion detectors, pressure sensors in furniture, heat-sensitive or normal cameras, RFID tags, light barriers, scales and switches. They indicate, where the person is (lies in bed, is moving in the kitchen), what she is doing (moving, sleeping, watching TV) and what she has done or has not done (forgotten to take pills). Bodily functions of interest are pulse, heartbeat, blood pressure and body temperature. [77]
Typically the situations are restricted to a single room or maybe two adjacent rooms and hence can be handled locally. When the person is in the bathroom, data from the living room is not needed, however, there are cases, where data from other sensors improve the reliability e. g. when the person walks to an other room, the data from that room confirms the change. Briefly there is a need to fuse data and the possibility of working locally, even if in some cases the exchange of data is useful as seen in the room swap example. The AIP module could therefore process data locally, thus fuse, filter and maybe exchange data, and depending on rules decide, whether the application must be notified of any changes. The rules should be formulated by application developers with medical knowledge or could be learned by a supervised learning algorithm. Whether the system should adapt its rules at runtime, is questionable, because the adaption could impair the security of the system.
Any changes and special events like "person goes to bed" should be logged by the system, exceptional circumstances must be reported to the doctor, and in case of emergency there must be a doctor who can reach and help the patient fast. But often it will be enough to notify the monitored person, for example when she has forgotten to take her pills, or to query her, whether anything has happened to her. Such notifications could be displayed on the TV screen or spoken over loudspeakers.
In addition to the observation there could be some supporting applications that can be found in so-called smart homes. Triggered by events doors could be opened or closed, lights and heating could be adjusted. For example if the person goes to bed, the "night program" is started that cools down the heating, turns off the lights and lowers the shutters. Moreover the supervised person could have a remote control or alternatively a mobile phone for starting programs like "close all shutters" and for setting off an alarm by himself. For this purpose one of the wireless technologies could be used.
As this scenario realizes elderly care in a smart home, it is called "Smart Elderly Care Home". It was implemented as a prototype for the AIP module (see chapter 5).

1.4 Requirements

After the examination of operational areas for the AIP module, a few requirements will be drawn from the scenarios:

- **Filtering.** An obvious task of a module processing sensor data is filtering. In the info point scenario the main task of the AIP module is to let only such information

pass that meets the user's criteria. In the elderly care scenario filtering is needed to provide that the medical staff is only informed of alerts and important changes and not burdened with irrelevant messages.

- **Sensor data fusion.** In applications, in which a software agent needs to perceive and picture its environment on the basis of sensory data, it is often necessary to fuse data to get a more accurate or more complete picture. We have seen in the automotive applications that the detection of overfatigue depends on multiple sensors. A single sensor cannot describe the whole situation sufficiently. If for example the driver's steering seems uncontrolled, it could be that he is forced to do so because of an obstacle or bad road conditions. But in combination with other sensors showing his bearing or his eye size, the uncertainty about the situation can be reduced.
  When data needs to be fused, we often deal with uncertainty, meaning that we have uncertain knowledge about the situation, and we fuse data from different sources to reduce the uncertainty, so that we are able to classify it. In 2.2 we will dwell on uncertain knowledge and in 2.5 we will investigate how fusion can be implemented.

- **Learning.** There are two ways to apply learning algorithms in the scenarios. On the one hand the AIP module could be trained, how to classify more complex situations, before it is started. On the other hand it could adapt itself at runtime.
  In the elderly care scenario for example, when the user or the doctor is notified, they can give feedback, whether the situation was rated correctly. Moreover in the smart home part of the scenario the system can learn the behaviour of the user. If the user usually turns on the TV set every day at eight o'clock to watch the news, the system will be able to learn this behaviour, so that it turns on the TV itself or reminds the user to do so.
  In the logistics scenario the agent could try different routing strategies and get feedback, when the container reaches its destination. As the destination can be remote, the feedback should be returned via a server providing the involved AIP modules with the feedback.

- **Classification.** The filtering of sensor data is a classification problem. Data is sorted into classes, and depending on the class it is processed. Fusion and learning algorithms can support the classification.
  The question is, which classification methods are necessary to meet the application's requirements. In the info point scenario plain criteria determine, whether data is displayed on the mobile device. For simple cases like this uncomplicated rules are sufficient for filtering the data.
  In some cases data first needs to be fused, especially when higher level data is to be sorted or when a situation needs to be estimated on the basis of uncertain knowledge.
  If the classification is so complex that the application developer is not able to specify rules by himself, then learning algorithms could help. When test data is available,

they can be used to train the system. Moreover, the system can adapt itself to the requirements during runtime.

- **Locality.** In spite of ever faster computers a central computer reaches its limits, when it has to read and process large amounts of raw data. Hence it is advisable, to process or preprocess locally. This separation of concern makes the program logic simpler and clearer and thus more reliable. Moreover local preprocessing diminishes the amount of communication. In the smart elderly care home scenario and the logistics scenario we have pointed out that most problems can be solved locally. That is also true for the automotive scenario, where many tasks are independent of each other, so that decisions can be made merely on locally provided sensor data.

- **Communication.** Although locality is given in all described scenarios, local areas often cannot be encircled strictly. In the elderly care home scenario the different rooms of the person's flat are appropriate sections, but changing to an other can be retraced better, when adjacent AIP modules are able to communicate. Another motivation for enabling communication between AIP modules is the joint access to sensors. In the automotive applications two applications need the yaw rate. The second sensor can be spared when one AIP module has remote access to the sensor of the other AIP module.

- **Generic sensor interface.** Because of the great variety of sensor types it is necessary to provide a generic sensor interface. Every AIP module must allow the connection to every sensor type – during initialization and preferably also at runtime.

- **Simple and generic management interface.** The various scenarios clearly showed the need for different means of data processing, but that not all of them are applied in all cases. Since memory space may be limited it should be possible to have an AIP module, which meets the requirements exactly, without wasting resources with unnecessary tools. That could be achieved by placing a set of fixed modules at the user's disposal. More appropriate is a construction kit with a fixed kernel, to which arbitrary tools can be attached. The application designer then can tailor the module to his needs.
  In order to initialize and manipulate these tools independent of the actual tool instances, a generic management interface – that is the interface used by the application above – is needed. Dependent on the demand the handling of this interface must be kept simple.

The scenarios showed that many tasks can be solved by simple rules, especially in the info point scenario and in the logistics scenario, that should be manageable by the application designer. But also the automotive applications are not so complex that learning algorithms have to be applied. And in the elderly care scenario, when using simple rules, the control is more easily retained. Of course, when necessary the application developer must use more sophisticated means like Bayesian networks or learning algorithms, but

they should not be used in every case, for their use requires a lot of preparation and maintainance.
Because of the potential complexity and the task specific design of an AIP configuration it is assumed, that one AIP module serves only one application. Though, of course, it is conceivable that several applications jointly address one AIP module regarding it as kind of a context server. Sections 2.7.1 and 3.1 take up this idea again, but in the implementation it is not taken into consideration (cf. chapter 4).

Summary

Based on the four scenarios, which point out several applications of AIP modules, we derived requirements to be satisfied by AIP modules, which we will refer to in the sequel. In the next chapter we shall describe some of these requirements in more detail to establish a base for investigating several architectures.

Chapter 2

Related Work

In this chapter we will discuss existing architectures and check them for the requirements stated in section 1.4. But first we will describe a few tools and techniques that will be mentioned in the analysis of the related architectures and of the AIP architecture in chapter 3.

Chapter 1 ended with the formulation of requirements for autonomous sensor information processing (AIP). As explanation and in consideration of the architecture to be designed, this chapter will study several tools and concepts of knowledge engineering. They include reasoning tools – in our case rule engines and Bayesian networks –, knowledge-based systems, related concepts like ontology and context, and means for data processing, namely fusion and learning algorithms. Finally an overview and evaluation of existing architectures is given.

2.1 Rule Engines

When in 1960 Newell and Simon introduced the idea of a *General Problem Solver* [70], the expectations about artificial intelligence rose very high. The machine was supposed to simulate human problem solving by applying a set of rules. The project failed, but from its ruins emerged in the 1970s the idea of an expert system, an advisor that supports men in problem solving. Expert systems are no longer all-round geniuses but experts with (human) knowledge in a restricted field. They were applied in business and medicine with varying success.
Expert knowledge can be represented in various ways. Today most common is the use of objects and rules. Here the objects represent the factual knowledge, while the rules expand this knowledge and generate answers to the queries. These rules are simple IF-THEN statements, as the ones well known from diverse programming languages. [20]
To be called an expert, the system must necessarily include a certain amount of knowledge. For that reason one also speaks of knowledge based systems. For Giarratano and Riley [20] a knowledge based system is a smaller system with rather flat knowledge that is not an expert in a field and that is much easier to program. In general any system using a

knowledge base is considered to be a knowledge based system. That definition includes expert systems and will be used in the sequel.
As knowledge-based systems do not necessarily incorporate a set of rules, rule based systems form a genuine subset of the knowledge-based systems. To interpret the rules an additional control system containing a rule interpreter is required. We denote the rule based system as whole also as rule engine and define:

> A rule engine or rule-based system is a software that consists of:
>
> - a knowledge base of facts
> - a rule base, that is a set of IF-THEN-rules
> - a control system with rule interpreter
>
> The control system chooses the rules to be applied, either data-driven (forward chaining) or goal-driven (backward chaining).

The terms forward chaining and backward chaining will be explained later.
The idea of providing a dynamic rule base and a dynamic knowledge base is certainly that they are easily adaptable at runtime, that rules and facts can be removed or added. A rule engine is a pluggable software component that separates the rules from the application code.
Expert systems were basically designed for symbolic reasoning, and besides *Prolog*, the best known language for artificial intelligence (besides *Lisp*), several special languages and rule engines were developed. We will introduce a few of them in chapter 4. For the sake of efficiency it is sometimes advisable to implement parts in procedural languages.

Forward Chaining A chain is defined as a sequence of inferences that lead from a problem to its solution or rather from the facts to the conclusion. Forward chaining means that such a chain is traversed from the facts toward the conclusion. The question is what conclusion can be reasoned from given facts. Such data-driven inferences are also termed bottom-up reasoning. They are applicable for planning, monitoring and control. [20]
In the following example, inspired by the elderly care scenario, we have the rules:

- `IF X has a low pulse AND X does not move AND X does not sleep THEN raise alarm for X`
- `IF X has pulse Y AND Y < 40 THEN X has a low pulse`

If the knowledge base just contains the facts that the monitored person, say her name is Mrs Krause, does neither move nor sleep, nothing will happen, since neither condition is satisfied entirely. But when we add the fact that Mrs Krause has a pulse of 38, the second inference will lead to the conclusion that Mrs Krause has a low pulse. Then all of the first rule's conditions are met, and the consequence is fired, thus the alarm is raised for Mrs Krause.

Backward Chaining The idea of backward chaining is traversing the chain from a hypothesis (or goal) to the facts. The hypothesis is fulfilled if all its conditions are fulfilled. So the question whether the hypothesis is true is recursively substituted by the question whether its conditions are true, until the facts are reached that are always true, when they exist. If cycles are obviated, this leads to a depth-first traversal in an and-or-tree made up of rules as inner nodes and facts as leafs.
Backward chaining rule engines are used for diagnoses, for example in expert systems. The goal-driven procedure, starting at a high-level hypothesis and proceeding to lower-level facts, is also called top-down reasoning. [20]
In the example, similar to the one above, we will use Prolog notation[1], for Prolog is a quasi-standard for backward chaining. Suppose, we have the following rules:

- `alarm(X) :- lowpulse(X), not(moves(X)), not(sleeps(X)).`
- `lowpulse(X) :- haspulse(X, Y), Y < 40.`

And the knowledge base contains the facts:

- `haspulse(krause, 38).`
- `sleeps(krause).`

Now, if we want to check the hypothesis that an alarm has to be raised (because Mrs. Krause might be in a state of emergency), we will ask: `alarm(krause).`
In this case, the answer will be `No`. Truly, Mrs Krause's pulse is less than 40 and therefore low. She does not move either, because there is no such fact in the knowledge base. But she is asleep, and we could add that the low pulse is ascribed to this fact.

2.2 Uncertain Reasoning

A software agent, that perceives its environments through sensors, has only a vague picture of it. Due to the values provided by the sensors it can only make suppositions that are not necessarily correct. So it only has uncertain knowledge at its disposal. Reasoning on such knowledge is called uncertain reasoning.
An example for the usefulness of uncertain knowledge and uncertain reasoning can be found in the elderly care scenario. Suppose, the system merely realizes, that the monitored person has picked up his pillbox. For she was instructed to hold the pillbox, tagged with RFID, close to a RFID reader, whenever taking a pill. Then the probable conclusion is

[1] Prolog rules (and facts) are Horn formulas. A Horn formula is a disjunction, in which at most one of the literals is positive: $b \vee \neg a_1 \vee ... \vee \neg a_n$. In Prolog such a rule is written as an equivalent implication: $b \Leftarrow a_1 \wedge ... \wedge a_n$, but using `:-` instead of $\Leftarrow$ and commas instead of $\wedge$. Facts have no positive literal (b is constant true).

that the pill was taken, but it cannot be guaranteed. Therefore it would be useful to represent the uncertainty by an estimated probability of, say, 5%.
The rule-based systems that were discussed in the previous section are quite inappropriate to deal with uncertain knowledge. In [79] it is explained that their intrinsically positive characteristics, namely *locality*, *detachment* and *truth-functionality*, contradicts uncertain reasoning.
Other approaches like the Dempster-Shafer theory and the use of fuzzy logic, which also were not particularly successful, are not described in this thesis. We refer to [79] and address ourselves to the Bayes Theorem, which is the basis for probabilistic reasoning in almost all modern AI systems, especially for Bayesian networks, which we deal with subsequently.

2.2.1 Bayes Theorem

The Bayes' rule is a basic theorem in the probabilistic theory and allows to determine the conditional probability $P(Y|X)$; given X, how probable is Y.

$$P(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)}$$

Normally we are interested in a posteriori probabilities $P(Y|X)$. We observe Y and want to know the probability that X is the cause. To calculate the probability, the three terms on the equation's right hand side must be known, the a priori probability $P(X|Y)$ and the unconditional probabilities $P(X)$ and $P(Y)$. Often it is the case that these values are on hand or that they can be estimated.
The standard example in [79, 38] is a medical diagnosis on the basis of a symptom. We consider a similar example taken from the elderly care scenario. Suppose, we want to know $P(a|l)$, the probability for an emergency, given that the person has a low pulse. From studies we know that an emergency is quite improbable with $P(a) = 1/10000$, that a low pulse has the probability $P(l) = 1/100$ and that in case of emergency the person has a low pulse with probability $P(l|a) = 0.7$. Then the a posteriori probability $P(a|l)$ is calculated as:

$$P(a|l) = \frac{P(l|a) \cdot P(a)}{P(l)} = \frac{0.7 \cdot 0.0001}{0.01} = 0.007$$

In spite of the 0.7 probability for a low pulse in case of emergency, the probability of an emergency is very small, no matter that the pulse is low.
Even if the terms, which are needed for the calculation, can be often determined, it is quite a disadvantage, that prior knowledge is required. Besides, to determine the values is usually time-consuming and nontrivial.

2.2.2 Bayesian Belief Network

A Bayesian belief network or simply Bayesian network like the one in figure 2.1 describes the probability distribution of a set of variables by specifying conditional probabilities and conditional independence assumptions. Before we go into detail, we define the conditional independence as:

$$\forall x_i, y_j, z_k : P(X = x_i | Y = y_j, Z = z_k) = P(X = x_i | Z = z_k)$$

Or shorter:

$$P(X|Y, Z) = P(X|Z)$$

It expresses that X is independent of Y, if Z is given.

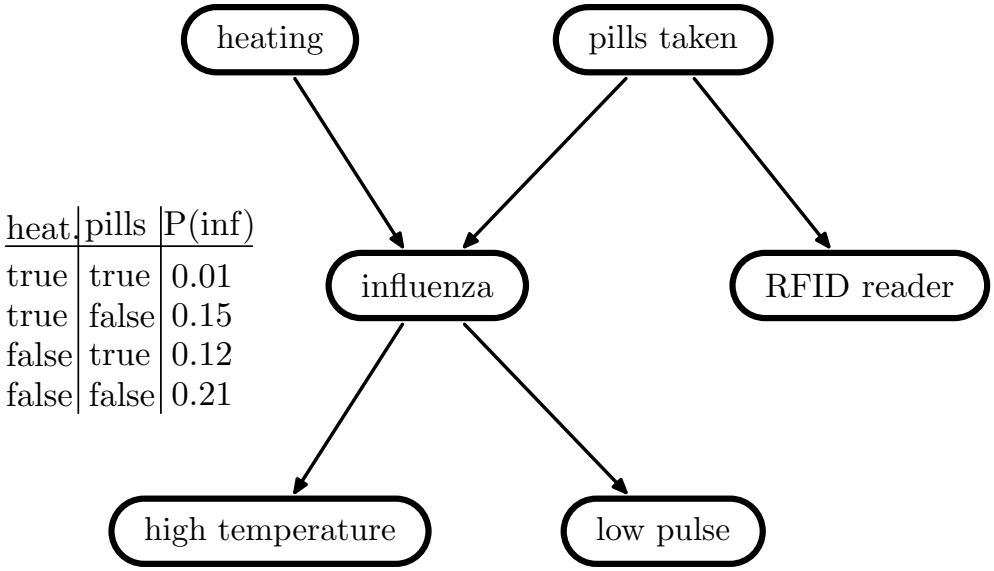

Figure 2.1: Bayesian belief network.

In the network dependencies are visualized by arrows. For each variable represented by a node, a table lists the dependencies. It specifies the probability distribution depending on the values of the immediate predecessors. In figure 2.1 only one table is given, namely for *influenza*, which depends on the logical values of *heating* and *pills taken*.
The simple example in this figure models a cut-out of the elderly care scenario. All variables are supposed to have type boolean. *heating* is true, if the temperature was properly adjusted. *pills taken* is true, if the person took her pills. Insufficient *heating* and omitted intakes of *pills* can cause an *influenza*. When pills are taken, it will be detected by the *RFID reader*. Effects of an *influenza* can be a *high temperature* or a *low pulse*.
Suppose, we want to know the probability $P(i \wedge l \wedge \neg t \wedge h \wedge \neg p \wedge \neg r)$ that the person has an influenza (i), a low pulse (l), though a moderate temperature ($\neg t$), that the heating was alright (h), but the pills were not taken ($\neg p$), and that the RFID reader has remained silent ($\neg r$). Because of the independence assumptions we can simplify:

$$P(i \wedge l \wedge \neg t \wedge h \wedge \neg p \wedge \neg r) = P(h) \cdot P(\neg p) \cdot P(\neg r | \neg p) \cdot P(i | h \wedge \neg p) \cdot P(\neg t | i) \cdot P(l | i)$$

With this formula the probability can be calculated, since all probabilities on the right hand side can be found in the tables. The probability for the influenza is given in figure 2.1 with: $P(i|h \wedge \neg p) = 0.15$.
In this case we only used information contained in the network, and we determined the probability of an assignment including all variables. But usually there will be additional knowledge, and we will be interested in the probability of particular values only. In the elderly care scenario for instance, we would get values by the RFID reader, the body thermometer and the pulse meter. Therefore we can consider *RFID reader*, *high temperature* and *low pulse* as known **evidence variables**, whereas *heating*, *pills taken* and *influenza* are unknown **nonevidence variables**. Moreover we can part the nonevidence variables into **query variables**, the ones we are interested in, and **hidden variables**. If we want to know the probability of an *influenza*, given a *high temperature*, a normal *pulse* and an "OK" from the *RFID reader*, the query would be:

$$P(i|t = true \wedge l = false \wedge r = true) = \frac{P(i, t, \neg l, r)}{P(t, \neg l, r)} = \frac{\sum_h \sum_p P(h, p, i, t, \neg l, r)}{P(t, \neg l, r)}$$
$$= \frac{\sum_h \sum_p P(h) \cdot P(p) \cdot P(r|p) \cdot P(i|h, p) \cdot P(t|i) \cdot P(\neg l|i)}{P(t, \neg l, r)}$$

Thus we loop over the possible values of the hidden variables and exploit the evidence variables to calculate the probability of the query variable. When the probability of no influenza $P(\neg i|t = true \wedge l = false \wedge r = true)$ is evaluated too, then the denominator need not be calculated explicitly, because it is the same for both terms and follows from the fact, that the probabilities of complementary events sum up to one.
The general case of probabilistic inference using Bayesian networks is known to be NP-hard [38, 7]. In [79] a few algorithms are listed for exact and approximate inference in Bayesian networks. Moreover, it is described how continuous variables can be handled using probability distributions, or avoided by discretization. Considering our example, pulse and temperature are such variables. They were discretized such that e. g. temperatures above a certain limit are considered high, and temperatures below are considered normal.

2.3 Ontologies

Before the term "ontology" was introduced into computer science, it was only known as a philosophical discipline which deals with the nature of being or the kinds of the existing and which can be sourced to ancient times (Aristotle, "Metaphysics" IV, 1) [21, 87].
In computer science an ontology forms the vocabulary of a knowledge domain and is used to represent data. In the context of *Semantic Web* [2] ontologies became very popular for establishing a common terminology between agents. Based on a shared ontology agents

are able to exchange knowledge. According to [88] a widely accepted definition was stated by Gruber [22]:

> An ontology is a formal explicit specification of a shared conceptualization.

Nevertheless there are differing definitions, which shows that there is little agreement so far. E. g., Swartout et al. [89] define:

> An ontology is a hierarchically structured set of terms for describing a domain that can be used as a skeletal foundation for a knowledge base.

