FAKULTÄT FÜR !NFORMATIK

Behavior Recognition and Prediction in Building Automation Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik
eingereicht von

Josef Mitterbauer
Matrikelnummer 0025067

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: O.Univ.Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
Mitwirkung: Dipl.-Ing. Dr.techn. Dietmar Bruckner

Wien, 9.11.2008 (Unterschrift Verfasser) (Unterschrift Betreuer)

Kurzfassung

Mit ständig besser und günstiger werdenen Technologien im Bereich der Gebäudeautomatisierung findet diese immer mehr Verbreitung, sowohl im öffentlichen Bereich als auch im Wohnbau. Durch die Verbesserungen in den Bereichen der Sensorik, Aktuatorik und Kommunikationssystemen, werden immer leistungsfähigere Systeme möglich. Jedoch steigt mit der Leistungsfähigkeit auch die Komplexität. Damit auch in Zukunft mit der zu erwartenden Komplexität umgegangen werden kann, sind neue Verfahren zur Verarbeitung von Sensorwerten erforderlich. Eine Möglichkeit besteht darin, mithilfe statistischer Methoden die erfasste Situation in Gebäuden zu beschreiben. Eine interessante Fragestellung ist, ob sich aus derartigen Modellen Prognosen für zukünftig zu erwartende Situationen ableiten lassen.

In dieser Arbeit wird untersucht, inwieweit sich die Theorie über Hidden Markov Modelle (HMM) eignet, um das Verhalten von Personen in einem Raum zu beschreiben und anhand der erlernten Beschreibungen Vorhersagen über das zukünftige Verhalten einer Person zu machen. Das Erstellen dieser Prognosen basiert auf bekannten Algorithmen. Weiters wird untersucht, inwieweit sich Daten unterschiedlicher Sensortypen in ein Modell integrieren lassen und welche Vorhersagen so ein System liefern kann. In dieser Arbeit meint „Vorhersage des Verhaltens“ die Berechnung der Wahrscheinlichkeit der möglichen Aktionen, die eine Person als nächstes ausführen kann und die auch vom System wahrgenommen werden können. Jene Aktionen mit der höchsten Wahrscheinlichkeit werden den Personen mitgeteilt. Aus diesem Grund sollte die Verarbeitung der Daten in Echtzeit erfolgen. Hierfür wurde eine Software Applikation entwickelt, welcher eine Liveanbindung an das Sensorwerterfassungssystem zur Verfügung steht. Diese Anwendung ermöglicht den Personen im Gebäude (d.h. in dem Raum, wo das System betrieben wird), einen Blick hinter die Kulissen des Systems, indem sie die relevanten Informationen auf einem Bildschirm darstellt, welcher von den Benutzern betrachtet werden kann.

Diese Darstellung zeigt einen Grundriss des Raumes und alle installierten Sensoren. Aus den ausgelösten Sensoren wird eine Abschätzung der vorhandenen Personen und deren Positionen errechnet, die dargestellt wird. Für jede dieser vermuteten Personen wird eine Vorhersage errechnet und ebenfalls dargestellt. Dadurch kann ein Benutzer sehen, an welcher Position er vom System vermutet wird und welche Vorhersage errechnet wurde.

Es hat sich gezeigt, dass HMMs für die Aufgabe der Modellierung derartiger Syteme gut geeignet sind und die Möglichkeiten des Erstellens von Modellen mannigfaltig sind. Die Art der Generierung ist entscheidend für die Qualität der Modelle. Weiters stellte sich heraus, dass in eher kleinen Räumen, wie der, der zur Verfügung stand, die Handlungen der Personen relativ ident sind, aber trotzdem große Unterschiede zwischen ständig auftretenden und eher seltenen Szenarios klar erkennbar sind.

Die Integration von Werten verschiedener Sensortypen in ein Modell ist insofern problematisch, als diese Werte untschiedliche Wahrscheinlichkeiten des Auftretens haben. Dies wurde durch eine Priorisierung bei der Auswertung kompensiert. Dieses Vorgehen bringt gute Lösungen in dieser Anwendung, da alle Aktionen sehr stark auf die räumliche Lokalität bezogen sind, kann aber mit hoher Wahrscheinlichkeit nicht verallgemeinert werden. Für einen allgemeinen Lösungsansatz sollte hier eine zusätzliche Abstraktionsebene eingeführt werden.

Abstract

Building automation systems are spreading more and more, thanks to technologies in that field constantly becoming better and cheaper; not only in public spaces but also in domestic architecture. Due to the improvements in the field of sensors, actuators and communication systems, increasingly efficient systems can be realized. But together with the efficiency also the complexity increases. For handling the expected complexity in future times, new methods for the processing of sensor values become necessary. The use of statistic methods is one possibility for describing recognized situations in buildings. It is an interesting question if it is possible to derive prognoses for expected future situations from such models.

How far the theory about Hidden Markov Models is useful for describing the behavior of persons in a room and to make predictions about a person's behavior out of the learned descriptions, will be determined in this work. The calculation of these predictions is based on common algorithms. In which way data of different sensor types can be integrated in one model and which predictions such a system can make, is also topic of this work. "Prediction of behavior" in this work means the calculation of the probability of possible actions (which can be perceived by the system) a person can do next. Those actions with the highest probability will be shown to the persons. For this reason data processing should be done in real-time. Therefore a software application, providing a live-data connection to the sensor value gathering hardware, was developed. This application allows the persons in the building (i.e. the room in which the system runs), a look behind the scenes of the system, because it shows the relevant information on a screen that can be watched by the users.

This illustration shows the layout of the room and all the installed sensors. From the triggered sensors an estimation of the number of present persons and their positions is calculated and depicted. For each of the assumed persons a prediction is calculated and also depicted. Therefore a user can see, where on which position the system assumes him to be and which prediction was calculated.

It has turned out that HMMs are very appropriate for the task of modeling such systems and that there are many possibilities for the creation of models. The quality of the models depends on the kind of generation. It also turned out that in rather small rooms, like the one we have used for our disposal, the actions of the persons are relatively identical, however, big differences between frequently occurring scenarios and rather rare ones can be observed.

The integration of values from different sensor types into one model is problematic so far, as these values have different probabilities of occurrences. This was compensated by a prioritization in the evaluation. This method brings good solutions for this application, because all actions are strongly referring to the spatial locality of persons, but it likely cannot be generalized. For a general approach an additional level of abstraction should be introduced.

Acknowledgements

Writing this diploma thesis would not have been possible without the support of several mentors. First of all I would like to thank Prof. Dietmar Dietrich for the opportunity to write this thesis at the Institute of Computer Technology. Special thanks go to my supervisor Dietmar Bruckner and the ARS project leader Gerhard Zucker for their great support to my thesis. Further, I would like to thank all members of the ARS team for fruitful hints as well as humorous private discussions. My deep gratitude goes to my parents, for the opportunity to study and their support during my student's career. Furthermore, I would like to thank my brother Ferdinand and my cousins Ludwig and Dieter for fruitful technical discussions and a humorous leisure time. Special thanks go to my girlfriend Silvia for loving and supporting me.

Table of Contents

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ARS	Artificial Recognition System
ASN.1	Abstract Syntax Notation One
BASE	Building Assistance System for Safety and Energy efficiency
CPU	Central Processing Unit
DB	Database
DER	Distinguished Encoding Rules
DBMS	Database Management System
FSA	Finite State Automaton
GUI	Graphical User Interface
HBM	Heart Beat Message
HMM	Hidden Markov Model
I^2C	Inter-Integrated Circuit
ICT	Institute of Computer Technology
ID	Identification
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
JDBC	Java Database Connectivity
MAP	Maximum A Posteriori
MHz	Megahertz
ML	Maximum Likelihood
MM	Markov Model
MSF	Micro Symbol Factory
MVC	Model View Controller
NFA	Nondeterministic Finite Automaton
NFS	Network File System
NP3	Neurology, Psychology, Psychoanalysis and Pedagogic

NSM	New Symbol Message
OOD	Object Oriented Design
OOP	Object Oriented Programming
PC	Personal Computer
pdf	probability density function
PIR	Passive Infra-Red
PVP	Probability of Viterbi Path
pmf	probability mass function
SmaKi	Smart Kitchen
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Introduction

Due to continuous improvements in technology, electronic equipment becomes more powerful and cheaper. This makes it possible to establish new fields of applications. Some of those new areas are ubiquitous computing and ambient intelligence, however, some ideas exist quite a long time, like described in [Wei91]. Technology is also getting smaller, this makes it possible to hide it from the users, to make it invisible and incorporate it into our everyday's surroundings. The vision is, to have an intelligent environment which is able to fulfill tasks self-contained. Explicit user interactions are not necessary any longer to execute everyday's duties, however, the system should offer support if wanted. One big research area for those new applications are building automation systems. Future houses should make life more comfortable, safe and secure for their occupants. To overcome this challenge, developments in many fields of computer technology are required, like chip design, power supplies, sensors, actuators, communication technology, data mining, data protection, data encryption, privacy, pattern recognition, artificial intelligence, and many others.

Today's and future automation systems will be equipped with more and more sensors for monitoring and for controlling functionality [Rus03]. This brings along new challenges to the information processing modules. To study these new arising problems the ARS project was founded, which is described in Section 1.1.

This chapter gives an overview of the context this work is related to. The first section is a motivation to the enclosing research project in the broader sense, which deals with basic problems of today's building automation systems. The second section describes the basic goals of this work, finally the third section gives an overview about the approach to solve the given problems.

1.1 ARS Project

The *Artificial Recognition System* (ARS) project was founded at the Institute of Computer Technology (ICT) in the year 2000. Originally the institute's focus was on research in field buses and their applications. Due to rising complexity of modern building automation systems, traditional approaches reached their limits. So the idea came up to find out what kind of methods nature provides to deal with complex problems. This was the start of the ARS project [Die00].

As the ARS project introduction [8] says, the aim of the project was to use results from other research areas than engineering: Studying the human brain is a task fascinating humans a fairly long time. However, especially the last 20 years the work of neurologists, psychologists, psychoanalysts and pedagogics (NP3) brought amazing results. It was intimating to use that expertises in the bionic field as well and to add them to traditional Artificial Intelligence (AI). Currently NP3 hold that consciousness and human consciousness cannot be explained by classical methods of mathematics.

As described in [PP05] there are two main fields of research within the ARS project, which are ARS-PC (Perceptive Consciousness) and ARS-PA (Psychoanalysis). The field of ARS-PC deals with the scientific theme, how to transform pure sensor data to images and scenarios which can be used at a higher level information processing unit. The field of ARS-PA researches how the Id-Superego-Ego model of Sigmund Freud [Fre23] can be used for technical concepts. This model is much more complex than other models which are currently used in automation systems [DLP+06].

Furthermore, there are two projects in the surroundings of the ARS which should be mentioned for the sake of completeness. This is the *Building Automation system for Safety and Energy efficiency* (BASE) project, described in [SBR05b, SBR05a]. This project uses statistical methods to model the situations in a building. The second one is the *Smart Kitchen* (SmaKi) project, described in [SRT00].

This work is related to the projects ARS-PC, BASE and SmaKi. The hardware which was installed for the SmaKi project delivers raw sensor data which has to be integrated into a system; statistical methods are used to build a model for making predictions. Inspired by the PHD-thesis of Dietmar Bruckner [Bru07], which uses HMMs for modeling scenarios in buildings automation systems, this work uses the approach of HMMs to make predictions of the behavior of persons, using the forward algorithm.

1.2 Problem Statement

In the course of the ARS and the BASE project, the idea emerged, to use the approach of HMMs for a statistical prediction of the behavior from persons in a building. As there is a room equipped with sensors available from a former project (SmaKi), this room was chosen as the object of interest. This room is the ICT's kitchen, henceforth called *SmaKi*. So within this work the term *SmaKi* denotes the kitchen of the ICT, i.e. the room with all the installed sensors, the hardware which is necessary to read their values and the software to transmit the perceived data.

The assignment for this work can be subdivided into two tasks, the generation of a prediction model on the one hand and the live-data visualization on the other hand.

Behavior Prediction Model

The goal is to model the behavior of a person inside the room with an HMM. Once a model is generated, common algorithms can be used to make predictions of the expected behavior of the person. The main focus is on the position data of a person, i.e. the task is to make predictions of the next position the person will take. As the test room is equipped with several other sensors, like closet door switches or a coffee machine vibration detection sensor, it should be analyzed if such sensor data can be integrated in one HMM. So it would be possible to make predictions on other activities than the movements, like the opening of the fridge or the preparation of coffee. This presumes that these activities can be associated with a certain position.

Real-Time Live-Data Visualization

The results of the system's prediction should be visualized to the kitchen's users, i.e. an application which displays the actual situation in the room on a screen needs to be developed. This should be done by showing the layout of the room in a (software-)window with all the occurring events, i.e. events which are recognized by the system. The application gets the current sensor values from the SmaKi's hardware in real-time and displays the triggered sensors. From the sensor values is calculated the number of persons which are currently present in the room and the positions of those persons. However, these are estimations. The system displays icons at the positions where it 'believes' that a person is. These positions are used for the *Behavior Prediction Model*, mentioned above. This model is realized by an HMM, which can be depicted as a graph. The graph should be visualized in a separate window, if wanted by the user. As the model calculates predictions, those values should be visualized as well, in case of predicted positions this is an arrow from the person to the position, in case of other predictions this should be visualized in some meaningful way. The window which is showing the room's layout gets the data live from the hardware and should display the person estimations and predictions in real-time. However, there are no hard constraints of deadlines, only for usability the persons in the room should see what the system actually recognizes. So this is a *soft* real-time system [Kop97]. The configuration of the basic properties of the application should be easy, i.e. without recompilation of the software.

1.3 Outline

The environment of this project is described in Chapter 2 ("Environment"). This includes the hardware which is used as well as software parts from former projects which are necessary for this work.
Chapter 3 ("Hidden Markov Models") gives an overview about the theory of Hidden Markov Models (HMM). The structure of an HMM is described as well as some basic algorithms.
The Chapter 4 ("System Design") presents a description of some problems which came to light during the development process and the approaches to solve these problems. The creation of HMMs from scratch is explained within this chapter.
The Chapter 5 ("Software Application Design") gives an overview about the design of the software application which was developed within this work.
The implementation of the software application described in the previous chapter is explained in Chapter 6 ("Implementation"). Simple parts of the software implementation, which do not contribute anything to a better understanding of the system, are omitted.
Finally the results of this work are depicted in Chapter 7 ("Results and Discussion") and some interpretations of these results are given too.
Appendix A ("Config Files and Resources") shows examples of the configuration files which allow to adapt the settings of the connections and to customize the visualization.
Appendix B ("List of Classes") shows a list of all classes which are used to build the application.

2 Environment

This chapter attempts to give an overview of the technical implementation of the environment for this project. At this work we focus on a higher level of data processing. It is not of interest how the hardware works. However, it is necessary to understand the basic entities which are used to accomplish this work. As one goal is to make predictions of a person's behavior in a building, we need an environment where this can be tested. A whole building would be an overkill and far too complex and expensive, therefore it is restricted to one room. This room is the kitchen of the ICT, called Smart Kitchen (SmaKi). For previous projects this room was equipped with different sensors for observation of occurrences in the room. Section 2.1 describes this room in detail as far as it is necessary for this project. This includes the layout and the different types of sensors which are installed. Information about the sensors is stored at the *ARS Sensor Database* which is described in Section 2.2. This database contains information about the mounted sensors as well as values retrieved from these sensors. Whenever the recognition system is running, each retrieved sensor value is stored at this database. The second goal of this work is to make a live data application. For this reason it is necessary to retrieve sensor data in real-time. However, this is not very strict for this work, so it is a soft real-time system. Nevertheless a live-data connection to the SmaKi's recognition hardware is required. This part is described in Section 2.3. Figure 2.1 shows an overview of the environment in concerning this work. *SmaKi Prediction Application* and *Visualization* are part of this project.

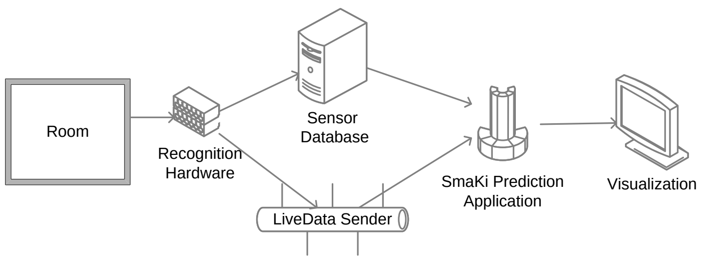


Figure 2.1: Environment Overview

2.1 Smart Kitchen

The Smart Kitchen Project (SmaKi) was started in the year 2000 at the Institute of Computer Technology (ICT) [7] in order to evaluate future aspects of building automation. The original intention was to elaborate field bus systems for smart applications in future buildings as well as the integration of many different sensors. The term SmaKi is used for naming the room where this project was implemented as well, since this is the kitchen of the ICT. The ICT is not a normal household, for this reason there is some office stuff in the kitchen, like a copier and bookshelfs[1]. The server cabinet contains some hardware of the SmaKi, for example the Octobus, described in Section 2.3.1.

Already for former projects, the room was equipped with several different sensors, which are tactile floor sensors, movement detection sensors, switch sensors, a vibration detection sensor and a camera. A description of the sensors of the SmaKi is given in Section 2.1.1. The camera is not used for this project. Figure 2.2 shows a picture of the SmaKi without floor cover. What looks like a black mat on the floor are the tactile floor sensors.

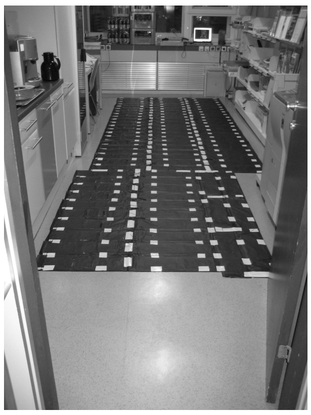


Figure 2.2: Picture of Smart Kitchen

This section describes the technical implementation of the SmaKi as far as it is necessary for this work, i.e. the room's layout and the sensor types. A detailed description of the SmaKi project is given at [SRT00]. As mentioned above, the interfaces to the SmaKi's hardware environment are the ARS Sensor Database (see Section 2.2) and the Octobus (see Section 2.3.1).

2.1.1 Layout

To get an overview of the installed sensors, this section describes the layout of the SmaKi. It may seem surprising, finding a copier and a bookshelf in a kitchen, this is because the room was not only used as a kitchen. For the project this has no effect. Figure 2.3 shows the layout of the SmaKi. At the left hand side there is the door, on the right a shelf. At the bottom (from left to right) there is a shelf, a copier and a bookshelf. On the top (again from left to right) there is the kitchenette with a coffee machine, a fridge, a server cabinet and a desk. The coffee machine

[1]In August 2008 there were made some modifications, so the office stuff has gone, but this project was started before.

and the fridge are most frequently used appliances in an office kitchen. In the server cabinet there is some hardware for the sensor evaluation and the computer for the data processing. The application, which was developed as a practical part of this work, is also running on this machine. On a screen in the server cabinet the people in the kitchen can see how the kitchen's system works, i.e. a visualization is running there. All up to here mentioned objects are static, i.e. they don't move around. This is a kind of a priory knowledge. At the center of the right third of the room there are a table and some chairs. These are moveable objects, however, they have no sensors and therefore they cannot be detected directly[2]. These objects can only be recognized by the sensors of the SmaKi, so they are just objects. A detailed discussion of this problem is given in Section 4.2.

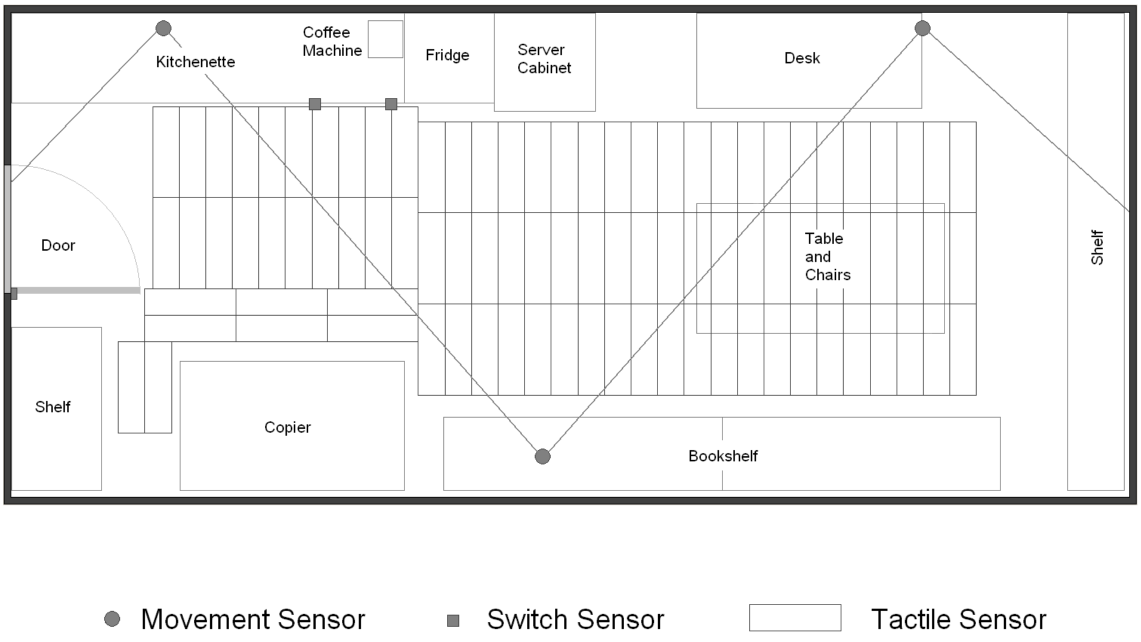


Figure 2.3: Layout of Smart Kitchen

As shown in Figure 2.3 there are three different kinds of sensors[3]: Several tactile sensors on the floor, three movement sensors, mounted at the walls and three so called switch sensors. These switch sensors are: A door switch, indicating if the door is open or closed, a fridge switch with the same functionality like the door switch and a coffee machine switch, which is indicating if the coffee machine is working. To be correct it should be mentioned that the last one is a vibration detection sensor, however, for this work this distinction is not necessary. The three movement detection sensors have a sphere of action which is indicated by the lines between the sensors, but be aware that this is only an estimation. The placement of the tactile sensors is also shown (the small, gridded rectangles). Note that in the area of the entrance there are no such sensors.

The positions of the sensors shown in Figure 2.3 are the real positions of the sensors taken from the ARS Sensor Database (see Section 2.2). To get the tactile sensor's identifications (IDs) corresponding to the layout, a helper function of the application can be used. When the application is started, select "Test/Test1" from the menu of the GUI (see Screenshot 6.1).

[2]In the implementation of the visualization there is a static object *table*, too.
[3]which are used for this project

2.1.2 Tactile Sensors

The floor of the SmaKi is equipped with 97 tactile sensors. The alignment of the sensors can be seen at Figure 2.3, a configuration schema for a single sensor is shown in Figure 2.4. A tactile sensor gives a signal *true* if some pressure is performed, i.e. if a person stands on it. The sensors have a size of 600 x 175 mm. They are described by three points, like shown in Figure 2.4. The center position c can be calculated by equation 2.1 using the three[4] positions p_i.

$$\begin{aligned} c.x &= \tfrac{max(p_i.x)+min(p_i.x)}{2} \qquad \text{for } i = 1,2,3 \\ c.y &= \tfrac{max(p_i.y)+min(p_i.y)}{2} \qquad \text{for } i = 1,2,3 \end{aligned} \tag{2.1}$$

Due to protection of physical influences the tactile sensors are placed between the blank bottom and the floor covering. Although this is a necessity it makes some problems, because the floor covering distributes the pressure performed by a person. Sometimes this results in erroneous triggering of sensors, especially at the border of the covered area.

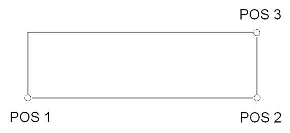


Figure 2.4: Tactile Sensor with Positions

2.1.3 Movement Sensors

In the SmaKi there are three Passive Infra-Red (PIR) motion detection sensors. They are called passive because they do not emit energy, but only observe the IR-spectrum of their environment. PIR sensors detect changes in the infrared spectrum. For this reason they can detect movements of persons since the body's warmness belongs into the IR spectrum. If the changing rate is beyond a threshold, the sensor is triggered. Because of the evaluation of changing rates a PIR sensor cannot detect very slow movements and accordingly IR emitting sources which are too far away.

The PIR sensors of the SmaKi are modeled with a pyramid-like field of perception [Goe06]. Thus this field is described by four points. Point one, the apex of the pyramid, is the point where the sensor is mounted. The points (2,3,4) define a rectangle as it is described for the tactile sensors. This rectangle is the base of the pyramid.

As experiments have shown, this field of perception is not very exact. To avoid using a 3D model of the SmaKi room in the software implementation and the given inexactness of the pyramid shape, a simplified calculation of the sphere of action for the movement sensors is used: The sphere of action is approximated by a simple 2D-rectangle.

[4]This ensures a correct calculation, independent from the order of the positions in the ARS Sensor Database.

2.1.4 Switch Sensors

The door, the fridge and the cabinets[5] are equipped with contact switches which indicate the status of their door. The status can be *opened* or *closed*, thus it can be represented by a boolean value whereas *true* means *opened* and *false* means *closed*. A vibration detection sensor, mounted at the coffee machine indicates if it is working. This is a boolean value too (*true*: coffee machine is working; *false*: not working). Combined, these sensors are called *switch sensors*, since a distinction of how the sensors work physically is not necessary at the level of abstraction we focus on. Switch sensors are defined by only one position which is the location where they are mounted in the room. The semantics of a switch sensor is given by the sensor itself, e.g. the fridge door contact switch indicates if the fridge is open or closed.

2.2 ARS Sensor Database

As known from Subsection 2.1.1 we have a room equipped with a high number of sensors. A question might be, where the important information about the sensors can be found. For example we have to know which type of sensor is located at which position in the room. Moreover sensors can have additional properties which are important to understand the information given by the sensor. A further question is, which type of data gives a sensor, i.e. its value domain. All this information is stored in a database. This is called *static data*, since once the room's system implementation is finished, this data is never changed, except in case of broken hardware or for maintenance. This is information of *how* the system perceives its environment. A description of this information is given in Subsection 2.2.1. Subsection 2.2.2 describes how the information coming from the sensors is stored. This is called "sensor data", the data which is collected when the system is running, i.e. *what* the system perceives. Finally, Subsection 2.2.3 gives an overview to the database management system which is used for the implementation.

In terms of reusability for other rooms or even buildings a relational database schema was developed to store such sensor information. This is the so called *ARS Sensor Database*. Figure 2.5 shows an entity relationship diagram of this database schema. All relations are one-to-many relations, where 'one' is indicated by the key-symbols (because there are foreign keys at those tables) and 'many' is indicated by the infinity symbol ∞.

2.2.1 Static Data

This Section gives an overview about the static data in the ARS Sensor Database. Static data is information which doesn't change after it is entered once. This is for example the number of sensors, the types of sensors and their properties. Only in case of a broken sensor this data will need a change, so changes of static data are only for maintenance, but not for operation. However, this static data is a necessity for running the SmaKi application since the information about the sensors comes from the database.

The database schema was designed to be flexible and to be able to handle several rooms, buildings, sensortypes etc. The design of the database is shown in Figure 2.5. Table 2.1 describes the relations, i.e. the tables of the database[6].

[5]However, the switches of the cabinets are not connected.

[6]The prefix"ARS_SEN" of all the tables is omitted here.

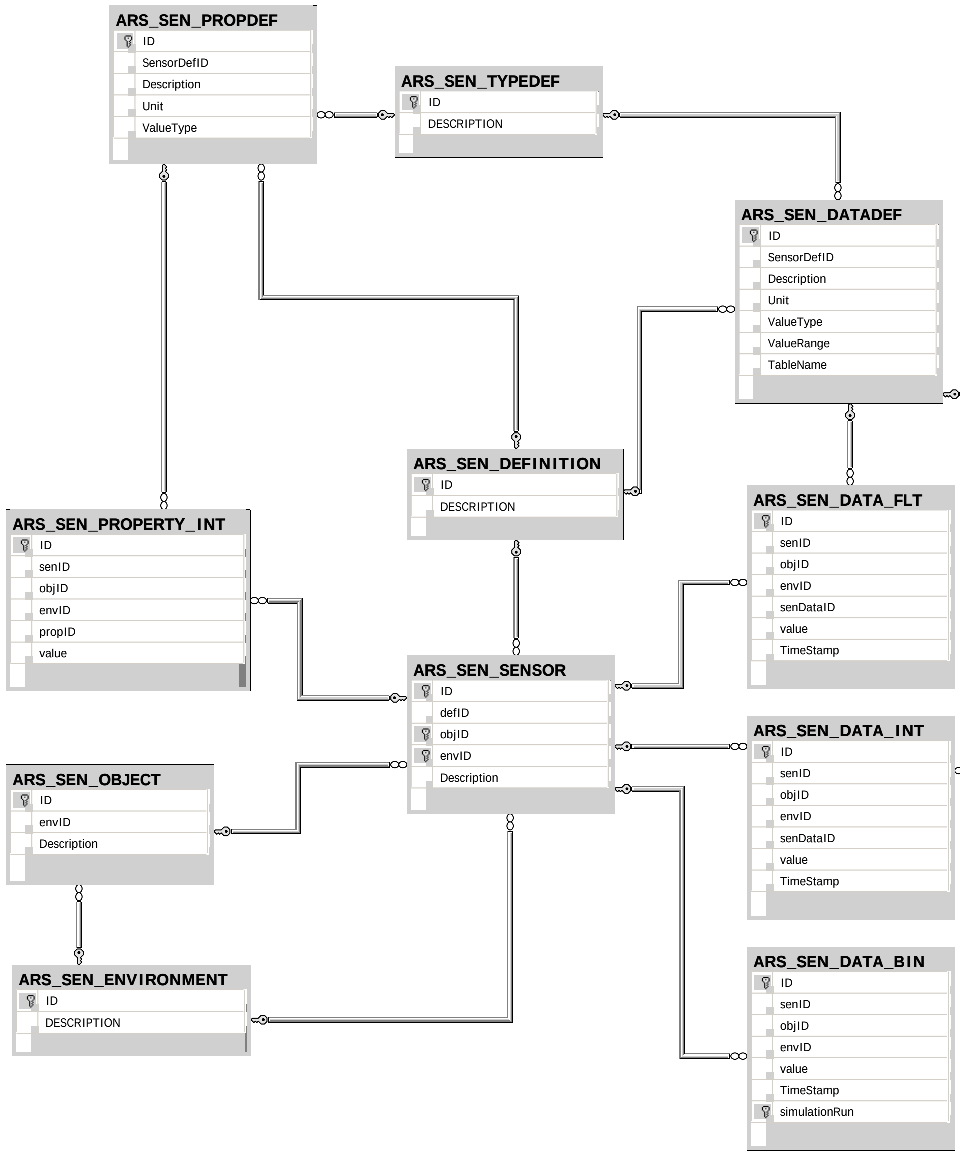


Figure 2.5: Entity Relationship Diagram of the ARS Sensor Database

The ARS Sensor Database is very powerful, however, for this work some attributes are constant. For this reason there could be made some simplifications which make the SQL-statements in the implementation part easier and shortens their execution time.

