diploma thesis

Adaptive Building Intelligence

Parallel Fuzzy Controlling and Learning Architecture Based on a Temporary and Long-Term Memory

Jonas Trindler
<trindler@ini.phys.ethz.ch>

Raphael Zwiker
<rzwiker@ini.phys.ethz.ch>

Advisors
Prof. Dr. Rodney Douglas, Institute of Neuroinformatics, ETH/University Zurich
Prof. Dr. Josef Joller, University of Applied Sciences Rapperswil

A cooperation between

HSR
UNIVERSITY OF APPLIED SCIENCES
RAPPERSWIL

COMPUTER SCIENCE

uni | eth | zürich

Computer Science Department
University of Applied Science Rapperswil
Oberseestrasse 10
8640 Rapperswil, Switzerland
http://www.hsr.ch

Institute of Neuroinformatics
University and ETH Zurich
Winterthurstrasse 190
8057 Zurich, Switzerland
http://www.ini.unizh.ch

December 15, 2003

Typeset by LaTeX

Abstract

Contemporary approaches to the architecture of living and working environments emphasize the simple reconfiguration of space to meet the needs, comfort and preferences of its inhabitants and to minimize the consumption of resources such as power. The configuration can be explicitly specified by a human building manager, but there is now increasing interest in the development of intelligent buildings equipped with standard sensors (e.g. presence, temperature, illumination, humidity) and effectors (e.g. lights, window blinds, wall-switches) that adapt to the needs of its inhabitants without human intervention. Occupants of a building can move, come or leave within short time and therefore also their individual desires. We regard inhabitants of a building as part of the structure in which they are working or living. The structure itself is a non-stationary environment where not only individual desires are constantly changing but also the physical structure (e.g. mobile walls in multi-office environments, integration or dismounting of devices).

We present in this paper an approach that is able to learn from non-explicit feedback and control a building with a collection of small controllers that act in sub-parts of the structure. A specifically designed controlling and learning architecture takes decisions in parallel based on a short- and long-term memory. A temporary controller implements a one-shot learning algorithm that learns online and unsupervised from temporary behaviors of the building's users. A long-term memory is managed by a memory-based controller that uses an evolutionary algorithm to learn an optimal set of fuzzy logic rules. The genetic encoded fuzzy rulebase is evolved and learnt with an iterative rule learning approach in collaboration with a fuzzy-variant adaptive boosting algorithm.

Furthermore, we introduce a novel multi-agent approach which is able to administer a dynamic structure where instances (e.g. devices, occupants) can change their location. Other services are providing an asynchronous interest-based messaging between different agents or preventing data losses with a reliable recovery system. Our approach is able to handle different types of multi-sensor environments simultaneously.

Further documentation, the API reference documentation and the sourcecode can be found at:
`http://www.trjo.ch/projects/abi/`.

Table of Contents

List of Figures

List of Tables

Part I

Introduction and Description of the Problem

Chapter 1

Introduction

1.1 Overview

Current research is concerning about the integration of autonomous intelligence into our everyday life. Many attempts are trying to interact, improve user comfort and provide security in modern working and living environments. We present in this paper a novel software framework that is able to administer, manage and control a typical commercial office building. It acts with the environment through common devices like lights, window blinds etc. and senses from illumination-, temperature-, radiation-, daylight-sensors and presence detectors. The integration of any other devices is possible (mobile devices, virtual computer network or household aids).

The framework serves as a fundamental component for any further analysis of a complex and non-stationary environment. It features numerous highly flexible, reliable and general services to access and control devices of a building on an abstact level.

Traditional building control requires a manual specification of rules that describe actions which have to be executed at specific time and sensor conditions. Although it is possible for a human building manager to specify a configuration explicitly, the size, sophistication and dynamic requirements of a modern building demands that they have autonomous intelligence which could satisfy the needs of its inhabitants without human intervention. We describe in this paper our approach to apply such an autonomous intelligence to a typical commercial office building.

Modern buildings are changing their nature from static structures to dynamic working and living environments that can actively support and assist their inhabitants. Occupants can move, come or leave within short time and large office environments are often under dynamic reconfiguration. More and more, such multi office environments are shared among different users and mobile walls are arranged to set boundaries between different working places. Traditional wall switches become mobile and are set-up based on the current structure configuration that implies also a frequent reconfiguring of the links to their effecting devices.