Considering both definitions, an ontology can be regarded as a basis for a specialized knowledge base, whose underlying concepts and structure can be shared. Often the processability has a great influence to the structure. In order to exploit the information inherent in an ontology, semantic reasoning is applied. And with the expressiveness of an ontology language the complexity of the reasoning process increases rapidly. For this reason different languages were developed in the context of *Semantic Web*.
RDF (Resource Description Framework) as the basis of the *Semantic Web* allows only for simple ontologies, as it is intended to describe merely binary relations (properties) between resources and to provide semantics for generalization among properties and classes. For further expressiveness the *Web Ontology Language (OWL)* [37] was developed. Advancing on the predecessors *DAML* and *OIL* and founded on the theoretical basis of *description logics*, the language provides three levels of expressiveness, namely *OWL Lite*, *OWL DL* and *OWL Full* with increasing expressive power and reasoning complexity. Dependent on the actual language type OWL provides further means to express relations between classes. Among others these are disjointness, cardinalities and equality. [21, 37]
Quite common in use is *OWL DL*. Its basis is the *description language* `SHIQ` or rather `SHIN`. Although reasoning on `SHIQ` is known to be ExpTime-complete, it normally behaves quite well. According to [88] the "pathological cases" are quite artificial and occur rarely. The *Context Engine* [17], which will be described in the following section, is an example for an OWL-DL-based ontology.

2.4 Context Information

The term **context** is widely used in different meanings. Besides computer science it can be found in various fields like archeology and contemporary art. Although the definitions differ considerably, there is a common understanding that context surrounds the actual interesting thing and might influence its signification. In the dictionary[2] is is circumscribed as:

- what comes before and after a word, phrase, statement, etc helping to fix the meaning.

[2] "Oxford Advanced Learner's Dictionary of Current English" [27]

- circumstances in which an event occurs.

In computer science (also in linguistics) the term is known from formal language theory in connection with context-sensitive grammars. For ten years the term has been using in artificial intelligence and ubiquitous computing. In one of the first works Schilit and Theimer [81] describe context-aware computing as the "ability of a mobile user's applications to discover and react to changes in the environment they are situated in". This idea of context was refined in the following years. Dey [11] formulated a widely accepted definition:

> Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.

The definition includes, that context is always associated with an entity and that it is subjective to it. But still there are aspects, like time-dependency, not mentioned. In recent works the notion of context became plainer and yet broader. Based on Henricksen's [26] and Krause's [32] work Fuchs [17] stated the following characteristics:

- *Context is a subset of the setting of use.*
- *Context is determined individually, but not mutually exclusive for each application.* Context depends on the entity and its current environment, but different entities can have the same context partially.
- *Context is always associated with at least one entity.*
- *Any type of information can be considered context information depending on the application.*
- *Context information is time-dependant.* In the automotive scenario the environment and the context change permanently while driving, which effects for instance the air conditioning system.
- *Context information can be represented in many different ways.*
- *Context information can often not be obtained directly.* Here directly obtainable context is synonymous with information provided by sensors. But often such low-level context is not sufficient, and it needs to be refined applying for example sensor data fusion.
- *Interpreting context information requires meta-information.* The quality of context information is determined by the means of its provision. For example, the quality of a sensor can be given by its resolution and reliability.

Modeling Approaches When for the first time context-awareness was explicitly named as a desirable feature (e. g. [81]), the utilization of context information was usually restricted to one application. The question of reusability or sharing of context was not reflected. Also the representation of context played a minor role, since it always could be adapted to the respective application. Only in the following years first approaches were made to heave context onto a conceptual level.
An early approach comparatively is **Dey's "Architecture to Support Context-Aware Applications"** [10] that is to ease the employment of context information by providing special context servers. Here multiple applications can use an infrastructure, which obtains low level context from sensors, processes it and offers the results. In 2.7.1 we will describe this architecture (as a related architecture) in detail.
A recent approach is **Fuchs's "Context Meta Model"** [17], a modeling technique oriented on the characteristics stated above. Here the focus is on how context data is actually modeled and represented. As it is used in the AIP implementation (see chapter 4), we will give a brief survey over the modeling technique and the so-called **Context Engine** implementation.
The implementation of the *Context Engine* consists of a frontend for the semantic web framework **Jena**, providing an interface for the *Context Meta Model*. The *Context Engine* uses a knowledge base with an underlying ontology. Ontology and knowledge can be written using the modeling constructs provided by the *Context Meta Model*.
The basic modeling constructs for representing context are *entity classes*, *datatype classes*, *property classes*, *datavalue classes* and *quality classes*. *Entity classes* and *property classes* are comparable with *entities* and *relationships* of an *entity relationship model*. An *entity class* represents a group of things, persons, places etc. with similar characteristics, a property class a group of relationships between entities. Examples for entities are: sensor, room, person, software agent. Examples for relationships are: contains, isUsedBy, hasValues. (see also fig. 5.3)
A *datatype class* is the base construct for representing a (higher-level) type of data values such as: temperature, location, pulse. It is based on *datavalue classes* that represent simple data types like *string* or *floating-point number*. The *datatype class* location, for instance, could be given by a string (like "kitchen") or by two floating-point numbers representing longitude and latitude.
Quality classes can be attached to property classes and provide meta-information about them. If, for example, a room is supposed to contain some object, the property *contain* could be specified by a quality class "certainty" for expressing the certainty of this assertion.
Based on OWL DL the *Context Meta Model* also provides means to express specialization and equivalence. Classes can be subclasses of other classes or equivalent to them. Moreover, rules can be stated in SWRL to transform or to derive data from available information. For further details see [17].

2.5 Sensor Data Fusion

Nowadays one of the biggest problems in designing new processors is the increasing emission of heat. One reason for the heating is that information is thrown away. Already in 1961 Rolf Landauer demonstrated that the energy dissipation of $k \cdot T \cdot ln(2)$ is the lower bound for a bit operation[3], and therefore every deletion of information inevitably causes heat dissipation [33].
But why do we destroy information? It is so that almost every logical or arithmetic operation causes a loss of information ($1 \wedge 0$ yields more information than the result 0) and that we are interested in the result, for we normally cannot grip all the information inherent in the related problem. In short, all computing is nothing but lossy fusion of information[4].
On a higher level information fusion, especially **sensor data fusion**, is a very up-to-date topic, because sensor data processing is vital in many ongoing research areas like man-machine-communication and robotics. In all domains, in which the computer needs to obtain environmental information by the use of multiple sensors, sensor data fusion is applied, because a single sensor cannot provide the acquired depth of information [97].
Different from the impression that is received considering the low level example, sensor data fusion does not have to be lossy. Below we will distinguish different ways of sensor data fusion. But first let us give an overall definition:

> Sensor data fusion means that data from different sources is combined to get refined information about an observed environment.

The resulting information is "often abstract, generalized or summarized" [29], and, like in the low level analogy, the amount of data is often reduced. How sensor data is fused, depends on the application. There is always a purpose to be served, and therefore the fusion process cannot be considered as an isolated system [29].
Instead of sensor data fusion the terms data fusion or simply sensor fusion are often used and not distinguished.

2.5.1 Classification and Problems

We have seen so far an overall definition of sensor data fusion. Now we will classify different types of sensor data fusion and mark a few problems that might occur.
Brooks and Iyengar [5] divide sensor data fusion into three classes:

[3] where k is Boltzmann's constant of $1.38 \cdot 10^{-23} J/K$, and T is the temperature of the environment into which unwanted entropy will be expelled

[4] A way to reduce heat dissipation could be reversible computing [1]. Fredkin and Toffoli developed the conservative logic and suggested reversible logical gates that conserve the signals. Signals should not be created or destroyed [16].

- **Complimentary sensors** act like pieces of a jigsaw puzzle. They do not overlap, but form a more complete picture of the environment, when they are put together. Their fusion is easy, because there is no conflicting information. In the logistics scenario there is such a situation, when data about the track of a container is exchanged. The data from the neighbouring AIP module only adds to the picture – provided that the monitored areas are disjoint.

- **Competitive sensors** deliver equivalent information each. Due to the redundancy potential fault and failures of single sensors can be tolerated. In automotive applications, for example in the *Electronic Stability Program*, failures can have serious consequences. A configuration with three identical sensors can tolerate the failure of one sensor.

- **Cooperative sensors** merge their data to derive information that could not be provided by neither sensor alone. An example of cooperative sensing would be the estimation of the driver's state in the automotive scenario. If the monitored pupils become smaller, it could be necessary to know the position of the sun as well to determine, whether the person is sleepy or just blinded.

Although the complimentary fusion is probably a lot easier to handle than the competitive fusion and the competitive fusion less difficult than cooperative fusion, there are main problems, specified by Wu [97], that actually concern all types:

- Sensors might use different physical principles.

- Data is given in different formats.

- Data is refreshed in different intervals, or depending on the sensor automatically or not automatically refreshed.

- The generated information may have different properties concerning resolution, accuracy and reliability.

2.5.2 Approaches

In recent years several approaches have been made to implement information fusion. The focus was on image fusion, which is beyond the scope of the AIP module, because the amount of data requires specialized means for a fast processing. Nevertheless we will describe image fusion in brief, before we change to the techniques more appropriate for the AIP module.

Image Fusion The reason why image fusion is so popular is that image data is digitized and can be represented by a matrix containing numbers. That is why image fusion is often a subject of applied mathematics [13]. Van Genderen and Pohl [19] define:

> Image fusion is the combination of two or more different images to form a new image by using a certain algorithm.

Application areas are medical imaging and remote sensing, e. g. with the merging of image data from different observation satellites. Further information about image fusion in general and its use in remote sensing, together with a survey of techniques can be found in [73].

Rule-Based Fusion A simple and coarse way of sensor data fusion is achieved by using a **rule engine** (see 2.1). All rule engines have a set of IF-THEN-rules that look like:

> IF $condition(data_1, ..., data_n)$ THEN $consequence$

Here we can consider rules as conditional functions that will fuse the values $data_1, ..., data_n$, if they satisfy the condition. The consequence can be interpreted as the result of the fusion.
Thus the rules control and execute the process, and it will not matter, if data is refreshed in different intervals, since it can be stored in the rule engine's knowledge base.

Bayesian Network It was already touched on in section 1.4 that sensor fusion often comes along with uncertain data. Either already the initial values or at least the results are uncertain. That is why a Bayesian network (see 2.2.2) could be applied.
Some of the problems stated above can be handled easily by a Bayesian network. Of course, it is not concerned with the physical principles of the sensor or the data format – that is part of the sensor interface –, but it can deal with data arriving at different times by storing the values in its nodes, and also with differences due to the reliability. The reliability can be modeled by a probability distribution (cf. 2.2.2).

2.6 Learning

Almost with the invention of the computer, the question arose, whether computers could be programmed to be able to learn. Today there exist a lot of learning algorithms for computers, but, even if progress is made, the computer is far from attaining human abilities. The initial euphoria about artificial intelligence was based on successes in domains, which seem to be tailored to computers, like chess. In chess there is only a rather small number of objects, 64 squares and 32 chessmen. The rules are strict and accurately defined. Therefore chess can be easily emulated by computers[5]. Meanwhile the best chess

[5]The first chess programs did not use learning algorithms at all, only fixed coded algorithms like *alpha beta pruning*.

programs are on par with the human world champion. But many tasks, which seem simple compared to chess and which every child can handle in early life like face recognition, are severe obstacles for computer programming, because they cannot be reduced to a handful of accurate rules. [3]
In order to ease the task and to perform well, the development and employment of learning systems is normally problem-oriented and reduced to a restricted domain. Even so the results surprise sometimes. E. g., due to learning systems computers are capable to drive cars along roads, to recognize handwritings and spoken words [38].
Even though everybody is convinced to know what learning means, it is not so easy, to give a definition. With respect to computers Mitchell [38] defined:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

The task T of a chess learning program would be playing chess, its performance measure P the percentage of games won against opponents. The training experience E would consist of games against humans, other computers or itself. A learning system requires in addition the specification of the type of knowledge to be learned, a representation for this target knowledge and a learning mechanism.
To explain a learning chess program, we can go along the lines of [38] (checkers example). The type of knowledge, that is to be learned, is a function *ChooseMove* that is to find the best move from a set of legal moves. Its representation is a compromise between accuracy and efficient computability. That is why the function that is learned by the system, is only an approximation to the ideal *ChooseMove*. Analogous to [38] we choose a simple function that rates a board state due to the weighted sum of board features, like the number and type of pieces left on both sides. The learning mechanism's task is to adapt the weights.

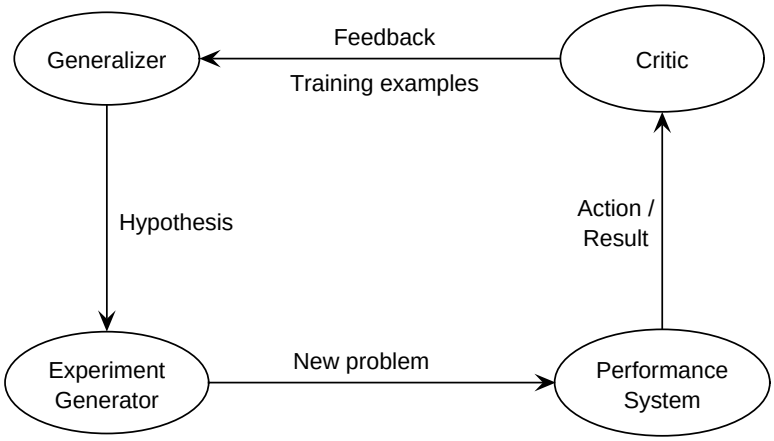

Figure 2.2: Learning program. [38]

In general a learning system is divided into four components, namely in the *Performance*

System, the *Critic*, the *Generalizer* and the *Experiment Generator* (see figure 2.2 and [38]). We consider these modules with regard to our chess example:

- The **Performance System** must solve the performance task, here playing chess by applying the learned target function. It takes a new problem, in this case a new game, and passes the game protocol as output to the *Critic*.
- The **Critic** evaluates each board state b_i with $V_{train}(b_i)$ taking into consideration, whether the game was won or lost and when in the game this position was reached. The pairs $< b_i, V_{train}(b_i) >$ are transmitted to the generalizer as training examples.
- The **Generalizer** fits the target function $\hat{V}$ to the training examples. This is frequently achieved using the least mean squares or LMS training rule, which determines the weights to minimize the sum of the squared deviations from training data $\sum_{<b,V_{train}(b)>} (V_{train}(b) - \hat{V}(b))^2$.
- The **Experiment Generator** generates a new problem for the *Performance System*, which might depend on the hypothesis about the target function $\hat{V}$. In chess this could be a special position or just starting a new game.

These four modules also constitute a learning agent (see 2.3, [79]). However, here the interactions with the outside world must be taken care of. In the case of chess the *Performance System*'s output is directly evaluated by the *Critic*. In case of a robot the *Performance System* initiates actions, like motions. These can only be evaluated, if there is a feedback through sensors.

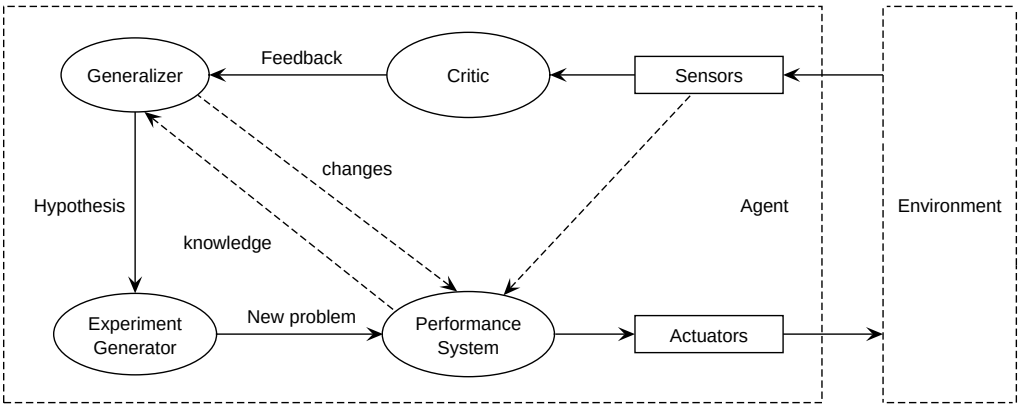

Figure 2.3: Learning agent. [79]

An other example for a learning agent could be the AIP agent in the logistics scenario. It could have an *Experiment Generator* providing different routing strategies and get feedback, namely the required time, when the container reaches its destination. The *Generalizer* in the AIP agent can learn a function that selects the next path depending on the traffic with the objective to speed up the overall flow.

As the destination can be remote, the feedback should be returned via a server providing the involved AIP modules with the feedback.
In the smart elderly care home scenario there are two possibilities for using learning algorithms. On the one hand a learning system could be trained prior to its use to classify situations correctly. On the other hand the behaviour of the user could be learned to start smart home applications automatically.
Considering the former possibility we possibly have a different type of learning. So far the learning systems were unsupervised, meaning that they adapt themselves without external interference. Supervised learning on the other hand is a technique for creating a function from user-defined training data. One example algorithm is the *nearest neighbour algorithm*.
Using the **nearest neighbour algorithm** [83] the learning system is trained before its employment. Training instances and instances that are to be classified, are given as vectors of attributes. After the training phase, new instances are rated like the nearest training instance (or selectively like the majority of the d nearest training instances). In order to measure the distance between instance vectors, the vector's arguments have to be real numbers. In the elderly care scenario, for example, an instance could be given as a vector containing the attributes *pulse* (real number), *body temperature* (real number), *pills taken* (boolean mapped to $-1, 1$) and state ($sleeping \rightarrow 0, sitting \rightarrow 1, moving \rightarrow 2$, etc.).
To reduce the time for the classification a binary decision tree is built up. For this the vector space is partitioned by hyperplanes (each rectangular to one axis). This is done recursively by dividing resulting partitions anew, until each training instance is separated from all others. Here every hyperplane is mapped to a tree node representing a binary decision, namely on which side an instance that is to be classified, is located.
To classify an instance means therefore to traverse the tree down to the leaf, which represents the cell of the instance. Unfortunately, due the partition into rectangular cells the cell's training instance is not necessarily the nearest training instance, so that, perhaps, the neighbouring cells have to be considered as well.
Beside separating supervised and unsupervised algorithms other distinctions are customary. But, as we do not need any special learning algorithms in the subsequent chapters, we go without further descriptions of algorithms and algorithm types and refer to the books and articles mentioned in this chapter ([38], [79], [83]).

2.7 Related Architectures

The need to process sensor data arises again and again. Hence, it is surprising that there exists hardly any literature describing architectures of frameworks for local sensor data processing. In the fall of 2005 an extensive query led to only a few architectures that satisfy just some of the requirements. We distinguish four types of architectures and present one representative of each:

- The "Architecture to Support Context-Aware Applications" belongs to the **con-**

text servers that extract sensor data, process it and offer the results as context information.

- The "Policy Based Adaptive Services for Mobile Commerce" is a **framework to generate agents** to gain and process context. In order to serve a higher-level application it has to be adapted to the task. This is quite similar to our idea of autonomous sensor information processing.
- The "Singularity Architecture" stands for an architecture that processes data from a **particular type of sensor** only, here RFID events. Such architectures are usually specialized for a single task and a destined type of application.
- "Motes" are little devices connected with sensors that transmit their data via radio communication. They are used in sensing networks pushing all the data to a single data sink, where it is processed. We sort them to the group of means for **centralized data processing**.

In the following sections these architectures will be presented. Each of the sections will be concluded by an evaluation based on the requirements stated in chapter 1.

2.7.1 An Architecture to Support Context-Aware Applications

In section 2.4 Dey's [10] architecture was briefly mentioned as a step towards a new way of handling context. Context should be reusable and shareable by several applications. For this uniform access must be provided. In order to ease the use of (additional) context in existing or new software, the architecture provides means to treat context similar to user input.
This architecture belongs to a type of software that is normally called **context server**. Such a server's task is quite similar to the one of the AIP module. It has to collect and process sensor data and it offers the refined results to applications. We will describe and discuss this architecture as a representative for all context servers.
As shown in figure 2.4 the architecture – in other papers [17] also referred to as toolkit – distinguishes *Widget*s, *Server*s and *Interpreter*s. All components can run on different computers and have the same means for communication, as they are all subclasses of the so-called *BaseObject* that provides the methods for the communication. *Widget*s encapsulate the sensors, that deliver the context information, and offer the data via polling or subscribing. They are defined by their attributes and callbacks. Although applications can retrieve information directly from such low level *Widget*s, they should request *Server*s that are extended *Widget*s. *Server*s combine the information of several *Widget*s and offer the compound context information to any application or component. The subscribing application or component can specify the callback event, the *Widget* attributes of interest and conditions under which the data is returned. By these options the information is filtered. *Interpreter*s are responsible for the interpretation of context and can be consulted by any *Widget* or application. Every *Interpreter* offers a function *interpretData()* that

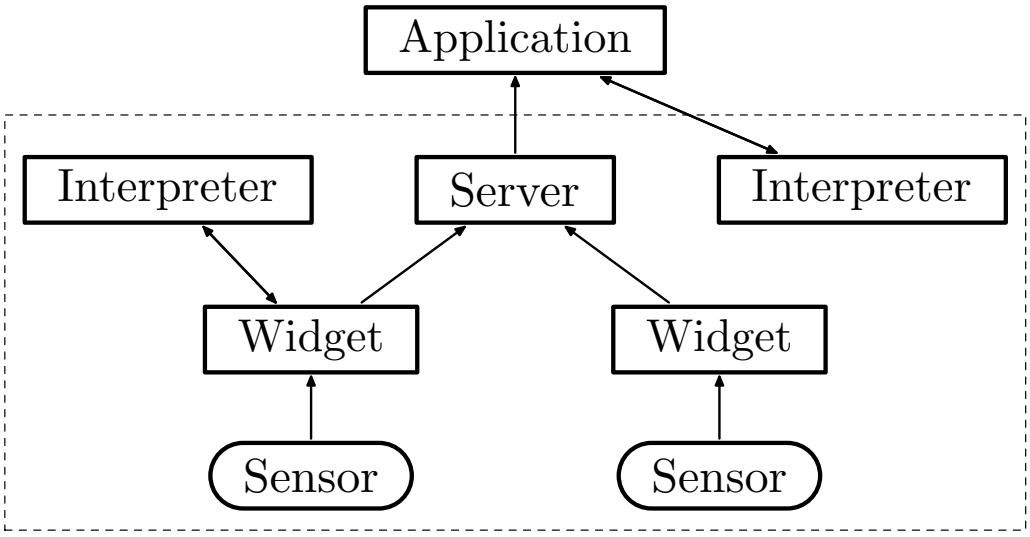

Figure 2.4: Architecture To Support Context-Aware Applications. Relationship between applications and the context architecture. Arrows indicate data flow. [10]

takes specified attributes, interprets them and returns the results.