Simplification Assumptions

These assumptions are based on the given configuration of the SmaKi at the time of this project.

Since we have only one room (and this room belongs to one building) there is only one environment and one object. In other words, these values are constant. So we don't care about these tables and their entries for a sensor, respectively. As a further consequence the field ID in the table SENSOR becomes primary key.

Table 2.1: ARS Sensor Database Relations

ENVIRONMENT	This is for example the building where the sensors are mounted.
OBJECT	An environment can have several objects, e.g. a building has multiple rooms.
SENSOR	A sensor belongs to an environment and a room. It can be identified by their IDs and a sensor ID. So the sensor ID does not need to be unique in a global sense, but only locally for the room of interest.
TYPEDEF	The definition of data types, like *boolean*, *integer*, *float*. It is used for the sensors's properties as well as for the sensor's data.
DEFINITION	Stores which kind of sensor is available. For example *Movement, Switch, Tactile.*
PROBDEF	Each type of sensor has specific properties. For example a tactile sensor has three positions and a position has three coordinates (x, y, z).
PROPERTY_INT	This table stores the properties of a concrete sensor. For example: The door switch sensor is at position x=1850 y=0 z=2000.
DATADEF	The data given by a type of sensor has a value domain, like switch sensors return boolean values.
DATA_{*}	For dynamic data see Section 2.2.2.

All sensors in the database have a boolean value domain, in the database they are called binary sensors or binary data in case of their output is supposed. So the table DATADEF is not needed.

All sensor-type's properties are positions. A position has three coordinates which are given in millimeters. This data is stored as an integer datatype. So we have only integers for the properties. Together with the assumption about the value domain of the sensor's data, the table TYPEDEF can be neglected, since now all used datatypes are known.

However, the sensor database does not only contain this static data but there is also the possibility to save dynamic data there. The term dynamic data addresses the values of the sensor when the SmaKi's recognition system is running. This means that it is possible to save every single sensor value.

2.2.2 Dynamic Data

As mentioned above, it is also possible to save every single sensor value during the SmaKi's recognition system is running, this is the dynamic data.

As we know from Section *static data*, every sensor can be identified by the sensor's ID[7]. Since all used sensors of the SmaKi are boolean, we need only the table ARS_SEN_DATA_BIN. The values in this table have positive logic, so one represents true and zero represents false. For each detection there is created a sensor value, so each sensor value is represented by a row in this table.

[7]the primary key at the table "SENSOR" in the database

Again we make a simplification assumption: Since we do not work with simulations the value of the field simulationRun is constant. Therefore it can be neglected and the field ID becomes primary key (instead of ID **and** simulationRun).

A whole data set entry is composed of the columns shown in Table 2.2.

Table 2.2: Columns of Table SEN_DATA_BIN

ID	the primary key
senID	foreign key to the field ID of table sensor
objID	not used since simplification assumption about static data
envID	not used since simplification assumption about static data
value	the value of the sensor (integer)
TimeStamp	a timestamp when the value was retrieved
simulationRun	not used since simplification assmuption about dynamic data

As can be seen in Table 2.2 we only need three entries of one row, therefore a sensor value can be represented by a triple (*Sensor ID, value, TimeStamp*).

Dynamic data is added to the database whenever the recognition system is forced to do so. This is when the sensor value observation system is started with the appropriate parameter. See Section Octobus 2.3.1 for detail.

2.2.3 Database Management System

A Database Management System is:

> "The software that manages and controls access to data in a database. In a database management system (DBMS), the mechanisms for defining its structures for storing and retrieving date efficiently are known as data management. A database is a collection of logically related data with some inherent meaning [GrH07]."

There are a lot of vendors for DBMSs. When the ARS Sensor Database was built, a DBMS of the company Oracle[8] was used. Due to the fact that it is no good idea to use original data during the developing process, it makes sense to have a clone of the ARS Sensor Database. Furthermore, in our application it increases performance if the database (DB) is locally, this is a benefit which is of interest especially during the developing process, because sometimes huge amounts of data are required. Since my experiences with Microsoft's DBMS, the circumstance that a 'light' version of it is available for free and the easy installation process, it was decided to make a clone of the original Oracle DB to a Microsoft[9] DB. This is the reason for the ability of the application to deal with different DBMSs.

[8]`www.oracle.com`
[9]`www.microsoft.com`

Oracle DBMS

As mentioned above, the original ARS Sensor Database is based on Oracle. For accessing Oracle DBs an open source tool called *iSQL-Viewer* (IndependentSQL-Viewer) was used. This tool is based on Java, and it is available for free at *sourceforge.net* [9]. The project description says:

> "iSQL(IndependentSQL)-Viewer is a JDBC 2.0-compliant application that is designed to exploit JDBC Features for all compliant drivers. Support for Interactive Transactions, Running Batches, Schema Viewing, and support for various import and export filters."

For connecting to an Oracle DB, an Oracle JDBC (Java Database Connectivity) driver is necessary. To get access to the ARS Sensor Database, a so called *service* has to be defined, which includes the following data:

Table 2.3: iSQL-Viewer Service Configuration

Connection Name	Example Name
JDBC-Driver	oracle.jdbc.driver.OracleDriver
JDBC-URL	jdbc:oracle:thin:@jupiter.ict.tuwien.ac.at:1521:ict
Username	ARS_SENSORS

The line *JDBC-URL* consists of the following parts:

Table 2.4: Connection configuration for Oracle DB

driver prefix	jdbc:oracle:thin:@
hostname	jupiter.ict.tuwien.ac.at
port	1521
sid	ict

Sid is the name of the database within an Oracle-DBMS, since a common DBMS can manage several databases. The path to the JDBC-driver has to be added at the *Resources* tab.

Microsoft DBMS

For this project the *Microsoft SQL Server 2005 Express Edition* was used. This is a 'light' version of the Microsoft's DBMS. It is available for free and can be downloaded from Microsoft's hompage. Simple and small Microsoft databases (as we have) consist of two files, which can be copied and attached to a new instance of *SQL Server 2005 (Express)* very easily. For maintainance of this product the tool *Microsoft SQL Server Management Studio Express* is necessary. It is available for free as well. Note that currently there is no way to write new sensor values automatically to a Microsoft DB, since this DB was designated for development only. It is essential to choose *Mixed Authentication Mode* when installing the *SQL Server 2005* product, because otherwise it is not possible to gain access with a Java application. The parameters as described for the Oracle DB are at the administrator's choice. The default port for TCP/IP connections is 1433, however, this service is disabled by default within the *Express* edition[10].

[10]Microsoft Knowledge Base Article 914277

2.3 Data Gathering Modules

This section describes the interface between the kitchen's hardware and the SmaKi application's software. For the hardware part an embedded Linux system is used. The application's software should be able to run at a standard PC. Subsection *Octobus* describes the hardware which collects the data of every single sensor, Subsection *SymbolNet* describes the communication protocol which is used for sending data from the embedded system to the PC.

2.3.1 Octobus

All sensors (except cameras) from the SmaKi are connected to an embedded system platform. This is a product of the company haag.cc®[11] and it is called Octobus®. The core of the Octobus is an IBM[12] PowerPC™ CPU module operating at 333 MHz. As operating system an embedded Linux is used. The sensors are attached to extension boards which are connected with the platform by an I^{2}C bus. Each extension board features 12 digital inputs and 12 digital outputs, the bus system can address up to 16 board, so in total the Octobus supports 184 inputs/outputs. The connectivity to a standard PC is given by two ethernet ports. Figure 2.6 shows an image of the Octobus.

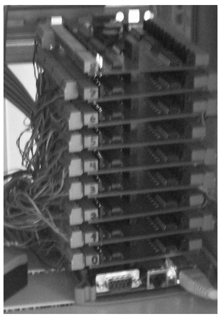


Figure 2.6: The Octobus®

For transmission of sensor data the micro symbol factory (MSF) is implemented at the embedded platform. The MSF is the starting point in the *SymbolNet* communication protocol which was developed for the ARS System. A description of *SymbolNet* is given in Section 2.3.2. Figure 2.7 shows a schematic illustration of the SmaKi hardware/software interface.

To make the software development for the target system easier, the filesystem of the Octobus is hosted on a remote PC. This is a linux machine running a *network file-system* (NFS) server. All necessary tools and the development software for the embedded system is available at this PC. Connectivity is given by a TCP/IP network. Furthermore, there is a proxy daemon process running at the remote PC which allows the forwarding (and possible transformations) of the Octobus's sensor data to the ARS sensor database. The Octobus itself, i.e. the hardware, is located at the server cabinet in the SmaKi.

[11] `www.haag.cc`
[12] `www.ibm.com`

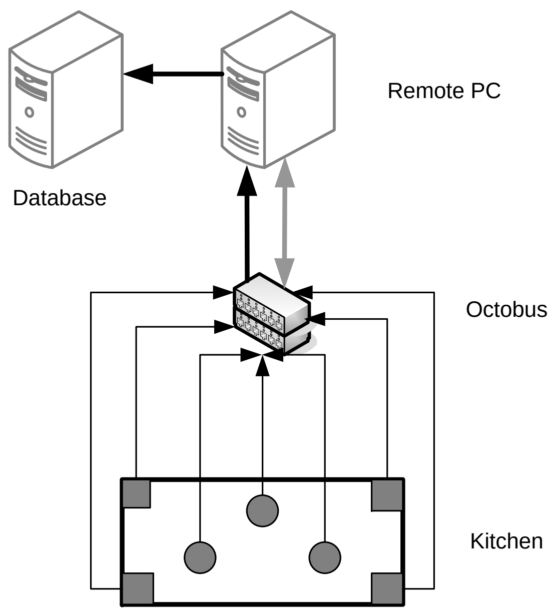


Figure 2.7: SmaKi Hardware/Software Interface

Hence, for correct data recording three systems have to be online:

- Octobus hardware
- Remote PC
- Database Server

If data recording is not necessary but only data recognition, the database server is not necessary, however, since the Octobus's file system is hosted on the remote PC, this PC is a necessity. The Octobus can be accesssed by a secure shell.

The software module running at the Octobus which collects and preprocesses all sensor information is called *sensaware* and can be started via command line. Furthermore, this software allows sending this data to a TCP/IP network. The frequently used command line arguments for the *sensaware* program are described in Table 2.5.

Example

We want to send sensor data to an application on the host 128.131.80.119 (port 59999). Moreover the sensor data should be stored in a database at a server 128.131.80.167[13] (port 2222). The name of the database within the DBMS is *ARSNEW*, the database username *username* and the password *password*. This can be done by typing the follwing command line at the Octobus:

[13]The real database might be on another server, this is only the address where the proxy daemon is running.

Table 2.5: Arguments for sensaware

`-help`	Get help on the target system.
`-c` ⟨v⟩	Read the sensors ‘v’ times. Default value is 0, i.e. endless mode.
`-F` ⟨IP:port⟩	Forward to specified host.
`-dburl` ⟨`url`⟩	Write to database with specified connection string ‘url’. If no argument of this type is specified, a default database is used. To avoid writing to any database, write ‘-dburl none’. Example for ‘url’: 128.131.80.167 2222 oracle:uid=user/pass@dbname.

```
sensaware -c 0 -dburl 128.131.80.167 2222 oracle:uid=username/password@ARSNEW
-F 128.131.80.119:59999
```

A more detailed description of the sensaware parameters can be found at [Goe06].

2.3.2 SymbolNet

At the SmaKi’s hardware/software interface the software package *SymbolNet* is used for transmission of sensor data. It was developed at the ICT for the ARS project to allow symbolic data processing. While *SymbolNet* is quite powerful here only a subset of the functionality is used and only the used parts are described in the following. A complete description can be found at [Hol06].

The three main parts of the *SymbolNet* framework are symbol container, symbol messages and symbol beans[14]. Relevant data for a given context is enclosed in beans. Beans are stored and manipulated in containers. These containers also provide functionality for creating, updating and deleting of beans. Messages are used for communication between the containers. The messages are transmitted by using the DER (Distinguished Encoding Rules) coding schema. This coding schema is defined in ASN.1 (Abstract Syntax Notation One) [Dub00].

The source of real world information is sensor data. The first processing step is done at the so called *microsymbol factory* (MSF) [Pra06]; it creates microsymbols based on sensordata. In this work only these microsymbols are used. So a MSF is implemented at the sender’s site of the communication, in our case, this is the Octobus.

Messages

Since only microsymbols are used, the Octobus needs only two types of messages:

- *New Symbol Message (NSM)*: When an NSM is received by a container, a new symbol is generated. All relevant data is inside the NSM. Each occurence of a modification of a sensor value results in sending an NSM with the sensor’s ID, the new value and the timestamp when the modification occured.
- *Heartbeat Message (HBM)*: This is designated for time synchronization of the communication partners. HBMs are sent every 100 milliseconds. They do not contain any additional data.

[14]For better readability the prefix ‘symbol’ is omitted at this paragraph.

Application Programming Interface

SymbolNet is implemented in Java. This excerpt of the Application Programming Interface (API) gives an overview to the most important functions for using the SymbolNet-package.

`public TcpSender(InetAddress address, int port) throws IOException`
: Establishes a connection to the given `address` and `port` and starts sending messages. Program flow remains in this constructor until connecting is successful or an `IOException` occurs.

`public TcpReceiver (InetSocketAddress address, boolean waitForPeer)`
`throws IOException`
: Listen at the specified port (which is part of the `InetSocketAddress`) for incoming connections. If `waitForPeer` is *true*, program flow remains in this constructor until a peer has connected successfully.

`public MessagePoller TcpReceiver.getPoller()`
: To get the received messages a `MessagePoller` is required.

`Message MessagePoller.PollMessage()`
: Returns a message if there is one, `null` otherwise.

`Message MessagePoller.PollMessageBlocking() throwsInterruptedException`
: Returns a message; waits until a message is available (or exception occured).

3 Hidden Markov Models

A Hidden Markov Model is a statistical model which can be described by two random processes. One process is assumed to be a Markov process which is responsible for transitions between states. These states are not directly visible. The other random process, which is influenced by the current state of the model, generates output symbols. Only these output symbols can be observed. However, since it is known that states influence the probability distribution of symbols, the sequence of output symbols allows conclusions to the sequence of states.

In comparison to the room which should be observed, there are some similarities: From the point of view of a computer system, there exists a kind of inner states of the room, i.e. the people inside the room. These people are not directly visible, only the triggering of sensors can be observed, but we know that this is influenced by persons. So it is possible to draw conclusions to the inner state from the observed sensor emissions. As we have a model of the inner state, it can be calculated what will happen next in the model. This abstract prediction is transformed to a prediction in the real world. Thus HMMs seem to be a promising approach to achieve the goal of predicting the behavior of persons in buildings. For this reason short introduction to HMMs is given here.

3.1 Definitions

Before introducing the theory of HMMs, some basic principles are explained first. Thus in this section some required definitions are depicted. These definitions are described in the following.

Stochastic Process

A stochastic process is a mathematical description of a system which generates temporally ordered random variables. A formal definition is given in [Doo96]:

> A stochastic process is a family of random variables $\{x(t, \bullet), t \in \mathcal{J}\}$ from some probability space $(S, \mathbb{S}, P)$ into a state space $(S', \mathbb{S}')$. Here, $\mathcal{J}$ is the index set of the process.

Markov Process

As is defined at [Wei99]:

> A random process whose future probabilities are determined by its most recent values. A stochastic process $x(t)$ is called Markov if for every n and $t_1 < t_2 \ldots < t_n$, we have
>
> $$P(x(t_n) \leq x_n | x(t_{n-1}), \ldots, x(t_1)) = P(x(t_n) \leq x_n | x(t_{n-1})).$$
>
> This is equivalent to
>
> $$P(x(t_n) \leq x_n | x(t) \forall t \leq t_{n-1}) = P(x(t_n) \leq x_n | x(t_{n-1})).$$

Markov Chain

A Markov Chain is a Markov Process with a discrete state space. Markov Chains can be discrete or continuous, depending on their parameter space. A sequence of random variates X_t

$$\begin{aligned} & P(X_t = j | X_0 = i_0, X_1 = i_1, \ldots, X_{t-1} = i_{t-1}) = \\ = \; & P(X_t = j | X_{t-n} = i_{t-n}, X_{t-n+1} = i_{t-n+1}, \ldots, X_{t-1} = i_{t-1}) \text{ with } 0 < n \leq t \end{aligned}$$

is called a Markov Chain of n^{th} order. If the random variates X_n take discrete values it is a discrete Markov Chain. Most common are discrete Markov Chains of first order, sometimes with Markov Chain this special case is associated. Assume random variates X_n that take discrete values $a_1, \ldots, a_N$ then a discrete Markov Chain of first order is a sequence of

$$P(x_n = a_{i_n} | x_{n-1} = a_{i_{n-1}}, \ldots, x_1 = a_{i_1}) = P(x_n = a_{i_n} | x_{n-1} = a_{i_{n-1}})$$

For short this is called a Markov Chain[1] [Pap84].

3.2 Markov Model

A Markov Model (MM) is a Markov Chain with a finite state space. For this reason an MM can be described by a directed graph, extended by an output alphabet. This is very similar to a finite state automaton (FSA) with the difference that the transitions in an MM are represented by probabilities. So it is a kind of probabilistic automaton. A probabilistic automaton is a generalization of a non-deterministic finite automaton (NFA), because of the transition probabilities, of course. Each state produces some *output symbol* which is an element of the output symbol alphabet Σ. According to the definition of Hidden Markov Models in [RJ86], a MM can be described by a quadruple $\lambda = (S, A, \pi, \Sigma)$:

$S = \{S_1, \ldots, S_n\}$	set of states
$A = \{a_{ij}\}$	transition probability matrix
π	initial state distribution vector
Σ	output alphabet

[1]This definition is used within this work.

As known from graph theory [BK96] a transition is a pair of states[2]. In a directed graph, this is an ordered pair. The set of such pairs can be represented by a $(n \times n)$-matrix, called adjacency matrix A, where the transition pair (S_i, S_j) is represented by an entry at $row(i)$ and $column(j)$ of the matrix. Because we have transition probabilities these entries take values between zero and one, i.e. $(a_{ij} \in [0,1], a_{ij} \in \mathbb{R})$.

We don't want to focus on trivial models, so we assume that all sets are not empty and that the norm of each row- and column vector of the transition matrix A is non-zero i.e. no state is isolated.

Due to the modeling by probability distributions, there also is no well defined start state as known from FSAs. Instead, here a distribution is given, denoting for each state the probability for being the start state.

An output alphabet is a finite set of output symbols.

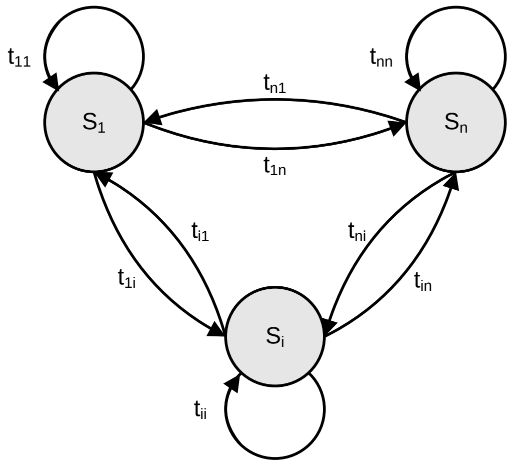


Figure 3.1: A Markov Model with n States.

Assume a synchronized process. With each step a transition is taken and every state which is visited emits a symbol. So the MM generates a random sequence of symbols. Figure 3.1 shows an MM with n states S and $(n \times n)$ transitions t.

In some cases such a model is not sufficient, especially if the states are not known. If there is no idea of the driving force behind a process, an MM can't be built. For such problems a Hidden Markov Model (HMM, see Section 3.3) could be an appropriate answer.

[2]States are corresponding to vertices and transitions are corresponding to edges.

3.3 Hidden Markov Model

A Hidden Markov Model (HMM) is a Markov Model (MM) extended by an emission probability distribution over the output symbols for each state. An HMM can be defined [RJ86] by a quintuple $\lambda = (S, A, B, \pi, \Sigma)$ where

$S = \{S_1, \ldots, S_n\}$	set of states
$A = \{a_{ij}\}$	transition probability matrix
$B = \{b_1 \ldots, b_n\}$	set of emission probability distributions
π	initial state distribution vector
Σ	output alphabet

The terms S, A, π, Σ are equivalent to the MM (see Section 3.2). B is a set of probability distributions. Due to the fact that Σ is an alphabet and therefore a finite set, B is a probability mass function (pmf). However, there are HMMs conceivable, with an infinite set of output symbols. In that case B would be a probability density function (pdf).

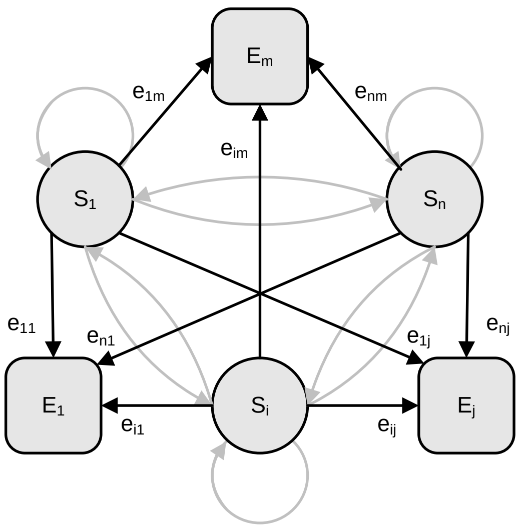


Figure 3.2: A Hidden Markov Model with n States and m Emission Symbols.

Figure 3.2 shows an HMM with n states (S) and m emission symbols (E). Emission probabilities are denoted by e_{ij}. For better view, the transitions are grayed out. In such a model the emission probabilities can be represented by a $(n \times m)$ matrix B with values $(b_{ij} \in [0, 1], b_{ij} \in \mathbb{R})$.

This model is called *Hidden* Markov Model because the states are not visible. An observer can only see a sequence of output symbols. This sequence allows conclusions to the sequence of states. The recognition of a specific output symbol at a current instant by an observer is called an observation.

The three problems for HMM's

As written in [RJ86] there are three key problems for HMMs:

Given the observation sequence $O = o_1, \ldots, o_T$ and the model $\lambda = (S, A, B, \pi, \Sigma)$ we can elaborate

1. Evaluation: How to compute $Pr(O|\lambda)$, the probability of the observation sequence.
2. Decoding: How to choose a sequence of states $I = i_1, \ldots, i_T$ which is optimal in some meaningful sense.
3. Learning: How to adjust the model parameters (A, B, π) to maximize $Pr(O|\lambda)$.

The evaluation problem is about how to compute the probability that the observed sequence was produced by the model. In other words, how to "score" or evaluate the model concerning the given observation sequence. This is of interest if we have several competing models. An efficient method to solve this problem is the forward algorithm which is described in Section 3.3.1.

The decoding problem is an attempt to uncover the hidden part of the model. The goal is to calculate the sequence of states which fits best to the given sequence of observation. This is a typical estimation problem. It can be solved by the Viterbi algorithm, which is described in Section 3.3.2.

The learning problem is about how to optimize the model parameters in a way that an observation sequence is described best. In this case the observation sequence is called a training sequence. The learning problem can be split in two subproblems: One is *structure learning*, which is about appointing states and (initial) connections (i.e. transitions) between them. The other one is *parameter estimation*, which means adjusting the transition and emission probability distributions. The second part of the learning problem can be done by the Baum-Welch algorithm, which is described in Section 3.3.3. For the fist part of learning (*structure learning*) there is no general solution available. Finding an initial model is very close to the real world phenomena which should be described by the HMM.

3.3.1 Forward Algorithm

Consider a scenario where we have one sequence of observations O (of length T) and several different models λ. It is obvious that someone would like to know, which model is most appropritate generating this sequence. To make them comparable we can calculate the probability $Pr(O|\lambda)$. The most straightforward way doing this is through enumerating every possible state sequence I of length T, calculate the probability of the observation sequence O for this specific I, calculate the probability of I itself, build the joint probability of them and summing up over all possible state sequences. This algorithm is unfeasable even for small models (e.g. for a model with $N = 5$ states and a sequence of length $T = 100$ the calculation would need $2 \cdot 100 \cdot 5^{100} \approx 10^{72}$ computations [RJ86].

Forward Probability

However, this problem can be solved efficiently[3] by the forward algorithm. First of all we define a forward variable or *forward probability* $\alpha_t(i)$ as

$$\alpha_t(i) = Pr(O_1, O_2, \ldots, O_t, i_t = q_i|\lambda)$$

This is the probability of a partial observation sequence with length t, for state q_i, at time t, given the model λ. $\alpha_t(i)$ can be calculated inductively by

1. $\alpha_1(i) = \pi_i b_i(O_1) \qquad 1 \le i \le N$
2. for $t = 1, \ldots, T-1; \quad 1 \le j \le N$
   $$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(j) a_{ij}\right] b_j(O_{t+1})$$
3. $Pr(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$

At step 1 the forward probability (for each state q_i) is initialized with the joint probability of state q_i and the initial observation O_1. At step 2 the inductive calculation is shown: Each state q_j can be reached from several other states q_i if the transition probability a_{ij} is not 0. This transition probability together with the forward probabilty of the source state is summed over all incoming transitions. This result together with the probability, making the observation O_{t+1}, is the forward probability for this state (q_j) at time $t+1$.

Finally the sum of the terminal forward variables $\alpha_T(i)$ gives the designated value $Pr(O|\lambda)$. With this algorithm the computation requires on the order of N^2T calculations (e.g. the example from above with $N = 5$ states and $T = 100$ observations will need about 3000 computations [RJ86].

3.3.2 Viterbi Algorithm

To solve the decoding problem the Viterbi algorithm can be used. The goal is to find the most probable sequence of states (which are hidden, of course) for an observation sequence O in an HMM λ. This is quite similar to the evaluation problem and as known from the forward algorithm, the simple straightforward approach is not feasable in terms of computation time. To make this problem solvable we define a Viterbi path probability (PVP) $\delta_t(i)$, which is the probability reaching a particular state along the most likely path. The difference to the forward probability $\alpha_t(i)$ is that $\alpha_t(i)$ sums up all path probabilities for the given state whereas $\delta_t(i)$ is the probability of the most likely path (to the state i, for an observation sequence of length t).

The goal of the Viterbi algorithm is to find the best path through a model (the *Viterbi path*) rather than calculating the probability of it, however, the latter is obviously an essential part. Therefore the predecessor state $\Psi_t(i)$ of the current maximum has to be stored at each step. Calculating the Viterbi path requires the following four steps:

[3]in terms of computational complexity theory

1. $\delta_1(i) = \pi_i b_i(O_1) \qquad 1 \leq i \leq N$

   $\Psi_1(i) = 0$

2. for $2 \leq t \leq T \qquad 1 \leq j \leq N$

   $\delta_t(j) = \max_{1 \leq i \leq N} [\delta_{t-1}(i) a_{ij}] b_j(O_t)$

   $\Psi_t(j) = \arg\max_{1 \leq i \leq N} [\delta_{t-1}(i) a_{ij}]$

3. $P^* = \max_{1 \leq i \leq N} [\delta_T(i)]$

   $i_T^* = \arg\max_{1 \leq i \leq N} [\delta_T(i)]$

4. for $t = T-1, T-2, \ldots, 1$

   $i_t^* = \Psi_{t+1}(i_{t+1}^*)$

The initialization step (step 1) is the same as in the forward algorithm. Predecessors are undefined, of course.

In step 2 of the algorithm for each state j of the model the PVP is calculated: Multiply the predecessors PVP and the probability of transition (predecessor $\rightarrow$ current state j) for each possible predecessor and take their maximum. Together with the probability of the current observation O_t for this state, this gives the current PVP. Do this for each observation of the given observation sequence. As mentioned above, we have to remember the predecessors, this can be done by taking the value of index i of the maximum function for which the maximum of the expression is returned. Since the emission probabilities are all the same within the maximum function they are not necessary to be computed[4].

Step 3 shows the calculation of the PVP $P^*(O|\lambda)$ of the model λ with a given observation sequence O and its correspondig state i_T^*. P^* is the maximum of the PVPs from all states at $t = T$, i.e. the whole observation sequence. The corresponding index i denotes the final state of the Viterbi path (as in step 2).

To get the whole Viterbi path it is necessary to step back the stored states, as shown in step 4.

3.3.3 Baum-Welch Algorithm

The Baum-Welch algorithm is an attempt to solve problem three of HMMs which is probably, the most difficult one. This is the learning problem, i.e. a method to adjust the model's (λ) parameters to maximize the probability of a given sequence of observations O. There is no general way to do this. All that can be done is to locally optimize $P(O|\lambda)$. For this task some more variables are necessary. One is the *backward probability* $\beta_t(i)$.

[4]When implementing in practice it is likely to do the two calculations of step 2 at once.

Backward Probability

This is the probability of a partial observation sequence from $t+1$ to T and it is quite similar to the forward probability $\alpha_t(i)$, but starting from the end. It is defined as

$$\beta_t(i) = Pr(O_{t+1}, O_{t+2}, \ldots, O_T | i_t = q_i, \lambda)$$

and can be calculated by using the following induction:

$$\begin{array}{ll} 1.\ \ \beta_T(i) = 1 & 1 \le i \le N \\ 2.\ \ \text{for}\ \ t = T-1, \ldots, 1 & 1 \le i \le N \end{array}$$

$$\beta_t(i) = b_{ik} \sum_{j=1}^{N} a_{ij} \beta_{t+1}(j)$$

Together with the forward probability $\alpha_t(i)$ from above we can define the probability of being in state q_i at step t (with O and λ) as *Current State Probability.*

Current State Probability

$$\begin{aligned} \gamma_t(i) &= Pr(i_t = q_i | O, \lambda) \\ &= \frac{\alpha_t(i)\beta_t(i)}{Pr(O|\lambda)} \end{aligned}$$

$\sum_{i=1}^{N} \gamma_t(i) = 1$ due to the normalization factor $Pr(O|\lambda)$. So it is a conditional probability.

Current Transition Probability

This is the probability of being in state q_i and making the transition $(q_i \to q_j)$ at the next step, with the given observation sequence O and the model λ. It is defined as

$$\xi_t(i,j) = Pr(i_t = q_i, i_{t+1} = q_j | O, \lambda)$$

Together with the current state probability we can write this as

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{Pr(O|\lambda)}$$

On the other hand, the current state probability $\gamma_t(i)$ is the probability of being in state q_i at t; so we can relate $\gamma_t(i)$ to $\xi_t(i,j)$ by

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j)$$

If we sum $\gamma_t(i)$ over the observation sequence index t, we get the number of times the state q_i is visited. If the last index T of the sequence is excluded, we get the expected number of outgoing transitions from q_i. Analogical we can use $\xi_t(i,j)$ to get the expected number of transitions from state q_i to state q_j.

$$\begin{aligned}
\sum_{t=1}^{T-1} \gamma_t(i) &= \text{expected number of outgoing transitions from state } q_i \\
\sum_{t=1}^{T-1} \xi_t(i,j) &= \text{expected number of transitions from state } q_i \text{ to state } q_j
\end{aligned}$$

With these definitions the Baum-Welch method can be described. This method changes the models's parameters (π, A, B) in a way that $P(O|\lambda)$ rises or does no change in case of a local optimum.