We present here an approach which is able to learn in such a dynamic environment from non-explicit feedback of users. There is no special interaction mechanism with which users can communicate with the controlling system. Occupants of a building should not notice that the building they are in, is actually intelligent which means that he can only interact indirectly through common devices (e.g. wall switch, presence detector).

Further, we tackle interesting challenges in such a non-stationary environment with a novel specifically developed learning and controlling architecture. Local controllers are acting and learning non-supervised in a sub-part of the whole structure and take decisions based on two different memories. One part of this controller is learning with a one-shot learning algorithm a temporary knowledge (short-term-memory). It is able to learn a new or unusual behavior immediately (e.g. a temporary new user in a room). But on the other side a long-term memory learns conditions which were repeated or occur frequently. The verification and transition of such higher-level information occurs online and can therefore change also dynamically the long-term memory.

An evolutionary algorithm, which is used in collaboration with a fuzzy-variant boosting algorithm, learns as few and as general as possible fuzzy logic rules. An iterative learning approach activates the evolution of an given rulebase. The fittest solution after each evolution will be part of the new ruleset before the adaptive boosting algorithm changes the relative distribution of the long-term memory in order to focus the genetic search on still uncovered areas. The whole evolving of the rulebase can be divided into two phases: a generation and afterwards in a refining phase.

1.2 Content

This document is structured into seven parts: *Introduction* (part I), *Learning and Controlling* (part II), *Architecture, Design and Implementation* (part III), *Results and Discussion* (part IV), *Conclusion and Future Works* (part V), *Appendix* (part VI) and the *Glossary and Bibliography* (part VII).

The first part (I) gives a general introduction in the problem of making an building really intelligent and we are discussing the aims of our project. We introduce in chapter 1 the conclusions of previous works of the *Adaptive Building Intelligence*. We mention related works which are also dealing with intelligent buildings and show the differences to our project. The chapter motivation (2) illustrates what interesting challenges in an intelligent building and gives reasons why it is worth to apply an ambient intelligence into everydays life.

Issues and challenges in a non-stationary and complex environment like a typical building is, are described in chapter 4. We propose our approach with a parallel fuzzy control and learning architecture, based on a temporary and a memory-based learning unit in chapter 5. The process of knowledge transition is explained in chapter 5 as well. The detailed temporary learning algorithm is explained in chapter 6. Based on the long-term memory we use an evolutionary learning algorithm (see chapter 7) to get an optimal fuzzy rulebase. The chapter 8 deals with the delayed behavior of specific devices within a building and explains our solutions to tackle these problems.

The part III gives a detailed overview of the whole software architecture and the real implementation of our framework with its multi-agent system. The chapter 9 points out the main requirements of our framework. Section 9.3 documents the advantages of choosing a multi-agent system instead of conventional software architecture. We describe the concept of the multi-agent system's basic services in detail. The actual implementation is described in chapter 10, where we explain major points within each agent as well as their internal class collaboration. The control agent (section 10.4 and 10.5) gives a detailed explanation of the implementation of our novel control and learning architecture.

In part IV, we discuss in chapter 11, 12 and 13 first results of our approach and list our assumptions. As this diploma thesis is very limited in time were were not able to make a detailed analysis after the development and implementation of the whole framework and algorithms. However, we are planning to set up different test cases where we are expecting to see more evidences how good our system performs in terms of user satisfaction.

In the final part V, we conclude our approach (14) and posse issues which have to be fixed in the near future (see chapter 15) in order to deploy our system in large size.

The appendix VI has a brief explanation of the fuzzy logic notation which we use in this paper. The glossary (VII) lists important abbreviations that are often used.

1.3 Related Work

This diploma thesis is mainly a continuation of the *Adaptive Building Intelligence* project and was conducted during the period from October to December 2003 at the Institute of Neuroinformatics, University / ETH

Zurich in Switzerland. After the work of our predecessors which is described in [RS02] and [RJD04] we analyzed firstly the behavior of typical multi-sensor environments [TZR$^+$03] [TZ03a] like the collection of common sensors in buildings are. We proved that the assumptions of our predecessors was sufficient which is saying that states of effectors are only dependent on a subset of all sensors within a building. This makes it possible to control a building with a large collection of small controllers which is also more efficient from a computational point of view and enables a fast and nearly-realtime decision making.