Review It is obvious that a context server providing data for several applications must be more general than the AIP module serving only a single application. In this connection it is not only interesting to see in how far this architecture fulfills the requirements of section 1.4, but also in how far the AIP module can function as context server.
Because of its powerful processing tools which can take over of a substantial share of data processing, the AIP module is more destined for special tasks. But it can be configured as context server as well, thus combining *Widget* and *Server* from Dey's architecture. In any case it makes sense to make the interface available to various applications. Then also several components of the same system could be notified by the AIP module, e. g. a backup data base in addition to the actual receiver.
The rest of this paragraph analyzes which requirements are met by Dey's architecture:

- **Means for processing.** As the *Widget*s only retrieve data from sensors and *Server*s only aggregate the information, the main processing task lies in the *Interpreter*s. Their implementations are not restricted, and so they can use a rule engine for instance or more sophisticated tools to interpret the input.
  Admittedly the architecture's philosophy would be violated, when an *Interpreter* stored data or when it were specialized for a certain *Widget*, such that it could no longer be used by all *Widget*s in the same way. Moreover, the reusability would be impaired. To allow demanding data processing by *Interpreter*s would mean, that several of the architecture's principals had to be given up.

- **Communication.** Via communication all components are accessible. Remote sen-

sors can be used.

- **Locality.** Since every component can communicate with any other and since this communication is crucial for any useful application, one cannot talk of local processing.
- **Generic interfaces.** The implementation of the component specific methods is unrestricted. The number of the methods' parameters and their types can be set by the application designer. Hence, the entire architecture is generic.

We can say that with some restrictions this design can be used as an architecture for sensor data preprocessing. One handicap is, that the *Interpreter*s must be called explicitly and that they should use only data transferred with this call. Therefore data is neither stored nor shared, so that the employment of sophisticated processing tools is quite useless. Moreover with the lack of local processing, one of the main AIP principles is infringed.

2.7.2 Policy Based Adaptive Services for Mobile Commerce

In mobile commerce it is important to ease the access to mobile services to improve their acceptance. Presetting should be simple or avoided entirely. Therefore in [78] it is suggested that context information is used to do the presetting automatically. The user types in his preferences once, for example that he wants low cost connections at any rate, and every time when a service is used the preferences are taken as context information for decision-making.
The architecture consists of agents that contain one to four (different) modules (cf. figure 2.5), namely the *Policy Decision Point* (consisting of the *Pattern Matcher* and the *Agenda*), the *Policy Enforcement Point*, the *Policies* and the *Context* module. These modules form the parts of a rule engine: *Policy Decision Point* and *Policy Enforcement Point* together represent a rule interpreter, *Policies* is the rule base, and finally *Context* provides the data. A priori every agent can have any subset of these four modules.
The decision-making process is done by such a split rule engine. At the *Policy Decision Point* based on the context information, the *Pattern Matcher* chooses the rule to be fired and adds it to the *Agenda*. At the *Policy Enforcement Point* the rule is fired. If the parts of the rule engine are located on different agents, the data has to be transmitted between them.
The architecture distinguishes three main agent types, namely services, mobile devices and sensors (cf. figure 2.6), that are distributed all over the network. Services and mobile devices can contain all four module types, whereas the sensors only provide context information. To reduce unnecessary transmissions the context is kept locally, and it is sent over the network only on request.
The prototype based on this architecture is a service for downloading movie trailers. The context information consists of technical information and user preferences. Quality, speed and cost are set by the user. Screen resolution, supported video encodings and available network interfaces are part of the mobile device that is used. Furthermore there is a

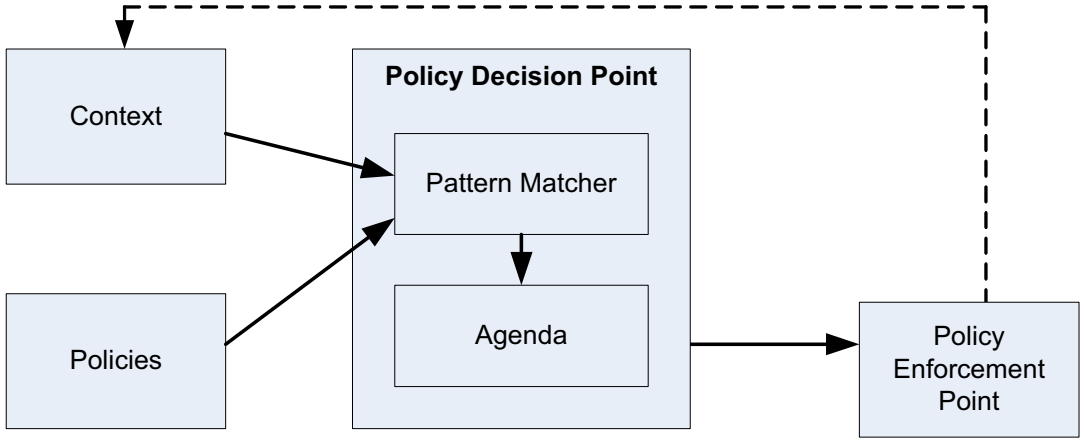

Figure 2.5: Policy-Based Adaptive Services. Basic modules of agents. [78]

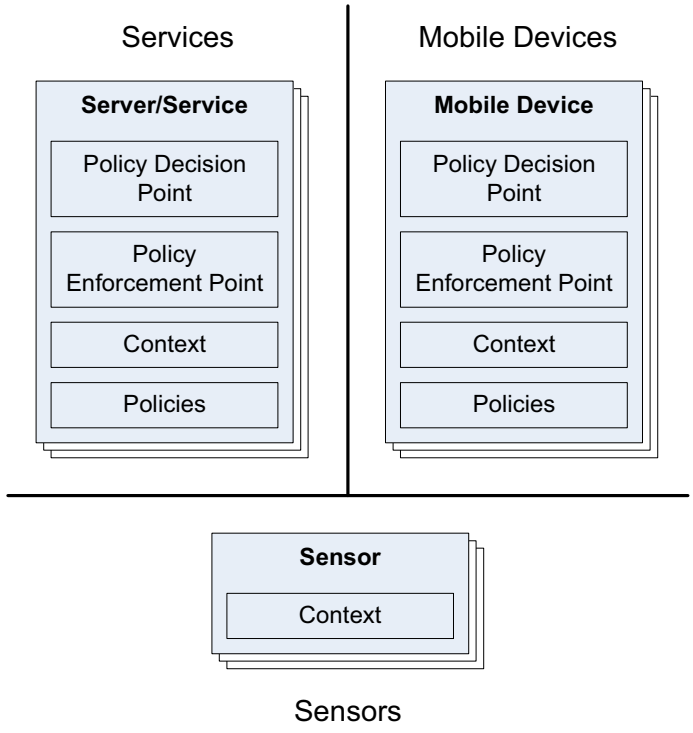

Figure 2.6: Policy-Based Adaptive Services. Agent types. [78]

selection of different network agents and video agents distributed over the network that provide context information about their services. A decision agent in the network chooses rules depending on the complete context, and the mobile device agent executes these rules.

Review Although this architecture has a different focus and is adapted to the requirements of mobile commerce, it roughly fits to our idea of autonomous sensor data processing. The low-level context information in this architecture is provided by sensors according to our extended sensor definition, and it is processed to serve an application on a higher level. Therefore we can investigate the architecture and see in how far our requirements are satisfied:

- **Filtering, fusion and learning.** The context information is processed by a rule engine. As stated in section 2.1, a rule engine allows filtering and simple data fusion. Learning of any kind is not supported.
- **Locality and communication.** Since in this case the emphasis is put on distribution and since module to module communication is essential, locality is present only in a marginal way. However, to keep communication costs low care was taken to process data locally as far as possible. For that purpose the rule engine was split up in its parts. In the prototype, for instance, only the rules have to be sent to the mobile device and not the context on the basis of which the rules were selected.
- **Generic interfaces.** The sensor entity of the architecture is not explicitly described. There is no framework supporting the connection of sensors. But in any case, seen from outside, the sensors behave the same way. As all context information is represented by RDF (Resource Description Framework), all sensor entities that do nothing but gaining context and sending it, must convert the sensor data into RDF. Equally there is no description how the user interface can be connected with the mobile device entity.
- **Rules and uncertainty.** Uncertain sensor data or uncertain knowledge in general is not considered. All decision-making is done by a rule engine.

This architecture can be seen as a set of rule engines with possibly shared knowledge and rules. The normal use-case will be that the sensor information contemplable for the processing is widespread, whereas we considered scenarios, in which local processing is useful and possible. Therefore this architecture is not appropriate for our aims. Moreover, beyond the lack of locality, the means of processing are restricted to rule engines, and there are no hints for possible extensions.

2.7.3 Singularity Architecture

Since new application areas are researched for the RFID technology and since it gains ground especially in logistics, there is an increased need for tools and frameworks. Retailing chain store giants like Metro and Walmart investigate in this technology and it is expected that competitors will follow.
Many RFID systems, especially in trade, use the Electronic Product Code (EPC) that

replaces the Universal Product Code (UPC) barcode system. In contrast to the UPC system every product gets its own specific identity number, the 96 bit long Global Trade Item Number (GTIN), which is contained in the EPC tag. The EPC system is managed by EPCGlobal Inc. [59].

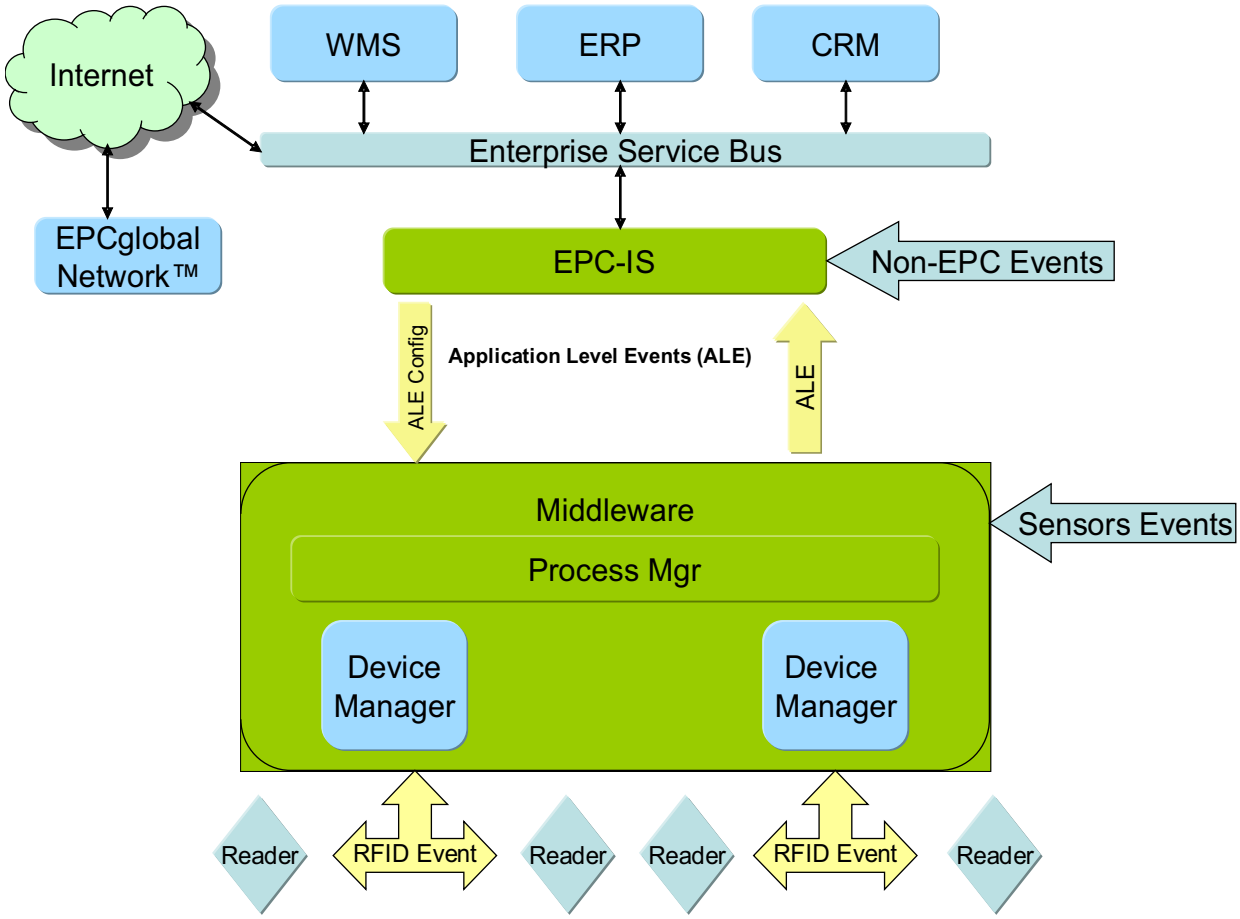

Figure 2.7: Singularity architecture [75].

The Singularity architecture [75, 76] is a solution for the application of RFID and EPC in supply chains. All levels are included starting at the lower end with the RFID scanners reading the tags, up to the distribution of processed data inside the enterprise and even further. The architecture consists of the *EPC Information Service (EPC-IS)* connected to the *Enterprise Service Bus* and a three layer middleware. Modules of one layer are notified about events by the modules of the layer immediately below.
At the lowest level the *Interrogator Agent*s act as drivers for the RFID readers, which can be of arbitrary type or brand. Several of these *Interrogator Agent*s can be connected to a *Device Manager* on the layer above, who is listening to their *Sensor Event*s. The *Device Manager* represents the first filter, it builds *Reader Event*s from one or more *Sensor Event*s and sends them to the *Event Message Space*. *Reader Event*s of different *Device Manager*s can belong to a *Logical Sensor Event*. On the next level the *Event Process*

Manager subscribes to that *(Logical) Sensor Event*s it wants to monitor and is notified, when these events are available in the *Event Message Space*. It uses a rule engine to filter events and to map them to higher level events, called *Business Event*s. These *Business Event*s then leave the middleware as *Application Level Event*s that are sent to the clients, namely the *EPC-IS* instances. When additional data from the company network is needed to handle the EPC events, it will be linked to the EPC-IS. Eventually the EPC events will be available via the *Enterprise Service Bus* inside the company and via the *EPCglobal Network* to business partners.

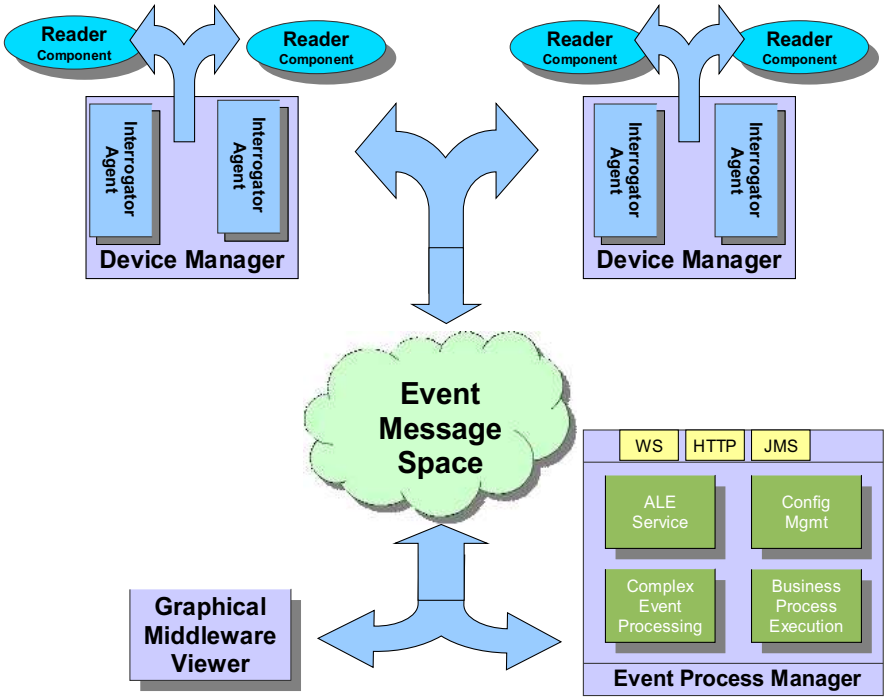

Figure 2.8: Singularity middleware [76].

As an alternative the architecture can be viewed without *EPC-IS* clients. So an arbitrary application can use the results offered by the *Process Manager*. Moreover in order to increase the feasibility, the *Process Manager*'s rule engine can process the *Business Event*s, its (intermediate) results, anew together with other events from the outside, which, however, are not specified in the paper.
The Graphical Middleware Viewer in figure 2.8 is used for the configuration and administration of the middleware. The Device Manager provides an interface that can be used to configure the Interrogator Agents. Besides the rule engine of the Process Manager must be provided with rules. And physical information like IP addresses must be set.

Review This architecture is tailored for RFID readers as only – or at least as essential – sensors and is not meant to fulfill all requirements. Nevertheless it will be interesting to

see, which requirements are met:

- **Filtering, fusion and learning.** By selecting Sensor Events for the (Logical) Reader Events, filtering takes place in the Device Managers. In the Process Manager a rule engine is used to map the low level *Reader Event*s to *Business Event*s. The rule engine allows filtering and simple data fusion. Learning is not possible.
- **Communication.** Due to the different layers "vertical" communication between *Device Manager* and *Process Manager* and between *Process Manager* and *EPC-IS* instances is necessary. Direct communication between modules of the same layer is not possible. *Sensor Event*s of *Interrogator Agent*s first come together in the *Device Manager*, *Reader Event*s of *Device Manager*s not before the *Process Manager*.
- **Locality.** Instances of the middleware preprocess RFID data locally and serve applications on a higher level, namely the *EPC Information Service*s. Thus local processing is granted.
- **Generic interfaces.** For the configuration and administration of middleware the *Graphical Middleware Viewer* is used. It provides a generic interface to configure all middleware components, especially the rule engine, the *Interrogater Agents* and eventually the RFID readers.
- **Rules and uncertainty.** As mentioned above the middleware is not able to deal with uncertainty and does not provide any means for learning and adaptation. It uses a rule engine for the processing.

We note that this architecture possesses a generic interface and that it obeys to the principal of locality. For our purposes the restriction to RFID or EPC data is a drawback, even if other sensors can be implemented as context sources. How this is achieved is not specified. A further disadvantage is the restriction to a rule engine as the only processing unit and the missing extensibility. Although rule engines are quite powerful, there are cases where additional tools could be used more efficiently.

2.7.4 Motes

For the "Habitat Monitoring on Great Duck Island" [44] *Motes* (developed by UC Berkeley) are used to collect data from sensors distributed over the habitat. *Motes* are small devices, powered by batteries, that are to read data from sensors and send it to the *Basestation*. Together they build a multi-hop network, i. e. they can forward messages from other *Motes* towards a gateway, which finally transfers them to the *Basestation* via the *Transit Network* (see figure 2.9).
In this scenario all sensor data is stored in a database which makes it available via the internet. It is not further processed. In general sensor data collected this way would be processed by a central computer like the *Basestation* forming a single data sink – in

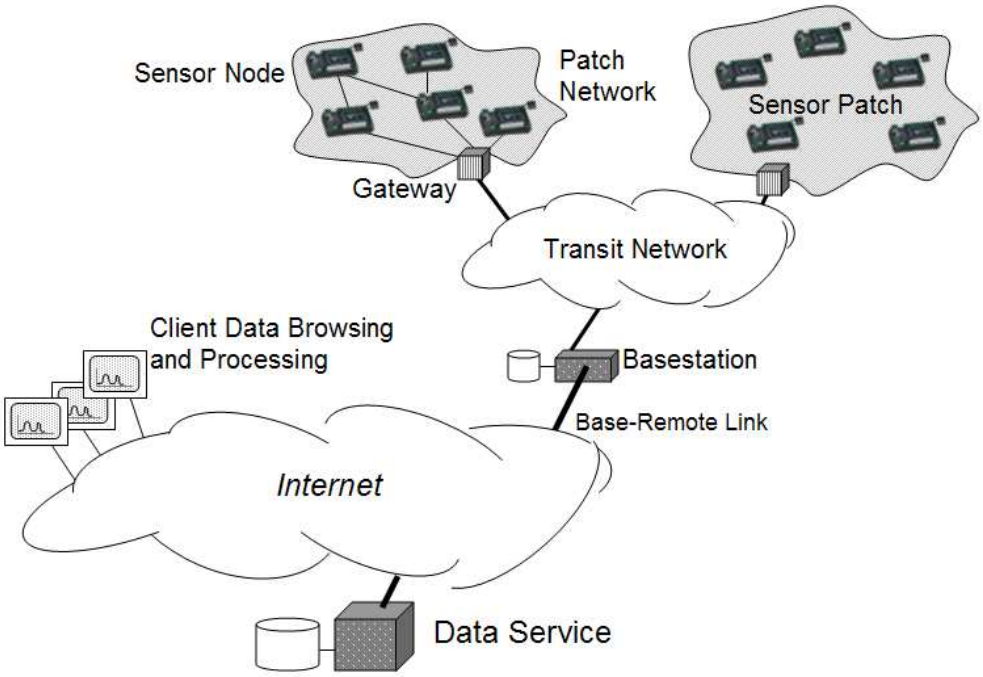

Figure 2.9: System architecture for habitat monitoring [35].

contrast to all the *Motes* being data sources. One example is the implementation of the elderly care scenario (see 1.3.4) with *Motes*, which is described in [77].