Baum-Welch Reestimation Formulas

The reestemated model's parameters $(\bar{\pi}_i, \bar{a}_{ij}, \bar{b}_{jk})$ are

$$\begin{aligned}
\bar{\pi}_i = &\quad \text{probability of being in state } q_i \text{ at } t = 1 \\
\bar{\pi}_i = &\quad \gamma_1(i), \qquad\qquad 1 \leq i \leq N
\end{aligned}$$

$$\begin{aligned}
\bar{a}_{ij} &= \frac{\text{expected number of transitions from } q_i \text{ to } q_j}{\text{expected number of outgoing transitions from } q_i} \\
\bar{a}_{ij} &= \sum_{t=1}^{T-1} \xi_t(i,j) \bigg/ \sum_{t=1}^{T-1} \gamma_t(i)
\end{aligned}$$

$$\begin{aligned}
\bar{b}_j(k) &= \frac{\text{expected number of times observing symbol } k \text{ in state } q_j}{\text{expected number of times being in state } q_j} \\
\bar{b}_j(k) &= \sum_{t \in T'} \gamma_t(j) \bigg/ \sum_{t=1}^{T} \gamma_t(j) \qquad\qquad T' = \{t' : O_{t'} = k\}
\end{aligned}$$

4 System Design

The previous chapter has introduced the theory of Hidden Markov Models (HMMs). This chapter represents a definition of a system design for using HMMs to fulfill the tasks described at the problem statement, see Section 1.2. The implementation of this design is described in Chapter 6.

Section 4.1 describes an abstraction of sensor data, so that this data can be used within the HMM algorithms. For this task the information from the sensors is encapsulated into emission symbols. In Section 4.2 a model of persons is introduced. To achieve our goals, sensor data has to be assigned to persons, the behavior of this persons is modeled with the HMMs. The reasons for introducing this model and its description is given at this section. In Section 4.3 an overview is given, how HMMs are modeled within this project. The section describes how the HMMs are used to make predictions, the necessity of having an own HMM for each person, the visualization of HMM's parameters and the data structure which was used at this work to represent HMMs. The learning of the structure of HMMs is described in Section 4.4. At this work HMM structure learning is done by creation of a very specific model from sensor data. This model is generalized by merging of similar states. Different approaches for defining which states should be merged are described in this section.

4.1 Sensor Data

As known from the environment description in Chapter 2, there are three different types of sensors: Switches, movement detection sensors and tactile floor sensors. These types differ in the kind of information the system retrieves from a triggered sensor. In case of a switch it was the opening (or closing) of a door. The specific door is identified by the sensor's identity (ID). For example, the sensor with the ID "4801" indicates if the fridge is open or closed. The mapping from sensor-ID to it's semantics is stored at the Sensor Database as a verbal description. Movement detection sensors indicate if persons are present in their sphere of action. Due to the size of the covered area of a movement detection sensor this information cannot be used for exact localization of persons. Movement sensors rather give some additional redundancy, however, they are a good example for showing the integration of different sensor types into one model. At the system the semantics of this type of sensors is given by their ID. This is the same like the switch sensors. For this reason these two types are grouped together at the implementation part, called *BinaryID*, because the values given by the sensor are binary and semantic is given by the sensor's ID. The third kind of sensors which are installed, are the tactile floor sensors. Such a sensor triggers if a person steps on it (or an object with a sufficient mass). These sensors can be used to localize persons, however, things aren't that simple, as can be seen in Section 4.2. In comparison to movement detection sensors the tactile floor sensors give very exact information about the location, because the point of happening is where the sensor is mounted. Thus, semantics of a tactile floor sensor is given by its position, which is stored in the Sensor Database as well. With one indirection the position of the sensor can be determined from its ID. So the model uses positions instead of tactile sensor IDs. Another advantage of using positions is that in case of having more tactile sensors activated, an average position can be calculated, this is important for the Person Model, described in Section 4.2.

To use these different types of information in one model, the semantics of the sensors and positions respectively, are encapsulated in symbols. As known from Chapter 3 one part of HMMs are the emissions. An emission has a probability per state and a corresponding value, i.e. *what* will be emitted. For flexibility, this value is encapsulated in a class, called *IEmissionSymbol*, and instances of this class can contain data of the three different sensor types. For the algorithms of the HMM this is transparent. Figure 4.1 shows the interface between the Markov part and the Data part.

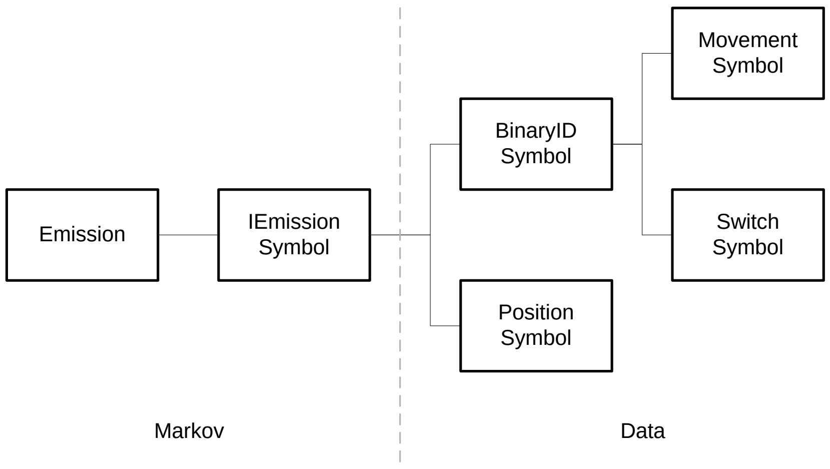


Figure 4.1: Interface Markov - Data

At the Markov side we have Emissions and Emissions contain IEmissionSymbols. This is an abstract form of any conceivable symbol, concrete data is not of interest. At the data side the functionality of an IEmissionSymbol has to be provided; important is a comparison operator. By means of semantics there are two kinds of symbols, Position Symbols and BinaryID Symbols, whereas the last ones are either Movement Symbols or Switch Symbols.

Another benefit of this symbolization is that the HMM package can deal with any kind of symbol as long as comparison operators are provided. This allows reusability of the HMM implementation.

4.2 Person Model

This project deals with the observation of persons in buildings. More precisely, this work is restricted to one room. Not the state of the building is of interest, but the behavior of the persons staying in the room, including arrival and leaving. As this is a technical project the interaction of a person with the technical environment is of interest. To give some demonstrative examples: The opening and closing of closet doors, activation and deactivation of kitchenware or just how a person moves around; more precisely, the positions of the person in the room at certain instants. External influences of natural phenomenons like temperature or humidity are not of interest.

4.2.1 Motivation

As the focus is on the behavior of persons with the goal of making predictions therefore, it is necessary to know *how many* persons are in the room and *which* person does *what*. From the view of a computer system, while observing the room the system has less information than a human who is located in the room and is doing the same. However, there is very exact information about the room itself, even more precise than the human observer. Sometimes the definition of what is part of the room is not quite clear, think about tables and chairs. Furthermore, there is exact information about the perception system, quasi the sense organs of the computer system. These are the sensors which are installed in the room. The type of sensor, their sphere of action, the delivered data and their positions are well known.

However, the driving force which triggers a sensor is mostly unknown. The perception of a human depends on the memory [ST02]. In some cases this is the same with computer systems, illustrated with the following example: Consider a tactile floor sensor like described in Section 2.1.2. The intended behavior would be: If a person steps over the sensor, this sensor will deliver a value *true*. At a higher level of data processing, the system 'knows' that at the position of this sensor a person is located. This is based on a priori stored knowledge, which can be seen as a kind of memory. But an essential question is, if this assumption holds in practice. There are a lot of possibilities which could cause the system to get a value from this sensor: For example, there could be an error at the sensor or an error at the transmission of data. For this work such errors are not taken into account, they are neglected. However, not only errors of the perception hardware can cause receiving an unintended value: An item or a pet can also trigger a tactile sensor. So we assume, that there are no pets at the institute's kitchen, but of course there are items: The table and some chairs. However, we can make a further assumption: An item does not move around by itself, this can only be done by a person. So if the first derivation of the sensor values is taken, only person-caused events remain.

Nevertheless this is not sufficient. With the assumptions above it is known that a person is the driving force for the changes of sensor values. But consider a scenario like shown in Figure 4.2. A corridor with two switches (l and r), one at each end of it, and two persons activating the switches in an alternating manner. When observing this corridor the sequence $(l, r, l, r, l, r, \ldots)$ is recognized. In fact, this is the behavior of both persons *together*. Each person separately causes a sequence $(l, l, l, \ldots)$ or $(r, r, r, \ldots)$ respectively. Since we want to model the behavior of *a person* rather than the behavior of a crowd, it is important to know which person is responsible for triggering a sensor, except the special case if we know that there is only one person present. In practice even the count of persons in the room is unknown. For this reason it is a necessity to have a person model and not to use sensor data directly for learning of HMMs, i.e. the system tries to assign the observed sensor values to already recognized persons.

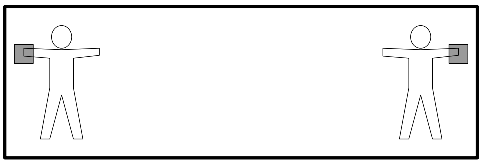


Figure 4.2: Corridor with two Switches and Persons

Assumption Coverage

As we use statistical methods there is no problem with taking those assumptions as long as their violations are rare enough. In fact the frequency of occurrence of such assumption violating events is estimated.

Privacy Statement

In terms of privacy it should be mentioned that the term *person* in this work means an abstract description of a human being. There are no personal properties collected or processed. For this reason the system has no discrimination criterions of human beings, therefore it is completely impossible to draw conclusions from an instance of the system's person model to a concrete human person in real life.

4.2.2 Recognition

The key questions are, how the system can recognize if there is a person present at the observed room, and, if it is only one person or more. With the assumptions made above, it is assumed that sensors are only triggered by persons. Obviously it seems to be a good idea to count the persons entering the room and make a person tracking. This was the first attempt.

A First Attempt

As can be seen at the SmaKi's layout (see Figure 2.3), there is only one possibility where a person can enter the room: The door. When observing the scenario where a person enters the room, it can be seen that the order of emitting sensors is approximately the same every time. However, there are small variations, i.e. which tactile sensor triggers first.

1. The door sensor indicates opening.
2. The movement sensor which is closest to the door indicates movement.
3. One of the tactile sensors near the door indicates pressure.

As can be seen in the following, this description does not hold, because only such a way of a persons's arrival is insufficient.

Person Distinction Limitation

Assume the following situation: Two persons going closely together enter the room, walk to the coffee machine and split there; one makes coffee, the other one goes to the bookshelf. The question is, how this situation will be recognized by the system. The start sequence would be the same as described above (First Attempt), because the persons are too close to be distinguished by the system. The system only perceives *one* person. At the point, where the persons split, the system will get confused. It 'sees' *one* person, who triggers sensors at different regions of the room.

Since our recognition system has limits it is not possible to get arbitrary exact information. As known from Chapter 2, we have three kinds of sensors: Switch sensors, which give no information about the count of persons in the room. There is for example no relation between an open door and the count of people in the room, even if we examine the door sensors's history. At the SmaKi there is a standard door, so it is possible that more than one person enter or leave the room once the door is open. Even worse are the movement sensors. As known from the sensor description in Section 2.1.3 they only trigger in case of human movements. However, it is not known if the change in the IR spectrum, which is observed by these sensors, is caused by one person or even more. So we only have the tactile sensors to distinct persons and in addition to get the count of persons which are currently present in the room. A tactile sensor has a size of 600 x 175 mm which is its sphere of action simultaneously. So we can assume that if exact one tactile sensor is activated, this is from exact one person. But if more than one tactile sensor is activated, the question about the count of persons is not easy to answer. Two adjacent sensors can be triggered by one or two persons. So we need a heuristic to get an estimation of the count of present persons in the room, this is given by the described model.

4.2.3 Design of the Model

For the reasons mentioned above a person model was developed. This is a rather simple model since this is only a subtask of this project. However, the estimations about the persons in the room which are taken from this model are sufficient to accomplish our task. The main focus is on the tactile sensors because this type gives the most accurate information. This is based in the number of these sensors and the sphere of action of each one.

The basic idea is to assign sensor values to persons. Therefore we have to manage two lists: For the room there is a list of the present persons. Each person manages a list of tactile sensors which are assigned to it. The policy how persons are created and sensor values are assigned to them is shown in Figure 4.3. If the System receives a value *true*[1] of a tactile sensor it searches the room's person list. If this list is empty, a new person will be created, the sensor which emitted the value is added to this person's sensor list and the person is added to the room's person list. The position of the newly created person is the center position of the tactile sensor. If the list is not empty, i.e. there is at least one person in the room, the system searches the person of which the position is closest to the center of the sensor which emitted the value. If the distance to the closest person is not within a predefined threshold, a new person will be created, the procedure is the same as if the room was empty. In the other case, i.e. there is already a person which is close enough to the emitting sensor, this sensor will be added to the person's sensor list and the person's position is recalculated; the new position is the average over all center positions of the assigned tactile sensors.

[1]This indicates the presence of a person.

As a consequence of this algorithm it is possible that persons can appear (and disappear) at any position in the room. So the system is able to deal with the problems described in paragraph *Person Distinction Limitation*.

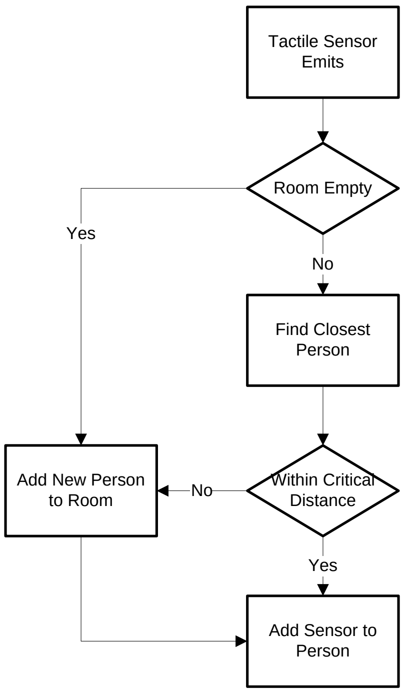


Figure 4.3: Assign Tactile Sensor to Person

The other sensors are treated in a similar manner. Switch sensors are assigned to the closest person. Movement sensors have a sphere of action. They are assigned to *all* persons within this area. The big difference to the tactile sensors is that deactivation is not evaluated. Doing so could be an improvement, but this would make it necessary to distinguish between the switch sensors, i.e. each switch sensor would need it's own implementation since their semantics are different. When comparing a closet door switch and the coffee machine vibration switch, we see the following difference: The 'deactivation' of a door, i.e. the closing of it, needs a person as driving force. The deactivation of the coffee machine, i.e. the coffee is ready, doesn't need a person, this happens automatically. Movement sensors are deactivated automatically as well. So for simplification only the activations are evaluated.

Sensor Value Removal

Now we know how the system creates persons and assigns tactile sensor values to them. The next important question is how these values are removed. When a person releases a tactile sensor the system receives a value *false* from that sensor. The system iterates the room's person list and the *false* emitting sensor is removed from every person's sensor list. Actually there should only be one person to which this sensor was assigned to, however, to be sure all persons are tested; uninvolved persons are skipped. The person whose sensor is removed from the list, will recalculate the position as average of the remaining sensors. In the case the person's sensor list is empty after removal, the position cannot be recalculated. A person with an empty sensor list is called an *orphan*.

The Role of Orphans

An orphan (in the context of this person model) is a person with an empty tactile sensor list, i.e. no sensor is assigned to this person. As a consequence there is no sensor indicating some pressure, i.e. at this current instant it seems that there is nothing present. When observing some activity in the room it can be seen that tactile sensors sometimes are bouncing. However, a debouncing is already implemented at the hardware part [Goe06] but this is still not sufficient for this person model. Another example (than bouncing) is given when a person walks quite fast through the room; the posticous sensor will release *before* the ahead sensor will trigger. To kill two birds with one stone, orphans are allowed. After removal of the last assigned sensor, the person is not removed, but a timer is started. The orphan's time is running, but otherwise it is handled like a normal person. If within a certain deadline a sensor is assigned to it, the orphan becomes a normal person again. If not, it will be removed.

Entrance Workaround

As can be seen at the SmaKi's layout (Figure 2.3) there are no tactile sensors nearby the door. As a consequence the system can recognize persons first, when they reach the area which is covered by tactile sensors. A similar problem is given when a person leaves the room, but this problem is already solved by allowing orphans. However, we don't want to lose the door switch value of the first person who enters a room. Remember, this is the start of a sequence. For this reason the system makes a check if there are already persons present in the room when a door switch value is received. In the case it can be assigned to a person (must be close enough) this is done. Otherwise it will remember this value. If a person is created within a few seconds (and this person is located nearby the door), the remembered value is assigned to that person. Due to the fact that movement sensors have a high frequency in emitting values (in comparison to the other sensors) they do not need a particular attention.

Person Deactivation

At the beginning of this section it was mentioned that the system cannot distinguish persons from objects when using values of tactile sensors. For this reason the first derivation of them is used. The whole application triggers on changing of sensor values. This system can be improved as follows: Consider a scenario where a person, carrying a chair, enters the room. The person walks until the center of the room, puts down the chair and leaves the room. At the instant when the distance between chair and person exceeds the specified threshold, the system recognizes two persons. There is no chance to identify the chair as an object. However, we can assume that no person will stay at exact one position for a very long time. For this reason each person has a timer which is reseted with every change of the person. If there is no changing for a predefined period of time, the system assumes that the force activating the sensor(s) is no person but rather an object. For the prediction algorithms the distinction between persons and objects is not necessary, however, a human observer of the visualization may get confused if there are shown predicted paths for the objects in the kitchen.

4.3 Modeling HMMs

This section describes how the theory of HMMs (described in Chapter 3) is used for this project. In Section 4.2 a Person Model was introduced to combine sensor values and assign them to persons. The person's behavior is modeled with an HMM. First of all we have to build such a model. This is based on the person's positions. How the models are built is described in Section 4.4. Once we have a model we can solve the *Evaluation Problem* (see Section 3.3) to make predictions. How this can be done under the given conditions is described in Section 4.3.1. For a look behind the scenes, the HMM can be visualized, this is described in Section 4.3.3. For efficient and fast calculation there was a special data structure implemented which is described in Section 4.3.4.

4.3.1 Prediction

If we calculate the forward probability $\alpha_t(i)$ for a partial observation sequence of length t as described in Section 3.3.1, we get for each state I the probability of being in that state at time t. To get a prediction about the next observed symbol, the joint probabilities of forward probability $\alpha_t(i)$, transition probability a_{ij} and emission probability at next state b_{js} have to be summed up. So for each symbol s a next step probability $\sigma_{t+1}(s)$ can be calculated as:

$$\sigma_{t+1}(s) = \sum \alpha_t(i) a_{ij} b_{js} \quad \forall i, j, s$$

The most probable symbol s^*_{t+1} at the next step, i.e. the *prediction* is

$$s^*_{t+1} = \arg\max_{\forall s}[\sigma_{t+1}(s)]$$

Example

Let us illustrate this algorithm by an example shown in Figure 4.4. For not losing track of things, we assume a lot of probabilities to be zero as it is in practice. The ellipses are states, the arrows are transitions (i.e. transitions with a probability greater than zero) and the digits in the states are emission symbols (with the probability to be emitted greater than zero); so we have no explicit labels of the states and we name them by the symbols they can emit, i.e. the most left state is called “240” because it can emit the symbols “2”, “4” and “0”. Furthermore, we assume the same probabilities for the shown objects, i.e. each transition from state “240” has a probability of 1/3 (because there are three transitions), the same with symbols, respectively. The bold states indicate a forward probability greater than zero at the current time t.

To keep things simple the shown example is following the software implementation of the algorithm, which is quite faster than a greedy implementation of the formal description above. This can be done due to the used data structure described in Section 4.3.4. As we see, the feasible current states are $\{(245), (250)\}$. Following the possible transitions, we get the feasible states at the next step $t + 1$, which are $\{(246), (241), (250)\}$; thus all possible next symbols are $\{(0), (1), (2), (4), (5), (6)\}$. The probability for a symbol (for example the symbol (2)) is calculated as:

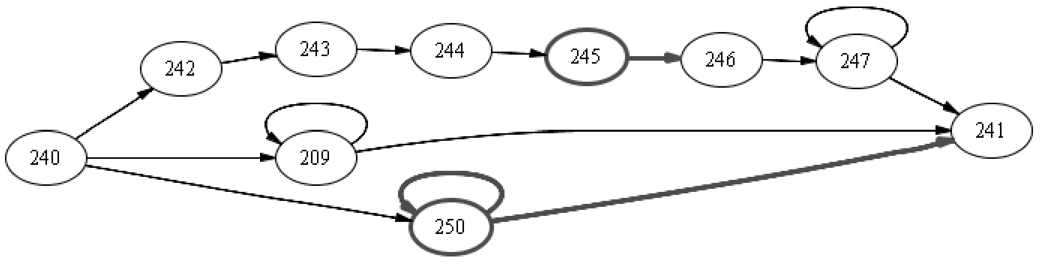


Figure 4.4: Prediction Example

$$
\begin{aligned}
\sigma_{t+1}(s) = &\ \sum \alpha_t(i) a_{ij} b_{js} \quad \forall i, j, s \\
\sigma_{t+1}(2) = &\ \alpha_t(245) a_{(245)(246)} b_{(246)(2)} + \alpha_t(250) a_{(250)(241)} b_{(241)(2)} + \alpha_t(250) a_{(250)(250)} b_{(250)(2)} \\
= &\ (1/2)(1)(1/3) + (1/2)(1/2)(1/3) + (1/2)(1/2)(1/3) \\
= &\ 1/3
\end{aligned}
$$

If this is done for each symbol (their probabilities will sum up to 1), we will get the corresponding values. For the symbols $\{(0), (1), (2), (4), (5), (6)\}$ we get $\sigma_{t+1} = \{1/12, 1/12, 1/3, 1/4, 1/12, 1/6\}$, thus the prediction would be the symbol (2):

$$s^*_{t+1} = \arg\max_{\forall s}[\sigma_{t+1}(s)] = (2)$$

4.3.2 Global and Local

Again, consider the example above. We assumed that feasable current states are $\{(245), (250)\}$, which are calculated by t iterations of the forward algorithm. Since we want to make a prediction at each step, it is reasonable to reuse the values from the previous iteration. These values are stored at the model. For this reason the model becomes person-related, i.e. the current forward probabilities belong to a specific person. To be able to deal with several persons simultanously, as this is required, each person gets its own HMM, i.e. a clone of the original one which can be modified in any conceivable way. So the person's clone is used to calculate the prediction. However, a person needs a second HMM for learning, see Section 4.4. This second model initionally is empty, it grows with the values a person receives. When a person disappears, the learned model can be merged with the global model. Figure 4.5 shows an overview of this structure. The class *Base* manages the (unique) global HMM. It provides two methods for cloning and merging. Each instance of class *Person* contains two HMMs, one for learning and one for predicting. So if there are n persons in the SmaKi, the system has to deal with $(2n + 1)$ HMMs.

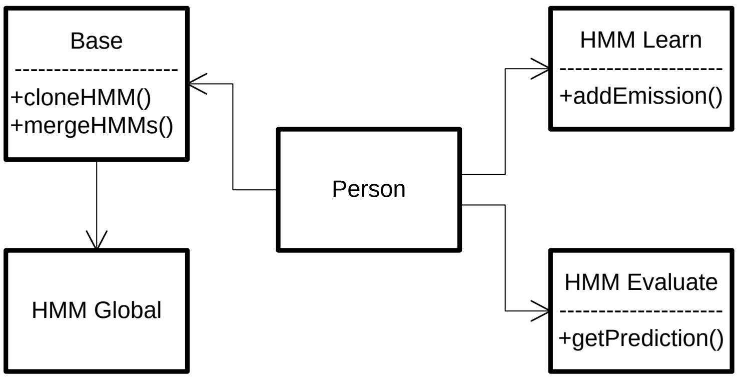


Figure 4.5: Global and Local HMMs of a Person

4.3.3 Visualization of Parameters

In figure 4.4 we assume some parameters of an HMM to make the calculation of predictions comprehensible, however, in practice we don't get such nice values and the symbols aren't numbers but rather symbols of a specific type, containing some data. Figure 4.6 shows how these parameters are visualized at the application. This is a detail view of one state of an HMM. Each row inside the state represents an emission, the string describes the type of the associated symbol, the first number (always '1' at this example) the number of occurences of this emission and the second number is the probability of this emission (always '0.2' at this example). For emissions containing position-symbols, the position represeted as string is shown, for example '(917.0,1542.0)' at the first line. The probability of transitions is shown as lable of the edge, '0.75' at this example.

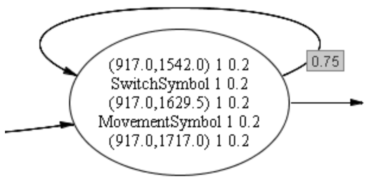


Figure 4.6: Visualization of HMM Parameters

4.3.4 Data Structure

HMMs can be represented by two matrices, one with the transition probabilities, the other one with the emission probabilities. A model with n states and m different emissions needs an $n \times n$ matrix for the transitions and an $n \times m$ matrix for the emissions. Depending on the application, these matrices may be sparce, i. e. lots of probabilities are zero. At this project this is expected. Furthermore, there is the problem that the count of emission symbols is not known a priori, since for example, position can take arbitrary values (at this level of abstraction[2]). If matrix

[2]At the Person Model the countable position values from the tactile sensors are averaged.

representation is used, adding of new values is expensive since this would need a new column for each value. All entries of this column would be zero except one. The implementation of this matrix representations with arrays would be a waste of resources (memory and time). For this reason a data structure as presented here was implemented for representing HMMs.

The used datastructure for representation of HMMs is based on linked lists [Sed98]. The model consists of a list of states and (for a basic description) each state contains two lists, one for the transitions and one for the emissions, see Figure 4.7. Each emission contains a symbol, i.e. the symbol which is emitted. However, here a kind of double linked list was used, so each state holds a third list with incoming transitions (*Backtransitions*). With this representation inserting and deleting of data (states as well as emission) is quite fast. Zero-probabilities are not stored; this saves memory and speeds up some algorithms. Table 4.1 shows a comparison of array representation and this list representation. The asymptotic time complexity of some actions is shown in big O notation [Knu97] for an HMM with n states, m emissions, k transitions to a particular state and l occurences of a particular emission, with $(k, l \leq n)$. The inserting at arrays is faster than removing because insertion can always be done at the end of the data structure. Note that the given values are related to this project and the concrete implementation of the operations which are used.

Table 4.1: Asymptotic Time Complexity Comparison

	Array	List
Inserting a state	$O(n)$	$O(1)$
Deleting a state	$O(n^2)$	$O(k)$
Inserting a new emission	$O(n)$	$O(1)$
Deleting an emission	$O(nm)$	$O(l)$

Deleting of states is a task which will be done very often at the learning phase, because at this project learning of HMMs is done by state merging (see Section 4.4). So this data structure will speed up state merging in comparison to array representation.

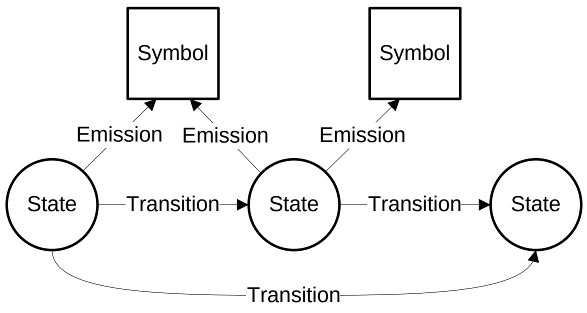


Figure 4.7: Data Structure for HMM

Implementation

A symbol contains some data about the concrete symbol which is not of interest here. A state contains three lists:

- A list of referencing States
- A list of Transitions
- A list of Emissions

A Transition is a triple $(SourceState, DestinationState, Probability)$.
An Emission is a tuple $(Symbol, Probability)$.
The 'source' of an Emission, i.e. the corresponding state, is given by its membership.

4.4 HMM Structure Learning

In Chapter 3 an overview of the theory of Hidden Markov Models is given. As mentioned, there are three key problems for HMMs, which are *Evaluation*, *Decoding* and *Learning*. Probable solutions are given by correspondig algorithms, which are explained as well. However, the introduced method for learning is only for estimating a fixed number of model parameters. This standard approach is called Baum-Welch algorithm, it uses dynamic programming to approximate a maximum likelihood (ML) or maximum a posteriori probability (MAP) estimate of the HMM parameters [BPSW70]. So this algorithm supposes a given model structure, but the problem how to find such a structure is not solved.

One approach is to choose the HMM topology by hand, however, this would be a long winded process and the optimization process would be very costly, since the Baum-Welch algorithm has to be applyed to many different models, because at the beginning there is absolutely no idea how the structure of such a model could be. In [SO93] the authors describe a technique for learning structures of HMMs from examples, i.e. to define the number of states and the connectivity (the non-zero transitions and emissions). The indroduced induction process starts with the most specific model consistent with the training data and generalizes by successively merging states. The basic ideas of this approach are used in this project. So we start building HMMs from scratch by using training data.

4.4.1 State Merging Principles

Two similar or even identical states can be merged to one new state to reduce the model's size. The decision when states should be merged is not part of this section, this is explained in the following section. Consider two states, I_1 and I_2, which should be merged, i.e. replaced by a new state I. So the model's probability distributions have to be replaced. The distributions for transitions and emissions of the new state I are a weighted average of I_1 and I_2. Transition probabilities into I are the summed up probabilities into I_1 and I_2. The weights for calculating the mixture distributions are the relative frequencies of visitations of the states I_1 and I_2. This is illustrated in the following example.

Example

Figur 4.8 shows the merging of two states, I_1 and I_2 to a new state I. As can be seen, the transitions *into* I_1, I_2 have to be redirected to I. Transitions and emissions *from* I_1, I_2 are from I after merging.

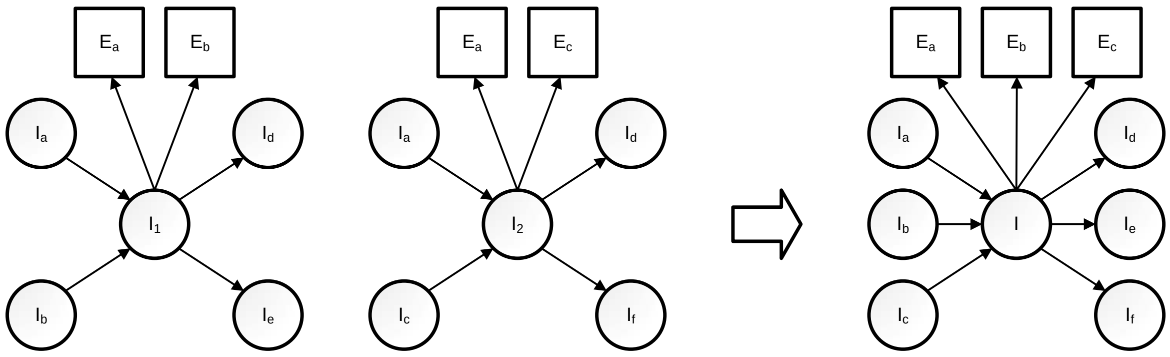


Figure 4.8: Merging two States

Using the notation from Chapter 3 (transition matrix A, emission matrix B) the transition probabilities *into* new state I are calculated as:

$$a_{I_x I} = a_{I_x I_1} + a_{I_x I_2} \quad \forall x$$

To get the values for emissions and transitions *from* new state I, we need the number of times the original states were visited, denoted as $c(I)$:

$$a_{II_x} = \frac{c(I_1)a_{I_1 I_x} + c(I_2)a_{I_2 I_x}}{c(I_1) + c(I_2)} \quad \forall x$$

$$b_{IE_x} = \frac{c(I_1)b_{I_1 E_x} + c(I_2)b_{I_2 E_x}}{c(I_1) + c(I_2)} \quad \forall x$$

Assuming $c(I_1) = 1$, $c(I_2) = 2$ and an intuitive[3] probability for all transitions and emissions at the left hand side of the example shown in Figure 4.8, the values of the merged model are calculated as shown in Table 4.2.

There are two special cases to be considered: Selftransitions ($a_{I_x I_x}$) and transitions between the two states which are merged, i.e. $a_{I_1 I_2}$ or $a_{I_2 I_1}$. For this reason the self transition probability of the new state after merging has to be recalculated as

$$a_{II} = \frac{c(I_1)a_{I_1 I_1} + c(I_2)a_{I_2 I_2} + c(I_1)a_{I_1 I_2} + c(I_2)a_{I_2 I_1}}{c(I_1) + c(I_2)}$$

The states I_1, I_2 and all probabilities concerning them are removed from the model after merging.

[3] p = 1.0 / count of outgoing edges; note: I_a has *two* outgoing transitions

Table 4.2: Calculation of Values

	Formular	Values	Result
a_{I_aI}	$a_{I_aI_1} + a_{I_aI_2}$	$0.5 + 0.5$	1.0
a_{I_bI}	$a_{I_bI_1} + a_{I_bI_2}$	$1.0 + 0.0$	1.0
a_{I_cI}	$a_{I_cI_1} + a_{I_cI_2}$	$0.0 + 1.0$	1.0
a_{II_d}	$\frac{c(I_1)a_{I_1I_d}+c(I_2)aI_2I_d}{c(I_1)+c(I_2)}$	$\frac{(1)(0.5)+(2)(0.5)}{(1)+(2)}$	1/2
a_{II_e}	$\frac{c(I_1)a_{I_1I_e}+c(I_2)aI_2I_e}{c(I_1)+c(I_2)}$	$\frac{(1)(0.5)+(2)(0.0)}{(1)+(2)}$	1/6
a_{II_f}	$\frac{c(I_1)a_{I_1I_f}+c(I_2)aI_2I_f}{c(I_1)+c(I_2)}$	$\frac{(1)(0.0)+(2)(0.5)}{(1)+(2)}$	1/3
b_{IE_a}	$\frac{c(I_1)b_{I_1E_a}+c(I_2)b_{I_2E_a}}{c(I_1)+c(I_2)}$	$\frac{(1)(0.5)+(2)(0.5)}{(1)+(2)}$	1/2
b_{IE_b}	$\frac{c(I_1)b_{I_1E_b}+c(I_2)b_{I_2E_b}}{c(I_1)+c(I_2)}$	$\frac{(1)(0.5)+(2)(0.0)}{(1)+(2)}$	1/6
b_{IE_c}	$\frac{c(I_1)b_{I_1E_c}+c(I_2)b_{I_2E_c}}{c(I_1)+c(I_2)}$	$\frac{(1)(0.0)+(2)(0.5)}{(1)+(2)}$	1/3

As we see in part two and three of Table 4.2, outgoing transitions and emissions probabilities of a state sum up to (1.0). To achieve highest possible accuracy, at the implementation not the number of times of visiting a state, but the number of 'using' a transition is stored at the transition, the same for emissions, respectively. As state merging is a basic concept of learning at this project, it is done very often. When using the standard formular above, there would be a lot of floating point additions and multiplications, since probabilities are represented by floating point variables. However, these operations are *not* associative[4] [DR06], so this could cause numerical errors. To avoid this, counters in the transitions and emissions are used, which are in integer arithmetic.

4.4.2 Learning from Sensor Data

The higher the number of states in a model, the more specific the underlying system is depicted. Whereas models with a lower number of states represent a generalization of the underlying process. The resulting inexactness is intended, because this raises the chance that a later perceived scenario can be represented by the model. However, with a too generic model it is not possible to elaborate differences between the scenarios. For this reason one of the key challenges is to find the balance between a very specific model with the risk of not finding a representation for a particular scenario on the one hand and a very generic model, which produces results for every situation but little significance, on the other hand.

In this project learning is implemented only by state merging, like described in Section 4.4.1. This form of learning HMMs from sensor data was introduced by [Bru07]. As this project deals with sensor data and HMMs as well, the basic principles are considerated. However, for future research it might be reasonable to additionally use other forms of learning, like the Baum-Welch algorithm, once the HMM's structure is fixed.

[4] $(a + b) + c = a + (b + c)$

As mentioned in Section 4.2, a person model is introduced to assign sensor values retrieved by the SmaKi's hardware to particular persons. This person models are used for the HMMs, so we don't use sensor values directly, i.e. there is some kind of pre-processing before. So we model the behavior of a particular person in order to make predictions out of that model. Exacting, this is the model of occurrences of sensor values which are assigned to a person. The way from recognition of a sensor value to the HMMs of a particular person is shown in Figure 4.9.

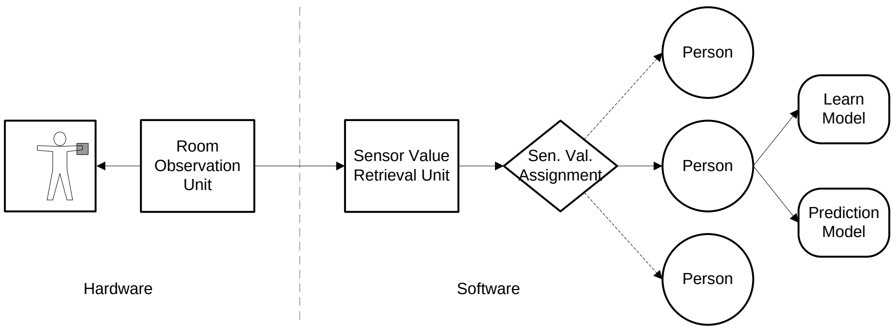


Figure 4.9: Assigning of Sensor Values to Person Models

In the following it is explained how an HMM is generated from the sensor values, received by the system. Furthermore, it is described how this model is generalized by state merging. To get designated results, it is necessary to do the follwing steps in the correct order.

1. Create a Chain
2. Preprocessing of Chains
3. Merge Horizontally
4. Merge Vertically
5. Merge Sequences

This steps are described in the following subsections.

4.4.3 Create a Chain

As known from 4.3.2, each person has two HMMs, one for continuously calculating the prediction algorithm and the other one for learning. The latter is empty at the beginning. In this section it is explained how learning works within this project. This includes the building of a single model for one person and the consolidation of several such models to one unique global model.

When a person is created, its learning HMM is empty. However, an 'empty' HMM is initialized by two artificial states, a start state and an end state. These artificial states never have any emissions. For each sensor value which is assigned to a person a new state is created which has exact one emission. The symbol of this emission contains a representation of the sensor value. A new transition is added to the predecessor state with the actually created state as destination. The

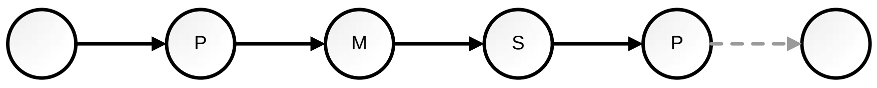


Figure 4.10: An Initial Chain of States

predecessor of the very first created state is the artificial start state. When a person disappears, a transition is added to the last created state with the artificial end state as destination. Thus, we get a chain like shown in Figure 4.10. The letters inside the states show the type of symbol the emission contains ('P' for *Position Symbol*, 'M' for *Movement Symbol*, 'S' for *Switch Symbol*). Such a chain fulfills the following conditions:

- Each state (except the artificial end) has exact one transition with the probability of 1.0.
- Each state (except the artificial start and end) has exact one emission with the probability of 1.0.

Each emission contains exact one emission symbol; there are three kinds of such symbols: Switch, Movement and Position. The first two are identified by the sensor's ID which emitted the value, Position symbols are identified by their value, i.e. the contained position. For short we call an emission which contains a Position symbol a *PositionEmission*, the same for Switch and Movement symbols, respectively.

Using this algorithm we get a chain for each person. Such a chain represents the exact episode of activity of the person, as far as this is detectable and correctly assigned to the person by the system. The most simple approach would be to consider a model as an accumulation of many parallel arranged chains. This would be the most specific model conceivable. Already the smallest deviation in data would cause a newly perceived scenario that it doesn't fit the model, and so a prediction would not be possible or gives unfeasable results. The goal is to find similarities between the chains and to combine equal patterns. So we get a smaller model which is a gerneralization of the original.

4.4.4 Preprocessing of Chains

The first consideration is about some preprocessing of chains before combining chains of several persons. As the factor time has a very weak representation within the used HMMs, the first goal is to avoid misinterpretations concerning temporal order. If we receive a value from a tactile sensor and a value from a movement sensor in series, we only know that this sensor values are perceived by the system in this order. However, at the models's level, we do not know, *how long* the duration between the occurence of emitting of the sensors is. That ranges from simultaneuously (and therefore a random order, due to transmission and processing is serial) to infinity, i.e. the person disappears. Taking into accout the whole system (described in Chapter 2) it can be seen that in order to the weak time modeling the order of triggering of different sensor types has no significance. To use this information it is necessary to have a model of time which would result in a far more complex system. This could be done by Hidden Semi Markov Models, like described in [Bru07].

This system mainly uses the tactile floor sensors, since they give the most accurate information about the location of a person. Due to the mentioned weak time model and the focus on positions calculated from the tactile sensors, we start with the following 'preprocessing' of a chain: It should be ensured that each state has a PositionEmission, i.e. each state is related to a position. It's obvious that states can have more than one emission after this step. So the states 'M' and 'S' in Figure 4.11 are merged to the state 'P', the result is a new state 'P,M,S', shown in Figure 4.12. The pseudocode of this preprocessing is shown in Algorithm 4.1.

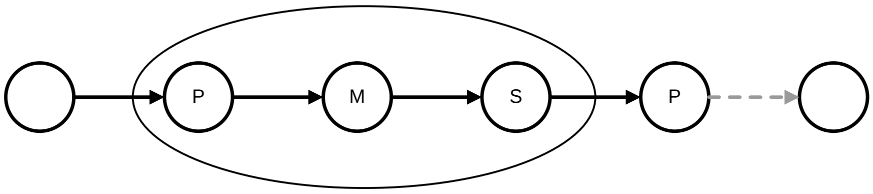


Figure 4.11: Raw Data Chain Merging

Algorithm 4.1: PreprocessChain()

```
Input: a state chain
// merge start
$posindex \leftarrow 0$
while $state[posindex].emission \neq PositionEmission$ do
    $posindex \leftarrow posindex + 1$
for $i \leftarrow 0$ to $posindex$ do
    $merge(state[i], state[posindex])$
// merge rest
for $i \leftarrow posindex + 1$ to $chain.length$ do
    if $state[i].emission = PositionEmission$ then
        $posindex \leftarrow i$
    else
        $merge(state[posindex], state[i])$
```

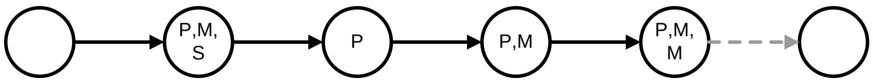


Figure 4.12: Preprocessed Chain

For this algorithm some preconditions have to be fulfilled. The input model is a chain, i.e. states have at most one incoming transition and at most one outgoing transition. All states, except the start and the end, have exactly one emission. At least one state has a PositionEmission.

After this preprocessing is done, we get a chain where each state (except start and end) has exactly one *PositionEmission* (and any number of other emissions). An example of a result is shown in Figure 4.12. The first[5] state ('P,M,S') is the reslut of merging the first states from the chain shown in Figure 4.11. The other states are examples.

Concerning the application domain, the information of movement and switch sensors is dependent on the persons's position. The movement sensors give a redundant information anyway, however, the switch sensors can only be triggered from a specific position of the person. For this reason only the activations of sensors are modeled, since only this is related to the persons's position. Consider the coffee machine vibration switch: The machine is started by a person, the sensor changes its value from *false* to *true*. When the coffee is ready, the machine stops vibrating and the sensor changes its value back to *false*. However, this cannot be assigend to a person, since this event happens independently. It also cannot be assigned to the person which started the machine, since there is no relation to the actual position of the person when the coffee is ready, yet the person may have left the room.

4.4.5 Merge Horizontally

A person standing in the SmaKi can trigger several tactile floor sensors. If a person's foot is placed over two sensors either the one or the other or both can be triggered, like shown in Figure 4.13. This depends on the person's shifting of weight, so this can cause bouncing. Although this is filtered by the introduced Person Model (see Section 4.2), we get a new position for each retrieved sensor value. At application level such differentiation of positions are useless. Moreover we want to model different positions in some meaningful way, so we don't need exact values. An accuracy which allows us to distinguish if a switch can be reached from a person at a position would be enough. For this reason the HMM is generalized by merging states with similar positions.

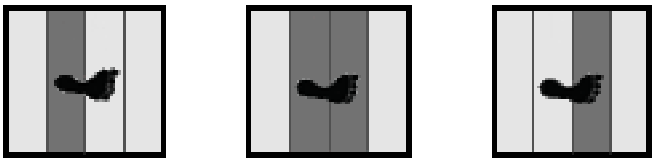


Figure 4.13: Persons's Foot Placement

Positions are defined as similar if the distance between them is within a certain distance *d*. As every state contains exactly one PositionEmission after the preprocessing, we can compare each pair of consecutiv states like shown in Figure 4.14 and merge them if the distance between their associated positions is within *d*.

When states are merged, where each state has a PositioEmission, the resulting state has *two* PositionEmissions. This has to be taken into account when the resulting state is compared again. The overall position of a state which has more than one PositionEmission is calculated as a weighted[6] average of its associated positions. So we apply Algorithm 4.2 on chains like shown in Figure 4.14 and get chains with combined positions like shown in Figure 4.15.

[5]first state with data, not the artificial start
[6]The emissions are probabilities.

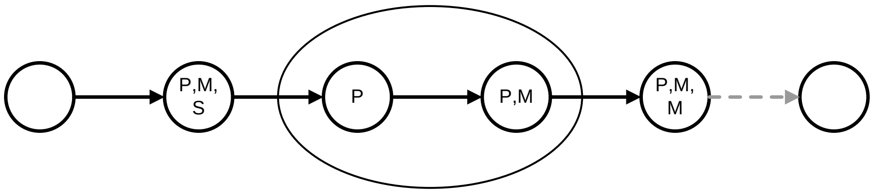


Figure 4.14: Find Consecutive States within Small Distance

Algorithm 4.2: MergeHorizontally()

```
Input: a preprocessed chain, mergedistance
$statesmerged \leftarrow true$
while $statesmerged = true$ do
    $statesmerged \leftarrow false$
    // skip the artificial start state:  start at 1
    for $i \leftarrow 1$ to $chain.length - 1$ do
        if $distance(state[i], state[i+1]) \leq mergedistance$ then
            $merge(state[i], state[i+1])$
            $statesmerged \leftarrow true$
```

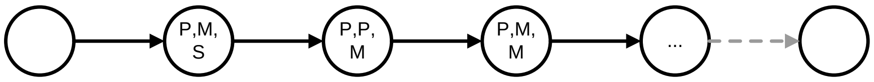


Figure 4.15: Horizontally Merged Chain

As the resulting chain (see Figure 4.15) shows, states can contain more than one PositionEmission. Such states are compared by the *distance function* which is already used in Algorithm 4.2. This function calculates a weighted average of the state's PositionEmissions.

Distance Function

As mentioned above, positions are treated as similar if they are within a certain distance d and the position of a state with more than one PositionEmission is a weighted average of all PositionEmissions of the state. In our algorithm the merging is done several times. After each merging of two states, the new position of the resulting state is calculated as a weighted average of *all* merged states.

If we would take this weighted average positions of the states for the decision whether to merge them or not, in worst case, each new state's position is exactly d away from the last calculated average $\bar{d}$. If we have only one state, $\bar{d} = 0$. The first added state can 'move' this to $d/2 = (0{+}d)/2$. The third state (the second added) influences it with a weight of $1/3$. So $\bar{d_3} = \bar{d_2} + d/3$ and so forth. As can be seen, in worst case the value $\bar{d}$ is:

$$\bar{d} = \sum_{i=2}^{n} \frac{d}{i}$$

When d is normalized, this is the *Harmonic Series*, which diverges for $n \to \infty$. In this application this means that $\bar{d}$ is unbounded. So a state can be of arbitrary size. To avoid this, the *distance function* calculates the maximum distance of any two positions within a state.

4.4.6 Merge Vertically

To construct a general model of the values learned from each person, the gained chains are joined together. As mentioned in the Section Person Model 4.2, persons can appear and disappear at any position in the room. For this reason the lifetime of a person might be very short, i.e. only a few sensor values are assigend to that person. With the merging of the chains (preprocessing and horizontally merging) this results in chains with only a few states. The decision, what 'a few' means, is taken by the user. We can assume that this short chains are errors of the recognition system, so they are not added to the global model. Even if this assumption doesn't hold, it would be no problem, since this very short actions are part of longer actions. So only chains with a minimal number of states are added to the global prediction model. Figure 4.16 shows such a global prediction model.

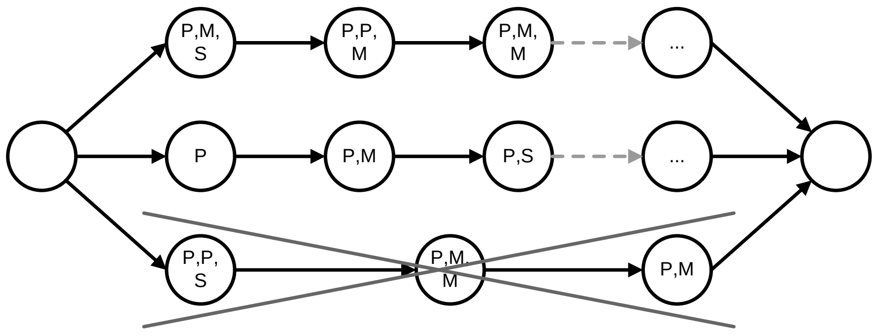


Figure 4.16: Joined Chains; Short Chains are Discarded

Once the chains are joined together we can merge the model globally. The idea is to reuse parts of chains. Consider the following example, shown in Figure 4.17. One person enters the room and walks along a path, i.e. the person produces the sequence of positions (A, B, C, D, E). Another person who walks around, produces the sequence (A, B, C, F, G). In our model this is represented by two chains, corresponding to the positions. These chains are similar until the point where one person goes to (D) and the other one to (F). So we can merge the start of the sequences in our model and split where the scenarios differ.

How the merging of similar start sequences can be calculated is shown in Algorithm 4.3. This algorithm is applied to the model whenever a new chain is added. As can be seen, this algorithm runs until no more states can be merged.

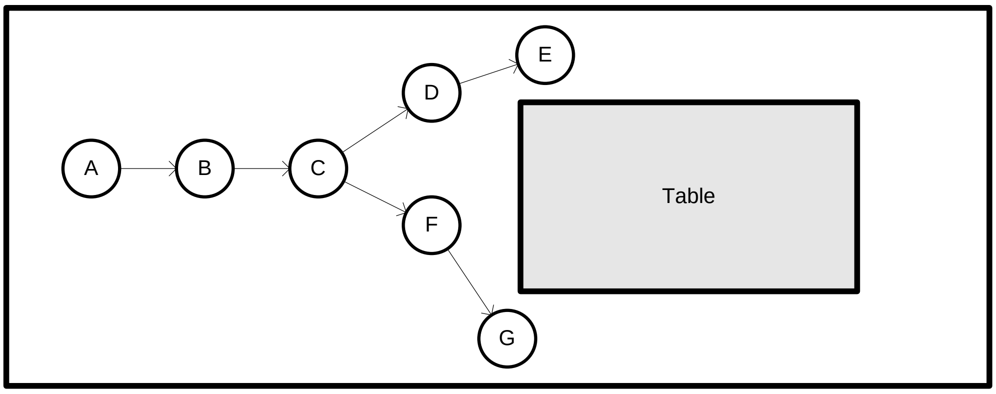


Figure 4.17: Splitting Paths of two different Persons in a Room

Algorithm 4.3: MergeVertical()

```
Input: HMM
$statesmerged \leftarrow true$
while $statesmerged = true$ do
    $statesmerged \leftarrow false$
    foreach $State\ s \in HMM$ do
        for $i \leftarrow 0$ to $s.transitions.count - 1$ do
            for $j \leftarrow i + 1$ to $s.transitions.count$ do
                $s_1 = s.transition[i].destination$
                $s_2 = s.transition[j].destination$
                // skip self-transition
                if $s_1 = s$ or $s_2 = s$ then
                    continue
                if similar($s_1$, $s_2$) then
                    merge($s_1$, $s_2$)
                    $statesmerged \leftarrow true$
```

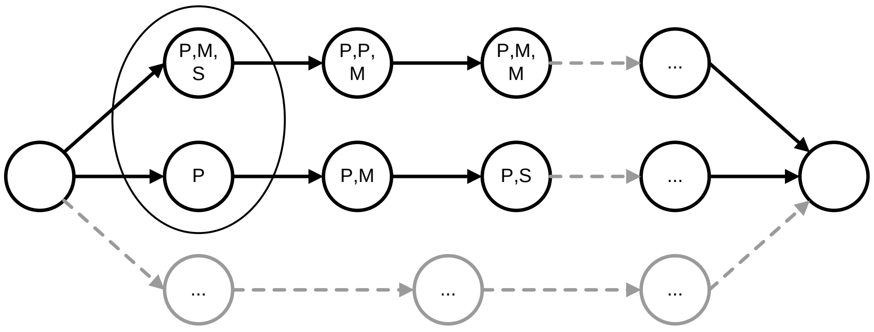


Figure 4.18: HMM before Merging Vertically

Figure 4.18 shows the model before merging. The grayed, dotted chain indicates that there are several other chains. Beginning from the artificial start state, any two successor states are compared if they are similar (within a similarity criterion, explained later). If similarity is given, the states are merged. So the arising new state has two successors, shown in Figure 4.19. Now the same is done for the newly created state. This procedure is repeated until nothing could be merged anymore. At the person-movement example above, the first three states would be merged, because the sequence (A, B, C) is the same for both persons. The resulting model of Horizontal Merging from the start is shown in Figure 4.20. The same can be done with the end of the sequences. So we use the same algorithm starting from the artificial end state.

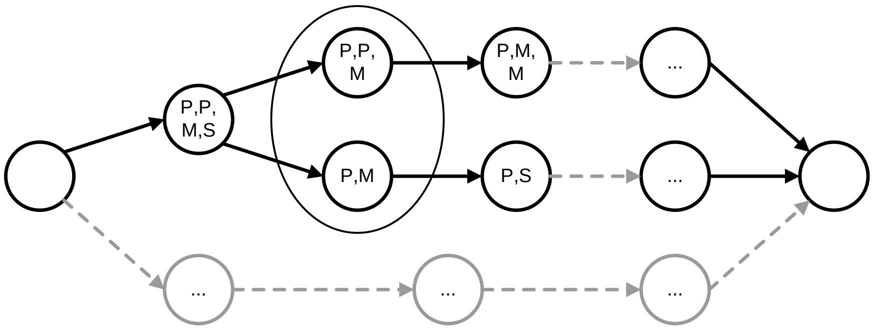


Figure 4.19: Two States Merged Vertically

When comparing Figure 4.18 to Figure 4.19 it can be seen that the states with the symbols $\{P, M, S\}$ and P are merged to a state $\{P, P, M, S\}$, i.e. all symbols are obtained. At the next step of the example, comparing Figure 4.19 and Figure 4.20, the states $\{P, P, M\}$ and $\{P, M\}$ are merged to a state $\{P, P, P, M\}$, i.e. at the resulting state is only *one* 'M'. The reason is that the same symbols are stored only once, with higher probability of course. This also can happen with *PositionSymbols* if they represent exactly the same position.

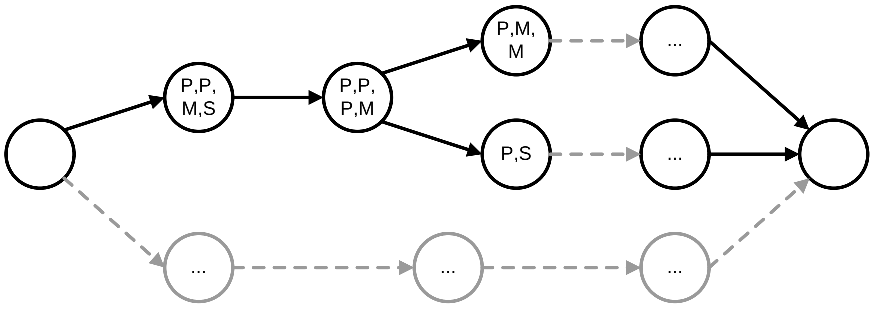


Figure 4.20: Vertically Merged HMM

Similarity Criterion

Unlike the horizontal merging at the vertical merging a *Similarity Criterion* is used to decide wether two states are merged. When merging horizontally it is sufficient to take the *PositionEmissions* of the states into account. All other emissions are merged to the predecessor state (or successor, if there is no predecessor). When merging vertically we have to distinguish whether a state contains a *SwitchEmission*[7] or not. The reason is that Horizontally Merging is a kind of information reduction *within* a scenario whereas vertical merging is finding similarities *between* scenarios. Walking to a point, activating a switch and walking back is *not* the same as just walking to the point and back.

4.4.7 Merge Sequences

As described in the previous section we get a model where similar start-sequences and end-sequences are merged together, with the limitation of comparing only two states. Another approach is to search the model for sequences of several states in the midst of the model, like shown in Figure 4.21. For better readability the more abstract notation (A, B, C) is used for the symbols, i.e. two states are similar if they contain the same letter. States with a '.' are not of interest for this example. If two similar sequences are found, the start and the end state of such a sequence is merged. The states between start and end of one chain become useless since they cannot be reached anymore. The result of merging the model from Figure 4.21 is shown in Figure 4.22. The grayed out sequence is the unused one, these states are removed from the model. However, this algorithm can cause *Backtransitions*. As can be seen in Figure 4.22 the same sequence can appear several times within one chain. If the algorithm is applied several times, all occurences of the sequence (A, B, C) are merged together. This results in a model shown in Figure 4.23. We get a transition from a state later in the scenario back to a state which occured earlier. This transition is called a *Backtransition*. However, *Backtransition* could be suppressed, but it might be of interest allowing them.

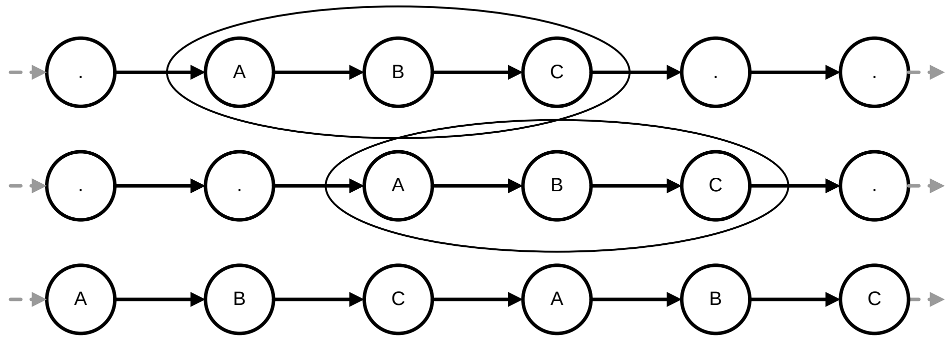


Figure 4.21: Comparing Sequences of States

Interpretation of Backtransitions

Consider the following scenario, shown in Figure 4.24. A person walks around the table several times. This results in a sequence $(A, B, C, D, E, F, G, A, B, C, \ldots)$. If *Backtransitions* are allowed, the laps are merged. This results in a sequence (A, B, C, D, E, F, G) where only the probabilities for incoming transitions of state 'A' have to be recalculated. The count of laps gets lost. If this is an intended behavior depends on the application.

[7] *MovementEmissions* can be neglected due to their probabilities at this application.