Review As *Motes* are not concerned with data processing, we will not examine the requirements in detail. Furthermore it is obvious that locality is not given, and therefore we consider this architecture only as an example for how data can be retrieved, but which contradicts the idea of the AIP module. Its use is appropriate for the collection of data, but not for the preprocessing in order to disburden the main application.

Summary

In order to establish a basis for the examination of architectures and their implementations, terms and tools were explained. At the end of this chapter related architectures were discussed, which we will be followed by the AIP architecture in chapter 3.

Chapter 3

Architecture

This chapter introduces a new architecture for autonomous (sensor) information processing (AIP). It is called AIP architecture. Based on the requirements from chapter 1 and the techniques described in chapter 2 a design rationale is framed that establishes a basis on which the architecture is developed.

This chapter describes the AIP architecture. It starts with a design rationale that establishes a basis for further reasoning in the succeeding sections. For this the requirements stated in 1.4 and the result of chapter 2 are consulted. After the general survey (3.2) the layers of the architecture and the inter-AIP-communication are described in separate sections, followed by an evaluation and further requirements concerning the software to choose.

3.1 Design Rationale

This section explains our approach to designing the AIP architecture. Besides the requirements stated in section 1.4, the tools and concepts for knowledge engineering, described in chapter 2, form the basis for our considerations.
The main task of an AIP module is getting raw data from arbitrary sensors and processing it for an application (cf. 0.1). Therefore a *generic sensor interface* is needed that can handle all kinds of sensors and that provides the module with standardized information.
As we assume that an AIP module is used by a single application (cf. 1.4), it is reasonable to let the AIP module be initialized and manipulated by this application. The necessary interface we call *management interface*. Its task is to receive commands from the application and also to provide it with processed information.
In 1.4 we found out, that a generic AIP module to which arbitrary tools can be attached, is preferable to a set of fixed AIP modules. Different Implementations of rule engines, Bayesian networks, reasoners, learning components and other tools thus can be connected to the AIP module. The advantage is, that for any application and hardware the optimal software can be chosen. E. g. in the info point scenario, where the AIP module is located in a mobile phone, saving memory space may be important, whilst in the automobile

performance is emphasized.
The decision to allow the use of arbitrary tools has the following consequences:

- For the initialization and manipulation of the processing tools, the application must have access to them via the management interface. For better exchangeability of tools the management interface should be generic.
- Since data must be available globally, it is advisable to install a **central knowledge base to which all tools have access** and into which all (sensor) data is written. Allowing direct exchange of processing tools is hardly feasible and would unnecessarily complicate program logic and consistent data storing.

To provide the connected tools with data, the central knowledge base should be able to notify the processing tools of any new data. Nevertheless the processing tools need to have active access to the knowledge as well. On the one hand they must write back results and on the other they can read old data, that is no longer kept in their local memory.
The knowledge base must be well structured to ease the input of data as well as to improve access to the bulk of sensor data, application data and processed data. For that purpose two concepts were introduced in chapter 2, ontology and context.
In order to open and close sensor connections, it must be possible to send sensor requests. For convenience it is advisable to allow polling as well as notification methods. The latter case is useful, for sensor data will be sent to the knowledge base and forwarded to the processing tools automatically.
These sensor requests as well as the notifications for the application, should be sent either by all tools or by a single tool. When all tools can write to the interface simultaneously, there are no means to coordinate the actions and concurrent requests and messages may arrive at the interface. Therefore one should specify one tool for this purpose. Of all the tools considered the rule engine probably is the best suited one, because it is quite conveniently controlled. Sensor requests and notifications from other tools can then easily be written into the knowledge base, whereupon the rule engine will forward them.
A further requirement from section 1.4 is the inter-AIP-communication. It is needed to exchange data in case of shared sensors or shared situations. One example is the “changing to an other room” in the elderly care scenario.
Before proceeding to the description of the architecture, we will summarize the results of our considerations:

- There should be one central knowledge base that underlies an ontology and that preserves correlations between pieces of data. Associated context data must be easily available, so that data can be identified or augmented with it.
- Various processing tools must be integrable. They need to have access to the knowledge base and must be notified of any new data. That puts demands to the knowledge base’s interface.
- For the maintenance of knowledge base and processing tools a generic interface should be provided. The same should be true for the sensors.

- There must be means to query the sensors and to notify the application about events or (intermediate) results. A rule engine seems to fit best, because it can be handled most easily.
- Communication between AIP modules for the exchange of (sensor) data is required.

The use of a central knowledge base, shared by all other components, suggests a star shaped representation (star topology) as shown in figure 3.1. The knowledge base is placed in the center surrounded by the processing tools, the interfaces and the communication module. These exterior elements supply or retrieve data.

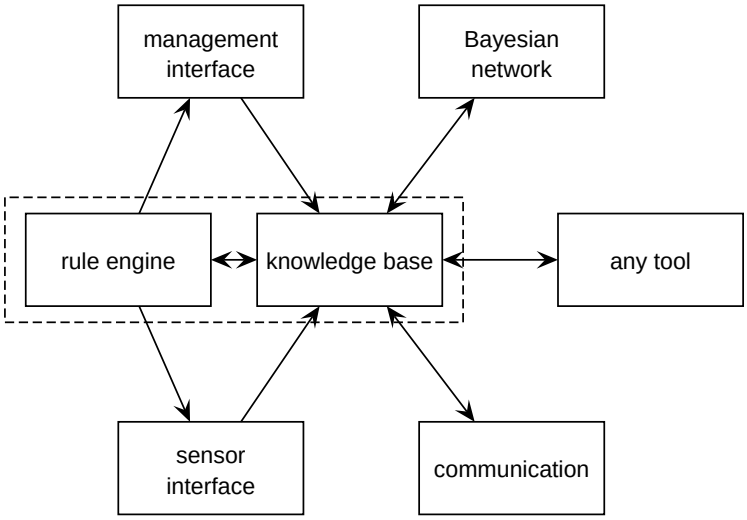

Figure 3.1: AIP architecture. Star topology.

This design has the advantage, that it displays the main data streams and highlights the knowledge base as central link. The rule engine's special role somewhat mars this design. That is avoided, when rule engine and knowledge base are considered as compound – as indicated in figure 3.1. A disadvantage is, that the interfaces are diminished in their importance and that this design neglects manipulations of the processing tools by the management interface.
An alternative to the star topology is a three layer model (see 3.2). The knowledge base is located in the middle layer, surrounded by the processing tools. The interface layers are over (management layer) and below (sensor layer) it. Compared with the star topology the interfaces now get an appropriate weight and are not leveled amongst the processing tools. Their fundamental function, the interaction with sensors and application, is emphasized and separated from processing. Moreover, the presentation can be extended by the levels of the physical sensors and of the application.
This design has the disadvantage, that inter-AIP-communication is not marked off. Hence a third design with four submodules can be thought of, where communication is a submodule of its own. But since in the case of local processing the relevance of inter-AIP-communication is rather low, we will stay with the three layer architecture. In the next

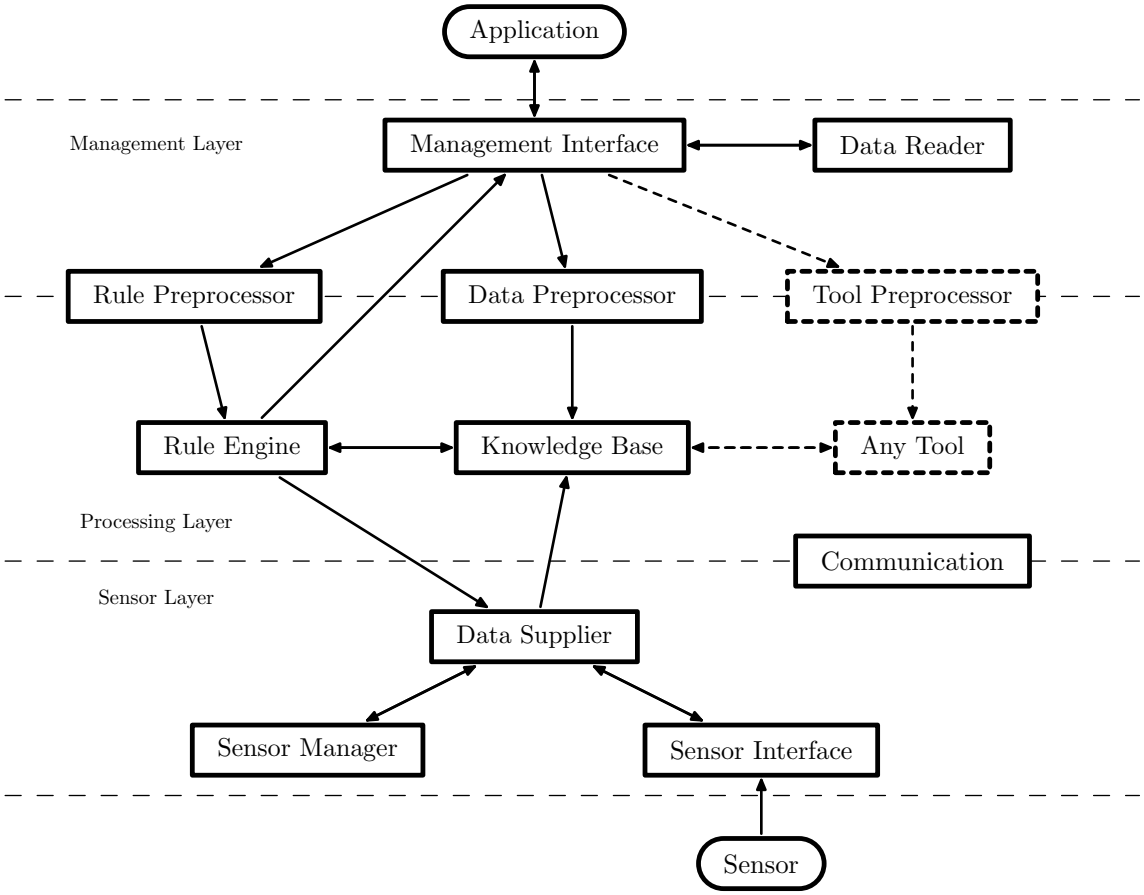

Figure 3.2: AIP architecture. Three-layer-design.

section we will give an overview of the AIP architecture and examine in which layer communication should be placed.

3.2 General Survey

The AIP-module is used for the autonomous preprocessing of sensor data. After the AIP-module is initialized, sensor data can be read automatically and processed with various tools. The results are forwarded as messages to the main application. Moreover, AIP modules can communicate with each other to exchange (sensor) data (inter-AIP-communication).
As mentioned and indicated by the dashed lines in figure 3.2, the AIP architecture consists of three layers, namely the sensor layer, the processing layer and the management layer. Sensors are connected to the sensor layer, which retrieves their data and makes it available by sending it to the knowledge base of the processing layer. There several tools extract the data, process it and write the results back to the knowledge base. One designated tool, a rule engine, can trigger two types of actions outside of the processing

layer, namely sensor requests and messages to the application via the management layer. The management layer will also receive input from the application, after the initialization mainly for registering new sensors and adapting the processing tools, e.g. by adding further rules to the rule engine or new facts to the knowledge base.
Inter-AIP-communication can either be associated with the sensor layer or with the processing layer, depending on where the transmission data is taken from. If just raw data is to be transmitted, the communication module can be viewed as part of the sensor layer. If in addition high level data is required, the knowledge base in the processing layer must be accessed, which demands greater effort.

3.3 Sensor Layer

The sensor layer is used to connect sensors to the AIP and to provide data to the knowledge base of the processing layer. The sensor layer consists of the sensor interface, the sensor manager and the data supplier.
The **sensor interface** needs to be generic so that every sensor consistent with the extended definition of 1.1 can be used. Due to the variety of sensor types and manufacturers it is impossible to provide drivers or adapters for all sensors on the market.
The sensor interface must allow for the pull mode (or polling mode) and for the push mode:

- In the **pull mode** or **polling mode** the sensor sends the last or actual measured value on request. Normally the communication is synchronous. The requesting thread waits until the sensor returns the value. In this case the value is not returned to the caller, namely the rule engine, but to the knowledge base that runs in its own thread. Call and return are decoupled and therefore the communication is not synchronous.

- For the **push mode** the sensor must provide a subscription mechanism. Subscribing to a sensor means that the subscriber is notified, whenever a sensor value is measured. The subscriber is sometimes called listener or event handler, as he listens to and handles notification events. The subscriber's function responding to the event is referred to as callback function.

The only subscriber to the sensors is the **data supplier** who forwards sensor data to the knowledge base in the processing layer. It is the only upward interface and it also receives the sensor requests (push or pull).
In sensor requests the relevant sensor can be specified directly by its ID or URL or by its properties, e. g. by the quantity it measures and additional context information. In both cases the **sensor manager** will be called on to choose a suitable sensor from its catalogue.
The AIP module is informed about newly available sensors via the management interface of the management layer. Mobile, wireless sensors communicating with the interface can

be marked "in range" or "out of range" via the push registry.

3.4 Processing Layer

The core of the processing layer is the **knowledge base**. It gets its data from the data supplier (cf. 3.3) and from the application via the management layer. On the knowledge base a **rule engine** accomplishes two tasks. On the one hand it transforms low level sensor data into high level application data, on the other hand it processes high level data and initiates two types of actions, namely sensor requests and notifications to the application (see below).
Selectively a second rule engine can be used, so that these two tasks are also physically separated and maybe performed more efficiently. In this case the rule engine for preprocessing sensor data can also be a front end of the knowledge base. Otherwise it can be an advantage to store the sensor data in the knowledge base allowing any tool that needs it to access the sensor data as well.
Conversion of low level data into high level data is necessary, to ease the application programmer's work. Rules that use data on a higher level are easier to program. As an example it is difficult and hence less reliable to state correctly a condition of the type "motion sensor 3 changes to 1". If we assume the elderly care scenario the application programmer would prefer something like "the person is in the bedroom".
The knowledge base underlies an **ontology** that can be logically divided into two parts. The first part describes the relationship between sensors and the AIP module and is provided by the AIP. Naturally any context data referring to AIP module and sensors is stored here as well as the incoming sensor data that is identified with the sensors depicted within the ontology.
The second part is the one that represents the ontology of the application and has to be written by the application developer. When high level data is gained from the sensor it needs to be sorted into the world model of the application.
The actions that are triggered by the rule engine are – as mentioned above – sensor requests and notifications to the application. Requests for local sensors are sent to the data supplier (cf. 3.3) whereas requests for remote sensors are processed by the communication submodule that obtains the data via inter-AIP-communication (cf. 3.6).
The notifications determined for the application are passed to the management layer (cf. 3.5) that does the forwarding.
The question which type of rule engine, backward or forward chaining, should be chosen will pursued in section 3.8.
Besides the rule engine, which has a special status because of its ability to trigger actions, other **processing tools** can be applied. Each of these tools needs data from the knowledge base that offers methods for polling and subscribing. During the initialization all tools to be installed are registered as listeners to changes of the knowledge base, so that they will be notified of any changes and new data. But they also can poll data and query the knowledge base themselves.

Actually the processing tools could directly work on the data of the knowledge base. Unfortunately there is no existing software that provides an appropriate knowledge base on which several tools can simultaneously work directly. Moreover usually the individual data representation is crucial for the efficiency of each processing tool. Therefore, before advanced tools are available, it is inevitable to keep copies of the data or at least parts of it at every processing tool.
The procedure of each tool comprises getting data from the knowledge base, processing it and writing the results back. Therefore any means of data processing must be forced into this scheme.
Bayesian networks or neural networks, for instance, could be other processing tools besides rule engines. For these we briefly describe knowledge management and procedures. In case the respective processing tool cannot work directly on the central knowledge base, it needs own memory to store data. For this the Bayesian network uses its variables, which are the network nodes. The neural network holds the knowledge in its input units. Whenever relevant changes to the knowledge base occur, the respective algorithm of the component needs to be started. The Bayesian network calculates the probabilities for the query variables anew, compares them with given limits and returns the results. The neural network starts the pass through the network, interprets the results and sends them to the knowledge base.
We will continue with the so-called preprocessors of the tools (cf. fig. 3.2) and the management interface in the next section.

3.5 Management Layer

The management layer consists of the management interface and the data reader. Moreover we will describe the preprocessors of the processing tools here, although they rather belong to the processing layer, as they are determined by the actually deployed tool instance.
The **management interface** is the interface to the application. The application receives messages from the AIP module (triggered by the rule engine of the processing layer) and adapts the AIP module over this interface by manipulating sensor manager, knowledge base and rule engine. It is possible to add and remove sensors, facts and rules, and furthermore to query the knowledge base and to send sensor requests, the latter being indirectly responded by inserting the results into the knowledge base. If other tools (see 3.4) are connected to the knowledge base, they also can be adapted via this interface.
The **data reader** interprets the configuration for the AIP given by the application or by a config file. Based on these values the initialization is started, preparing the sensor manager and the components on the processing layer, especially the rule engine, the knowledge base and its ontology.
During the initialization and at runtime the inputs for the knowledge base and the processing tools are sent to the **preprocessors** first. Due to the preprocessors the management interface can be kept generic, because their task is the transformation of the inputs to the

tool's specific format. The preprocessor for the rule engine is called **rule preprocessor**, and the one for the knowledge base is called **data preprocessor**.

3.6 Inter-AIP-Communication

Although a basic idea of this framework is that the AIP modules act locally, it happens that data must be exchanged between AIP modules without the involvement of the application. As mentioned in the survey (cf. 3.2) there are two ways to establish inter-AIP-communication, namely either in the sensor layer or in the processing layer.
If the communication module is part of the sensor layer, only raw sensor data is remotely available. As in the local case, the requesting AIP module should be able to choose, whether it wants to receive data in pull or push mode from the remote sensor. In pull mode a single sensor value or an array of values is transferred. The communication is synchronous between the requesting communication submodule and the requested AIP. In push mode the communication is asynchronous, the inquirer is registered in the data supplier and is informed about any new data from the specified sensor.
When placed into the processing layer, the communication submodule can act in a similar way compared to the processing tools described in 3.4, because it pulls or gets data from the knowledge base; however, it also puts data into it. The only difference to processing tools is that the communication process needs to be triggered, e. g. by the rule engine.
As an ontology transfer might be necessary before high level data can be exchanged between two knowledge bases of different AIP modules, such an exchange becomes much more difficult to accomplish than a low level data transfer. The software engineer has to implement an automatic ontology transfer to be performed, before data is exchanged. Otherwise the application developer is overburdened, when he must design rules to import parts of the ontology before data can be requested. But even then, the software engineer has to provide means to import ontologies.

3.7 Evaluation of Architecture

After the specification of the architecture we should **check the requirements**. For this we define the architecture with knowledge base and rule engine, but without further processing tools as **basic architecture**.

- **Filtering and fusion.** Considering the basic architecture there is only the rule engine that processes data. Filtering and fusion are possible, but for more subtle applications a Bayesian network should be added.
- **Learning.** A rule engine alone is inappropriate for learning. A learning Bayesian network or other learning algorithms must be added to the basic architecture, if the application demands learning.

- **Classification in general.** As we are free to add any tools that can work on the data provided by the knowledge base, we have all means at our disposal to classify data.
- **Locality.** The AIP architecture was designed for the local (pre)processing of sensor data. Nevertheless due to the modularization within it is possible to divide and distribute the AIP module.
- **Communication.** We suggested two ways to establish the inter-AIP-communication. Both meet the requirements, for the exchange of low level sensor data suffices for the use cases described in 1.4. Nevertheless situations are imaginable, where transfer of high level data becomes necessary or at least desirable. In such cases communication should be implemented in the processing layer.
- **Generic interfaces.** In sections 3.3 and 3.5 generic interfaces are described and demanded.

3.8 Software Requirements

Before we turn to the implementation of the AIP module, we should see, whether we can find any software requirements or restrictions beyond the the listed ones:

- The discussed application of the AIP module in different environments and on different platforms imply the use of a **platform independent programming language**.
- The reusability of the processing tools and the encapsulation inherent in the architecture can be achieved best by using an **object-oriented programming language**. The design process was already done object-oriented.
- As incoming sensor data is to trigger rules, a data-driven and thus **forward chaining rule engine** is recommendable. Nevertheless it might be advantageous to be able to question the rule engine. Then, for example, requests from the application could be answered more easily. Therefore, if possible, a **combination of both, forward and backward chaining**, would be perfect.

Summary *This chapter was devoted to the AIP architecture that satisfies the requirements of section 1.4. The sensors in combination with the sensor interface form the main data source; the management interface and thus the application the main data sink. In between, in the processing layer, the data is transformed and processed by arbitrary and user defined tools. In the sensor layer or in the processing layer the inter-AIP-communication enables the exchange of (sensor) data.*

Chapter 4

Implementation

Based on the architecture introduced in the previous chapter we continue with the implementation of the AIP module. This chapter's structure follows the previous one. After an overview of the entire implementation each layer will be considered in a separate section.