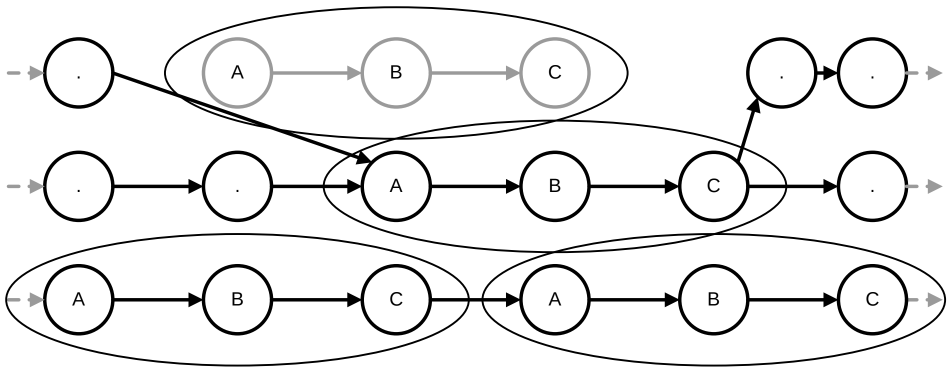


Figure 4.22: Merged Sequence of States

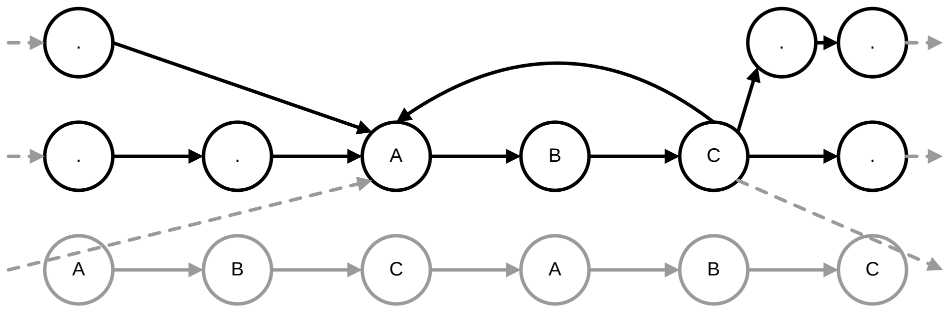


Figure 4.23: Backtransition as a Consequence of Sequence Merging

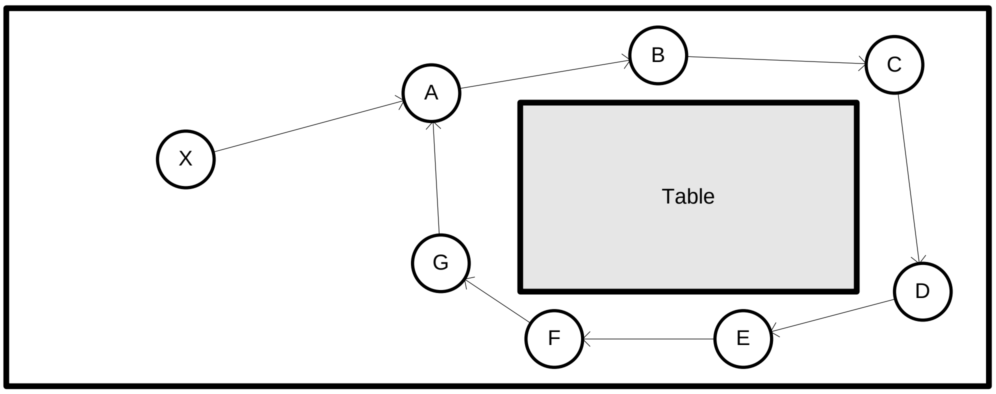


Figure 4.24: Person going around the Table

5 Software Application Design

In order to achieve the two main goals, the prediction system modeling and the visualization which are defined in Section 1.2, a software application was developed. This is the *SmaKi Prediction Application*. It has to fulfill several tasks; for this reason it was subdivided into packages correspondig to these tasks. Figure 5.1 gives an overview to the most important parts and the interactions between them. The core of the application is called *Base*. All other parts are controlled by this core element. The core element itself is controlled by the part called *Graphics*, because the *Graphics* part contains the Graphical User Interface (GUI). In Chapter 6 this part is called *GUI* since the package at the Java implementation is called *gui*[1]. Since a GUI handles user interactions it is obvious that this part can control the core of the system. The *Persistence* part is the application's interface at data level. This contains sensor values, information about sensors and sensor meta-data, i.e. information about sensor values. Since the visualization should show (among other things) the sensors and their values, the *Graphics* part can control the *Persistence* part. For reusability all source code concerning HMMs is accumulated in part *Markov*. To have a clear separation, there is a part *Data* between *Markov* and *Persistence*. The *Data* part contains symbolized data from the sensors. *Markov* controls *Data* and *Data* controls *Persistence*.

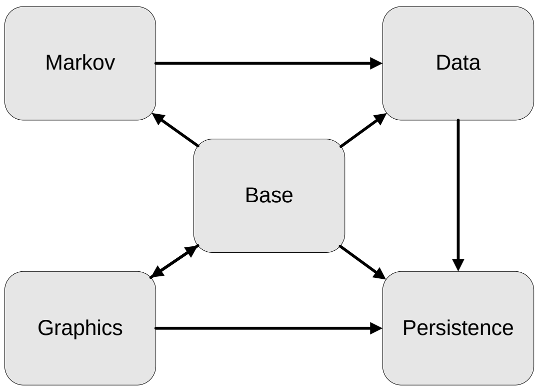


Figure 5.1: Schematic Drawing of the Application Design

[1]In Java package names start with a lower case letter by convention [LL08].

5.1 Terminology

This chapter gives a high-level description of the software application's design, the detailed description is given in Chapter 6. According to [Kai99], "*an OOD*[2] *model is a model of the proposed software system's internal construction*", in this chapter the chosen solutions are defined by describing the objects of the software system.

The Term 'Interface'

The term 'interface' is used in a double manner within this chapter. Once it is used in the computer science manner, mostly (at this work) this means the separation of classes from different packages. The other meaning is the interface of the Java programming language, i.e. an abstract class that contains only abstract methods and constants. For better distinction, the Java-Interface is written with a capital letter or extended by the prefix 'Java-' as in this sentence.

5.2 Base Part

The part *Base* is the core of the SmaKi Prediction Application. As can be seen in Figure 5.1, it interacts with all other parts in some way. For better overview this part is subdivided into *Application Core*, *World Representation* and *Sensor Representation.* This subdivision is discretionary and only for better explanation, at the implementation this is one package.

Application Core

The core of the application is the class *MainModel.* Figure 5.2 shows the most important classes which are used by *MainModel.* The *MainModel* manages the persons which are currently present in the SmaKi, i.e. persons which are recognized by the system. Real world persons are represented by instances of the class *WorldPerson*, for the task of managing those, we have a *WorldPersonList.* New persons are added to this list, persons which disappear are removed from the list. The class *HiddenMarkovModel* represents the global HMM. Whenever a new *WorldPerson* is created, it gets a clone of this global HMM for its prediction calcualtions. *HiddenMarkovModel* provides XML-serialization, i.e. the HMM can be stored in an XML-file. At startup of the application a default HMM is loaded, when the application is terminated, the changes of *HiddenMarkovModel* are saved in another XML-file to keep the original unchanged. However, it is possible to load and save HMMs from/to any XML-file manually. The class *MergeProvider* provides the algorithms for HMM structure learning, described in Section 4.4. This learning is done on the global HMM, represented by *HiddenMarkovModel.*

Class *MainController* is the interface to the graphical user interface (GUI). All instructions from the user are passed to the application via *MainController.* User interactions concerning this part are loading/saving of HMMs, adding a specific HMM which is stored in an XML-file to the system's global HMM, applying a specific merge algorithm on the global HMM, configure merging, force the *LiveDataReader* to establish a connection, disconnect the *LiveDataReader* and starting the live-data processing.

[2]Object-oriented Design

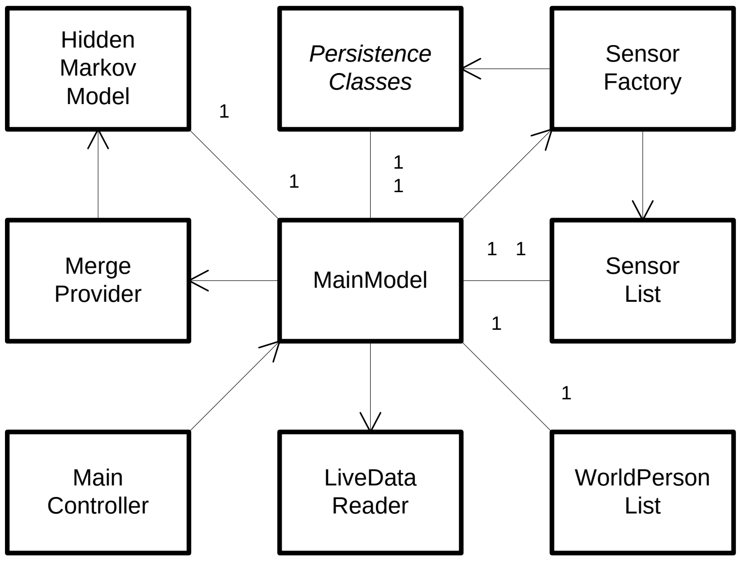


Figure 5.2: OOD of the Application's Core

The *LiveDataReader* receives sensor data from the SmaKi's perception hardware via a TCP/IP connection and passes it to the *MainModel*. For each sensor value which is retrieved by the *MainModel*, the corresponding sensor is searched from the *SensorList* and updated. Furthermore, the *MainModel* tries to assign the sensor value to a person from the *WorldPersonList*. If this is possible, the affected *WorldPerson* is updated, otherwise a new *WorldPerson* is created in case of the retrieved sensor value is from a tactile sensor. Other sensor values which cannot be assigend to a person are ignored.

The class *SensorList* represents a list of all available sensors of the SmaKi. At startup of the application this sensor list is created by the *SensorFactory*. The *SensorFactory* uses the *Persistence Classes*, which are described in detail in Section 5.6, to establish a connection to the ARS Sensor Database and to create the *Sensors* from the so called *static data* (see Section 2.2.1) of the database. Then the *Sensors* are added to the *SensorList*.

World Representation

The class *WorldPerson* is the core of the world representation. As mentioned above, the class *MainModel* tries to assign received sensor values to *WorldPersons*. In case of tactile and switch sensors, the person is chosen, which is closest to the position of the emitting sensor. Values of movement sensors are assigned to all persons within the sphere of action of the emitting sensor.

As can be seen in Figure 5.3 a *WorldPerson* contains two *HiddenMarkovModels*. One is for learning, the other one for the prediction. When a *WorldPerson* is created it gets an empty HMM for learning and a deep-copy[3] of the global HMM from the *MainModel* for the prediction. With each sensor value that is assigned to a person, a new state is added to the learning model. This state

[3]The clone() method of Java only makes a flat copy. For making a deep copy the XML-serialization functionality of *HiddenMarkovModel* and its referenced object is used.

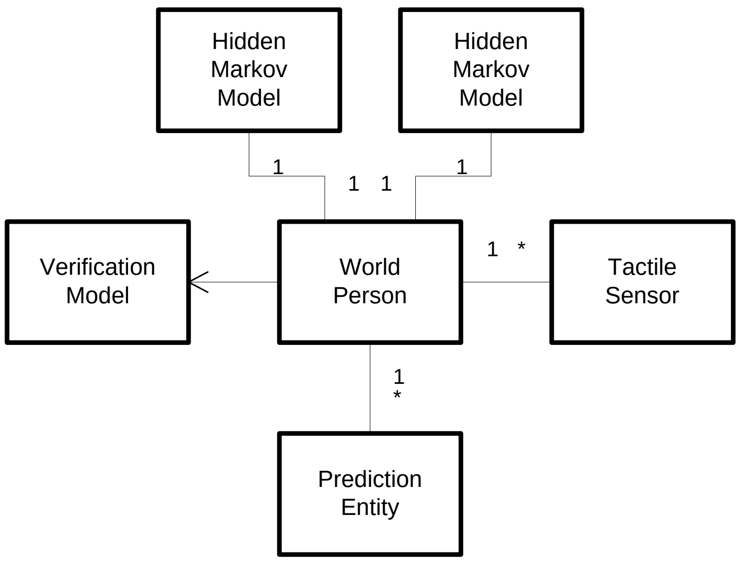


Figure 5.3: OOD of the Classes concerning *WorldPerson*

has one emission with the current sensor value as emission symbol. At the other HMM one step of the forward algorithm is calculated. The *WorldPerson* contains a list of *n* *PredictionEntities*, where *n* is a predefined number, i.e. there are always at most *n* *PredictionEntities*. The content of the *n* most probable emissions from the result of applying the forward algorithm is stored at the *n* *PredictionEntities*. The *VerificationModel* 'remembers' the prediction(s) and calculates in the next step the difference from the predicted value to the occured value. This only works for positions, since only for positions the calculation of a difference is useful. With the *Verification-Model* it is possible to compare the quality of predictions of different HMMs, i.e. of different merging strategies.

Furthermore, the class *WorldPerson* manages a list of *TactileSensors*. For each sensor value of a tactile sensor which is assigned to the *WorldPerson*, the corresponding tactile sensor is added to this list if the content of the value is *true*, or removed if the content is *false*. For each modification of the list of tactile sensors, the *WorldPerson* updates its position. This is done by calculating a weighted average of the center positions of the tactile sensors in the list. If the last sensor is removed from the list, the position cannot be updated, but the *WorldPerson* stays for a certain duration. If within this duration no new tactile sensor value is assigned to the *WorldPerson*, it is removed. If a *WorldPerson* with a non empty tactile sensor list, gets no new sensor value for a fairly long time, the person becomes an item. This is a smoth process. Once this process is completed, i.e. the object is an item with 100 %, there are no new sensor values assigned to it.

Sensor Representation

Figure 5.4 shows an overview of the modeling of the different sensor types which are installed at the SmaKi. The class *Sensor* is the base class of all sensor types, which are *MovementSensor*, *SwitchSensor* and *TactileSensor*. These classes contain the static sensor data from the ARS Sensor Database. Most important for this application is the location where a sensor is mounted in the room. This is given in a different way for each type of sensor. As sensors emit values in the real world, this is realized by the use of corresponding *SensorValues*, i.e. a *MovementSensor* uses a *SensorValueMovement* and so forth. The *SensorValueKeepAlive* is not from a real sensor, it is an artificial value, created by the *LiveDataReader* to indicate that the system is running. The corresponding 'Sensor' is *Heartbeat*, but it is not inherited from *Sensor*.

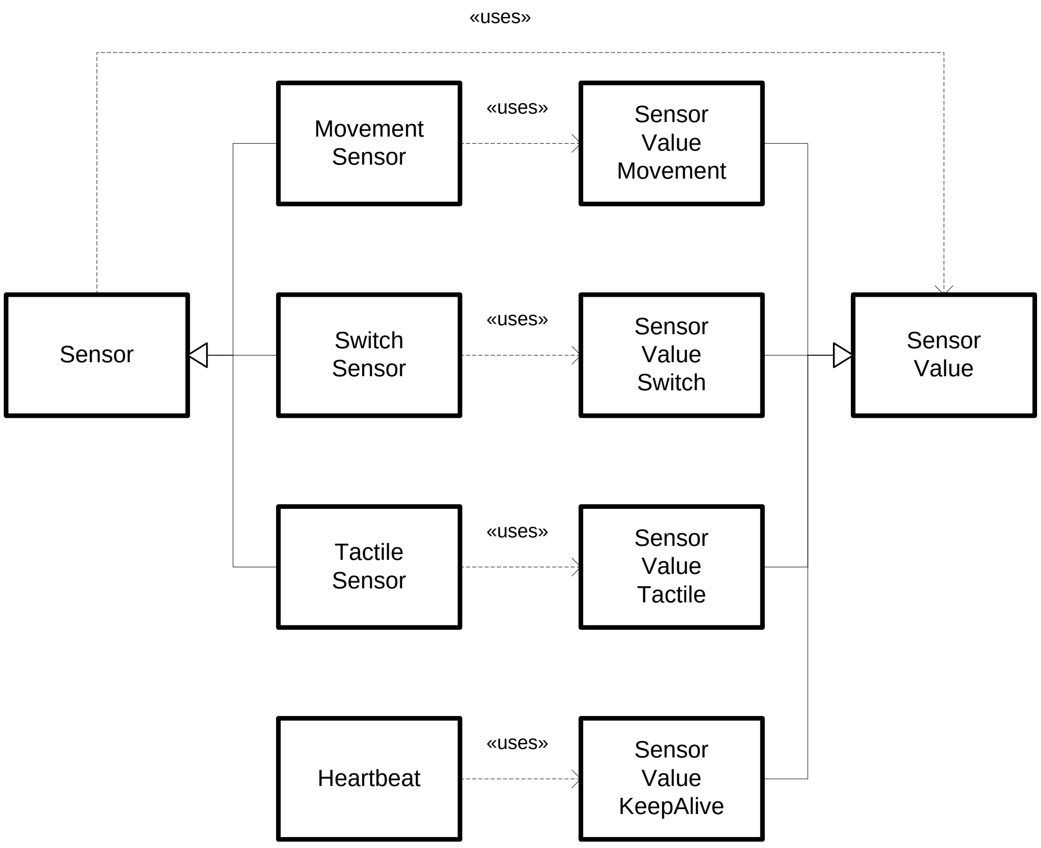


Figure 5.4: OOD of the Sensor Representation

5.3 Data Abstraction

To have a clear encapsulation of the Markov algorithms there is a part *Data* between *Markov* and *Persistence*. As the algorithms of the *Markov* part work with symbols, *Data Abstraction* provides the interface to the data which is represented by those symbols. Figure 5.5 shows an OOD of this package. This part is used to get the data from the predicted symbols, which can be positions, movement sensor activations or switch sensor activations, i.e. it is used to get the data which was filled into the HMM during the learning phase back from the model at the prediction phase.

The root class is *DataAbstractionProvider* which is the interface to the *Markov* and *Base* packages. It controlls the *DataAccessor* which is part of the *Persistence* package and therefore the interface to it. The *SensorList* contains all sensors of the SmaKi. As can be seen in Figure 5.5, there are

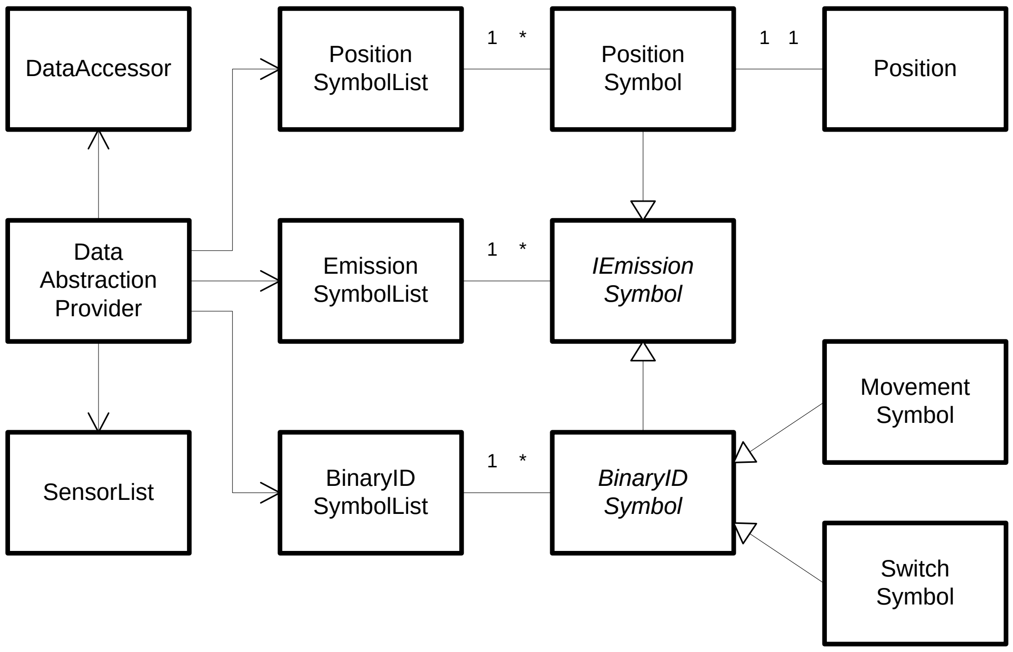


Figure 5.5: OOD of the Package Data

several lists of *EmissionSymbols*. Each emission of the HMM implementation contains exact one *IEmissionSymbol*, i.e. the symbol which is emitted. In the implementation of the Hidden Markov Model this is as abstract as in the theory of HMMs. So *IEmissionSymbol* is a Java-Interface of the Markov package. The *DataAbstractionProvider* manages a list of such symbols. As can be seen, this Java-Interface is implemented either by a *PositionSymbol* or a *BinaryIDSymbol*, whereas the last one is an abstract class. The concrete implementations of a *BinaryIDSymbol* are either *MovementSymbol* or *SwitchSymbol*. Furthermore, the *DataAbstractionProvider* has two separated lists, one for *PositionSymbol* and the other one for *BinaryIDSymbol*. These lists are used if the type of an incoming *IEmissionSymbol* is known to prohibit the `instanceof` operation. A *PositionSymbol* contains exact one *Position*.

Prediction Data Retrieval

From the prediction algorithm we get a probability distribution over all available symbols of the model. Those symbols with the highest probability are the prediction which is visualized to the user. As we have three different types of sensors and therefore three different types of symbols, it is obvious to have three different types of representations. However, we can handle *MovementSymbols* and *SwitchSymbols* in the same manner, this is the reason for indroducing the common super-class *BinaryIdSymbol*. As *PositionSymbols* contain a *Position* and the visualization of a position prediction is realized by drawing an arrow from the person's current position to the predicted position, this is simply done by using the contained *Position* directly. This is not that simple with *BinaryIDSymbol*, which contains the ID of the appropriate sensor. The prediction of triggering a movement or switch sensor is visualized by drawing a line from the person's current position to the sensor, for which triggering is predicted. To get the position of the sensor from a sensor-ID, the *SensorList* is used.

5.4 Graphics

The Graphics part is responsible for everything that should be displayed to users. It is subdivided into three parts: Building Visualization, Graph Visualization and the Main part. The Main part contains the GUI, whereas the two others are for information displaying only. For this reason the main part has control over the others. A quite useful approach for the design of graphical computer applications is the Model View Controller software design pattern, which is used in a nested way, as explained later.

Model View Controller Pattern

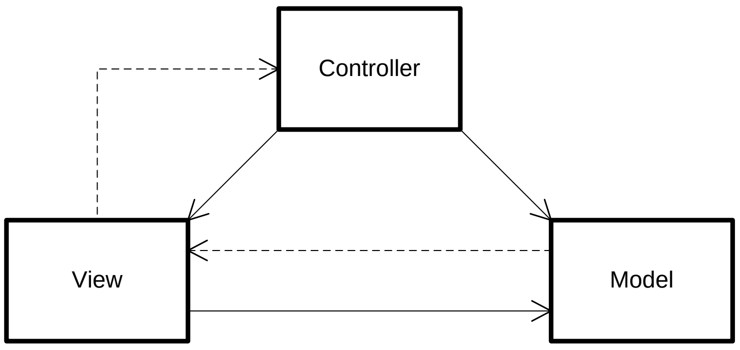


Figure 5.6: MVC Pattern

In software engineering so called *design patterns* are elaborated to reuse collective experiences. In [AIS$^+$77] Christopher Alexander says:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

The Model-View-Controller (MVC) pattern was introduced in [KP88] to build user interfaces in Smalltalk-80[4]. There are three kinds of objects within the MVC. The Model contains everything concerning the application, the View object is responsible for screen representation, and the Controller handles user inputs. MVC is a good solution for decoupling these things to increase flexibility and the chance for reusability. The design of the MVC triad is shown in Figure 5.6. User input can concern the Model or the View, the Controller processes these events and may invoke changes on the Model or cause the View to change the way how information is displayed on the screen. The View renders the Model and visualizes it to the user. If the inner state of the

[4]Smalltalk-80 is the first generally released version of the Smalltalk programming language which is object-oriented, dynamically typed and reflective. See `http://smalltalk.org`

Model is changed, the View is notified to enforce redrawing. It is possible to have multiple Views of one Model for different purposes.

Graphics Control Structure

As mentioned above, the basic design element of the Graphic package is the MVC pattern in a nested way. We have a Main part, which is responsible for the GUI and controlling the two other parts, which are the visualization of the building and the visualization of the underlying HMM. The HMM is a graph of course. For each visualization part a MVC pattern was used. Due to the given control structure, the overall structure is given as shown in Figure 5.7.

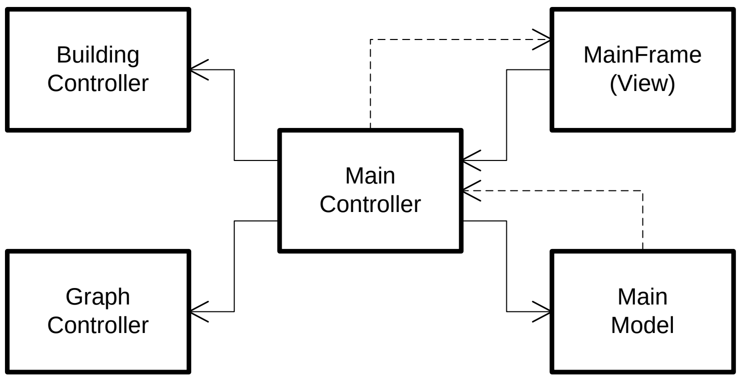


Figure 5.7: Graphics Control Structure

As can be seen, control flow concerning *MainFrame* is inverted in comparison to the original MVC pattern. This is motivated by the used framework for developing the software. The class *MainFrame* encapsulates the whole MFC structure of a GUI using Java Swing[5] [LEW+02]. The class *MainController* is an interface to the two other controllers and the *MainModel.*

The Class *MainFrame*

The class *MainFrame* is also the main entry of the whole application, it contains the (static) main method. To be platform independent it was decided to use *Lightweight Components* i.e. *Swing* elements for implementation of the GUI [Kru00]. As recommended for drawing frames with Swing, the class *MainFrame* inherits from *JFrame. MainController* is the interface to the application. A look to the application design (see Figure 5.1) shows that this is the interface to the package *Base.* Since *MainFrame* contains the user interface it also provides some dialogs which are inherited from *JDialog.* Note that class Dialog in Figure 5.8 is only a schematic drawing to show the structure of the Graphics Main part, in the implementation there are several different dialogs with extended names, like *Dialog_Suffix.*

[5]See Chapter 6 for more information about Java Swing

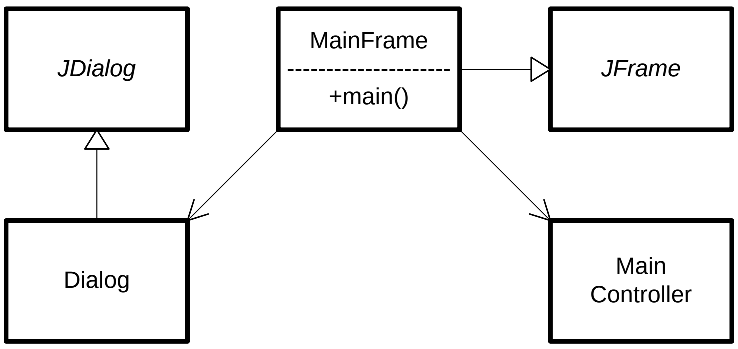


Figure 5.8: Entry Point of the Application

Building Visualization

This is the part which is responsible for visualizing the results of the prediction. For better imagination, the input data, i.e. the sensor values which are delivered to the system, are shown as well. The aim of this part is a real-time[6] live animation of the system, which shows the results of the prediction process to the user. Figure 5.9[7] shows the static structure of this part. As can be seen it is based on the MVC pattern. The real screen representation is done by *ViewPanel*. This class is inherited from the Java Swing class *JPanel* (for the screen drawing stuff) and implements the Java-interface *Runnable* for the animation stuff. The implementation of the *Runnable* interface makes it possible to run it in an own thread, which is a good idea for graphics animations [Knu99]. The *Model* contains all entities which should be rendered. To ensure a correct registration and for better factorization of the software, the creation of all entities is centralized in a factory class, called *ObjectFactory*. Note that *Object* is the abstract superclass of all visualizable entities, the concrete subclasses are neglected here since this gives no further information for understanding the structure.

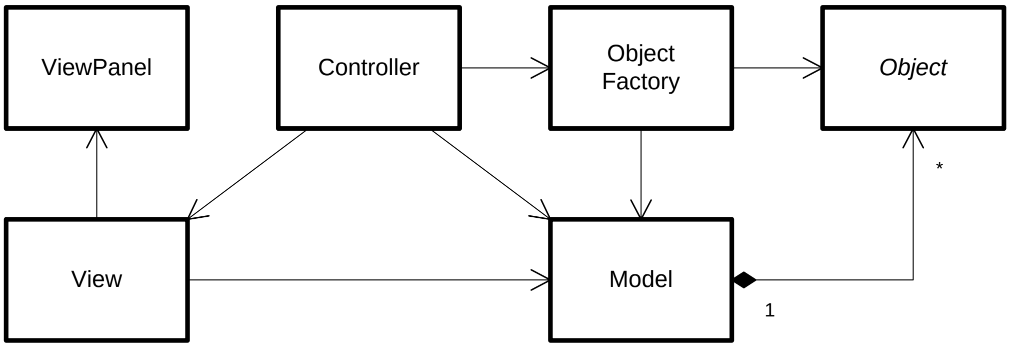


Figure 5.9: Structure of the Building Visualization

[6]a soft real-time system in terms of [Kop97]
[7]The prefix *Building* is omitted within the class names.

The Building Visualization shows the ground plan of the SmaKi's room and the actions inside the room. the predicted actions should be shown as well. So it has to fulfill several tasks. It has to show the static layout of the room, i.e. the walls, the door, several closets and the positions of the sensors. These are so called static[8] entities, since once shown, they do not change their appearance. On the other hand are the dynamic entities which can be distinguished between persistent and non-persistent. Persistent entities are sensors, they are created at startup and never deleted. However, they change their appearance according to the value the sensor takes. Non-persistent entities are persons and the prediction according to them. Persons are created and removed dynamically, as well as the predictions about their behavior.

Graph Visualization

The driving force behind the prediction are Hidden Markov Models (HMMs). Simplified (with sufficient correctness for this chapter) it can be said, HMMs are directed graphs. We are interested to look at those graphs to analyze their structure and their properties. According to a common saying *"One Picture is Worth a Thousand Words"* these graphs should be visualized. This is done by this part of the software.

However, the automated visualization of graphs, called *Graph Drawing* is a great field in computer science [KWE01] which is still not completed. A scientific community [5] exists, which is engaged in esthetic drawing of graphs. Besides the annual Graph Drawing Symposium, they organize the annual Graph Drawing Contest, actually the 15th [4].

Because of that complexity, it was decided to use a third party tool to accomplish this task. An open source project exists, called Graphviz [1] which provides a binding to the Java programming language, called Grappa [2]. With this tool it is possible to make nice visualizations of the HMMs. An example of such graphs can be seen in Figure 5.10.

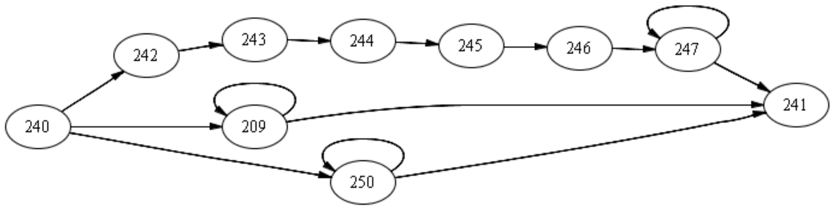


Figure 5.10: Example of a Graph visualized by Graphviz & Grappa

Details about how to use this third party tool are described in Chapter 6. This section describes the structure of the software and the envolved classes. An overview is given in Figure 5.11. As can be seen here the MVC pattern is used again. The class *Model* is a container for all objects of the graph which are nodes, i.e. instances of class *State*, and edges, i.e. instances of class *Edge*. *Object* is the abstract super class of them. The interface to the third party tool is within the *Model*.

[8]Static in the sense of not moveable in the room, not to be confused with static objects in terms of computer programming language.

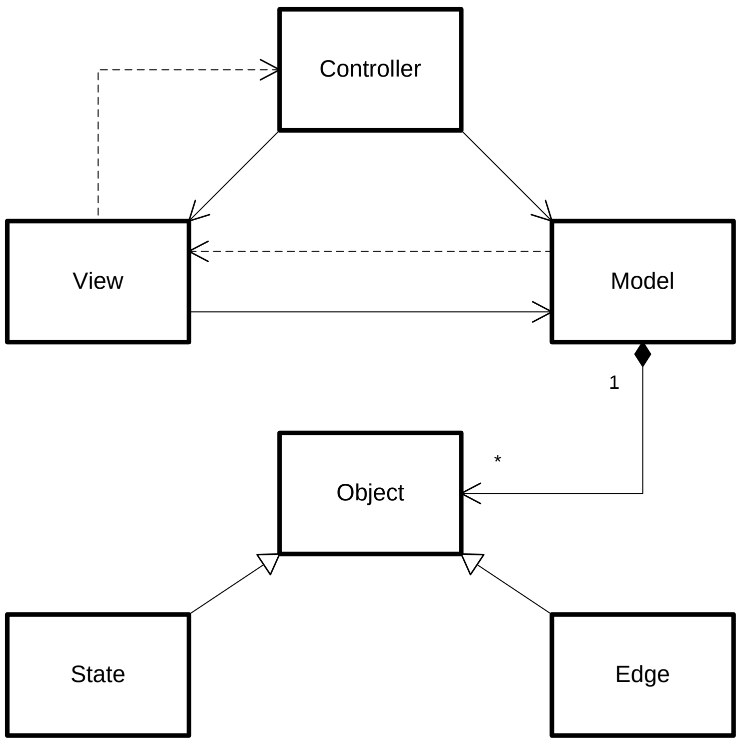


Figure 5.11: Structure of the Graph Visualization

5.5 Markov

This part contains the program logic about Hidden Markov Models (HMM). This software does not use matrix representation for the HMM, but rather a datastructure according to linked lists. As can be seen in Figure 5.12 an emission contains a symbol. The decision of using only symbols rather than application specific data, decouples the markov part from the data part. This rises the possibility of reuseability and allows to make changes easier.

As Figure 5.12 shows, the algorithms, the functionality for merging and the interface to the application specific data (*DataAbstractionProvider* are separated from the HMM's implementation. *HiddenMarkovModel* represents an HMM. As described in Chapter 3, an HMM is defined by a quintuple $\lambda = (S, A, B, \pi, \Sigma)$ which is represented as follows: S, the set of states, is represented by a list of *States* of *HiddenMarkovModel*. The initial state distribution π is a property of *States*, i.e. a variable of type IEEE-double[9]. The transition probability A is represented by *Transition* which is associated to *State*. A *State* can have $(0 \dots n)$ *Transitions* (in a model with n states), a *Transition* contains exact two *States*, the source and the destination. The probability is a property of *Transition*. If the transition probability between two states is zero, there is no *Transition* necessary. This saves memory and enhances performance. The emission probability B is represented by *Emission* which is associated to *State*. A *State* can have $(0 \dots \infty)$ *Emissions*, an *Emission* is part of one *State*. The probability is a property of *Emission*. *Emissions* with a

[9]IEEE 754-1985, Standard for Binary Floating-Point Arithmetic; the standard double datatype in Java

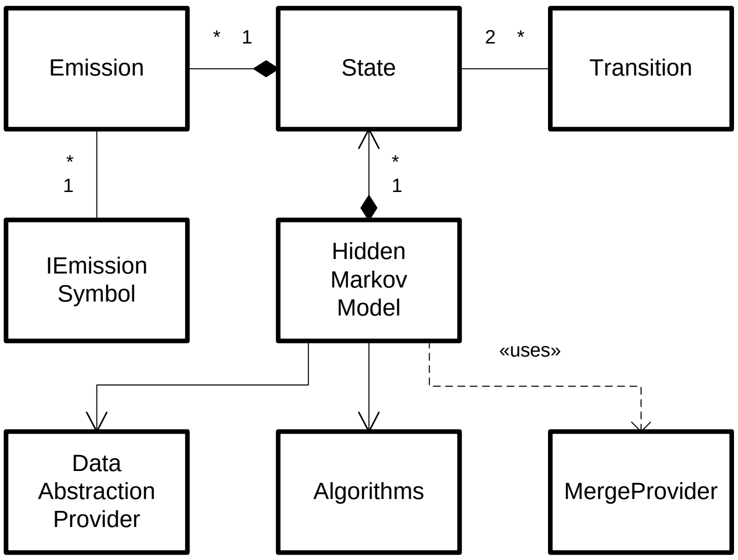


Figure 5.12: OOD of the Package Markov, Prefix 'HMM' is suppressed

probability of zero are not stored. The output alphabet Σ is represented by the collection of all *IEmissionSymbols* within the model, however, this is not required at this application. The *IEmissionSymbol* represents one output symbol, new *IEmissionSymbols* can be created at runtime, so the set of output symbols is unbounded and therefore it is no alphabet. This approach is chosen since the possible output symbols are not known a priori, they are learned. This is described in Section 4.4. This design of the HMM representation allows efficient learning, because if a new symbol is added to the system, this affects only the state[10], which can emit this symbol.

Learning

For the learning task, the *HiddenMarkovModel* is initialized with two *States*, the artificial start state and the artificial end state. For each new added *State*, a *Transition* is added to the immediate previously added *State* with the new *State* as destination. For the first added *State* the immediate predecessor is the artificial start. Merging of the learned models is provided by the *MergeProvider*. As the system manages several HMMs for learning (one for each person) the *MergeProvider* takes the *HiddenMarkovModel* which should be merged as a parameter for its `static` methods. However, *MergeProvider* only provides algorithms for defining which states should be merged, the real merging of two *States* is done within *State* itself.

Prediction

What *MergeProvider* is for learning, *Algorithms* is for the prediction. It provides the methods for calculating the forward probabilities and based on them the prediction for the next step. It is extended to look ahead more than one step, so up to three steps in the future are provided.

[10]With the learning principles at this work, this is always exact one state.

However, *Algorithms* only provides algorithms for calculating the probability distributions. The prediction is a selection of the most probable emissions. This selection is done by *HiddenMarkov-Model*. As we have integrated sensor values of different sensor types into the same model, sensor values of sensors, which emit a lower number of values than the others, are priorized. In a first round the most probable next states are selected. In a second round the most probable emission of these states are selected, only in the second round the prioritization is used. To get the real world values from the *Emission's IEmissionSymbols*, the *DataAbstractionProvider* is used. *DataAbstractionProvider* provides methods to get information from the *IEmissionSymbols* which can be visualized, i.e. the position of a switch sensor, since the ID (which is the data of the corresponding *EmissionSymbol*) would not be satisfying for a user.

5.6 Persistence

As the name supposes, this part deals with data which should be made persistent, i.e. the interface to the database. However, the name 'persistence' is not to be seen too narrow, the interface to the SmaKi's sensors is within this part of software, too. So we can distinguish two subparts, the connection to the database and the connection to the SmaKi's sensor network. The first one simply is called *Database* whereas the second one is called *Livedata*, because with this interface it is possible to retrieve sensor values from the SmaKi's room in real-time[11]. As described in Section 2.2.3, the system provides two different DBMS, from the companies Oracle and Microsoft. The whole structure of the persistence part is shown as a schematic drawing, depicted in Figure 5.13.

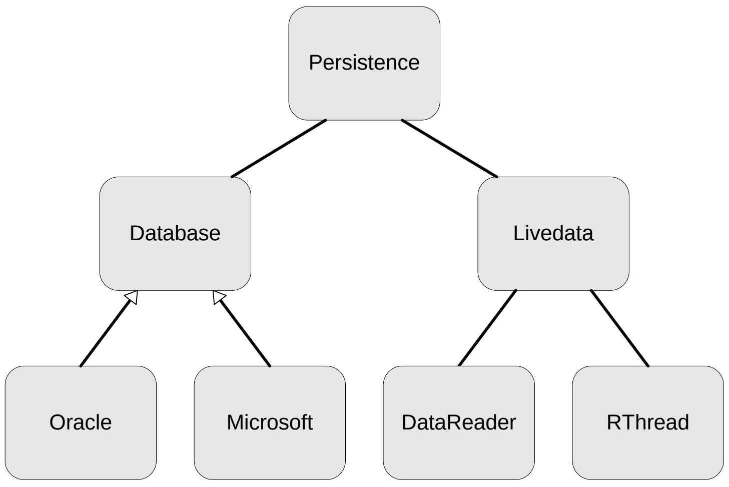


Figure 5.13: Schematic Drawing of the Persistence Part

To use the SmaKi Prediction Application it is a necessity to have a connection to a ARS Sensor database, because the information about the available sensors is taken from that storage. The values of the sensors can be taken from the database as well, however, this is rather a task for debugging during development. For the intended suppose of this application, a connection to SmaKi's observation hardware is needed, to get live-data.

[11]soft real-time

Database

The application supports two kinds of DBMS, but this is an ‘either - or’ relationship, i.e. at runtime only one database can be chosen. There are two classes which need access to the database. These are the *DataAccessor* and the *SensorFactory*. The *DataAccessor* provides functionality to create HMM-Chains from sensor values which are stored in the database. The aim of *SensorFactory* is to create the system’s internal representation of the sensors at startup of the application. Since some important information about the sensors, like type and position, is stored in the ARS Sensor database, *SensorFactory* needs a connection. If the connecting fails, the system cannot start.

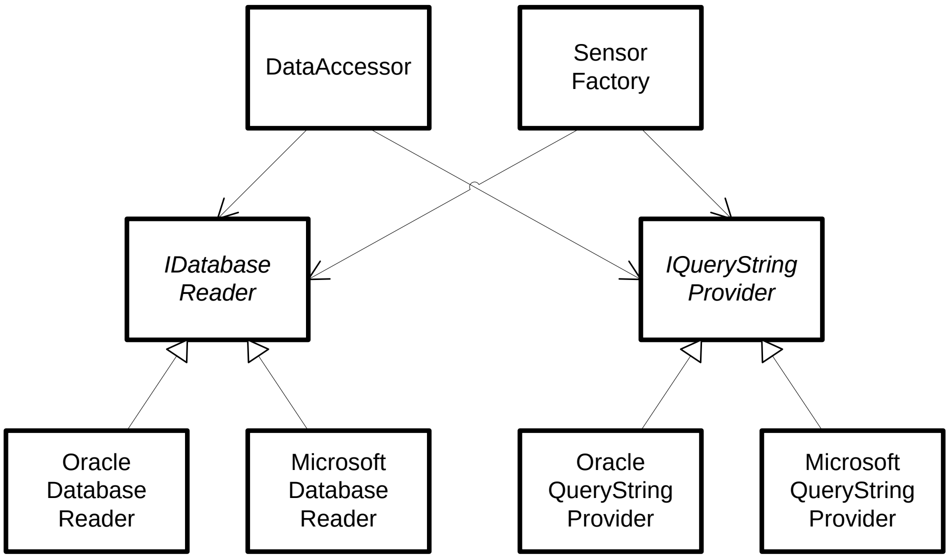


Figure 5.14: OOD of the Connection Classes to the Database

Both classes need a *IDatabaseReader* and a *IQueryStringProvider*, which are Interfaces. As can be seen in Figure 5.14, the concrete implementation of this interfaces depends on the available DBMS (provided are DBMS from Oracle and Microsoft). *IDatabaseReader* contains information which is necessary to establish a connection to the DBMS-server and to get access to the appropriate database. All SQL queries are collected in *IQueryStringProvider*. The reason for this approach is that each producer of a DBMS has his own SQL-dialect and this dialects are not compatible.

Livedata

This part provides the interface to the SmaKi’s observation hardware which is described in Section 2.3. Sensor values are sent over a TCP/IP network using the SymbolNet protocol (see Section 2.3.2). *LiveDataReader* establishes the connection. If this is successfully it starts the *ReaderThread* which listens on a TCP/IP socket for incoming messages. The messages contain a tuple (ID, value, timestamp) which represents a sensor value. This data is converted to the appropriate datatypes of this software by the *ReaderThread*. The converted tuples are added to a buffer of the *LiveDataReader*. *LiveDataReader* runs in an own thread, too. If a tuple is received, the appropriate sensor value is created by using the *DataAccessor*, since the *DataAccessor* ‘knows’ which sensor-ID corresponds to which sensor-type. An OOD of the involved classes is shown in Figure 5.15.

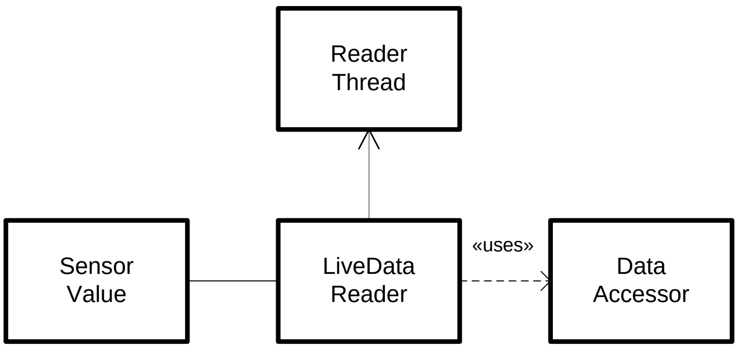


Figure 5.15: OOD of the Live-Data Connection Classes

6 Implementation

This chapter describes the implementation of the *SmaKi Prediction Application.* As described in Section 1.2 this software has to fulfill two main tasks: On the one hand the creation, parameter-optimization and evaluation of HMMs and on the other hand a live animation, which makes the SmaKi system's behavior and the results visible to the users. It was decided to use Java as programming language for the following reasons:

- Java is a state-of-the-art programming language.
- Java supports Object Oriented Programming (OOP) which is state-of-the-art for applications of this size.
- There are several sub projects concerning the ARS system which are implemented in Java.
- Interfaces to the Live-Data Modules are available for Java.
- The performance is sufficient; this is no hard-real-time application.
- An open source package for graph drawing is available for Java.
- Most of the students have sufficient programming skills in Java (for reusability).

For development the *Eclipse* IDE (Integrated Development Environment) was used, which is an open source project that focuses on building open development platforms [6]. For creating the graphical user interface (GUI) a WYSIWYG[1] GUI editor was used. This editor is named Jigloo and is available as a plugin for the Eclipse IDE. Jigloo "*is free for NON-COMMERCIAL use only*" [3]. For this project the latest version of Java was used, this is Java Standard Edition 6 (1.6.0).

Applications which are written in Java are subdivided into packages. Packages encapsulate the visibility of variables and methods and provide a structure for related modules. This is reasonable for this project, so this form of organization was followed. Note that package names in Java start with a lower case letter by convention, whereas names of classes start with a capital letter [LL08]. This chapter is organized like the software application, so the sections correspond to the packages in the source code.

[1]acronym for What You See Is What You Get

Important Classes, Methods and Member Variables

In this chapter the term ‘important’ is used in relation to classes, methods and member-variables. In a computer program *all* those entities are necessary for running correctly. However, there are a lot of very simple entities, like *Getter* and *Setter* methods[2], which are essential for the program, but provide no information to a programmer for a better understanding of the system’s structure. Moreover, the description of such entities is annoying, for this reason this is omitted in this work. In this chapter the term ‘important’ means, “important for a reader to understand the system’s structure”.

6.1 Package `base`

The package `base` contains the basic classes of the SmaKi Prediction Application. It interacts with all other packages and therefore it contains interfaces to all those packages. In the following an overview of the classes of this package is given. Simple classes are neglected. Methods of classes are mentioned only if they are important for understanding the system.

Class `MainController`

This class is the interface to the GraphController and the BuildingController of the package `gui`. As the name of the package promises, this includes being the interface to the graphical user interface. So this class has a lot of simple methods to pass the instruction given by the user to the correct module of software. A detailed description of this methods is not required to understand the system.

Class `MainModel`

The class `MainModel` contains basic data of the application, like a `SensorList` and a `WorldPersonList`. It has references to many other classes, however, those are already described in Section 5.2. The important methods are:

`init(MainController)`
: This has to be called before calling any other method. The parameter `MainController` is registered at the model. In order to the chosen database, it instantiates the appropriate implementations and establishes a connection.

`loadHMM(String)`
: Loads an HMM from the XML-file, which is specified by the string. Throws an `Exception` if the specified file contains not a valid HMM representation.

`saveHMM(String)`
: Saves the global HMM to the XML-file, which is specified by the string. Throws an `Exception` if the specified file cannot be written.

[2]accessor methods to access member variables

`addHMMChain(String)`
: Adds a Chain to the global HMM. The Chain is stored in an XML-file which is specified by the string. If the file doesn't contain a Chain, but a model, it is not added.

`addHMMModel(String)`
: Adds an HMM to the global HMM. The HMM which should be added is stored in an XML-file which is specified by the string. If the specified file contains a Chain the user is asked whether to add it as a Chain, which does some preprocessing.

`liveDataRetrieved(SensorValue)`
: This method is called from the `LiveDataReader` for each retrieved sensor value. The `SensorValue` is passed as argument. The corresponding sensor is updated and a method is called, to handle the specific type of the sensor value. Furthermore, all `WorldPersons` of the `WorldPersonList` are updated with the SensorValue's timestamp. If the lifetime of a WorldPerson is expired, it is removed from the list.

`tactileSenValRetrieved(SensorValueTactile)`
: This method is called from the method `liveDataRetrieved(SensorValue)` if the `SensorValue` is of type `SensorValueTactile`. If the raw value of the sensor value is equal to '1' (i.e. *true*), the appropriate `TactileSensor` is assigned to the closest `WorldPerson`. If the distance to this `WorldPerson` is too big or if the `WorldPersonList` is empty, a new `WorldPerson` is created. If the raw value of the sensor value is equal to '0' (i.e. *false*), the appropriate `TactileSensor` is removed from each `WorldPerson` in the `WorldPersonList`.

`movementSenValRetrieved(SensorValueMovement)`
: This method is called from the method `liveDataRetrieved(SensorValue)` if the `SensorValue` is of type `SensorValueMovement`. If the raw value of the sensor value is equal to '0', nothing has to be done. If the raw value of the sensor value is equal to '1', the `SensorValueMovement` is assigned to each `WorldPerson` in the `WorldPersonList` which is in the sphere of action of the appropriate movement sensor.

`switchSenValRetrieved(SensorValueSwitch)`
: This method is called from the method `liveDataRetrieved(SensorValue)` if the `SensorValue` is of type `SensorValueSwitch`. If the raw value of the sensor value is equal to '0', nothing has to be done. If the raw value of the sensor value is equal to '1', the appropriate `SwitchSensor` is assigned to the closest `WorldPerson`. If the distance to this `WorldPerson` is too big or if the `WorldPersonList` is empty, it is ignored. The door switch needs some extra handling due to the entrance workaround (see Section 4.2.3).

Class `VerificationModel`

This class provides a verification mechanism. The resulting values are written to a text file the name-suffix of which is generated using the current time. This ensures that verification files are not overwritten. These files can be read with any text editor or imported by *MS Excel.* The first column of the file represents the values from the prediction which got the highest probability. The second column represents the values of the second highest prediction and so forth. Sometimes there are blank values, they occur if there was no position emission at this rank.

Class `PredictionEntity`

This class is the representation of a prediction. It contains a `Type` which denotes which kind of prediction it is, since at the visualization there are differences in drawing predictions of different types. `Type` is an `enum` and can take the values (`POSITION`, `SWITCH`, `MOVEMENT`, `NONE`). All provided methods are *Getters* and *Setters*, so a detailed description of these methods is not required to understand the system.

Class `WorldPerson`

This is the class representing the abstraction of the persons in software. It contains the Person Model, described in Section 4.2. The most important member variables are the `Position`, which represents the position of the person in the room. For the calculation of this position it contains a list of assigned tactile sensors, which is a list of type `LinkedList<TactileSensor>`. A value which indicates if the person is active is also within this class. For the predictions it contains a list of type `ArrayList<PredictionEntity>`, with `PredictionEntity` as argument. The important methods are:

`Constructor(TactileSensor, HiddenMarkovModel)`
: The constructor takes two arguments. As described in Section 4.2.3 a new person is created if a sensor value of a tactile sensor cannot be assigned to a existing person. The emitting sensor of this unassigned value is passed to the constructor as the first argument (`TactileSensor`). The second argument is a reference to a **copy** of the global `HiddenMarkovModel`, which is needed for the prediction calculation. The constructor creates a new `HiddenMarkovModel`, the HMM which is used for learning; it is empty at the beginning. The list of tactile sensors which is necessary for the calculation of the position is allocated, as well as the list of prediction entities.

`movementSenValRetrieved(SensorValueMovement)`
: This method is called if a sensor value of a movement sensor is retrieved. The appropriate `SensorValueMovement` is passed to it as an argument. The method creates a `MovementSymbol` and forces the learn-HMM to 'learn' this symbol.

`switchSenValRetrieved(SensorValueSwitch)`
: This method is called if a sensor value of a switch sensor is retrieved. The appropriate `SensorValueSwitch` is passed to it as an argument. The method creates a `SwitchSymbol` and forces the learn-HMM to 'learn' this symbol.

`add(TactileSensor)`
: If the current value of the `TactileSensor` is equal to '1' it is added to the persons's list of tactile sensors. So the position of the person has to be recalculated, this is indicated by setting a `PositionChanged` bit.

`remove(TactileSensor)`
: If the current value of the `TactileSensor` is equal to '0' it is removed from the persons's list of tactile sensors. So the position of the person has to be recalculated, this is indicated by setting a `PositionChanged` bit.

`double weightDistance(Position)`
: The distance from the person (i.e. the `WorldPerson's Position`) to the `Position` given by the parameter is calculated and returned as a value of type `double`. The method is called *weight*, because if the person is not active, i.e. it is an item (as described in Section 5.2), the method returns `Double.MAX_VALUE` and if the person is active, but has an empty sensor list, the distance is shortened. This is done to prioritize a person with an empty sensor list when assigning tactile sensor values to persons.

`boolean update(long timeMillis)`
: The parameter of this method represents a timestamp which is used to calculate if the lifetime of an inactive person has expired. This method uses the `PositionChanged` bit which is set by `add(TactileSensor)`.
If this bit is *true*, this indicates that a step of `this` person occurred. The term *step* has two meanings in this context: On the one hand, it is a footstep of a real person in the SmaKi, on the other hand it is a step in the HMM (prediction) representation. So a new `PositionSymbol` is created and learned by the learning-HMM. The prediction-HMM calculates the next step by use of the created `PositionSymbol`. After that, the `PredictionEntities` and the `VerificationModel` are updated.
If the `PositionChanged` bit is *false*, the lifetime-timers are updated. If lifetime has expired, the person is marked as inactive in case of having an assigned tactile sensor. If there is no tactile sensor assigned, the learning-HMM is saved to a file an the person is marked to be removed from the `MainModel`'s `WorldPersonList`.

`delMe()`
: This is a simulation of a destructor. It sets all references to `null` and forces the Java's garbage collector to run immediately. This is a necessity to avoid `OutOfMemoryExceptions` when the system is running several weeks.

`saveLearnmodel()`
: This method creates a file name by the use of a predefined prefix and the current time of the system. This ensures unique file names. The learned HMM is saved to this file.

Class `WorldPersonList`

This class represents a list of `WorldPersons`. The basic operators `add`, `remove` and an `iterator` are implemented. The extension (due to standard lists) of interest is the method `WorldPerson getClosest(Position)`. It takes a `Position` as parameter and returns a reference to the `WorldPerson` which is closest to this `Position`. If the list of `WorldPersons` is empty or if all `WorldPersons` of the list are inactive, the method returns `null`.

Class *`Sensor`*

This is an abstract class representing a (generic) sensor. All implementations of specific `Sensors` are inherited from this class. It contains the sensor's identity (ID) as an `integer` value and a description as `String`. It also provides a field `Timestamp` where the point in time where the last modification of those sensor's value occurred, can be saved. There are two important methods within this class. The first one is `updateValueRaw(SensorValue)` with the purpose to update the sensor's value. This method has to be overwritten by a subclass. The second important class is `Point getPosition()` which is abstract, so it has to be overwritten by a subclass. This method returns the center `Point` of a sensor. The concrete implementations are `MovementSensor`, `SwitchSensor` and `TactileSensor`.

Class `SensorList`

A list of all available sensors is represented by this class. The basic operators `add`, `remove` and a `iterator` are implemented. The extension (due to standard lists) of interest is the method `Sensor get(int ID)` which returns a Sensor with the given `ID`, `null` if there is no sensor with the given `ID`. Since this functionality is used frequently (for each sensor value retrieved by the system) and searching a list takes a long time ($O(n)$) compared with a dictionary-search ($O(log(n))$), the internal representation of the list is a `TreeMap<Integer, Sensor>` with the `Sensor`'s `ID` as key. The class `TactileSensorList` represents a list of the `TactileSensors` only.

Class *`SensorValue`*

This class represents a sensor value retrieved from the SmaKi's hardware. This is an abstract class which contains only one field, an `integer` with the `ID` of the sensor[3] which emitted this value. All implementations of specific `SensorValues` are inherited from this class, these concrete implementations are `SensorValueMovement`, `SensorValueSwitch`, `SensorValueTactile` and the 'pseudo' value `SensorValueKeepAlive` for the keep-alive messages.

6.2 Package `data`

This package is the interface between the package `markov` and the package `pers`. It is used to get the underlying data from the `IEmissionSymbols` at the prediction phase. When learning from live-data this package is not necessary, however, it is used for learning from the database.

Class `DataAbstractionProvider`

This class provides the interface to the package `pers`, which is a reference to class `DataAccessor`. The interface to the package `markov` is given by a reference to class `HiddenMarkovModel`. Furthermore, it provides functionality for getting data of the predicted emissions and for the creation of simple HMMs from data from the database. The most important methods are:

`setPredictionEntity(PredictionEntity, Emission)`
: The first parameter is a `PredictionEntity`. This is a class which contains data for the visualization of predictions. The second parameter is the `Emission` which should be visualized. An `Emission` contains an `IEmissionSymbol` which denotes *which* type of `PredictionEntity` should be displayed. Furthermore, the `IEmissionSymbol` contains data which denotes *how* the `PredictionEntity` should be displayed. This method uses the information stored in the `Emission` to configure the content of the `PredictionEntity`.

[3]The sensor of the real world, not the internal representation which is denoted `Sensor`.

`IEmissionSymbol createEmissionSymbolFromXML(Node)`
: This method takes an XML-node as parameter which contains an `IEmissionSymbol` and creates this symbol from the XML-data. If `Node` contains no symbol data, an exception is thrown.

Class `Position`

This class represents a position in world coordinates. It is like the Java's `Point2D.double` class, but it implements the `Comparable Interface`. This allows ordering of `Positions` and therefore a dictionary-search in a sorted collection.

Symbol Classes

The classes `MovementSymbol` and `SwitchSymbol` inherit from `BinaryIDSymbol`, `BinaryIDSymbol` and `PositionSymbol` implement the `IEmissionSymbol Interface` of the package `markov`. All these classes provide functionality for creating an XML-structure to save their content to an XML-file.

There are three classes representing lists of symbols, which are `PositionSymbolList` for `PositionSymbols` only, `BinaryIDSymbolList` for `MovementSymbols` and `SwitchSymbols`, and `EmissionSymbolList` for *all* kinds of symbols. The last one is used if the type of symbol is not known. These lists provide special `get()` methods which implement a dictionary-search in the internal collection.

6.3 Package `gui`

This section describes the package `gui`. This package covers everything which should be displayed to the user. Not every subpart of this package allows user input (except default operations, like closing the window) since they are intended for displaying information only. Almost all user interaction elements are located within the class *MainFrame*. Since building graphical user interfaces from scratch is a long winded work, it was decided to use a graphical GUI editor for creating these classes. This editor is named Jigloo and is available as a plugin for the Eclipse IDE.

This package can be subdivided into three parts, the user interface which is encapsulated in class `MainFrame`, the Building Visualization which is the real-time part that shows the current situation to the users of the SmaKi and the Graph Visualization for the drawing of the HMM's structure as a directed graph.

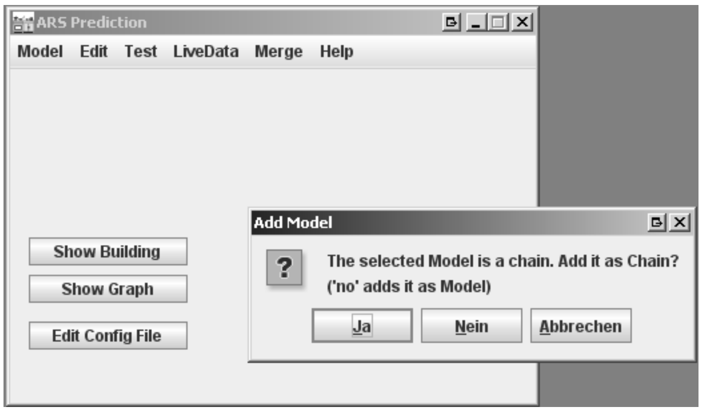
6.3.1 User Interface

As mentioned above, the user interface is in class `MainFrame`. This class also contains the entry point of the application. All user actions are forwarded to the class `MainController`, which is located in package `base`.

The user interface was designed with *Jigloo*, a plugin for *Eclipse* which allows rapidly developing of GUIs, based on a visual GUI editor. It generates the source code for the GUI elements automatically, only the actions of the handlers have to be added manually. The handlers themselves are autogenerated too, the source code is appended at the end of the file. Their name ends with "`ActionPerformed`" and an `ActionEvent` argument is passed to the implementation of them, but this is never used.

Since this class contains the entry point of the application, i.e. the static `main()` method and it is responsible for some initialization stuff which takes some time, it was decided to show a splash screen to the users when the application starts. However, this increases usability, because the initial connection to the database may take some time, this is not a necessity. Java provides a module for creating splash-screens which is started at the very beginning to reduce the time in which no feedback is given to the user.

Figure 6.1 shows a screenshot of the GUI. As can be seen, the system will display dialogs to suppose actions the user might have done. For each action a user starts, this class delivers an `ActionEvent` to the `MainController`.



The selected Model is a chain. Add it as Chain?
('no' adds it as Model)

Figure 6.1: Screenshot of the GUI

6.3.2 Building Visualization

This section describes the classes for the visualization of the SmaKi's layout. Since each type of item which should be drawn on the sceen is represented by its own class, not all of these classes are described here. The important classes are described in the following.

Class `BuildingController`

This class is the interface to the package `base` of the application. It gets instructions from the `MainController` and handles the affected building-visualization classes.

Class `BuildingView`

This class provides a `JFrame`[4] to draw the `BuildingViewPanel`. The constructor gets a reference to the `BuildingController` as parameter. This is necessary to tell the the controller if the size of the window, represented by the `JFrame` is changed. Resizing of windows can be done by the user, i.e. it is not disabled.

Class `BuildingViewPanel`

This class is responsible for drawing the scenes. It extends `JPanel` and implements the `Runnable Interface`, i.e. it runs as an own thread. Once a screen is rendered, the thread sleeps for a certain period. As rerendering is not necessary if no changes happen, the frequency of redrawing is reduced if no sensor value is retrieved for a while. This is done to save power and reduce generation of heat. All periods and durations are configurabel. The important methods are:

`start() and stop()`
: Starts or stops the animation thread.

`paint()`
: The overriden method for painting of the `JPanel`. It calls the `paintObjects()` method of class `BuildingModel` to redraw all objects on the screen.

`setAnimationSpeed(AnimationSpeed value)`
: Sets the durations between the redrawing of the scenes. `AnimationSpeed` is an `enum` which can take three values: `fast`, `slow`, `powersave`.

Class `BuildingModel`

This class contains all objects inside the room which have to be visualized. For this task it manages two lists: A list of `BObjects` and a list of `BPersons`. The transformation from world coordinates to screen coordinates is also done within this class. For this task, the class manages the viewing area of the screen and the size of the world representation. The important methods are:

`addObject(BObject)`
: This method takes a `BObject` as argument and adds it to the model's object list. If the world coordinates of the added object are outside the model's world representation range, this range is updated.