4.1 General Survey

In the prototype essentially everything was implemented that is necessary for the AIP module's employment. The sensor layer is fully functioning and includes the inter-AIP-communication. The processing layer with knowledge base and rule engine is also ready for use, its basic version is extended with a reasoner (as part of the knowledge base). At the moment the management interface is restricted to the basic functions for sensor manipulation, rule engine and knowledge base. An extension for additional processing tools is not yet implemented. The entire software was written in Java making it platform independent and allowing object-oriented programming. The software employed is explained in the corresponding sections. In the sequel class- and method-names will be characterized by `monospaced type`.
Figure 4.1 shows the static software structure as class diagram. For the sake of clarity just the most essential methods are included. Attributes and other `public` methods are omitted. The class `AIPInterface` serves as stub for the application. Together with `ApplicationServer` and `ApplicationThread` it will be explained in the Management Layer section.

4.2 Sensor Layer

For the implementation of sensor manager and sensor interface we use the "Mobile Sensor API", i. e. the unfinished Java Specification Request (JSR) 256 [71] of the Java Community Process (JCP) [47] in the version 0.10. The JSR 256 is meant "to fetch data easily and in a uniform way from sensors" and thus fits the requirements.

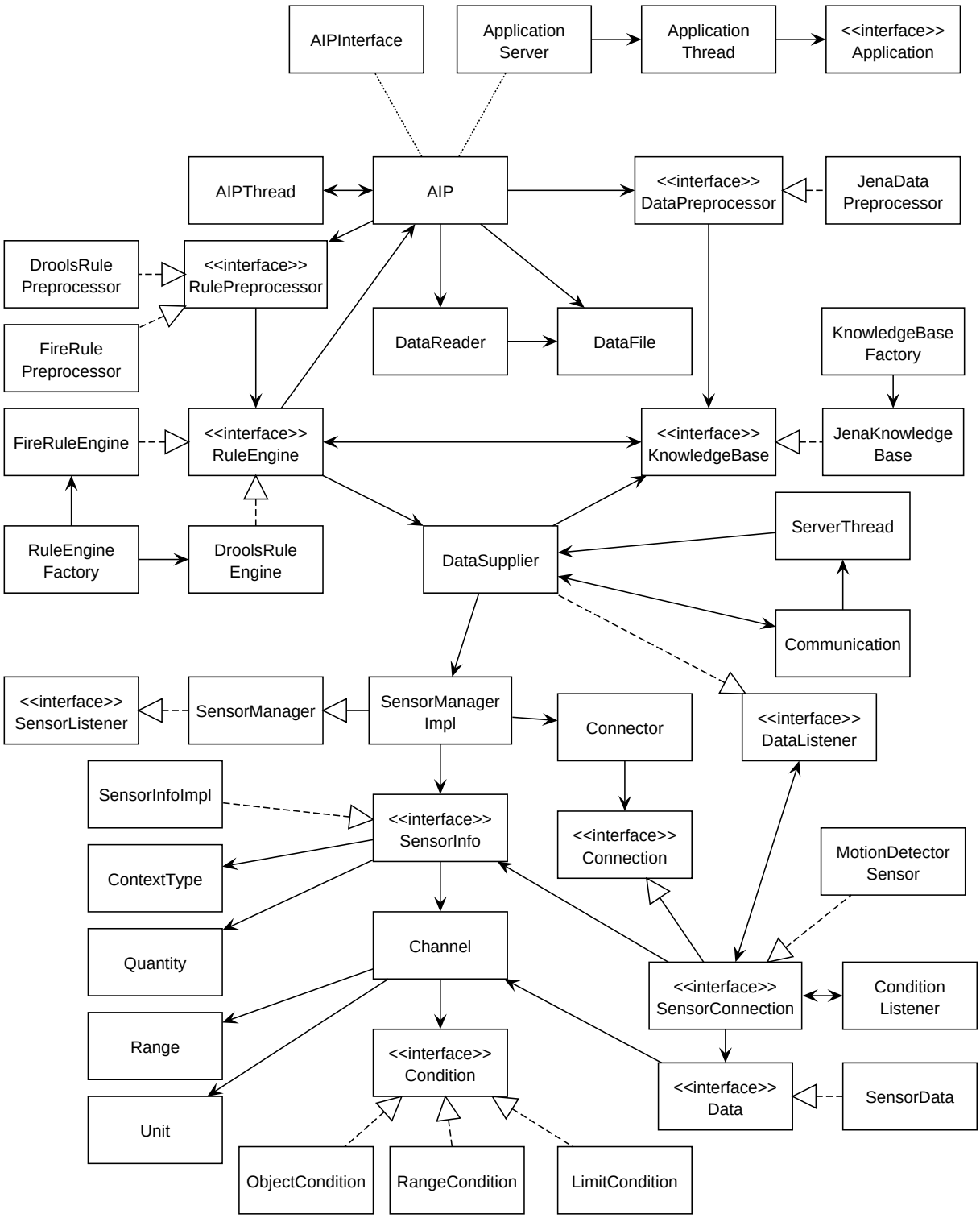

Figure 4.1: Class diagram.

In several places the JSR 256 relates to the "Generic Connection Framework" (GCF) that is to be used for the connection of sensors. Although the GCF is only part of the Java 2 Micro Edition (J2ME) and not of the Java 2 Standard Edition (J2SE), there is an open source implementation of it. The "ME4SE" [25] is based on the JSR 197 ("Generic Connection Framework Optional Package for the J2SE Platform") [8] and presents indeed the GCF as an optional package for J2SE. Nevertheless it is not used in the AIP module, because the only code besides the `Connection` interface that is needed by the JSR 256 has to be added to the `Connector` class anyway. Therefore we implemented the JSR 256 and the required parts of the JSR 197 ourselves.
Because of the size of the interface description, the definition of the classes and interfaces is restricted to the important methods. For further information we refer to "Mobile Sensor API" [71].

4.2.1 Sensor Interface

One of the main requirements for the sensor interface is that every sensor that meets the extended definition of section 1.1 can be connected to it and that it provides the data in an uniform way, i. e. independent of the sensor type and vendor. For this purpose the "Mobile Sensor API" provides the `SensorConnection` interface that has to be implemented by every sensor that is to be connected. `SensorConnection` involves `getData()` methods for polling information and `setDataListener()` methods for subscribing to the sensor. In both ways of data retrieval it is possible to set the buffer size (how many values are to be returned at one time) and the buffer period (how long data is to be collected). Every sensor needs `SensorInfo` that includes all its basic and context information. The sensor should be identifiable by the URL (Uniform Resource Locator) that consists of the `Quantity` to be measured, one `ContextType`, the sensor model and a location string. The indication of context via `ContextType` instances renders more precisely what kind of sensor is meant, if the `Quantity` alone is not significant enough. For example, if pressure is measured, and the context types are "user" and "blood", the sensor is a blood pressure meter for clinical use. A URL according to our example would look like:

```
sensor:pressure;contexttype=blood;model=bpm-xxx;location=hospital
```

The rest of the sensor information includes the channels (see below), a description string, the position given by latitude and longitude, the version of the `SensorConnection` implementation, the vendor of the sensor, the connection type (wired, embedded, short range, remote), the maximum sampling rate given in Hertz and further properties if needed. Unfortunately JSR 256 specifies the maximum sampling rate as an integer, hence sensors with sampling rates below 1 Hz are poorly approximated by 1 Hertz.
If a sensor measures different values simultaneously, each value is presented by an extra `Channel`. So a motion detector has only one `Channel`, whereas a sensor that determines

a position in three dimensions has three. The attributes of a `Channel` object are name, data type (`int`, `double`, `Object`), `Unit`, scale, accuracy, an array of `Range` objects and a `java.util.Vector` of `Condition` objects. Here the scale relates to `Unit`. If for example the `Unit` is meters and the scale is -2, the actual unit is centimeters. A `Range` is defined by lower bound, upper bound and resolution. All values that are not within one of the ranges, are invalid. A `Condition` is either a `RangeCondition`, an `ObjectCondition` or a `LimitCondition`. The `Condition`s are not used by the AIP module itself, but can be applied by the application designer. A `RangeCondition` is met, when the measured value is within the defined range. A `LimitCondition` checks, whether a numeric value is less than, greater than or equal to a predefined limit. An `ObjectCondition` at last tests the value for equality to a given one. `ConditionListener`s that will be notified, when the `Condition` is met, can be added to the `SensorConnection` implementations. The `Condition`s can be used as preliminary filters.
Each `Channel` has an URL that is composed of the name and the URLs of the `Condition` objects. It is used only by the push registry of sensors. The push registry is closely connected with the `SensorListener` interface, that is described in the following section 4.2.2.

4.2.2 Sensor Manager

The sensor manager administers the sensors connected to the AIP module. New sensors can be added, and registered sensors can be requested. The sensor manager includes the SensorListener interface, the abstract class `SensorManager` and its implementation.
The main task of the `SensorManager` is to find the sensors that are requested via the `DataSupplier`. For this it provides the `findSensors()` methods that get either a `String` or `Quantity` and `ContextType`. Both methods return an array of fitting `SensorInfo`s. The `String` that is passed to `findSensors()` is a URL as defined in section 4.2.1. If only the `Quantity` to be measured and the `ContextType` are specified, the second method must be applied to select a proper sensor.
In the AIP implementation the `SensorManagerImpl` itself keeps the list of `SensorInfo`s. Various methods are available to add and remove `SensorInfo`s.
The `SensorListener` interface is not implemented by any class of the AIP, but still can. It is meant for listening to events providing information about changes due to the availability of registered sensors. In case of wired sensors it will receive notifications, when the cable is plugged in or out. In case of wireless sensors a notification will be sent, when the sensor enters or leaves the range of reach.
In the smart elderly care home scenario `SensorListener`s would be of much help. As the bodily functions of the monitored person have to be measured directly on or in the body, the sensors cannot be assigned permanently to one of the static AIPs. But it may be useful to make bodily data available to AIPs in the actual room. This could be achieved by enabling communication between the room's AIP module and the one on the body. More elegant is a solution, where the `SensorListener` automatically detects body sensors in reach and opens a link. Then the person does not have to carry an AIP module, and

inter-AIP-communication between mobile and static AIP is not needed.
The underlying principle is called "push registry". Further information about the topic can be found in [82].

4.2.3 Data Supplier

Basically the data supplier is identical with the `DataSupplier` class that is not part of the JSRs. It uses the `Connector` class from the "Generic Connection Framework" and the `SensorManagerImpl` that was described in the previous section. Furthermore it is the interface to the processing layer, is fed with sensor requests by the `RuleEngine` or by the `communication` object and sends `SensorData` to the `KnowledgeBase` or to other AIP modules via the `communication` object.
For starting a **sensor request** one of the `DataSupplier`'s `sensorRequest()` methods need to be called, passing the sensor's URL or alternatively `Quantity` and `ContextType`, the command ("open" or "close") and in case of opening the mode ("pull" or "push"). Additionally it has to specify the consumer, thus the AIP that started the request, and the producer, the AIP module that is supposed to have such a sensor. The producer AIP can be addressed by its ID or by one of the following keywords:

- **LOCAL.** Only the local AIP is checked, whether the requested sensor, specified by its URL or by `Quantity` and `ContextType`, is available. For this one of the `SensorManager`'s findSensors() methods is called and the `SensorInfo` returned first is taken.
- **REMOTE.** The request is broadcast to all (neighboring) AIPs. If multiple AIPs have sensors that meet the specification, all sensors will be treated equally, meaning that they are all opened (in push or pull mode) or closed. Any returned data will be written into the `KnowledgeBase`.
- **LOCAL_REMOTE.** If the local AIP has an adequate sensor, the request will be handled like the LOCAL case, otherwise like the REMOTE case.

If the producer is remote the request will be passed to the `Communication` object. If it is local, the `sensorRequest()` method will determine the `SensorInfo` and put the request to a queue as a `ConnectionRequest`.
With the attributes of the `ConnectionRequest` the method `openConnection()` is called by the `DataSupplier`'s thread. By calling `SensorInfo`'s `getUrl()` the URL of the sensor is determined and passed to the `open()` method of the `Connector` class. At last the `Connector` returns the newly created `SensorConnection` instance that represents the sensor. (Regrettably due to the design of the `Connector` class the `SensorInfo` has to be added to the `SensorConnection` afterwards.)
If the sensor is not yet contained in the `KnowledgeBase`, it will be inserted with all the context information that is provided by its `SensorInfo` and its `Channel`s. In case of a

remote sensor, the SensorInfo has to be imported from the referring AIP module.
For every sensor (in push mode) the DataSupplier holds a SensorConsumerRelation object providing a list of the sensor's consumers. They all are notified, whenever new data is pushed by the sensor. Pull requests are directly answered, and the consumer is not added to the list.
The DataSupplier's second task is passing SensorData to the local KnowledgeBase or to the Communication submodule. If SensorData is requested via polling, the getData() method of the referring SensorConnection will be called that returns the SensorData directly. On the other hand in push mode the DataSupplier is notified of every new sensor value. For all SensorConnections the DataSupplier is the only DataListener.
In pull mode as well as in push mode the SensorData is transformed to a TripleFact that is then passed to either KnowledgeBase's addFact() or Communication's sendFact(). The latter also needs the consumer.

4.3 Processing Layer

The implementation of the processing layer consists of a KnowledgeBase, a RuleEngine and the related preprocessors. In the JenaKnowledgeBase a reasoner is integrated and can be regarded as an additional processing tool. But so far there is no preprocessor implemented for it, and it cannot be controlled via the the management interface (AIP class).
Moreover there are two so-called factories, namely the KnowledgeBaseFactory and the RuleEngineFactory. Based on the config file that is read in during the initialization, they instantiate the chosen KnowledgeBase or RuleEngine. In the AIP implementation one KnowledgeBase, namely the JenaKnowledgeBase, and two RuleEngines, namely DroolsRuleEngine and FireRuleEngine, are available. The factories are based on the design pattern *Factory* [18].
Everytime, when a fact is added to the KnowledgeBase, it is automatically forwarded to the RuleEngine and will be forwarded to any other registered processing tool. As the reasoner works directly on the KnowledgeBase, it need not be notified of changes.

4.3.1 Knowledge Base

The KnowledgeBase supplies all processing tools with data and stores their results. Moreover it is fed with data by the DataSupplier, which is the collecting point for all sensor data, and by the application. All method calls coming from the application pass the DataPreprocessor, whose principal role is to adapt data formats to the specifications of the KnowledgeBase instance. Therefore the DataPreprocessor must be replaced also, when the KnowledgeBase implementation is replaced.

4.3.1.1 Data Exchange Format

The format for the exchange of data between the components is defined by the `TripleFact` class, i. e. before transferring data it has to be transformed to `TripleFact` objects. A `TripleFact` is equivalent to a triple of the Context Meta-Model (see 2.4 or [17]) with fixed attributes and quality classes. Both specify a *property* instance, which connects two *entity* instances or alternatively one *entity* with one *datatype* instance, and add the attributes timestamp and uncertainty (see table 4.1 for details). In case of sensor values these attributes are the measuring time and the standard deviation of the estimated error.

`TripleFact`	Context Meta-Model	Description
`PropertyClass`	*Property Class*	
`PropertyInstance`	*Property Instance*	
`SubjectClass`	*Entity Class*	
`SubjectInstance`	*Entity Instance*	
`ObjectClass`	*Entity* or *Datatype Class*	
`ObjectInstance`	*Entity* or *Datatype Instance*	
`Timestamp`	timestamp of *Property*	
`Uncertainty`	*Quality Instance*	displays measuring error
`isDataType`		boolean defining the object
`DataValues`	*Datavalue Instance*s	list of *Datavalue* instances

Table 4.1: Relation between `TripleFact` and Context Meta-Model.

The exchange format was adapted to the *Context Meta-Model*, and the implementation `JenaKnowledgeBase` of the `KnowledgeBase` interface is based on the related *Context Engine*. Nevertheless it is possible to use other `KnowledgeBase` implementations. To use OWL (Web Ontology Language) or at least RDF (Resource Description Framework) would be an advantage for these implementations but it is not a prerequisite. One could also think of a relational data base, but software details will be discussed later. At first we will examine the `KnowledgeBase` interface.

4.3.1.2 Interface

The interface `KnowledgeBase` mainly provides methods for adding (`addFact()`), canceling (`removeFact()`) and finding (`findFact()`) facts. The `query()` method allows to formulate complex queries in RDQL (RDF Data Query Language) [84] (W3C submission), similar to SQL (Structured Query Language), used with relational data bases. The results of the `query()` are returned in form of a `java.util.Iterator`. Processing tools are registered as `KnowledgeBaseListener` by the method `addKnowledgeBaseListener()`.
The initialization of the `KnowledgeBase` is triggered by calling `initKnowledgeBase()`. For this the caller, namely the `AIP` instance, must pass three files containing respectively

the AIP specific ontology, the application-oriented ontology and initial knowledge. All files need to be written in OWL DL to conform to the management interface, but can be transformed into other formats by the `DataPreprocessor`. For writing the OWL files it is recommended to use one of the numerous OWL-compliant ontology editors. The AIP specific ontology and the ontology and knowledge files for the prototype were generated with "Protégé" of Stanford University [53].

4.3.1.3 Ontology

The ontology's application specific part must be written by the application developer. The AIP-specific part of the ontology is the same for all AIP agents and it is built according to the entity-relationship diagram[1] of figure 4.2 in OWL DL.
The main entities (or objects) are *AIP*s, *Sensor*s and *Value*s. When a new sensor value is added, it is assigned to that *Sensor* instance which represents the actual sensor. The *Sensor* instances again are connected to the corresponding *AIP* instances.
All other entities connected to the*Sensor* entity represent context information to the sensor. This context information is essentially the one listed in section 4.2.1 when describing the `SensorInfo`.
At start time the knowledge base includes the local AIP module and in addition that AIP modules it can communicate with. During initialization these must be specified in the config file. Only local sensors are written to the `KnowledgeBase`. But, when at runtime data is exchanged via inter-AIP-communication, sensors of other AIP modules are entered also.
The application-specific part of the ontology can be connected to the AIP-specific part sharing the *Application* entity. But this is not necessary, the partial ontologies can also remain separated.

4.3.1.4 Software

For the implementation the *Context Engine* of Fuchs [17] is used, which is based on the semantic web framework Jena (see 2.4, [46]). It is accessed through the wrapper class `JenaKnowledgeBase` implementing the interface `KnowledgeBase`.
If a different software is to be used one must program a corresponding wrapper class and an extra `DataPreprocessor`. As mentioned above, a software is advisable that utilizes OWL or RDF. However, particularly through the usage of the entity-relationship model it becomes obvious, that a relational database also is apt as `KnowledgeBase`. Then the database scheme serves as ontology in which data can be placed.
How to get from an entity relationship model to a relational database is well defined and

[1] Entity-relationship diagrams were proposed by Chen in 1976 [6] and are primarily intended for the design of relational database schemes. Parts of the real world are mapped to entities and relationships between them.

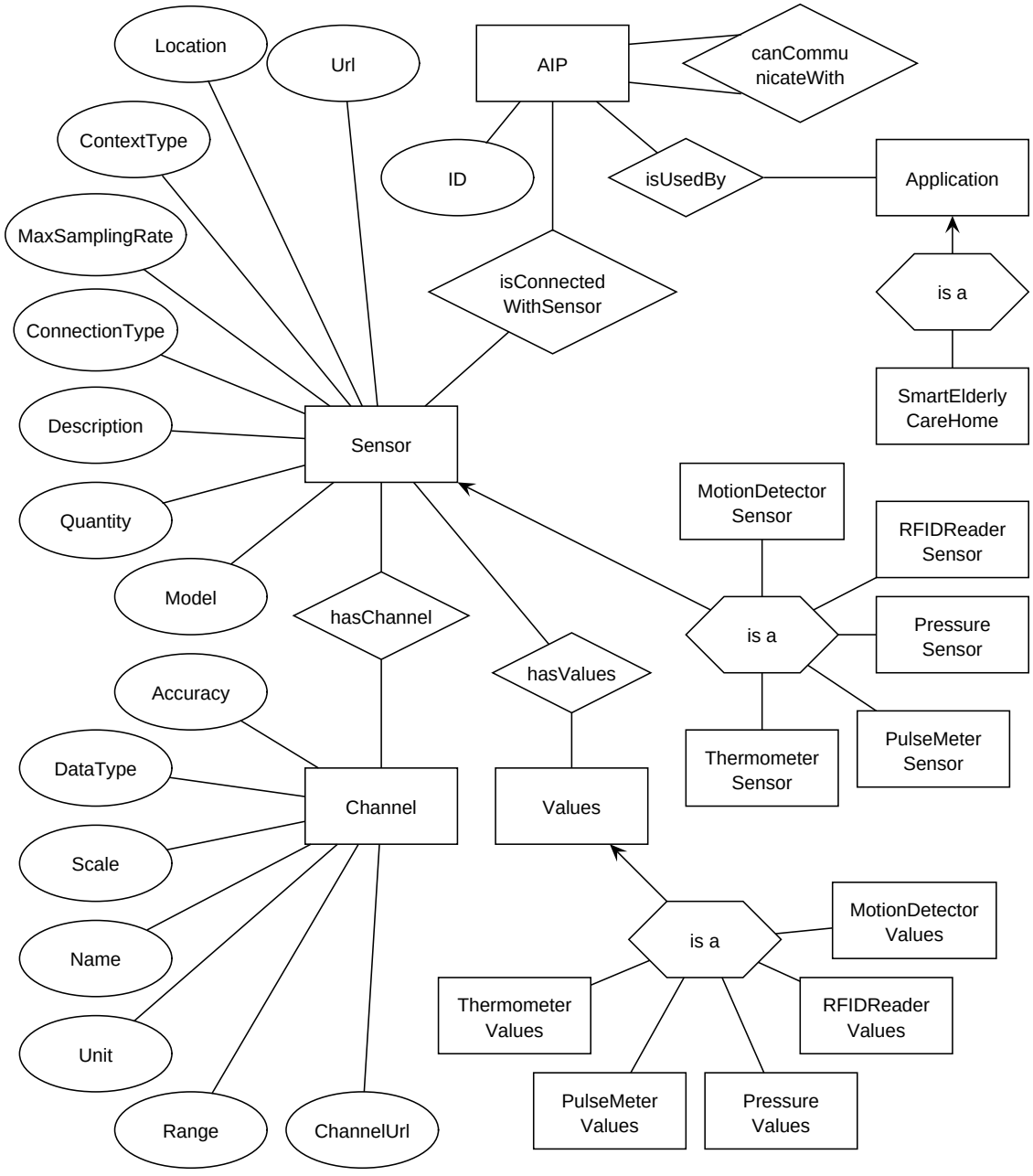

Figure 4.2: Entity relationship diagram of the AIP specific part of the ontology. It is extended with subentities of *Sensor* and *Values* from the smart elderly care home scenario (see 5.1). Using the *Context Engine* software, the entities' attributes need to be transformed to triples: entity – relationship – entity.

can be easily accomplished with the help of a textbook (e. g. [30]).