`addPerson(BPerson)`
: This method takes a `BPerson` as argument and adds it to the model's person list. Additionally the person is added to the model's object list, by using the method `addObject(BObject)`. As a `BPerson` is inherited from `BObject` this can be done.

`paintObjects(Graphics)`
: This method is called from the `BuildingViewPanel` and gets the `Graphics` object from there. It iterates over all `BObjects` in the model's object list and calls the `BObjects's paint()` method. This is the reason that `BPersons` are added to this list too.

[4]JFrame is used to draw windows with Java Swing.

`updatePersons()`
: This method updates all `BPersons` of the model's persons list, i.e. it recalculates their positions and predictions and their appearance (inactive persons become an item). Fully inactive persons are marked for deletion.

`removePersons()`
: `BPersons` which are marked for deletion are deleted. Furthermore, all the `BPredictio-nentities` which are associated with this `BPerson` are deleted too.

`setWindowSize(Dimension)`
: This method is called if the size of the window where the visualization is displayed has changed. So the scale factor for the transformation from world to device coordinates has to be recalculated and all objects which are displayed have to be updated.

`toDeviceCoordinates()`
: This method calculates the device coordinates from the world coordinates for all `BOb-jects`. This method is used at startup and when the size of the device has changed, i.e. the viewing window is resized by the user.

`toDeviceCoordinates(BObject)`
: This overloaded method which takes a `BObject` as parameter only recalculates the device coordinates of the given `BObject`. This method is used for recalculation of moveable objects, which are for example `BPerson` and `BPredictionEntity`.

Class `BuildingObjectFactory`

This class creates all objects which should be visualized and adds them to the `BuildingModel`'s object list. At startup of the application this class is used for creating all the visual entities of the displayed SmaKi's layout. At runtime it is used to dynamically produce `BPersons`. The important methods are:

`createSensors(SensorList)`
: This method takes a `SensorList` as parameter. For all sensors within this list, which can be of different types, the appropriate graphical representation of the sensor-object is created and added to the `BuildingModel`.

`createPerson(WorldPerson)`
: This method takes a `WorldPerson` as parameter. For this `WorldPerson` a corresponding graphical representation is created and added to the `BuildingModel`.

`createStaticItems()`
: This method creates all static items which should be displayed. These are rectangles showing the kitchenette, shelfs, table et cetera. The values are hardcoded.

`createStaticItemsFromFile(String)`
: To make modifications of such trivial things of the SmaKi's layout easier, this method provides functionality to read the static configuration of the layout from an XML-file. The filename is given by the parameter. If wanted, the system can draw images inside the rectangles.

`Color getColor(String)`
: This method takes a `String` as parameter and tries to convert the represented value to a `Color` object and returns it. The paramter can contain rgb-values or the name of a system color. The `String "[128,128,128,128]"` represents rgb-values with the optional alpha parameter, the `String "yellow"` represents a known system color. To ensure that all system colors are provided, the method uses reflection to get the correspondig values of the string.

Class `BObject`

This is the *abstract* super-class of all entities which can be visualized by the Building Visualization. It only contains one method, this is the `abstract` method `paint(Graphics)` which takes a `Graphics` object as parameter. However, there are two member variables worth mentioning: An array of `WorldPoints` which are of type `Point2D`, i.e. take `double` values for 'x' and 'y' and an array of `DevicePoints` which are of type `Point`, i.e. take `int` values for 'x' and 'y'. Memory allocation for this arrays is done by the implementing class. The class `BuildingModel` which manages all `BObjects` which are displayed uses these two arrays of points to transform world coordinates to device coordinates.

All concrete implementations of this class override the `paint()` method to draw the appropriate graphic on the screen. All classes have a "`B`" as prefix in their name. Implementations of dynamic items contain a reference to the corresponding world representation object, e.g. `BTactileSensor` contains a reference to a `TactileSensor`. To get the values from the world-object an `update()` method is used. This is all the same for the implementations of `BObject`, so a detailed description is not necessary here. All concrete implementations of `BObject` are: `BRectangle`, `BImage`, `BHeartbeat`, `BMovementSensor`, `BSwitchSensor`, `BTactileSensor`, `BPerson` and `BPredictionEntity`.

6.3.3 Graph Visualization

This section describes the classes for the visualization of the global HMM as a directed graph. The local HMMs do not contain further information since the learn models are simple chains and the prediction models are clones of the global one. The graph's layout is calculated with a tool called *Graphviz*, for visualization the library *Grappa* is used, see Section 5.4 for the motivation using this. The important classes for Graph Visualization are described in the following.

Class `GraphController`

This class is the interface to the package `base` of the application. It gets instructions from the `MainController` and handles the affected graph-visualization classes.

Class `GraphView`

This class provides a `JFrame`[5] to draw a `Panel` and it has two important member variables, which are `GrappaPanel` and `Graph` which are both from the Grappa library. `Graph` contains the internal

[5]JFrame is used to draw windows with Java Swing.

representation of the graph and is filled at the class `GraphModel`. So it is forwarded to this class by the use of `GraphController`. The `Graph` is passed to the constructor of the `GrappaPanel`. This panel manages the visual representation of the graph.

Class `GraphModel`

This class provides the generation of the graph from the HMM's structure. The constructor takes a `HiddenMarkovModel` as parameter which represents the HMM that is visualized. The layout of the graph is calculated by a call of the method `createGraph()`. This method uses the inner class `GraphFactory` to render the graph's layout. Furthermore, this class provides methods for marking and unmarking of edges and nodes. Edges are represented by the class `GEdge`, nodes are represeted by `GState`; they have a common super class, named `GObject`.

Class `GraphFactory`

This is an inner class of `GraphModel`. It creates the graph from the HMM by using the Graphviz and Grappa packages. For this task the following methods are used.

`createGraphSpec()`
: This method creates a graph specification file. This file contains all nodes, a description of these nodes, all edges, a description of the edges and some general properties of the graph. The file name is a specified constant. For the creation of the nodes two alternatives are possibel, one which represents the correspondig `HMMState`'s label as node label, the other one uses a description of the `HMMState`'s emissions as label for the nodes in the graph. For the description of a graph-specification file, see the online documentation[6].

`createGraphLayout()`
: This method creates the graph layout file from its specification file. The file names are constants which are defined within the class. The method uses an external tool which is part of the Graphviz application. This tool is an executable which is started by the command line "`dot -Tdot specfile -o layoutfile`". It is executed as a `Process` in a `Runtime` environment.

`readGraphLayout()`
: This method reads the graph layout file and uses Grappa to parse it. If parsing is successful, the result is a `grappa.Graph` object which can be visualized by the class `GraphView`.

`bindGrappa()`
: When the specification of the graph is created, for each object of the `HiddenMarkovModel` which is added to the specification file, a corresponding `GObject` with a reference to the `HMMObject` is created. As the corresponding `grappa.Objects` are created when the layout file is parsed, these `grappa.Objects` are binded to the corresponding `GObject` with this method.

[6] `www.graphviz.org`

Class `GObject`

This abstract class represents an entitiy of the graph, concrete implementations are `GEdge` and `GState`. It provides two overloaded methods `equals()` for testing if two `GObjects` represent the same entity. One takes a `GObject` as argument, the other one a `String` which represents the name of the entity, i.e. one label in case of a state, two lables separated by an arrow in case of a transition. Furthermore, it contains two abstract methods `mark()` and `unmark()`.

Class `GState`

This class represents one node of the graph. It contains a reference to a `grappa.Node` and a reference to an `HMMState`. The constructor is private and takes an `HMMState` as parameter. The `HMMState`'s label is used to call the constructor of the `super`-class `GObject`. A `static` method `createGState(HMMState)` is used for instantiation of an object.

Class `GEdge`

This class represents one edge of the graph. It contains a reference to a `grappa.Edge` and a reference to a `HMMTransition`. The constructor is private and takes the label of the edge as parameter. So a `static` method `createGEdge(HMMTransition)` is used for instantiation of an object. This method builds a string which is the label of the edge from the given `HMMTransition`.

6.4 Package `markov`

The package `markov` encapsulates the software for internal representation of HMMs. This includes the data structure described in Section 4.4 and the algorithms for learning, merging and predicting. The root class is named `HiddenMarkovModel`. This class represents the core of an HMM and is the interface to the package `base` as well. The important classes of this package are described in the following.

Class `Algorithms`

This class contains the prediction algorithms. This is the forward algorithm, known from Chapter 3 which gives a probability distribution over all next states. To take an appropriate selection is part of another algorithm. Calculating the forward probability is done iterative with each retrieved *EmissionSymbol*. The current forward probability is stored at the class `HMMState` for each state separately. The prediction algorithm allows to see ahead up to three steps in the future. The method `calcForward(IEmissionSymbol)` takes the actually received Emission Symbol and calculates the forward probabilities for the next three steps for all states. Due to the forward algorithm inherent the probabilities become very small numbers. This will cause problems with the value range of the IEEE-floating point numbers, even if double precision is used. A definition can be found at [IEE85]. One approach is to use the logarithm of the probabilities, however, this is not possible due to the necessary summation of probabilities. So the `shiftProbs()` method was introduced. If the highest forward probability of any state is below a certain threshold, the values of all states are multiplied by this threshold. So it is ensured that at least one forward probability always is within the value range of the IEEE-double, the others could be zero.

Class `Emission`

An `Emission` belongs to an `HMMState` and contains an `IEmissionSymbol` which contains the intrinsic data. The `Emission` itself contains a counter of occurrences and its probability. The method `update(count)` recalculates the probability dependent on the given value `count`, which is the sum of all emissions belonging to a state.

Class `HiddenMarkovModel`

At startup two states are created, the artificial start state and the artificial end state, which are necessary for the systems structure learning, see Section 4.4. This class provides several methods which are described in the following.

`learnSymbol(IEmissionSymbol)`
: For a `PositionSymbol` a new state is added to the model and a new emission containing this `PositionSymbol` is added to this new state. All other `EmissionSymbols` are added to the `CurrentLearnState`, which is a reference to the last created state, `null` at startup.

`getPredictionEntities(ArrayList<PredictionEntity>)`
: This method calculates the most probable predictions. It takes a list of `PredictionEntity` which has to be allocated at the caller. There are several strategies how to calculate the most probable prediction, since the different types of sensor produce different probability distributions. For this reason this method exists with several suffixes, like "_1", which are not used at the current compiled version, but might be of interest.

`addModel(HiddenMarkovModel)`
: This method allows to combine two models. As known from Section 4.4, so called Chains are created at the learning phase. These Chains are combined to one model using this method.

`saveToFile(filename)`
: This method provides functionality for saving the HMM to an XML-File, the filename is given by the parameter. An inner class `XMLSerializer` provides the serialization of the HMM.

`loadFromFile(filename)`
: This method provides functionality for loading the HMM from an XML-File, the filename is given by the parameter. An inner class `XMLSerializer` provides the deserialization of the XML-structure.

`delMe()`
: This is a simulation of a destructor. It sets all references to `null`, so the Java's Garbage Collector can remove this object from memory. The removal of the reference to `this` object is in response of the caller, which is the method `delMe()` of the class `WorldPerson` in the package `base`.

Class `MergeProvider`

This class provides algorithms for state merging which is necessary for the learning of HMMs (see Section 4.4). The methods are `static` since they operate on the `HiddenMarkovModel` which is passed to them as a parameter. This is done because there are several different HMMs within the learning phase (each person has its own HMM, see Section 4.3.2 for details). Originally it was planned to use gaussian distribution (μ, σ^2) of position's distances to define whether to merge two states. But as the parameter σ has to be estimated as well as the resulting variable x, with the effect of cancelling each other out (and for simplification), it was decided to use distances. The name 'Gauss' remained in the method names to indicate that this is no merging of exact data.

`boolean isChain(HiddenMarkovModel)`
: This function checks whether the assigned `HiddenMarkovModel` is a chain, i.e. each state (except the artificial end state) has exact one transition.

`processChain(HiddenMarkovModel)`
: This method does the preprocessing of chains like described in Section 4.4.4. Additionally states with exact the same positions are merged, the input `HiddenMarkovModel` has to be a chain.

`mergeHorizGauss(HiddenMarkovModel)`
: Does the horizontal merging like described in Section 4.4.5. The input `HiddenMarkov-Model` has to be a chain.

`mergeStartGauss(HiddenMarkovModel)`
: The vertically merging described in Section 4.4.6 is implemented in this method. The used algorithm begins at the artificial start state.

`mergeEndGauss(HiddenMarkovModel)`
: The same as `mergeStartGauss(HiddenMarkovModel)`, but using an algorithm that begins at the artificial end state.

`mergeConsecutive(HiddenMarkovModel)`
: This method provides the merging of sequences which is described in Section 4.4.7.

`boolean haveSameSwitch(HMMState, HMMState)`
: Having same *SwitchEmission*[7] means that if a state S_a contains a *SwitchEmission* E_a, a state S_b also contains E_a and vice versa. *SwitchEmissions* are distinguished by the ID of their contained `SwitchSymbol`.

Class `HMMState`

This class represents a state within the HMM. As can be seen in Figure 5.12, a `HiddenMarkov-Model` contains $0 \ldots n$ `HMMStates`. Each `HMMState` contains $0 \ldots n$ `Emissions` and $0 \ldots n$ `HMM-Transitions`. Thus the `Emissions` and `HMMTransitions` are represented as `ArrayLists` of the appropriate type. Furthermore, each `HMMState` contains an `ArrayList` of referencing `HMMStates`, i.e. of states which contain a transition with this state as destination.

[7]An `Emission` which contains a `SwitchSymbol` is called a *SwitchEmission* for short.

`Constructor`
: The constructors of this class are private, so it provides a method for creating which is called `createState(HiddenMarkovModel)` and takes the HMM where the state should be assigned to, as a parameter. This ensures that an `HMMState` is part of a `HiddenMarkov-Model`. The constructor and the appropriate creation-method which take a name as second parameter, create a state with the given name, this is used when a `HiddenMarkovModel` is created from a file. Those without the name as parameter define their own name by using a `static` counter variable which is incremented for each created state. Since a model is either learned *or* loaded from a file, overlappings are impossible. However, overlappings can occur when combining models at the learning phase as described in Section 4.4.6, so it has to be ensured that the name is unique and if necessary the state has to be renamed.

`merge(HMMState)`
: This method merges the given state with `this` state. The method ensures that the emissions and transitions from the given state are added to `this` state, or if already contained that their probabilities are updated. Also the list of referencing states is updated and the transitions *to* the given state are redirected to `this` state.

Class `HMMTransition`

A `HMMTransition` has a source state and a destination state. Furthermore, it contains a counter of occurrences and its probability. The method `update(count)` recalculates the probability dependent on the given value count, which is the sum of all `HMMTransitions` with the same source state.

Interface `IEmissionSymbol`

This represents the interface to the package `data`. Each concrete *EmissionSymbol* has to implement this interface.

Class `XMLSerializer`

This is an inner class of class `HiddenMarkovModel`. It provides the functionality to save HMMs to XLM-Files. The method `removeTextNodes(Node root)` is necessary to remove unintended text nodes from the XML-structure when an XML-file is read.

6.5 Package `pers`

This package has to fulfill two tasks. First it has to provide connectivity to the ARS-Sensor database. The second task is to provide the interface to the live-data, which is sent from the SmaKi's observation hardware via TCP/IP. The OOD class diagrams for this two tasks are shown in Figures 5.14 and 5.15.

6.5.1 ARS Sensor Database Connectivity

The classes which are described here provide connectivity to the static data and the dynamic data of the sensor database. The software provides DBMS of different vendors, here only the interfaces are described in detail. Concrete implementations are given for *Microsoft* and *Oracle* databases.

Class `DataAccessor`

This class is the interface to the package `data`. It provides methods for creation of `SensorValues`.

`Constructor(IQueryStringProvider, Connection)`
: The constructor takes a `IQueryStringProvider` and a valid connection as parameter. Thus the connection has to be established at the creator.

`ArrayList<SensorValue> getSensorValues(Calendar start, Calendar end)`
: The ARS Sensor Database is able to store all sensor values of the SmaKi's recognition system if this is wanted. To get these values from the database this method can be used. It takes two `Calendar` arguments, where the first one is the start timestamp and the second one the end timestamp. So the method returns an `ArrayList` of all sensor values between start and end.

`static SensorValue createSensorValue(senID,value,timestamp)`
: The return value of this static method (`SensorValue`) has several subtypes. If the system receives a sensor value from the hardware, this is a triple (sensor ID, value, timestamp). To create the correct subtype of `SensorValue`, corresponding to the ID, this method is used. For simplification the mapping ('sensor ID' - 'sensor type') is hard coded, for this reason this method doesn't need a database connection and can be static.

Class `SensorFactory`

As the name promises this class is used to create the sensors. The constructor takes three parameters, a `IQueryStringProvider`, a valid `Connection` and a `SensorList`. The connection to the database has to be established at the creator. Once an instance of this class is created there is only one method of interest, the method `createSensors()`. If it is called, the `SensorList` is filled with all sensors which are stored in the database. At this project there are three types of sensors, which are `TactileSensor`, `SwitchSensor` and `MovementSensor`, they all are subtypes of `Sensor` and therefore stored in one `SensorList`.

Interface `IDatabaseReader`

For connecting to a database this interface has to be implemented. The method `Connection getConnection()` returns the `Connection` which has to be established by calling the `connect()` method. The implementations used at this project are called `MicrosoftDatabaseReader` and `OracleDatabaseReader`. They contain the data which is necessary to connect to a database, however, this data comes from a config file over one indirection.

Interface `IQueryStringProvider`

An implementation of `IQueryStringProvider` contains the required SQL queries. As each vendor of DBMS has its own SQL dialect, all queries which are used by the system are collected within this class. There is nothing done in terms of security, so we assume that there is no malicious person at the institute who tries to make SQL injections.

6.5.2 Live-Data Interface

For using live data from the SmaKi's oberservation hardware, two classes are necessary: `LiveDataReader` as the interface to the application and `ReaderThread` for listening on a TCP/IP socket for incoming data. A tool called *Message Distributor* provides the distribution of data from the Octobus to several receivers.

Class `LiveDataReader`

This class represents the interface to the live data. It implements a thread for polling on incoming sensor values. It has a synchronized buffer, which is filled by the `ReaderThread`. If a value is added to the buffer, the `LiveDataReader`'s thread is notified. As the thread starts, the value is consumed, transformed in a `SensorValue` and passed to the `MainModel` (both part of package `base`). For handling data from SymbolNet it implements an inner class `SymbolNetTuple`, which contains three values: *Sensor-ID*, *value*, *timestamp.* The buffer is implemented as a double ended queue. If for a certain time (which is configurable) no data is retrieved, it tells the graphics part to reduce the animation speed. Important methods are:

Constructor(MainModel)
: The constructor gets a reference to the `MainModel` as parameter to be able to pass `SensorValues` to it. Some initialization and memory allocation for the buffer is also done within the constructor.

`addTuple(int id, int value, long timestamp)`
: This method is called by the `ReaderThread` to add data to the buffer. The three single values are converted to a `SymbolNetTuple`, this tuple is added to the buffer by the synchronized method `addLast(SymbolNetTuple)`.

`run()`
: This is the implementation of the thread. It blocks until the buffer contains a value. If there is a value in the buffer (of type `SymbolNetTuple`), this is converted to a `SensorValue` by using class `DataAccessor`. Afterwards this `SensorValue` is passed to the `MainModel`. However, there is a timeout implemented when waiting for a new value in the buffer. If this timeout expires, a keep alive message is sent to the `MainModel`.

`connect()`
: When this method is called, the `ReaderThread` is created and started. Once the `ReaderThread` is running, a connection from the Octobus can be established by using the appropriate command line parameters (see Section 2.3.1). This can be automated by using the `sendConnectMessage(int port)` method which uses a tool, called *MessageDistributor.*

`sendConnectMessage(int port)`
: If the *MessageDistributor* is used, this method is used to force the *MessageDistributor* to establish a connection to the caller. The port to which should be connected, is given by the parameter of the method.

`disconnect()`
: First the `ReaderThread` is stopped, then the `this` thread is stopped.

Class `ReaderThread`

This class implements a thread, which is listening the TCP/IP-Socket for incoming symbol-messages of the SymbolNet-Protocol. These messages are added to the `LiveDataReader`'s buffer. As communication is done by using the SymbolNet-Protocol (see Section 2.3.2), it has an inner class `SymbolReceiver` which inherits `TcpReceiver`. `TcpReceiver` is part of the SymbolNet library. This inner class decodes the SymbolNet messages and passes it to the `LiveDataReader`. The methods of `ReaderThread` are:

`Constructor(InetSocketAddress, LiveDataReader)`
: The first argument is the address of incoming data, the second is a reference to the `LiveDataReader`, which is necessary for adding messages to its buffer.

`run()`
: The thread's runnable implementation.

`disconnect()`
: Disconnects the `SymbolReceiver` and stops the thread.

`isConnected()`
: Returns `true` if the thread is running correctly.

Message Distributor

For automated establishing of connections there is a tool called *Message Distributor* (MD). The MD can be started on any PC with a TCP/IP connection. Instead of connecting from the Octobus to a computer with an application that needs data from it, the Octobus is connected to the MD. MD listens on a port for incoming connect-messages (which contain a port as parameter). If such a message is retrieved, the MD tries to establish a connection to the IP-address, from which the message was retrieved, with the specified port. If this is successful, the data from the Octobus is forwarded to this machine. Data can be forwarded to several connections.

6.6 Config Files and Resources

The SmaKi Prediction Application is configurabel by config files and it produces some data files. A description of these files is given in this section. A list of the used resources and libraries as well as examples of the config files are shown in the Appendix.

`prConfig.xml`
: Important constants of the application are summarized in the class `Constants`. At startup the application tries to open the file `prConfig.xml` to override the constants in this class, hence these are not really constants, however, once set, they are never changed. The file has to be located at the root directory of the application.

`SmaKiItems.xml`
: This is the default filename of the static items which are displayed at the Building Visualization. This items can be illustrated as simple rectangels or as images.

`MDConfig.xml`
: Config file for the *MessageDistributor* tool; it has to be located where the MessageDistributor is started (if it is used).

`Graph Generation`
: For the generation of the graph, two files are built. One contains the specification of the graph, the file name is `input.txt`. The other one, which is generated by the Grapgviz-tool 'dot.exe' is named `out.dot`.

`HMM Files`
: The default global HMM is loaded from the file `HMM.xml` at startup. When the application is closed, the actual global HMM is saved to the file `HMM.mod.xml`. The global HMM is changed if some merging is applied. The Chains which are learned are saved to files with the name `LearModel_suffix.xml`, where *suffix* is the current system time in milli seconds since 1^{st} January 1970. This is done to have unique file names which can be ordered.

7 Results and Discussion

The aim of this work was to build a system for behavior recognition and prediction in building automation systems with the use of the theory about Hidden Markov Models (HMM). The creation of HMMs from scratch is not a straight forward task and there exists no universal solution because this is application dependent. In this work HMMs are created from sample data which results in very specific models. These models are generalized by merging of states. For this task several approaches are tested.

The second part of this work is a software application for a real-time prediction, which is called *SmaKi Prediction Application*. This software receives sensor values from the SmaKi's observation hardware by a TCP/IP connection and shows the values on a screen. Furthermore, an estimation of the currently present persons and their positions is calculated and visualized. For each recognized person, the system calculates a prediction of its next action and displays it on a screen. Last but not least the underlying HMM can be visualized as a directed graph.

Section 7.1 shows and describes the results of the room's visualization. Section 7.2 depicts the results of some merging strategies by showing the corresponding graphs. In Section 7.3 it is described how the quality of the predictions of different HMMs can be compared. Finally Section 7.4 gives an outlook how the system could be improved.

7.1 Visualization of Prediction

This section describes the visualization of the SmaKi's room. A description of the static layout is given in Section 2.1.1 at Figure 2.3. Figure 7.1 shows a screenshot of the application of a situation where two persons are recognized. The depicted situation can be interpreted as follows. At the top of the figure there are three circles. An exact comparison to the layout description reveals that the filled circle in the center is not mentioned. The reason is that this is not the representation of a movement sensor like the other (small) circles, but the heartbeat indicator. At the original application the circles can be distinguished by their color. The heartbeat indicator shows the user that the system is running. If it doesn't change its value periodically, the system has crashed. The left circle is filled, this indicates that this movement sensor has triggered. The unfilled right circle indicates that this sensor has not triggered, respectively.

For better readability some details are depicted in Figure 7.2. The black filled rectangles in Subfigure 7.2(a) indicate that these tactile sensors have triggered. The bright square shows an object to which these sensors are assigned, so the object is located at the center of these two

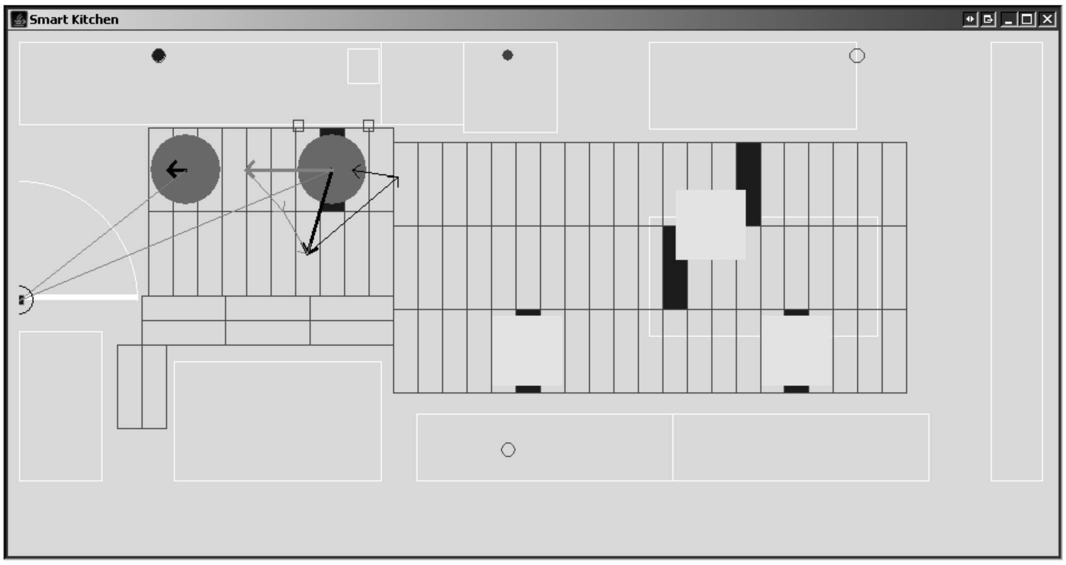


Figure 7.1: Screenshot of the Smart Kitchen's Layout

sensors. The Subfigure 7.2(b) shows a person (the big gray filled circle) where one triggered tactile sensor is assigned to. The arrows indicate the predictions. The heavy arrows point to the location, where the person is estimated at the next step. The darker arrow represents the prediction with the higher probability. As can be seen, there are thin arrows starting at the head of the heavy arrows. These are the predictions for a second and a third future step. Subfigure 7.2(c) shows a person where no tactile sensor is assigned to, i.e. an *orphan.* The thin line to the (triggered) door switch sensor indicates the prediction that the person will activate this switch at the next step. As can be seen, there are no more tactile sensors at the left side of the person and the single arrow which predicts a position, points to the most left tactile sensor's center. This altogether can be interpreted that the person is currently leaving the room and is at a location that cannot be recognized by the system, this includes that the person already may have left the room. The *orphan* will disappear within a few seconds. Subfigure 7.2(d) shows at the left side a person which becomes an item. This is shown by the almost transparent square around the circle. At the right side, there is a person where this process is almost completed; it already looks like an item, however, there are still predictions; items do not have any predictions.

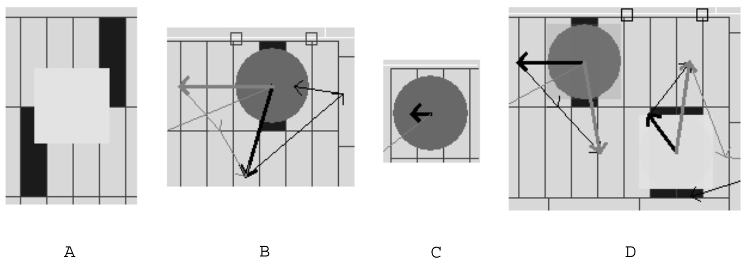


Figure 7.2: Detail View: A: Item; B: Person with Predictions; C: Orphan; D: Person Morphing to Item

7.2 Graph Drawing

The automated visualization of graphs, called *Graph Drawing*, is a great field in computer science. It makes no sense to reinvent the wheel, so a third party tool, called *Graphviz* was used to accomplish the graph drawing task. *Graphviz* provides a binding to the Java programming language, called *Grappa*. Figure 7.3 shows the graph representation of an HMM which is already merged. Except the method of merging sequences (described in Section 4.4.7) all merging strategies are applied on the shown HMM. The numbers inside the states are the labels of them. When the program is running it is possible to get the probabilities of the transitions and emissions by pointing with the mouse cursor to a transition or a state, respectively. Then the probability is shown as a tool-tip. Furthermore, it can be seen that the chain structure of the learned models is preserved. The effects of vertically merging can be seen at the start and at the end. One splitting which is a result of vertically merging occurs at state 92 (left side of figure). At state 89 (right side of figure) a conflation occurs, which results from vertically merging from the end state.

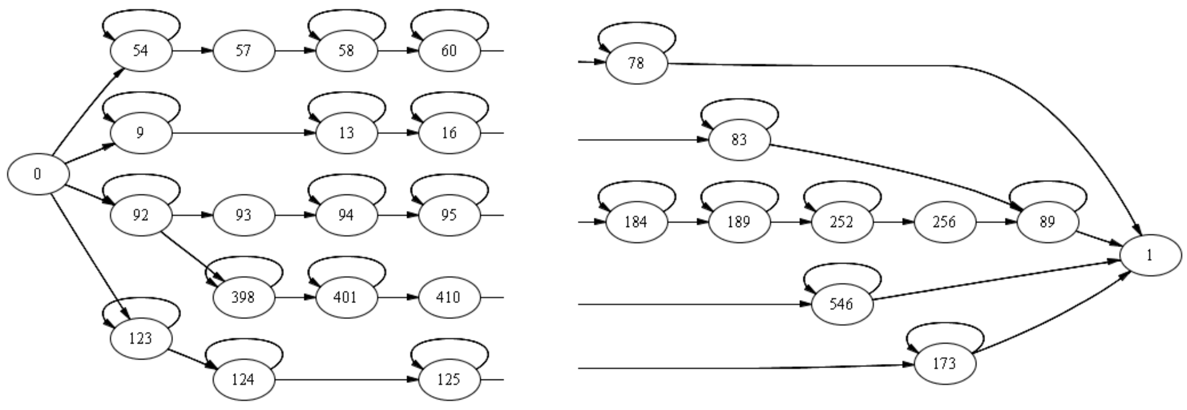


Figure 7.3: Parts of a Merged HMM, drawn with Graphviz and Grappa

As can be seen in Figure 7.3, all transitions point from left to right (except the self transitions), i.e. there is no possibility to come back to a previous state. So we have a time order from left to right at the model. If we apply the sequence merging algorithm on a HMM, this order is not given any more as well as the chain structure is disbanded. Such a model is shown in Figure 7.4. As can be seen, there is a *Backtransition* from state 81 to state 53.