4.3.2 Rule Engine

The rule engine is the only mandatory processing tool, because it does not only write back its results to the `KnowledgeBase`, but also triggers actions. These actions are on the one hand sensor requests sent to the `DataSupplier`, on the other hand messages for the application transmitted via the management interface, i. e. the `AIP` object.

4.3.2.1 Interface

It is assumed, that the `RuleEngine` possesses its own database, so that there are not only methods for adding, removing and modifying rules, but also methods for adding, removing and modifying facts. Rules and facts are either passed as `java.lang.String` or as `java.lang.Object`.
The method `fireAllRules()` must be used, when the rule engine does not fire automatically as soon as all conditions are met. This is the case for "Drools" (`DroolsRuleEngine`), which is used in the AIP module, but not for "Fire" (`FireRuleEngine`).
The rule engine is initialized through `initRuleBase()`, where the rule base is turned over either indirectly by a file name or directly as `java.io.InputStream`.
The remaining methods of importance execute actions mentioned above. `SendAction()` sends messages to the application by just passing `java.lang.String`s or additionally a `java.util.HashMap` that may contain arbitrary Java objects (provided they are serializable). The objects in the `HashMap` are merely considered as add-ons to the `String` message.
Sensor requests were discussed in section 4.2.3. They are triggered by calling one of the `sensorRequest()` methods.

4.3.2.2 Rule Preprocessor

The `RulePreprocessor` is actually intended to adapt the rules, which arrive via the management interface, to the rule language of the rule engine used. Thus the management interface can work with a fixed rule language independently of the actual rule engine. Because of the great variety among rule engines, however, it is difficult to find a rule engine that can be translated to all the other rule languages.
E. g. in the rule base of "Drools" both, the left hand sides (conditions) and the right hand sides (consequences), are pure Java code, whilst "Fire" permits only predicate symbols with string variables on left hand sides. A compromise in this case would be, that Drools also is restricted to conditions with predicates and string variables.
To avoid such painful compromises it would be necessary to narrow the set of admissible rule engines. Standards, apt for that, are:

- **Prolog.** Because of its long history prolog's notation is a well established quasi

standard. Even if there are minor differences between the various prolog implementations, their kernels are essentially the same.

- **Rule Markup Language (RuleML).** This rule language, based on XML and RDF, is designed to comprise many different rule languages (forward and backward chaining) and to unify their formats. It is also devised for the exchange of rules and of course can be processed by rule engines. The rule engines "Mandarax" and "jDREW" use RuleML for example. RuleML is proposed as a W3C standard by the Rule Markup Initiative [4], but not yet submitted.
  Unfortunately RuleML is badly documented, and only parts are finished so far, mainly the Datalog sublanguages, for which a tutorial is provided ([4]). Since Datalog uses restricted Prolog rules and since Prolog rules can be mapped to RuleML (e. g. with "Prolog2RuleML" [12]), RuleML is comparable with Prolog.

- **Semantic Web Rule Language (SWRL).** SWRL [28] is a combination of OWL and the RuleML Datalog sublanguages, so that it extends OWL by Horn-like rules. Thus, SWRL is also comparable with Prolog. Yet there is hardly any documentation available.

Hence as a consequence all (more or less) suitable rule languages use notations similar to Prolog with Horn-like rules. Although forward chaining (e. g. jDREW) and backward chaining (e. g. Mandarax) are possible, we abstained from a standardized language for the management interface. One reason is that the examined languages did not satisfy all requirements, an other reason is that there are many rule engines that do not use Horn-like rules (e. g. Drools).
As a consequence of doing without a standardized language the rule base must be specified in the rule engine's rule language. That is not necessarily a disadvantage, even if the reusability of the rule base is impaired when the rule engine is to be replaced.

4.3.2.3 Software

Since the rule engine in the AIP module is exchangeable, the software chosen for the AIP implementation need not be definite.
When the AIP is to be installed in a system and when the AIP's function is specified within the overall software, the selection can be changed to achieve a better adaption to the requirements. This section will list several rule engines, which were investigated during implementation.

Prolog Prolog allows merely backward chaining and therefore does not conform to the software requirements in 3.8; nevertheless several Prolog implementations were studied with the intention to adapt Prolog to these requirements. The wrapper class could take over the firing of rules, which, by the way, is also necessary with some forward chaining

rule engines like Drools. Yet the necessity to check everytime all the goals of interest presents a severe drawback, unless the application permits the rules to be fired periodically instead at any change of data.
Imaginable is a combination of Prolog and a forward chaining rule engine, which can work with the same set of rules. As stated in 3.8 a combination of forward and backward reasoning is very desirable.
Forgy [14] showed that forward and backward reasoning are not basically different aside from the direction. In both cases subgoals are defined as waypoints, which usually match in spite of the different approaches.
A further reason to consider Prolog is that because of the findings from 4.3.2.2 the Prolog notation or something similar was initially thought of as a standard rule language for the management interface.
Vaucher's **XProlog** [91] represents an extension to Winikoff's **WProlog** [96]. Like WProlog it is written in Java and can easily be attached to Java programs. It was not selected, because it does not support the adding and deleting of rules. At present van Schooten extends XProlog under the name of **YProlog** [90]. His answer to the question, whether he will add the commands, was "maybe".
The same problem, that no rules can be added, occurs for **tuProlog** [74] of Bologna University. In tuProlog an `assert` command is available, but still not working properly.
There are more Prolog descendants, e. g. Wielemaker's SWI-Prolog [92]. It was disregarded, because its integration into a Java program seems to be more complicated than it would be with XProlog and tuProlog.

Drools, Fire and Jena **Drools** [43], **Fire** and the rule engine included in **Jena** [46], do **forward reasoning** and use the **Rete algorithm** by Forgy [15]. *Rete* is latin for *net*. The algorithm creates a network fusing conditions to a single node, if they appear on the left hand side of more than one rule. When new facts are asserted or modified, all nodes satisfied by that fact will be marked. If the entire left side thus becomes true, the rule will be triggered. The Rete algorithm achieves a speedup at the cost of memory. It is the base for all recent forward chaining rule engines.
Although the **Jena rule engine** could work directly with the knowledge base chosen, which also is based on Jena, it was not studied more deeply, because according to [78] it is not yet mature. In future it might be a good choice because of its support for forward and backward chaining.
Drools is a Java based, JSR-94-compliant [85] rule engine that is used in the AIP module. The conditions and consequences of the rules can be stated in Java, Python and Groovy simplifying the collaboration with the rule engine. The rule base for initializing the rule engine is a XML file into which the code is integrated. The facts added to the "working memory" are arbitrary Java objects.
A disadvantage of Drools is that adding new rules is problematic, since the network must be newly generated by the Rete algorithm everytime. In the AIP module the method `addRule()` of the wrapper class was implemented the following way: The rule

base is extracted, the rule is added and finally the network is generated anew by calling `newWorkingMemory()`.
Fire was developed by Siemens Corporate Technology. It is written in Java and can easily be integrated into Java software. The left hand side of a rule consists of predicate symbols. If they are satisfied, the rule is fired automatically and a function is called that is specified by the right hand side. This is either a built-in function to manipulate the facts in the working memory (knowledge base) or a user-defined `fireRHS()` function. For the latter only the function's class is given, which has to implement the `RightHandSide` interface that includes the `fireRHS()` method.
One advantage of Fire is that at runtime rules and facts can be added without any problem. A wrapper class and a `RulePreprocessor` were completed. For lack of time the rule engine was not completely integrated into the AIP module.

4.4 Management Layer

Since application and AIP module communicate via socket connections, the management interface consists of two parts. At the AIP module's side the classes are `AIP` and `AIPThread`, `DataReader` and `DataFile`. The application houses the classes `AIPInterface`, `ApplicationServer` and `ApplicationThread`. Moreover it must implement the interface `Application`.

4.4.1 Management Interface

The `AIPInterface` class represents the actual **management interface**. The public interface of this class is identical with the public interface of the `AIP` class and serves therefore as a **stub**. If a method is called by the application, a socket connection is opened and the name of the method and the parameters are sent to the `AIP` instance together with a `boolean` value indicating, whether a return value is expected.
The `AIP` class extends `java.lang.Thread`. Its `run()` method listens to the specified port, to which the method calls are sent by the application, and passes each method call to a new `AIPThread` instance. The `AIPThread` calls the specified method of the `AIP` and sends, if requested, the return value back to the `AIPInterface`.
In the reverse case that a message is to be sent to the application the `RuleEngine` wrapper class calls `AIP`'s `notifyApplication()` method passing two `java.lang.String`s ("id" and "message") and a `java.util.HashMap`. The `HashMap` must contain only serializable objects, otherwise it cannot be transmitted via the socket connection.
A message, sent by the `AIP` object, is received by the `ApplicationServer` that starts an `ApplicationThread`. The thread calls the `messageReceived()` method of the implementation of the `Application` interface.

The Interface in Detail The `AIPInterface` can be divided into the part handling the initialization, the part that attends to the `DataPreprocessor`, the part for the `RulePreprocessor` and the one serving the `SensorManager`. Table 4.2 lists all these methods and separates the four parts by double lines.
The constructor of `AIPInterface` and the method `initAIP()` will be discussed more closely in the following section "Initialization".

Method	Description
`AIPInterface`	Constructor, see section "Initialization"
`initAIP`	see section "Initialization"
`insertTripleFact`	sends `TripleFact` to `KnowledgeBase`
`removeTripleFact`	removes triple from `KnowledgeBase` that equals passed `TripleFact`; this method is not supported by `JenaKnowledgeBase`
`findTripleFact`	returns `TripleFact` that has the passed ID; this method is not supported by `JenaKnowledgeBase`
`queryKnowledgeBase`	takes RDQL query and returns `Iterator` of results
`addRule`	adds rule given as `String` or `Object` to `RuleEngine`
`removeRule`	removes rule from `RuleEngine` that equals rule given as `String` or `Object`
`addSensorInfo`	forwards `SensorInfo` to `SensorManager`
`addSensorInfos`	forwards array or `InputStream` of `SensorInfos` to `SensorManager`
`removeSensorInfo`	removes `SensorInfo` specified by its URL or by `Quantity` and `ContextType`

Table 4.2: Public interface of `AIPInterface`.

4.4.2 Initialization

Before an AIP module can be initialized by the application, it has to be started in its Java Virtual Machine (JVM). When calling the `AIP`'s `main()` method, the config file together with two port numbers must be transmitted. On the first port the AIP module listens for messages from the application, on the second one for messages from other AIP modules. Moreover the sensors must be readied, i. e. especially the `SensorConnection`s must be available.
The application has to start the `ApplicationServer` that receives messages from all AIP modules. Then, to initialize an AIP module the application has to call the constructor of `AIPInterface` and its `initAIP()` method. The constructor gets IP address and port number of the AIP module, and `initAIP()` gets an array of `SensorInfos`.

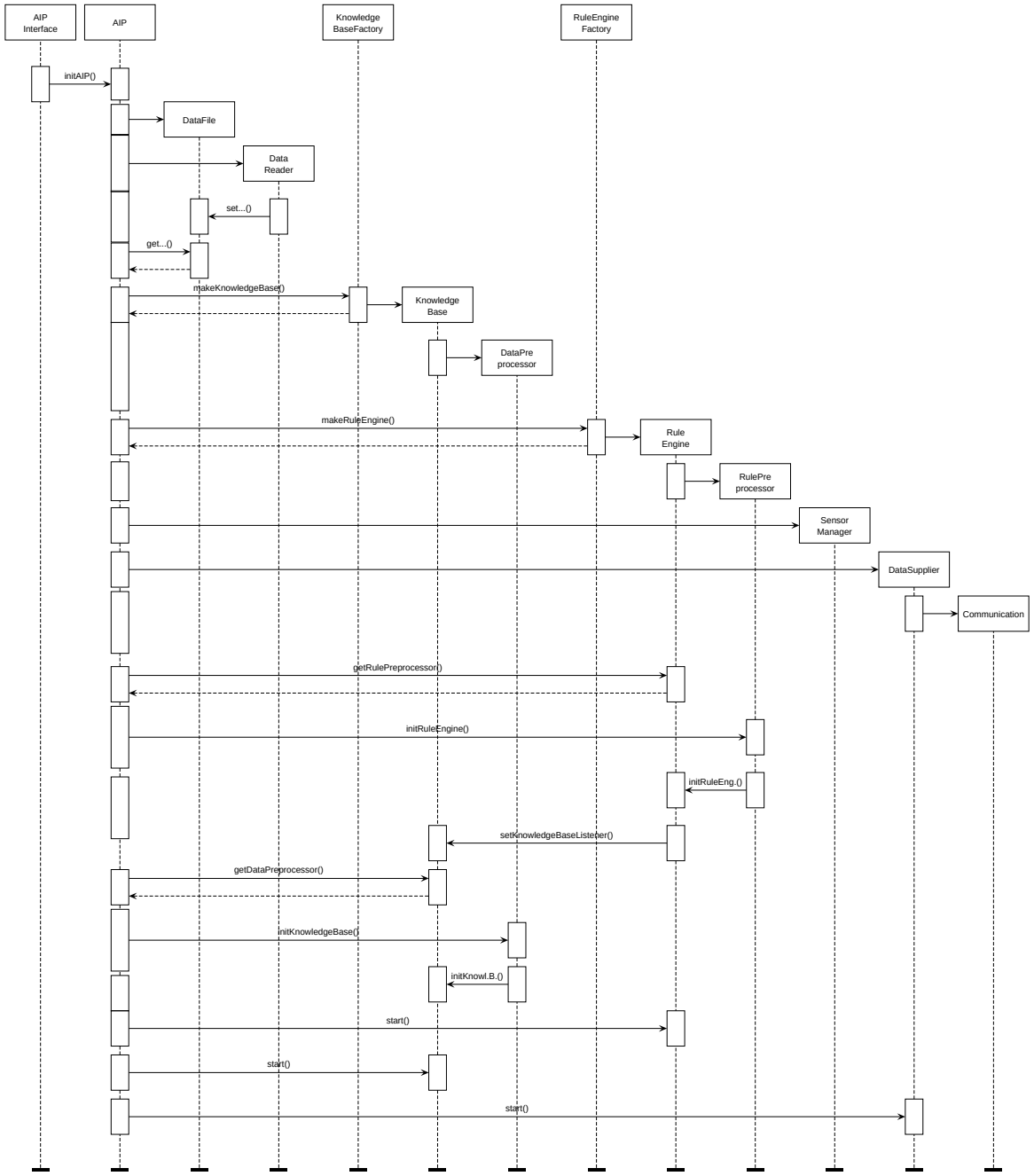

Figure 4.3: Sequence diagram of the AIP module's initialization.

The `initAIP()` call is transferred to the AIP module, so that the `AIP`'s `initAIP()` is called, which starts the initialization (see figure 4.3). Therefor the config file written in XML is read out by the `DataReader` and its contents are put into the `DataFile` object. For reading the file the `DataReader` uses the Xerces DOM-parser of the "Apache XML Project" [69].
The config file specifies:

- The ID string of the AIP module.
- IP address and port of the application.
- IDs, IP addresses and ports of the neighboring AIP modules. Later the IDs are used to identify the owners of the requested remote sensors. Naturally IP addresses and ports are needed for the inter-AIP-communication.
- The (fully qualified class) name of the `KnowledgeBase`.
- The path of the file that contains the application specific ontology.
- The path of the file that contains the initial knowledge.
- The (fully qualified class) name of the `RuleEngine`.
- The path of the file that contains the rule base.

The `SensorInfo`s, passed to `initAIP()`, must contain further specifications in their `java.util.HashMap` "properties", namely "type", "class" and "command". "type" is the type or name of the sensor and is used, when triples are written into the `KnowledgeBase`. That way the application developer can use the name in the rules. "class" is the fully qualified class name of the `SensorConnection`'s implementation. It is used by the `Connector`. "command" specifies by PULL or PUSH, how the sensor is to be started. Either the sensor sends a first value for the `KnowledgeBase` and then waits for requests, or in push mode starts immediately to notify the `DataSupplier`, whenever a new sensor value is available. With the `SensorInfo`s and the information provided by the config file the AIP module can be initialized completely.

4.5 Communication

The inter-AIP-communication is realized on the sensor layer. The communication submodule interacts with the `DataSupplier` and consists of the two classes, `Communication` and `ServerThread`.
The `Communication` class that provides the interface for the inter-AIP-communication is instantiated by passing the related `DataSupplier` instance, the IDs of the neighbouring AIP modules and the communication port to the constructor. At runtime further "neighbours" can be added by calling `addNeighbour()`. The inter-AIP-communication does not support the use of AIP modules as intermediaries to allow a connection between AIP modules via a third one.
The communication process is similar to that between AIP module and application. Here also socket connections are established for transmitting sensor requests and sensor values. The `Communication` object's thread passes incoming sensor requests and sensor values to `ServerThread`s, which call methods of the `DataSupplier` for further actions.

Sensor requests by the `DataSupplier` intended for other AIP modules are handled by the `sensorRequest()` methods. Their parameters equal the ones of the corresponding methods of the `DataSupplier` (cf. 4.2.3), i. e. the sensor's URL or alternatively `Quantity` and `ContextType` (to identify the sensor), consumer and producer (the involved AIP modules), the command (OPEN or CLOSE) and the mode (PUSH or PULL).
In order to transmit sensor values the method `sendFact()` is called, for which beside the `TripleFact` the local AIP must be stated as consumer.
Finally a method `sendSensorInfo()` is needed. When sensor data is requested remotely and the calling AIP module is not yet registered as a consumer in the `DataSupplier`, then with the first data the sensor's `SensorInfo` is sent also, to provide the consumer's `KnowledgeBase` with the necessary sensor data.

Summary

In the first part of this chapter, a survey of the implementation of the AIP architecture was given. Subsequently the three layers were described in detail, followed by the inter-AIP-communication that is part of the sensor layer.

Chapter 5

Realization of Scenario

This chapter is devoted to the implementation of the "Smart Elderly Care Home" scenario (see 1.3.4), which is based on the AIP module software introduced in chapter 4.

As a demonstration and test of architecture and implementation the smart elderly care home scenario was implemented. For this a Java application was written that uses six AIP modules in order to monitor the person's behaviour and her bodily functions. Person, environment and sensors are simulated by software.
This chapter begins with a recapitulation of the scenario, before it describes user interface and fake sensors (5.1.1), the system's topology (5.1.2), the application specific ontology and rules (5.1.3) and finally special features of the initialization (5.1.4).

5.1 Smart Elderly Care Home

In section 1.3.4 the elderly care scenario was outlined as a way to guarantee that in the future old or sick persons receive the necessary attendance. A computer system equipped with various AIP modules and a vast number of sensors takes over the the health monitoring and notifies physicians and nursing stuff when necessary.
This scenario is extended with smart home applications, which provide additional help and comfort in everyday life. This might include switching on the light automatically when a person enters a room and adjusting the temperature. Two scenarios are merged in this case, hence the name "Smart Elderly Care Home" was chosen.
The "Smart Elderly Care Home" scenario was implemented as a Java applet. The monitored person's actions and the sensors are simulated through software and input from keyboard or mouse. AIP modules process the data. This application will now be described in detail; both aspects, elderly care and smart home, are regarded jointly.