7.3 Verification of Predictions

The creation of HMMs is done by learning from sample data. The so created HMMs are generalized by merging of states. The decision which states to merge, is a crucial task. Several strategies are supposed. These strategies have to be applied in the correct order. To make the resulting HMMs comparable, a kind of evaluation function is needed. The verification process is limited to the predicted position, since there are too little other sensors at the SmaKi to make meaningful statements about the accuracy of the prediction. To make predictions of positions comparable,

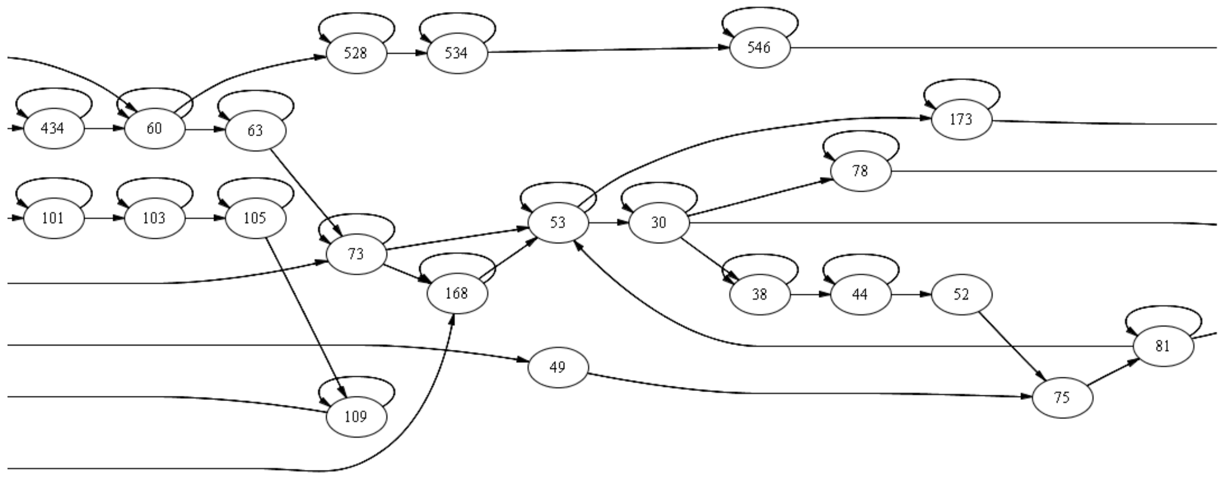


Figure 7.4: Part of a Fully Merged HMM with Backtransitions, drawn with Graphviz and Grappa

the distance between the predicted position and the real position at the next step is calculated. These distances can be compared, smaller values are from more exact predictions.

To make the distances of different models comparable, the same sample data was used to create the models. After this initial procedure, the different merging strategies are applied to the model. Then the same scenario is tested with each model. The test scenario is quite simple: A person enters the empty room (i.e. there are no other persons), walks to the fridge, opens the fridge, closes the fridge, goes on to the server cabinet (where the screen with the visualization is located), turns around and leaves the room. The results are shown in the following.

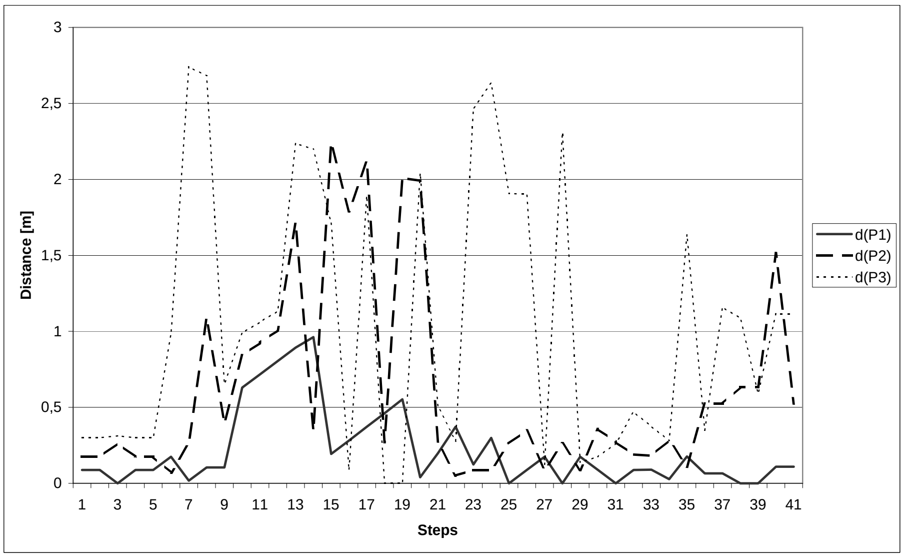


Figure 7.5: Evaluation of a Horizontally Merged HMM

Figure 7.5 shows the evaluation of a horizontally merged HMM. At the x-axis there is the count of steps, at the y-axis the distance as mentioned above. The series d(P1) represents the prediction with the highest probability, series d(P2) the second highest and d(P3) the third one. Before we start the interpretation of this diagram, we should remember the tactile sensors: They have a size of 600 x 175 mm, so all distances below 0.6 m are only one sensor-size away (in the dimension of the lower resolution).

As can be seen, d(P1) is only from step 10 to 14 above the value of 0.6 m, however, d(P2) has lots of higher deviations. Finally d(P3) has still higher deviations than d(P2). This is what we expected: The prediction with the highest probability gives the best estimations.

If we compare this to Figure 7.6, which shows the evaluation of a *fully* merged HMM, it can be seen that the predictions get better. The term *fully* means that merging of sequences, like described in Section 4.4.7, is also applied. The vertically merging, which is the next step after the horizontal merging is not mentioned here, since there are only very little differences for this example. The reason therefore is that little sample data was used. However, for models with several hundreds of states, this would be an important task.

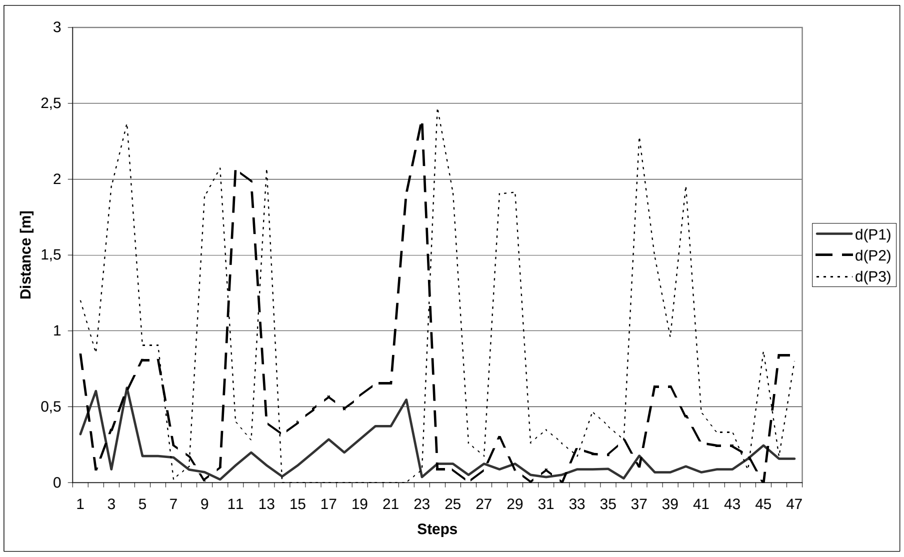


Figure 7.6: Evaluation of a Fully Merged HMM

Figure 7.7 shows another *fully* merged HMM. It cannot be compared to the others directly, since it is learned from other sample data as well as the tested scenario was another one. This evaluation is shown to describe things which have not occurred in the other example. At step 16 the distance of d(P1) gets higher than the distance of d(P2) and remains higher until step 34. This can be interpreted as follows: At step 16 the person had two alternative ways. Here the second most probable path was taken. There is no intersection between the two alternative paths until state 34. The other point of interest in this figure is the lack in d(P3) between step 8 and 13. This indicates that the system could not calculate a third prediction of positions. The reason could be that there are only two predictions possible, but this is not very likely. The system calculates at most six predictions, where *all* types of predictions are included, i.e. additionally to position predictions these are movement sensor predictions and switch sensor predictions.

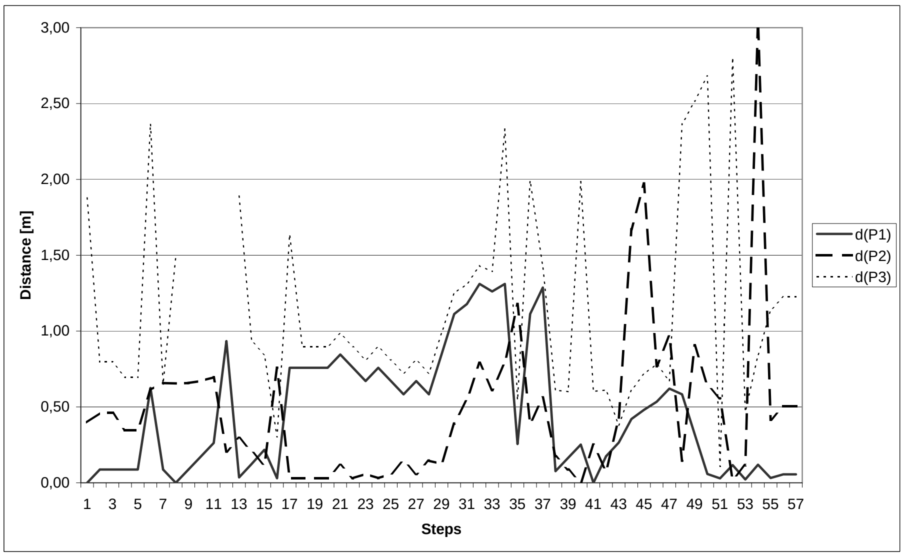


Figure 7.7: Evaluation of another Fully Merged HMM

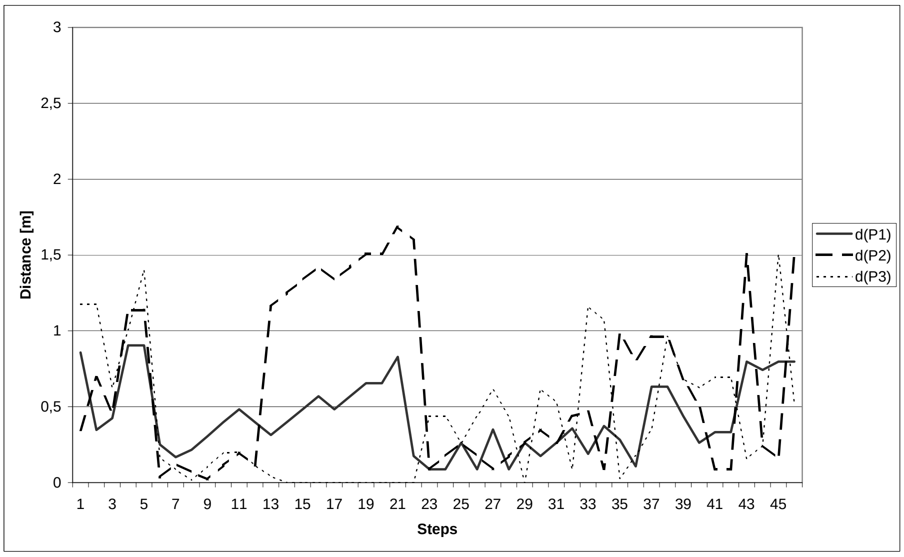


Figure 7.8: Evaluation of the Second Step Prediction

As the system provides to look more than one step into the future, these predictions are evaluated as well. Figure 7.8 shows the distances for a second future step. The underlying model which was used for this example, is the same like in the Figures 7.5 and 7.6. Finally Figure 7.9 shows the distances when making three steps into the future. It can be seen that all distances are longer. So for a third future step the predictions are unexact.

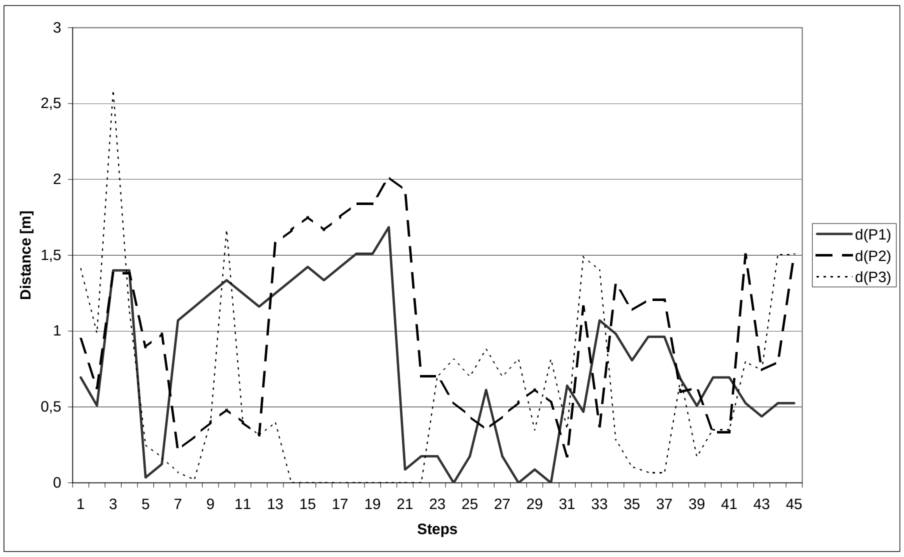


Figure 7.9: Evaluation of the Third Step Prediction

7.4 Outlook

The represented model works quite well for the intended purpose. However, it has a weak time representation. The occurrence of events is ordered, but the duration between two occurring events is not evaluated. Consider the following two scenarios: Scenario one, a person enters the room, goes to the fridge, opens it, closes it and leaves the room. Scenario two, a person enters the room, goes to the fridge, opens it, has a look at it, does nothing for a while, closes the fridge and leaves the room.

With the simple time model of this system, these scenarios will be identified as the same, because the time span where nothing happened is not taken into account. To provide a better representation of time, the standard approach of HMMs is not sufficient. In [Bru07] a system of a higher order model, the *Hidden Semi Markov Model* is described. Such models could be used to overcome the weak modeling of time within the standard HMMs. So the system should be able to distinguish better between persons who trigger tactile floor sensors and objects doing this.

A system with a higher resolution of the tactile floor sensors would be able to define the positions of persons and objects more exactly. The currently installed sensors have a size of 600 x 175 mm. Since a person can be located between two adjacent sensors, the circumcircle of a person raises to 1200 mm. At such an area there can be several persons, this makes it difficult to make a good estimation of the number of persons in the room. If the area near the door would be equipped with tactile sensors, the system would be able to detect a scenario of leaving the room more precisely.

The status of the door is indicated by one switch. If the door is open, the system cannot recognize how many persons are arriving or leaving. Even better would be a light barrier, but still not optimal. However, an optimal person counter, like a security turnstile, would be an overkill and not accepted by the users.

The person model described in 4.2 could be improved by applying a physical model to it. Moreover the implementation of (person) tracking algorithms could overcome the problem that persons could disappear or make jerky leaps.

A more sophisticated integration of the movement detection sensors could improve the distinction between persons and items. The current system can recognize a very calm person as an item. This can be eliminated in some cases, e.g. if a movement detection sensor triggers and there is only one person present in its sphere of action.

If such a system is equipped with pressure detection floor sensors it would be possible to determine the number of persons very exact. However, such a system might be too exact for many applications: If the sensor values are stored in a database, as it is possible within this project, it might be possible to draw conclusions which person has entered the room. Even worse, if the habit of a person is known (e.g. time of arriving in the morning), such a system would make it possible to determine that person's weight.

It becomes apparent that more precise systems can reach a level of performance which even might be rejected by the affected people. For this reason it is necessary to find the balance between a surveillance system which violates our privacy and a supporting system making life more comfortable.

A Appendix

Config Files

This chapter shows examples of the config files which are used by the SmaKi Prediction Application. They are located at the application's root directory.

prConfig.xml

This is an example of the main config file of the application. The values of the database connection need to be adapted.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<PredictionConfiguration>
  <Database Type="Microsoft">
    <Connection Server="128.131.80.10" Port="1433" Sid="ARS" User="josef" Pass="josef1234"/>
  </Database>
  <GUI>
  <!-- names of colors are specified in java class Color -->
  <!-- "[r,g,b,a]" color by rgb values, 4th is alpha value  -->
    <MovementSensor Diameter="0.1" ActiveColor="yellow" InactiveColor="red"/>
    <SwitchSensor Diameter="0.075" ActiveColor="yellow" InactiveColor="red"/>
    <TactileSensor ActiveColor="yellow" InactiveColor="light_gray"/>
    <AnimationDelay Fast_ms="40" Slow_ms="200" Powersave_ms = "4000"/>
    <AnimationWait Slow_s="15" Powersave_s="300"/>
    <HeartbeatSymbol Show="true" PosX_mm="100" PosY_mm="3500" Diameter_mm="75"/>
    <SplashScreen Time_s="0"/>
    <!-- <HeartbeatSymbol PosX_mm="10" PosY_mm="10" Diameter_mm="5"/> -->
    <StaticItems Filename="SmaKiItems.xml"/>
    <Prediction MaxPredictionEntities="6" FutureSteps="3">
      <Arrow_1 HiColor="red" LoColor="[255,128,128,192]" HiStroke="7" LoStroke="5"/>
      <Arrow_2 HiColor="red" LoColor="[255,128,128,128]" HiStroke="3" LoStroke="1"/>
      <Arrow_3 HiColor="red" LoColor="[128,128,128,128]" HiStroke="1" LoStroke="1"/>
    </Prediction>
  </GUI>
  <Livedata KeepAliveMessageIntervall_ms="200">
    <LiveDataReader MDIP="128.131.80.119" MDPort="65523" ConnectionPort="12346"/>
    <Person NewPersonDistance_mm="1000"/>
    <PersonTiming TimeToStartDisappear_0_ms="4000" TimeToStartDisappear_1_ms="8000"
                  DisDeferment_0="20" DisDeferment_1="100"/>
  </Livedata>
  <DataFiles Path="data">
    <VerificationFile Write="true" Path="verification" PrefixName="Verification" Extension="txt"/>
    <LearnmodelFile Write="true" Path="learnmodel" PrefixName="Learnmodel" Extension="xml"/>
  </DataFiles>
  <Editor name="C:\Program Files\PSPad editor\PSPad"/>
</PredictionConfiguration>
```

SmaKiItems.xml

This is an example of the file which stores the data of the static items in the SmaKi layout.

```xml
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<SmartKitchenItems>
  <!-- Attention: If ShowImages is true, the Application would require much more HeapSpace -->
  <!-- Start with appropriate option to avoid Exceptions -->
  <StaticItems ShowImages="false" ImagesPath="D:\\DA\\prj\\resources\\" Color="black">
      <!-- Name is not optional, except Boundary: it is invisible -->
      <Rectangle Name="table" x1="1.258" y1="4.514" x2="2.109" y2="6.142" ImageFile="table1_big.jpg"/>
      <Rectangle Name="cupboard" x1="2.072" y1="0.0" x2="3.145" y2="0.592" ImageFile="cupboard1_big.jpg"/>
      <Rectangle Name="copier" x1="2.294" y1="1.110" x2="3.145" y2="2.590" ImageFile="copier1_big.jpg"/>
      <Rectangle Name="bookshelf1" x1="2.664" y1="2.849" x2="3.145" y2="4.68" ImageFile="bookshelf1_big.jpg"/>
      <Rectangle Name="bookshelf2" x1="2.664" y1="4.68" x2="3.145" y2="6.512" ImageFile="bookshelf2_big.jpg"/>
      <Rectangle Name="shelf" x1="0.0" y1="6.956" x2="3.145" y2="7.326" ImageFile="shelf1_big.jpg"/>
      <Rectangle Name="desk" x1="0.0" y1="4.514" x2="0.629" y2="5.994" ImageFile="desk1_big.jpg"/>
      <Rectangle Name="servcab" x1="0.0" y1="3.182" x2="0.65" y2="3.848" ImageFile="servcab1_big.jpg"/>
      <Rectangle Name="kitchenette" x1="0.0" y1="0.0" x2="0.592" y2="3.182" ImageFile="kitchenette1_big.jpg"/>
      <Rectangle Name="fridge" x1="0.0" y1="2.590" x2="0.592" y2="3.182" ImageFile="fridge1_big.jpg"/>
      <Rectangle Name="coffee" x1="0.05" y1="2.35" x2="0.295" y2="2.58" ImageFile="coffee1_big.jpg"/>
      <Rectangle Name="Boundary" x1="0.0" y1="0.0" x2="3.18" y2="7.36" ImageFile=""/>
  </StaticItems>
</SmartKitchenItems>
```

MDConfig.xml

If the *Message Distributor* is used, it can be configured by this file. The `Port` specified at node `Clients`, is the same as the `MDPort` in node `LiveDataReader` of the file `prConfig.xml`.

```xml
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<MessageDistributorConfiguration>
  <Octobus Port="59999"/>
  <Clients Port="65523"/>
  <Andi IP="128.131.80.119" Port="65000"/>
</MessageDistributorConfiguration>
```

Icons and Libraries

The application uses some icons, one image and some libraries which are listed here, together with their corresponding directories.

```
.\resources\s_ki_klein1.gif
.\resources\smki.GIF
.\resources\SplashScreen.png

.\lib\grappa1_2.jar
.\lib\ojdbc14.jar
.\lib\sqljdbs.jar
```

B Appendix

List of Classes

Here is shown a list of classes which are necessary to run the SmaKi Prediction Application. Each class is stored in a file with the name `Classname.java`.

```
Package base

Heartbeat
MainController
MainModel
MovementSensor
MovementSensorList
PredictionEntity
PredictionProcessor
Sensor
SensorList
SensorValue
SensorValueKeepAlive
SensorValueMovement
SensorValueSwitch
SensorValueTactile
SwitchSensor
TactileSensor
TactileSensorList
VerificationModel
WorldPerson
WorldPersonList


Package data

BinaryIDSymbol
BinaryIDSymbolList
DataAbstractionProvider
EmissionSymbolList
IDataAbstractionLayer
MovementSensor
Position
PositionSymbol
PositionSymbolList
SwitchSymbol
```

```
Package gui

BArrow
BDoor
BHearbeat
BImage
BMovementSensor
BObject
BPerson
BPredictionEntity
BRectangle
BSwitchSensor
BTactileSensor
BuildingController
BuildingModel
BuildingObjectFactory
BuildingView
BuildingViewPanel
DialogHelpAbout
DialogMerge
GEdge
GObject
GraphController
GraphModel
GraphView
GState
MainFrame
TactileAlignment
```

```
Package markov

Algorithms
Emission
HiddenMarkovModel
HMMComparator
HMMState
HMMTransition
IEmissionSymbol
MergeProvider
PositionEmissionComparator
TactileDistanceComparator


Package pers

DataAccessor
IDatabaseReader
IQueryStringProvider
LiveDataReader
MicrosoftDatabaseReader
MicrosoftQueryStringProvider
OracleDatabaseReader
OracleQueryStringProvider
ReaderThread
SensorFactory


Package utils

Constants
MyMath
```

Literature

[AIS+77] ALEXANDER, Christopher ; ISHIKAWA, Sara ; SILVERSTEIN, Murray ; JACOBSON, Max ; FIKSDAHL-KING, Ingrid ; ANGEL, Shlomo: *A Pattern Language.* Oxford University Press, New York, p. X, 1977 56

[BK96] BARON, Gerd ; KIRSCHENHOFER, Peter: *Einführung in die Mathematik für Informatiker, Band 3, Zweite, verbesserte Auflage.* Springer-Verlag, Wien New York, p. 149ff, 1996 19

[BPSW70] BAUM, L.E. ; PETRIE, T. ; SOULES, G. ; WEISS, N.: A maximization technique occuring in the statistical analysis of probabilistic functions in Markov chains. In: *The Annals of Mathematical Statistics 41(1)*, 1970, p. 164–171 37

[Bru07] BRUCKNER, Dietmar: *Probabilistic Models in Building Automation: Recognizing Scenarios with Statistical Methods*, Vienna University of Technology, Institute of Computer Technology, Diss., 2007 2, 39, 41, 91

[Die00] DIETRICH, Dietmar: Evolution potentials for fieldbus systems. In: *IEEE Int. Workshop on Factory Communication Systems WFCS 2000*, 2000 1

[DLP+06] DEUTSCH, Tobias ; LANG, Roland ; PRATL, Gerhard ; BRAININ, Elisabeth ; TEICHER, Samy: Applying Psychoanalytic and Neuro-Scientific Models to Automation. In: *Institute of Computer Technology at Vienna University of Technology, Wiener Psychoanalytische Vereinigung, Sigmund Freud Institut Wien, Anton Proksch Institut Klinikum*, 2006 2

[Doo96] DOOB, J. L.: The Development of Rigor in Mathematical Probability (1900-1950). In: *Amer. Math. Monthly 103*, 1996, p. 586–595 17

[DR06] DAHMEN, Wolgang ; REUSKEN, Arnold: *Numerik fuer Ingenieure und Naturwissenschaftler.* Springer-Verlag, Berlin Heidelberg New York, p.39, 2006 39

[Dub00] DUBUISSON, Olivier: *ASN.1 - Communication between heterogeneous systems.* Morgan Kaufmann Publishers, 2000 15

[Fre23] FREUD, Sigmund: The ego and the id. In: *The Standard Edition of the Complete Psychological Works of Sigmund Freud*, 1923, p. 12–66 2

[Goe06] Goetzinger, Sigfried: *Scenario Recognition Based on a Bionic Model for Multi-Level Symbolization*, Vienna University of Technology, Institute of Computer Technology, Diplomarbeit, 2006 7, 15, 32

[GrH07] *McGraw-Hill encyclopedia of science and technology - 10th ed.* McGraw-Hill Companies Inc., USA, 2007 11

[Hol06] Holleis, Edgar: *SymbolNet - Ein Application Framework für symbolische Kommunikation.* 2006. – ICT, TU Wien 15

[IEE85] IEEE Standard for Binary Floating-Point Arithmetic. In: *ANSI/IEEE Std 754 1985*, 1985 77

[Kai99] Kaindl, Hermann: Difficulties in the transition from OO analysis to design. In: *IEEE Software*, 1999, p. 94–102 51

[Knu97] Knuth, Donald: *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition.* Addison-Wesley, pp.107-123, 1997 36

[Knu99] Knudsen, Jonathan: *Java 2D Graphics.* O'Reilly, Sebastopol (USA), chap. 14, 1999 58

[Kop97] Kopetz, Hermann: *Real-time systems.* Kluwer, Boston, 1997 3, 58

[KP88] Krasner, Glenn E. ; Pope, Stephen T.: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. In: *Journal of Object-Oriented Programming*, 1988, p. 26–49 56

[Kru00] Krueger, Guido: *Go To Java 2, Zweite Auflage.* Addison Wesley, Germany, chap. 35, 2000 57

[KWE01] Kaufmann, Michael ; Wagner, Dorothea ; (Eds.): *Drawing Graphs, Methods and Models.* Springer-Verlag, Berlin Heidelberg New York, chap. 1, 2001 59

[LEW$^+$02] Loy, Marc ; Eckstein, Robert ; Wood, Dave ; Elliott, James ; Cole, Brian: *Java Swing 2nd Edition.* O'Reilly, Sebastopol (USA), p. 13, 2002 57

[LL08] Liguori, Robert ; Liguori, Patricia: *Java Pocket Guide.* O'Reilly, Sebastopol (USA), chap. 1, 2008 50, 65

[Pap84] Papoulis, A.: *Probability, Random Variables, and Stochastic Processes, 2nd ed.* New York: McGraw-Hill, p. 532, 1984 18

[PP05] Pratl, Gerhard ; Palensky, Peter: Project ARS - The next step towards an intelligent environment. In: *Vienna University of Technology, Institute of Computer Technology*, 2005 2

[Pra06] Pratl, Gerhard: *Processing and Symbolization of Ambient Sensor Data*, Vienna University of Technology, Institute of Computer Technology, Diss., 2006 15

[RJ86] Rabiner, Lawrence R. ; Juang, Biing-Hwang: An Introduction to Hidden Markov Models. In: *IEEE ASSAP Magazine 3* (1986), January 18, 20, 21, 22

[Rus03] Russ, Gerhard: *Situation-dependent Behavior in Building Automation*, Vienna University of Technology, Diss., 2003 1

[SBR05a] Sallans, B. ; Bruckner, D. ; Russ, G.: Statistical Detection of Alarm Conditions in Building Automation Systems. In: *Proceedings of the 5th IEEE International Conference on Industrial Informatics*, 2005 2

[SBR05b] Sallans, B. ; Bruckner, D. ; Russ, G.: Statistical Model-based Sensor Diagnostics for Automation Systems. In: *Proceedings of the 6th IFAC International Conference on Fieldbus Systems and their Applications*, 2005 2

[Sed98] Sedgewick, Robert: *Algorithms in C.* Addison-Wesley, pp.90-109, 1998 36

[SO93] Stolcke, Andreas ; Omohundro, Stephen: Hidden Markov Model Induction by Bayesian Model Merging. In: *Hanson, Stephen J. ; Cowan, Jack D. ; Giles, C. L. (eds.): Advances in Neural Information Processing Systems Bd. 5*, Morgan Kaufmann, San Mateo, 1993, p. 11–18 37

[SRT00] Soucek, S. ; Russ, G. ; Tamarit, C.: The Smart Kitchen Project - An Application on Fieldbus Technology to Domotics. In: *Proceedings of the 2nd International Workshop on Networked Appliances (IWNA2000)*, 2000, p. 1 2, 5

[ST02] Solms, Mark ; Turnbull, Oliver: *The Brain and the Inner World.* Karnac/Other Press, Cathy Miller Foreign Rights Agency, London, England, p. 152ff, 2002 28

[Wei91] Weiser, M.: The Computer for the 21st Century. In: *Scientific American*, 1991, p. 66–75 1

[Wei99] Weisstein, Eric W.: *CRC Concise Encyclopedia of Mathematics.* Chapman and Hall / CRS, Boca Raton, London, New York, Washington p.1139, 1999 18

Internet References

[1] *Graphviz - Graph Visualization Software*, September 2008. http://www.graphviz.org/.

[2] AT&T Labs. *Grappa - A Java Graph Package*, September 2008. http://www.research.att.com/~john/Grappa/.

[3] CloudGarden. *Jigloo*, September 2008. http://www.cloudgarden.com/jigloo/.

[4] Committee. *15th Graph Drawing Contest*, September 2008. http://www.graphdrawing.de/contest2008/.

[5] Committee. *graphdrawing.org*, September 2008. http://graphdrawing.org.

[6] eclipse.org. *eclipse*, September 2008. http://www.eclipse.org/.

[7] Institute of Computer Technology. *Homepage*, September 2008. http://www.ict.tuwien.ac.at.

[8] Project ARS, Institute of Computer Technology. *ARS - Artificial Recognition System*, September 2008. http://ars.ict.tuwien.ac.at/.

[9] sourceforge.net. *iSQL-Viewer*, September 2008. http://sourceforge.net/projects/isql/.