5.1.1 User Interface and Sensors

Figure 5.1 shows the applet's graphical interface, depicting a person that can be moved by mouse clicks, in her apartment with living room, bedroom, kitchen, hall and bathroom. The right margin marks the outside surroundings. The margin is also used to display text messages from the computer system doing the monitoring. Shown are time of day, the person's presumable location and activity, temperature and pulse, both measured at the body, and finally the room temperatures.
Normally the AIP would store such values locally and alert the application only, when something essential happens. For this demonstration however, unimportant data too is transmitted to the application, to make it easier for the viewer to follow the ongoing development. In order to get these values permanently, all sensors are started in push mode. Inside the rooms several sensors and actuators are denoted. They are explained together with the invisible ones:

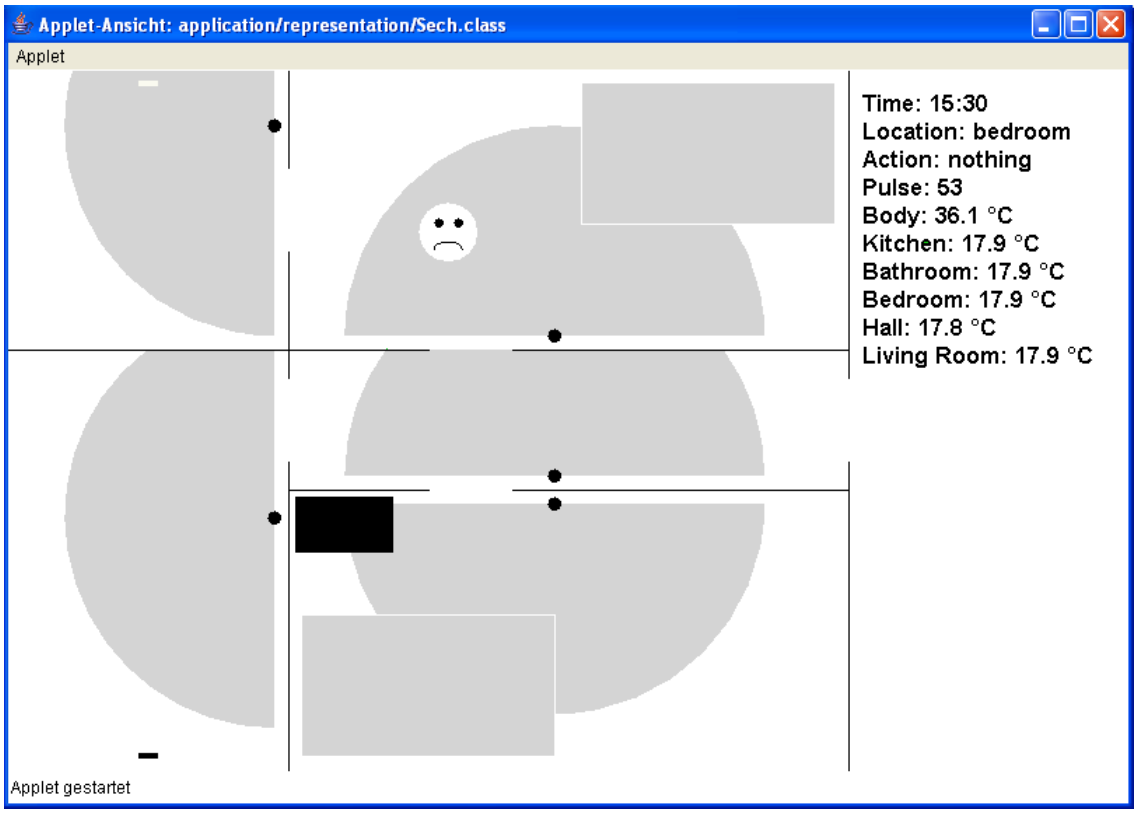

Figure 5.1: Smart Elderly Care Home. Applet showing the flat.

There is a **motion detector** in every room, indicated by a black dot in the plan. When a motion is recognized, the AIP sends a message to the application. The grey coloured circle around the motion detector turns red, *Action* changes to "moving" and the room, housing the detector, is assigned to *Locality*. *Action* changes to "nothing" when no motion is detected, which means that the system does not know, what the person is momentarily

doing.
In bathroom and kitchen two objects are marked by short lines. The white line in the bathroom indicates a **RFID tagged** toothbrush, the black line in the kitchen a pill box. When any of these objects is removed from its place, the corresponding **RFID reader** detects that the RFID tag is out of reach and notifies the application. It is assumed that the person indeed takes his pills or brushes his teeth. The *Action* "pills taken" or "toothbrush taken" will be displayed.
Pressure sensors are installed in the bed in the bedroom and in the couch in the living room. The sensor in the bed notices when the person goes to bed. Then the *Action* "sleeping" is displayed and the "night program" is started which lowers the room temperatures. The couch sensor detects when the person is sitting down and displays the *Action* "sitting". Leaving bed or couch changes *Action* to "nothing".
Opening and closing the front door is detected by a **switch** and indicated by the *Action* "opening door" or "closing door". In cooperation with the motion detector in the hall, it can be found out, whether the person entered or left the apartment.
In every room there is a **thermometer** (temperature sensor). Attached to the person's body is a **clinical thermometer** and a **pulse meter**. For display the AIP module transmits temperatures and pulse values directly to the application.
As there are no real sensors, the sensor values must be simulated by software. For each fake sensor there are three classes, namely its manager, its value generator and the implementation of the `SensorConnection`:

- The **managers** are used on side of the application to provide `SensorInfo`s and to handle requests from value generators that need data for the simulation. Of course, there would be no need for such requests, if the sensors were real.
- The **value generators** use application data and random generators to determine the sensor values and send them to the associated `SensorConnection` according to the sampling rate.
- The `SensorConnection` implementations push data to the `DataListener`, here the `DataSupplier`, or allow access by `getData()` methods (cf. 4.2.1).

In case of the RFID reader, for instance, the classes are `RFIDReaderManager` (manager), `RFIDReaderFake` (generator) and `RFIDReaderSensor` (`SensorConnection`). In order to determine, whether the person uses the RFID tagged item, the `RFIDReaderFake` asks the `RFIDReaderManager` for the person's position in the applet and calculates the distance to its own position – given by the `SensorInfo`'s attributes *longitude* and *latitude*.
`MotionDetectorFake` and `PressureFake` work quite similar. After the determination of the room, the distance between person and sensor is calculated. A motion detector perceives a motion, when the person is in range and when three successive positions of the person are not all equal. For the `PressureFake` counts: The lesser the distance, the higher the pressure.
For the front door's switch `SwitcherFake` determines, whether the connecting line between two successive positions of the person intersects the door line.

The person's pulse and body temperature and the room temperatures change randomly, though the temperatures depend on the heating. The radiators' values are provided by the application, and `ClinicalThermometerFake` and `ThermometerFake` can request their managers to get them.
The radiators can be turned on and off from the keyboard: "1" (kitchen), "2" (bathroom), "3" (bedroom), "4" (hall) and "5" (living room). The TV set (black box in the living room) cannot be switched on by keys unfortunately, but it can be used by the application to display advices, e. g. when the person has forgotten to take her pills.

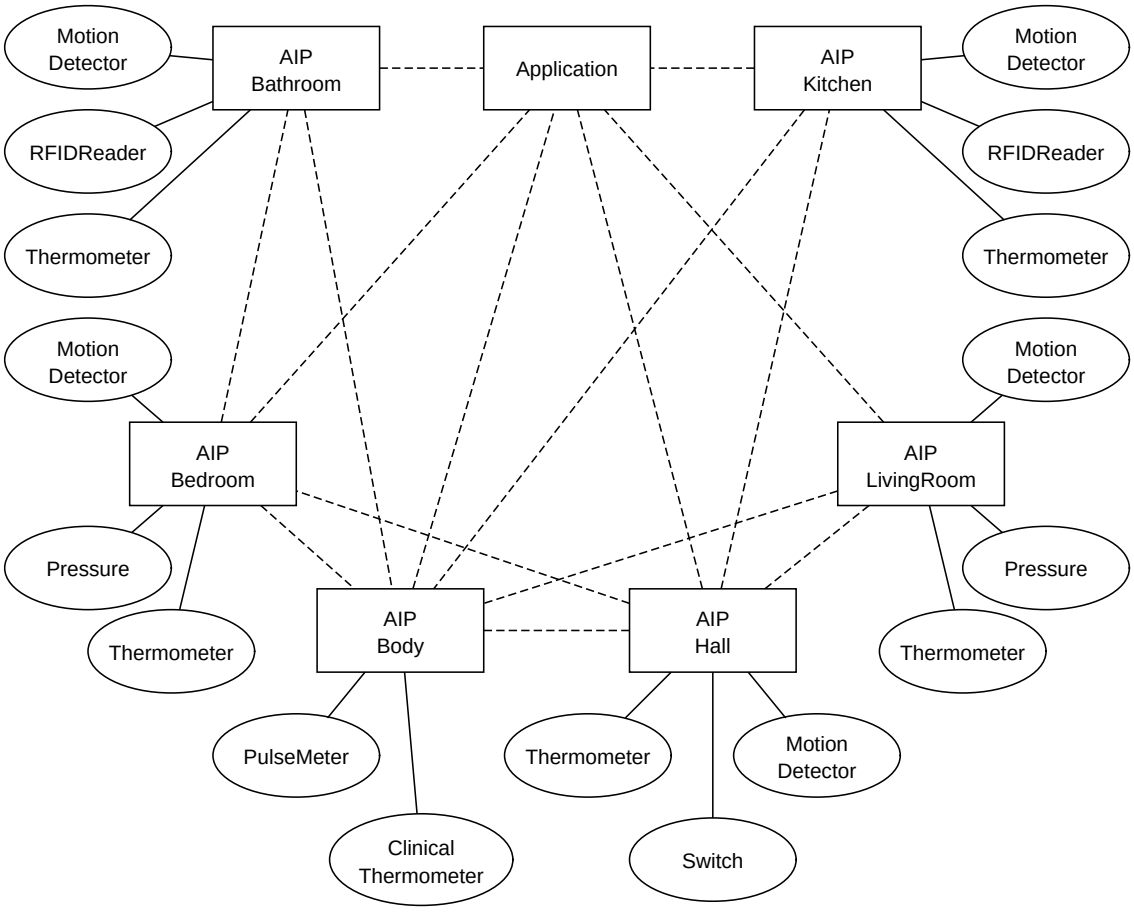

Figure 5.2: Smart Elderly Care Home. System topology.

As mentioned above, the person movement is controlled by mouse clicks. So the program's user can choose the action by navigating the person through the flat, let her sit or move her to the kitchen to take her pills.
Due to the measured behaviour and due to the AIP modules' logic messages about the person's state are sent to the medical staff. Three states, namely "quick" (or healthy), "weak" and "emergency", are distinguished. The actual state is indicated by the person's facial expression. Moreover to abridge the simulation the person's state can be forced to change by pressing "q" ("quick"), "w" ("weak") or "e" ("emergency").

5.1.2 Topology and Communication

As argued in section 1.3.4 local data is sufficient to judge the situation. If every room was furnished with an AIP module, data exchange via inter-AIP-communication would be needed only, when the person moves to an other room. But the local information is, beside the sensor data from the actual room, also the body data. Because of the missing push registry (cf. 4.2.2) an extra AIP module must be fixed to the person's body, and data has to be exchanged between the room's and the body's AIP module as well.

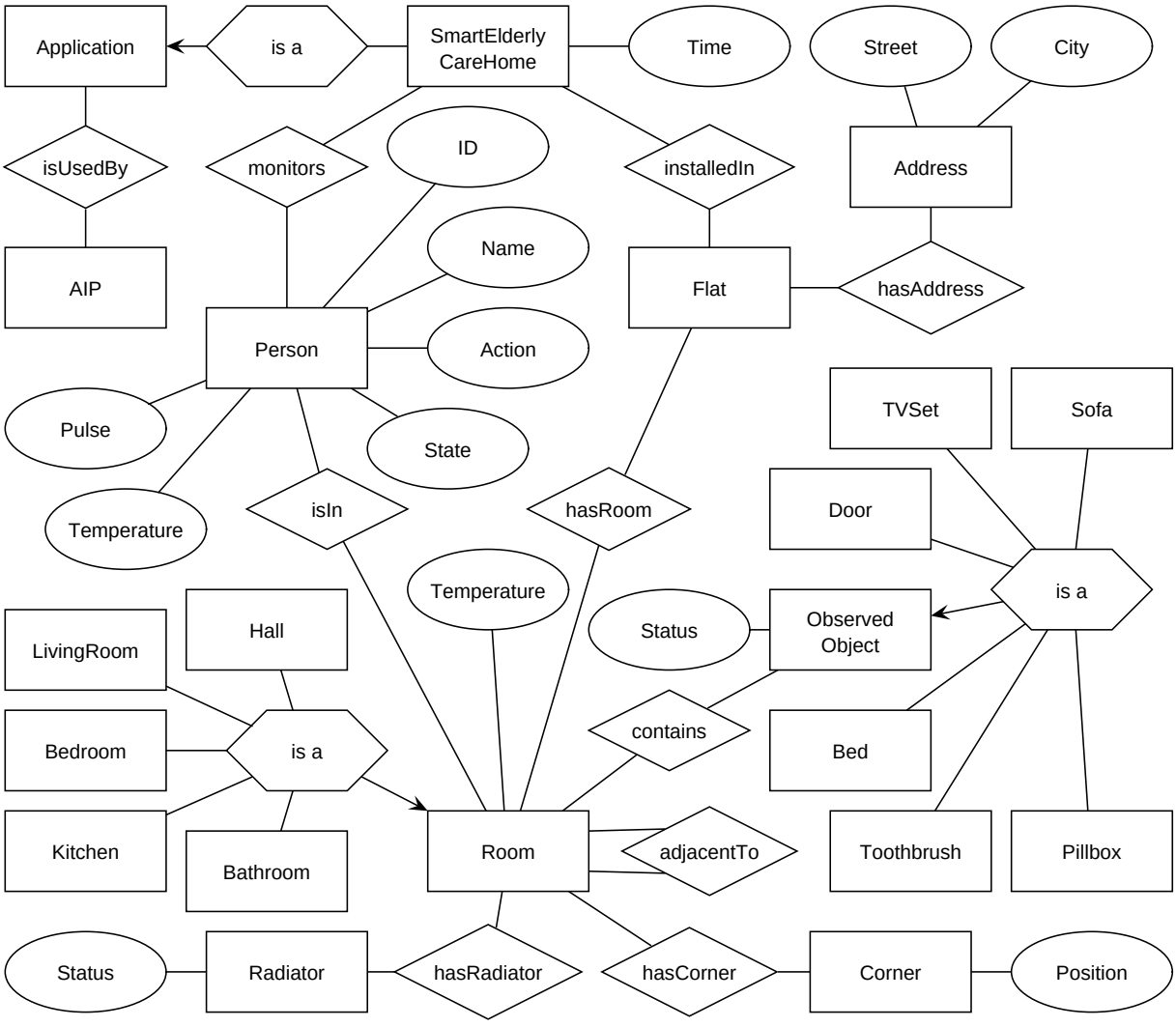

Figure 5.3: Smart Elderly Care Home. Application specific ontology.

Therefore the monitoring system consists of the application and six AIP modules, one for each room and an extra one for the body. The topology is shown in figure 5.2. Dashed

lines between components signify the ability to communicate.
Although application and AIP modules are located on the same computer in the implementation, they naturally can all run on different computers. In that case only the loopback addresses (127.0.0.1) had to be replaced by suitable IP addresses.

5.1.3 Ontology and Rules

As described in 3.4 and 4.3.1.2 the ontology splits into the AIP specific part, provided by the AIP module, and the application specific part that has to be written by the application designer. Both parts can be linked using the *Application* entity of the AIP specific ontology (see figure 4.2).
For the smart elderly care home scenario a slightly simplified ontology is shown in figure 5.3. It can be used in any of the six AIP modules. The main entities are *Person* and *Room*. They are directly linked with all important entities and attributes, like the *Person*'s *State* and the *Room*'s *Observed Object*s. So queries or rules that naturally use one of these entities as entry point, can be kept short and concise.
Note, in order to use the *Context Engine* (see 4.3.1.4 and [17]), the attributes in the entity relationship diagram need to be expressed as triples. For example, the entity *Person* and the attribute *Name* could be extended to a triple: *Person – hasName – Name*, where *Person* is an `EntityClass`, *hasName* is a `PropertyClass` and *Name* is a `DatatypeClass`, whose instance will keep the actual name later.
At runtime all data that is sent to the knowledge base must be sorted according to the ontology. It is forwarded to the rule engine, whose task is refining low level sensor data as well as processing high level data. The rules' conditions may depend on both, the actual knowledge and the underlying ontology. This is shown in the following simplified exemplary rule, written in a pseudo rule language. The rule is fired, when a `TripleFact` "fact" from the kitchen's motion detector arrives:

> IF subject instance of fact is the ID of motion detector
> AND subject class of fact is "Sensor"
> AND property class of fact is "hasValues"
> AND first data value of fact is 1
> THEN send `TripleFact` with object class "Action" and object instance "moving"
> AND send `TripleFact` with object class "Room" and object instance "kitchen"

The actual rule in Drools's rule language is more laborious, because for building the `TripleFact`s quite a lot of data is needed, and both, conditions and consequence, must be expressed in Java code.

5.1.4 Initialization

The general initialization process is described in 4.4.2. Each AIP module has to be started in an own *Java Virtual Machine (JVM)*, before the application can initialize it via a socket connection. To ease the program's launch a small *Perl* program was written. On *Windows* calling `perl jvm.pl` will start the application and the six AIP modules in different JVMs, each in its own console, but all on the same computer.
The AIP specific files containing the configuration, the rule base, the ontology and the initial knowledge respectively must be provided for the AIP modules locally. An example configuration file for the AIP module in the kitchen is shown in B. In this implementation the ontology file is the same for every AIP module.

Summary

In this chapter the implementation of the smart elderly care home scenario is described. The system's topology consists of six AIP modules and several sensors simulated by software. After the explanation of the ontology and an exemplary rule the chapter concludes with scenario specific features of the initialization.

Chapter 6

Conclusion

6.1 Summary

The thesis aims at developing a software architecture for the local and autonomous (pre)processing of sensor data.
Unlike in many other approaches that favour complete or almost complete central data processing, we here strive for local preprocessing as far as possible.
In various scenarios we point out areas of application where such preprocessing can be reasonably applied. It turns out that the processing requirements vary with different tasks and that a variable architecture fits best.
To facilitate that, diverse concepts and tools for processing sensor data are investigated and adapted for the use in our architecture. These concepts and tools include rule-based and probabilistic reasoning, learning and fusion algorithms, knowledge-based systems, ontologies and context.
Based on the requirements, deduced from the scenarios, architectural designs are developed and discussed. Eventually a three layer presentation is chosen, in which inter-AIP-communication is not appointed to a definite layer:

1. On the **sensor layer** a generic *sensor interface* for arbitrary sensors is provided. The *sensor manager* is responsible for finding the specified sensors and controlling the connections to them. Finally, the *data supplier* is the interface to the *processing layer*. It receives all sensor data and forwards it to the *knowledge base*.

2. The **processing layer** contains the *knowledge base* of the entire AIP module. Diverse processing tools have access to it, among others the *rule engine*, which in addition transmits sensor requests to the *data supplier* and messages – addressed to the application – to the *management interface*.

3. The **management layer** contains the *management interface* to the application. Messages to the application are transmitted via this interface. Vice versa it is used by the application to install new sensors and to control the processing tools through their *tool preprocessors*.

4. The **communication submodule** is situated either on the *sensor layer* or on the *processing layer* and permits an exchange of raw data from the sensors or high level data from the *knowledge base*.

On the basis of this architecture the AIP module is implemented in Java. Here the *communication* submodule is connected to the *data supplier* which is designed to handle remote sensor requests as well.
During implementation various software was checked for its usability in the AIP module and in some cases adopted.
The thesis is concluded with implementing the "smart elderly care home" scenario. In this health monitoring for a person is provided by a computer system that uses six AIP modules and several sensors to observe the person's behaviour and her bodily functions. The environment and the sensors are simulated by software.

6.2 Critical Discussion and Open Questions

Besides the positive and innovative aspects there are also aspects which may draw critique and aspects which are not treated in the thesis but deserve being investigated by future work.
The greatest benefit is also the most obvious field for critique. The use of arbitrary tools for the corporate processing of (sensor) data is not yet investigated and maybe difficult to coordinate. In principle any tool can depend on the data provided by any other tool. Hence the order of accesses to the data base can be decisive.
An other critical point is processing speed. Often AI algorithms have a poor performance, and in the case of knowledge bases and rule engines it will get even worse with the amount of data. For critical real time applications like the *Electronic Stability Program* from the automotive scenario, AIP a priori is not qualified. It is an open question, how various configurations of the AIP module will behave in such a case.
Moreover one can ask, whether the configuration effort is justified. After all it may be necessary to initialize additional tools besides the provision of ontology, initial knowledge and rules for knowledge base and rule engine. In the case of learning algorithms that could even mean training the system prior to its use. Yet, considering the free choice of tools, the effort for the configuration can be estimated and controlled. It is self-evident that a simpler tool is to be preferred, when it does the job satisfactory well.
Finally there remains the question, already discussed in chapter 3, in which layer to place the communication. An exchange of sensor data is sufficient for the scenarios under consideration. In scenarios with more emphasis on communication, the exchange of high level data gains importance. However, one should keep in mind, that the focus of this thesis is on **local** processing, which should do without communication as far as possible.

Appendix A

Used Software

Apache Ant *Java* based build tool, similar to *Make*. Used for building the AIP software and the scenario. [40]

Automatica Selection of macros for *Metapost* to draw automatons and graphs. Used for drawing several pictures. [39]

Context Engine Knowledge base implementation based on *Jena* and on Fuchs's *Context Meta Model*. Used in the AIP implementation. [17]

Drools Open source forward chaining rule engine. Used in the AIP implementation. [43]

Fire Forward chaining rule engine by *Siemens Corporate Technology*. Used in the AIP implementation, but it is not yet applicable in it.

Java SDK 1.4.2 Java Software Development Kit by *Sun Microsystems*. Used for implementing the AIP software and the scenario. [45]

Jena Open source *Java* framework for building semantic web applications. It is used by the *Context Engine*. [46]

MetaPost Graphics language with *PostScript* output by John Hobby, based on Knuth's *METAFONT*. Used for drawing architectures. [24]

MikTex Implementation of *TeX* and related programs for *Microsoft Windows*. Used to write this thesis. [80]

Protege Open source ontology editor and knowledge-base framework. Used to build the ontology and knowledge files in OWL DL. [53]

Toolkit for Conceptual Modeling (TCM) Graphical tools for creating diagrams, tables and trees. Used for drawing entity relationship diagrams, the class diagram and the topology. [93]

tuProlog *Java* based *Prolog* for internet applications by Universita di Bologna. Tested for use in the AIP implementation. [74]

WinEdt Editor for *Microsoft Windows*, qualified for *TeX* and *LaTeX*. Used to write this thesis. [86]

Xalan *Java* based XSLT processor by the *Apache XML Project*. Used in the AIP implementation. [68]

Xerces XML Parser (DOM, SAX, JAXP) in *Java* and other languages. Used in the AIP implementation. [69]

XProlog Compact *Prolog* for *Java* based agents. Tested for use in the AIP implementation. [91]

Appendix B

Config File for the Kitchen's AIP Module

```
01  <?xml version="1.0" encoding="UTF-8"?>
02
03  <AIP>
04    <ID>aip1_kitchen</ID>
05
06    <Application-Message>
07      <ID>sech</ID>
08      <IP>127.0.0.1:3001</IP>
09    </Application-Message>
10
11    <Neighbour>
12      <ID>aip6_body</ID>
13      <IP>127.0.0.1:5006</IP>
14    </Neighbour>
15
16    <Neighbour>
17      <ID>aip4_hall</ID>
18      <IP>127.0.0.1:5004</IP>
19    </Neighbour>
20
21    <KnowledgeBase>aip.JenaKnowledgeBase</KnowledgeBase>
22    <OntologyFile>sech_ont.owl</OntologyFile>
23    <KnowledgeFile>sech1_kb.owl</KnowledgeFile>
24    <RuleEngine>aip.DroolsRuleEngine</RuleEngine>
25    <RuleFile>sech1.drl</RuleFile>
26  </AIP>
```

Bibliography

[1] C. H. Bennett

Logical Reversibility of Computation.

IBM Journal of R&D, vol. 17, pp. 525-532, 1973.

[2] T. Berners-Lee, J. Hendler, O. Lassila

The Semantic Web.

Scientific American, May 2001.

[3] D. Beste, M. Kälke, K. Mainzer, E. Pöppel, H. Ritter

Das Wechselspiel zwischen KI- und Hirnforschung.

Spektrum der Wissenschaft, Dossier 4/97, Heidelberg, Germany, pp. 14-23, 1997.

[4] H. Boley et al.

The Rule Markup Initiative.

http://www.ruleml.org/ (URL, Oct. 2005).

[5] R. R. Brooks, S. S. Iyengar

Multi-sensor fusion: fundamentals and applications with software.

Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[6] P. Chen

The Entity Relationship Model: Toward a Unified View of Data.

ACM Transactions on Database Systems (TODS), Vol. 1, Number 1, pp 9-36, 1976.

[7] G. F. Cooper

Probabilistic inference using belief networks is NPhard.

Technical Report KSL-87-27, Medical Computer Science Group, Stanford University, 1987.

[8] J. Courtney et al.

JSR 197: Generic Connection Framework.

http://jcp.org/en/jsr/detail?id=197 (URL, Nov. 2005).

[9] G. Dargan, B. Johnson, M. Panchalingam, C. Stratis
The Use of Radio Frequency Identification as a Replacement for Traditional Barcoding.
Carnegie Mellon University, March 2004,
http://www.andrew.cmu.edu/user/cjs/index.html (URL, Dec. 2005).

[10] A. K. Dey, D. Salber, M. Futakawa, G. D. Abowd
An Architecture To Support Context-Aware Applications.
Technical Report GIT-GVU-99-23, Georgia Institute of Technology, USA, 1999.

[11] A. K. Dey
Understanding and Using Context.
Personal and Ubiquitous Computing, Vol 5, No. 1, pp. 4-7, 2001.

[12] A. Eberhart
Prolog2RuleML.
http://www.aifb.uni-karlsruhe.de/WBS/aeb/prolog2ruleml/ (URL, Oct. 2005).

[13] L. Fatone, P. Maponi, F. Zirilli
Data fusion and nonlinear optimization.
Siam News, 35, pp. 4-10, 2002

[14] C. Forgy
Forward and Backward Chaining: Part 2.
http://www.rulespower.com/forgy_chaining_2.htm (URL, Nov. 2005).

[15] C. Forgy
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence, 19, pp 17-37, 1982.

[16] E. Fredkin, T. Toffoli
Conservative Logic.
International Journal of Theoretical Physics, vol. 21, no. 3/4, pp. 219-253, 1982.

[17] F. Fuchs
A Modeling Technique for Context Information.
Mobile and Distributed Systems Group, Technische Universität München, Germany, 2004.

[18] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides
Design Patterns: elements of reusable object-oriented software.
Addison-Wesley, first edition, 1995.

[19] J. L. van Genderen, C. Pohl

Image fusion: issues, techniques and applications.

in Proceedings of EARSeL Workshop on Intelligent Image Fusion, J. L. van Genderen and V. Cappellini (eds.).

[20] J. Giarratano, G. Riley

Expert Systems: Principles and Programming.

PWS Publishing, Boston, USA, 3rd edition, 1999.

[21] G. Groh

Ad-Hoc-Groups in Mobile Communities – Detection, Modeling and Applications.

Doctoral thesis, Dept. of Applied Informatics and Cooperative Systems, Technische Universität München, Germany, 2005.

[22] T. Gruber

A translation approach to portable ontology specifications.

Knowledge Acquisition, 5(2), pp. 199-220, 1993.

[23] W. A. Günthner, M. Wilke

Materialflusstechnologie - Anforderungen und Konzepte für wandelbare Materialflusssysteme.

Technische Universität München, Germany, 2003.

[24] J. Hobby

MetaPost.

http://cm.bell-labs.com/who/hobby/MetaPost.html (URL, Nov. 2005).

[25] S. Haustein, M. Kroll, J. Pleumann

ME4SE.

http://kobjects.sourceforge.net/me4se/index.shtml (URL, Oct. 2005).

[26] K. Henricksen, J. Indulska, A. Rakotonirainy

Modeling Context Information in Pervasive Computing Systems.

In Proceedings First International Conference on Pervasive Computing, Pervasive 2002, pp. 167-180, 2002.

[27] A. S. Hornby, A. P. Cowie, A. C. Gimson

Oxford Advanced Learner's Dictionary of Current English.

Oxford University Press, 1982.

[28] I. Horrocks et al.

SWRL: A Semantic Web Rule Language Combining OWL and RuleML.

http://www.w3.org/Submission/SWRL/ (URL, Dec. 2005).

[29] R. Johansson

Information Acquisition in Data Fusion Systems.

Licentiate Thesis, Dept. of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, 2003.

[30] A. Kemper, A. Eickler

Datenbanksysteme.

Oldenbourg Wissenschaftsverlag GmbH, Munich, Germany, 5th edition, 2004.

[31] A. Knoll, T. Christaller

Robotik.

Fischer Taschenbuch Verlag, Frankfurt, Germany, 2003.

[32] M. Krause

Eine Sprache zur dynamischen Komposition von Kontextinformationen.

Diploma thesis, Ludwig-Maximilians-Universität München, Germany, 2003.

[33] R. Landauer

Irreversibility and Heat Generation in the Computing Process.

IBM Journal of Research and Development, vol. 5, pp. 183-191, 1961.

[34] E. K. Liebemann, K. Meder, J. Schuh, G. Nenninger

Safety and Performance Enhancement: The Bosch Electronic Stability Control (ESP).

Robert Bosch GmbH, Paper Number 05-0471,

http://www-nrd.nhtsa.dot.gov/pdf/nrd-01/esv/esv19/05-0471-O.pdf (URL, Nov. 2005).

[35] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, J. Anderson

Wireless Sensor Networks for Habitat Monitoring.

Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications, Atlanta, Georgia, USA, pp. 88-97, 2002.

[36] J. Marek, H.-P. Trah, Y. Suzuki, I. Yokomori

Sensors for Automotive Technology.

WILEY-VCH Verlag, Weinheim, Germany, ed. J. Hesse, W. Göpel, J. W. Gardner, 2003.

[37] D. L. McGuinness, F. van Harmelen
OWL Web Ontology Language Overview.
http://www.w3.org/TR/owl-features/ (URL, Nov. 2005).

[38] T. M. Mitchell
Machine Learning.
McGraw-Hill, New York, USA, 1997.

[39] L. Nagel
Automatica.
http://vs242099.vserver.de/~lars/automatica/ (URL, Jan. 2006).

[40] N. N.
The Apache Ant Project.
http://ant.apache.org/ (URL, Dec. 2005).

[41] N. N.
What are the differences between Wi-Fi (802.11b) and the Bluetooth wireless technology?.
https://www.bluetooth.org/admin/bluetooth2/faq/view_record.php?id=49 (URL, Oct. 2005).

[42] N. N.
Das BMW Techniklexikon.
http://www.bmw.de/de/faszination/index_techniklexicon.html (URL, Aug. 2005).

[43] N. N.
Drools.
http://www.drools.org/ (URL, Aug. 2005).

[44] N. N.
Habitat Monitoring on Great Duck Island.
http://www.greatduckisland.net/ (URL, Dec. 2005).

[45] N. N.
J2SE 1.4.2.
http://java.sun.com/j2se/1.4.2/ (URL, Dec. 2005).

[46] N. N.
Jena - A Semantic Web Framework for Java.
http://jena.sourceforge.net/ (URL, Aug. 2005).

[47] N. N.

Java Community Process.

http://www.jcp.org/en/home/index (URL, Aug. 2005).

[48] N. N.

Standard Ecma-340 – Near Field Communication Interface and Protocol (NFCIP-1).

Ecma International, Rue de Rhone 114, Geneva, Switzerland, 2004.

[49] N. N.

Near Field Communication White Paper.

Ecma International, Rue de Rhone 114, Geneva, Switzerland, 2005.

[50] N. N.

NFC Forum.

http://www.nfc-forum.org/home (URL, June 2005).

[51] N. N.

Weltpremiere in Hanau: RMV startet mit Nokia und Philips Pilotprojekt zum Handy-Ticketing.

Rhein-Main-Verkehrsverbund GmbH, Hofheim, Germany, 2005,

http://www.rmvplus.de/getin/NFCHandyTicketing.pdf (URL, Dec. 2005).

[52] N. N.

Nokia announces the world's first NFC enabled mobile product for contactless payment and ticketing.

Nokia Communications, Press Releases, 2005,

http://press.nokia.com/PR/200502/979695_5.html (URL, July 2005).

[53] N. N.

Protégé.

Dept. of Medical Informatics, Stanford University, USA,

http://protege.stanford.edu/ (URL, Oct. 2005).

[54] N. N.

Bevölkerung Deutschlands bis 2050.

Statistisches Bundesamt, Wiesbaden, 2003,

http://www.destatis.de/presse/deutsch/pk/2003/Bevoelkerung_2050.pdf (URL, Dec. 2005).

[55] N. N.

Sozialhilfe in Deutschland 2003.

Statistisches Bundesamt, Wiesbaden, Germany, 2003,

http://www.destatis.de/presse/deutsch/pk/2003/sozialhilfe_2003i.pdf (URL, Dec. 2005).

[56] N. N.

Drahtlos – aber wo und wie?.

In elektro AUTOMATION, Leinfelden-Echterdingen, Germany, p. 23, Feb. 2004,

http://www.3soft.de/_content/presse_e/_pdf/trendinterview.pdf (URL, Dec. 2005).

[57] N. N.

Wachsende Vielfalt für die Praxis.

In elektro AUTOMATION, Leinfelden-Echterdingen, Germany, p. 14, Nov. 2004,

http://www.3soft.de/_content/presse/_pdf/trendinterview11e04.pdf (URL, Dec. 2005).

[58] N. N.

Bluetooth.

http://en.wikipedia.org/wiki/Bluetooth (URL, Sept. 2005).

[59] N. N.

Electronic Product Code.

http://en.wikipedia.org/wiki/Electronic_Product_Code (URL, Nov. 2005).

[60] N. N.

Elektronisches Stabilitätsprogramm.

http://de.wikipedia.org/wiki/Elektronisches_Stabilit%C3%A4tsprogramm (URL, Nov. 2005).

[61] N. N.

Near Field Communication.

http://en.wikipedia.org/wiki/Near_Field_Communication (URL, July 2005).

[62] N. N.

Radio Frequency Identification.

http://de.wikipedia.org/wiki/Rfid (URL, July 2005).

[63] N. N.

Radio Frequency Identification.

http://en.wikipedia.org/wiki/RFID (URL, July 2005).

[64] N. N.

Robotik.

http://de.wikipedia.org/wiki/Robotik (Dec. 2005).

[65] N. N.

Sensor.

http://de.wikipedia.org/wiki/Sensor (URL, Sept. 2005).

[66] N. N.

Wireless LAN.

http://de.wikipedia.org/wiki/Wireless_LAN (URL, July 2005).

[67] N. N.

ZigBee.

http://en.wikipedia.org/wiki/Zigbee (URL, Oct. 2005).

[68] N. N.

Xalan.

http://xml.apache.org/xalan-j/ (Nov. 2005).

[69] N. N.

Welcome to Xerces.

http://xerces.apache.org/ (Oct. 2005).

[70] A. Newell, J. C. Shaw, H. A. Simon

Report of a general problem-solving program for a computer.

In: Proceedings of an International Conference on Information Processing, UNESCO, Paris, France, pages 256–264, 1960.

[71] P. Niemela et al.

JSR 256: Mobile Sensor API.

http://www.jcp.org/en/jsr/detail?id=256 (URL, Nov. 2005).

[72] H. S. Nwana, D. T. Ndumu

An Introduction to Agent Technology.

In: Software Agents and Soft Computing, Springer-Verlag, London, UK, ed. H. S. Nwana, N. Azarmi, pp. 3-26, 1997.

[73] C. Pohl, J. L. van Genderen

Multisensor image fusion in remote sensing: concepts, methods and applications.

International Journal of Remote Sensing, 19, pp. 823-854, 1998.

[74] A. Ricci et al.

tuProlog.

http://www.alice.unibo.it:8080/tuProlog/ (URL, Aug. 2005).

[75] T. Rose

Singularity Architecture.

http://www.i-konect.com/singularity/docs/SingularityArchitecture.pdf (URL, Dec. 2005).

[76] T. Rose

Singularity Middleware Design.

http://www.i-konect.com/singularity/docs/SingularityMiddlewareDesign.pdf (URL, Dec. 2005).

[77] P. E. Ross

Managing Care Through The Air.

IEEE Spectrum, 3 Park Avenue, New York, USA, 2004.

[78] E. Rukzio, S. Siorpaes, O. Falke, H. Hussmann

Policy Based Adaptive Services for Mobile Commerce.

Media Informatics Group, Institute of Computer Science, Ludwig-Maximilians-Universität München, Germany, 2005.

[79] S. Russell, P. Norvig

Artificial Intelligence: A Modern Approach.

Prentice Hall Series in Artificial Intelligence, Englewood Cliffs, New Jersey, USA, 2003.

[80] C. Schenk

Miktex.

http://www.miktex.org/ (URL, Dec. 2005).

[81] B. N. Schilit, M. M. Theimer

Disseminating Active Map Information to Mobile Hosts.

IEEE Network, 8(5), pages 22-32, 1994.

[82] K.-D. Schmatz

Java 2 Micro Edition: Entwicklung mobiler Anwendungen mit CLDC und MIDP.

dpunkt.verlag GmbH, Heidelberg, Germany, 2004.

[83] A. Schweikard

Maschinelles Lernen.

Lecture Slides, Technische Universität München, Germany, 2002,

http://www9.informatik.tu-muenchen.de/personen/schweikard/AS_Lehre.html (URL, Dec. 2005).

[84] A. Seaborne

RDQL - A Query Language for RDF.

http://www.w3.org/Submission/RDQL/ (URL, Dec. 2005).

[85] D. Selman et al.

JSR 94: JavaTM Rule Engine API.

http://www.jcp.org/en/jsr/detail?id=94 (URL, Dec. 2005).

[86] A. Simonic

WinEdt.

http://www.winedt.com/ (URL, Nov. 2005).

[87] S. Staab.

Intelligent Systems on the World Wide Web.

Lecture Slides, University of Karlsruhe, Germany, 2000.

[88] S. Staab.

Handbook on Ontologies.

Springer Verlag, Berlin, Germany, 2003.

[89] B. Swartout, R. Patil, K. Knight, T. Russ.

Toward distributed Use of Large-Scale Ontologies.

In Ontological Engineering, AAAI-97 Spring Symposium Series, pages 138–148, 1997.

[90] B. van Schooten

YProlog.

http://wwwhome.cs.utwente.nl/~schooten/yprolog/ (URL, Oct. 2005).

[91] J. Vaucher

XProlog.

http://www.iro.umontreal.ca/%7Evaucher/XProlog/ (URL, Aug. 2005).

[92] J. Wielemaker

SWI-Prolog.

http://www.swi-prolog.org/ (URL, Dec. 2005).

[93] R. Wieringa

Toolkit for Conceptual Modeling (TCM).

http://wwwhome.cs.utwente.nl/~tcm/ (URL, Nov. 2005).

[94] M. Wilde

Entwicklungstendenzen bei Fahrerassistenzsystemen.

heise online, Helstorfer Str. 7, Hannover, Germany, 2004,

http://www.heise.de/newsticker/meldung/49859 (URL, Dec. 2005).

[95] M. Wilke

Wandelbare Materialflusssysteme für Minifabriken.

Technische Universität München, Germany, 2004.

[96] M. Winikoff

W-Prolog.

http://goanna.cs.rmit.edu.au/~winikoff/wp/ (URL, Aug. 2005).

[97] H. Wu

Sensor Data Fusion for Context-Aware Computing Using Dempster-Shafer Theory.

Doctoral Thesis, The Robotics Institute, Carnegie Mellon University, Pittsburgh, USA, 2003.

Abbreviations

- **AI**: Artificial Intelligence .... *Special field in computer science dealing with (ostensibly) intelligent machines*
- **AIP**: Autonomous Information Processing ..... *Software architecture providing the local preprocessing of sensor data*
- **API**: Application Programming Interface ..... *Interface provided by a computer program*
- **DAML**: DARPA Agent Markup Language ..... *DAML+OIL was a predecessor of OWL, it is also based on RDF*
- **DL**: Description Logics ..... *Family of languages for the representation of knowledge, most of them are subsets of first order logic*
- **DOM**: Document Object Model ..... *Interface for the access to HTML and XML documents*
- **EPC**: Electronic Product Code ..... *Successor of UPC that supports the use of Radio Frequency Identification*
- **EPC-IS**: EPC Information Service ..... *Specification for a standard interface for accessing EPC-related information.*
- **GTIN**: Global Trade Item Number ..... *Identification number for trade items*
- **IP**: Internet Protocol ..... *IP specifies the format of packets and the addressing scheme.*
- **JSR**: Java Specification Request ..... *Collection of requests on Java features to be modified or added.*
- **JVM**: Java Virtual Machine ..... *Virtual machine for running Java on different platforms.*
- **NFC**: Near Field Communication ..... *Wireless technology for short-range wireless interaction in consumer electronics, mobile devices and PCs.*
- **OIL**: Ontology Inference Layer ..... *DAML+OIL was a predecessor of OWL*

- **OWL**: Web Ontology Language ..... *Description-Logics-based declarative ontology language*
- **PC**: Personal Computer.. *Certain type of microcomputer, used as workstation or home computer*
- **PDA**: Personal Digital Assistant ..... *Small mobile computer*
- **RDF(S)**: Resource Description Framework (Schema) ..... *Declarative Semantic-Web Ontology language for semantic nets*
- **RDQL**: RDF Data Query Language ..... *Query language for RDF based on SquishQL.*
- **RFID**: Radio Frequency Identification..... *Wireless technology for short range identification.*
- **RMV**: Rhein-Main-Verkehrsverbund ..... *Public transport association in Hessen, Germany*
- **RuleML**: Rule Markup Language ..... *Shared rule language permitting both, forward and backward rules*
- **SIG**: Bluetooth Special Interest Group..... *Group of companies interested in the further development of Bluetooth (see 1.2)*
- **SQL**: Structured Query Language ..... *Computer language for the manipulation of data in relational database systems*
- **SWRL**: Semantic Web Rule Language ... *Rule language based on OWL and Datalog RuleML*
- **UPC**: Universal Product Code ..... *Original barcode language used e. g. by retail store chains*
- **URI**: Uniform Resource Identifier ..... *Global identifier in mark-up languages and rsp. declarative standard*
- **URL**: Uniform Resource Locator ..... *Global address of documents and other resources on the World Wide Web.*
- **WLAN**: Wireless Local Area Network..... *Wireless technology for connecting a local computer network.*
- **WPAN**: Wireless Personal Area Network ..... *A WPAN is a short range wireless network interconnecting devices centered around a person's workspace.*
- **XSLT**: Extensible Stylesheet Language - Transformations ..... *XML language for the transformation of XML-trees*

List of Tables

List of Figures

Danke

Mein Dank gilt allen, die mich im letzten halben Jahr bei der Erstellung dieser Arbeit unterstützt haben.

Besonderer Dank gilt meinem Aufgabensteller Prof. Dr. Johann Schlichter für sein Interesse am Thema der Arbeit und für die Gelegenheit, diese Arbeit in Kooperation mit Siemens Corporate Technology durchzuführen.

Ebenso danke ich Rudolf Kober und der Abteilung CT IC 6 bei Siemens für die Zusammenarbeit und die zur Verfügung gestellten Ressourcen.

Mein Dank gilt auch Dr. Georg Groh, Dr. Michael Berger und Michael Pirker, die die Arbeit betreuten und viele nützliche Ratschläge gaben.

Für eine äußerst angenehme Arbeitsatmosphäre danke ich Raimund Steger und Dr. Christian Seitz, mit denen ich nebenbei ein paar nicht minder interessante Turmbauprojekte vorantrieb. Ein Dankeschön geht auch an Florian Fuchs für seine Ausdauer dabei, mir seine Software zu erklären, die in dieser Arbeit verwandt wurde.

Zudem danke ich Dr. Klaus Nagel, Nadine Perera, Raimund Steger und Jochen Rösch fürs Korrekturlesen und nicht zuletzt meinen Eltern und meiner WG für die seelische Unterstützung.