After a long-term analysis of the proposed architecture and learning algorithm in [RJD04] we conclude that the main idea of the architecture itself is an ideal basis to work with. The learning algorithm is however not able to maintain a memory of already learnt knowledge. Rules which have been learnt can be rejected with only one new instruction although they have been learnt and rewarded by many samples before.

We implemented therefore a new framework which is able to cope with a dynamic structure and manage different input systems (multiple fieldbuses, computer networks, etc.) simultaneously. With an abstraction layer all devices can be processed anywhere in the software without knowing its specific implementation and properties. Furthermore, we extended the system with new goals such as energy reduction and developed a novel learning and controlling architecture.

There are also a variety of other related projects being conducted at other places in similar environments. A group in Sweden [DB00] [BDY99] applies a multi-agent system as well in order to control an intelligent building. Their research focus on the energy consumption reduction but with still satisfying users with a high comfort. Individual preferences indeed has to be programmed in advance and their agents do not learn behavior of inhabitants. However, they could prove that their approach could reduce energy consumption around 40% compared to the same building which was only controlled by its users.

Another group [Moz98] [Moz99] is concerning more about home automation. Their system is able to program itself and maintain traditional elements in a house (light, air- and water-heating, ventilation) by observing the lifestyle and desires of inhabitants. In difference to our work, they use a centralized control architecture.

Perhaps the most similar approach to our work is presented in [HCCC03], where they describe a system with different sensors of a room connected via a sensor network to a single embedded agent that is located physically in this particular room. This agent learns with a genetic learning paradigm fuzzy rules. The learning is indeed supervised and receives explicit feedback from the user with proactive interaction mechanisms.

Robotics can be seen as a related field where research is encountered to more or less similar problems. However, a building has special requirements that need novel solutions because a building completely contains its environment not like a robot which moves trough an environment.

Chapter 2

Motivation

Within this chapter we motivate why it is worth trying to apply an autonomous intelligence in a working or living environment like a typical building. At first glance, it looks maybe like an easy task to control a building by configuring it with some specific rules. Traditional building control systems are able to operate with manually or sometimes also semi-automated rule generation. However, this rule specification is neither convenient nor cost effective. A building has to be regarded as a non-stationary environment where structure changes due to the fact that users move, come and leave more or less continuously. Changes of the physical structure have also to be considered as in nowadays multi office environments mobile walls can be rearranged, devices can be installed or replaced. A manually configured system will not satisfy all these constraints because once it is programmed, the building its structure has perhaps already changed.

A control system demands therefore an autonomous intelligence that could satisfy the needs of the building's inhabitants without explicit human intervention. It must be able to configure itself to the dynamic requirements of different instances: individual users which work or live in the building and the human building mananger.

Preferences and desires of each individual inhabitant can not or only hardly be described with a mathematical model and the collaboration with a building results in a complex environment. The control system has to sense in the environment and react to specific conditions. The occurrence of such conditions is indeed not predictable.

However, we want to show with our approach that it is possible to increase the comfort of individual users and still reduce the energy consumption of the whole building. One major issue is the learning from non-explicit feedback where the user does not notice our system which is running in the background. He has no possibility to communicate directly with the system.

As we have seen in our anaylsis in our last project [TZR$^+$03], it is possible to control individual effectors (such as lights, window blinds) only with a subset of sensors (e.g. daylight, presence signal, outdoor illumination, temperature). This enables us to use a distributed artifical intelligence (DAI) that learns a knowledge on local basis. The sharing of experiences between such local controllers poses many interesting challenges.

Imagine, at one day in future we are living in homes and working in offices which are controlled by such ambient intelligence. They can assist people with the simplest task which is to switch on the light if they enter into a dark room. But the scope of such an intelligent system can reach far beyond this simple scenario. Future intelligent systems are able to identify persons and maintain their individual perferences on which the system can adapt the environment in order to increase the comfort for the inhabitants. People may have different ways to interact and control their surrounding environment with mobile devices such as simple mobile control devices or even small portable computers.

Other scenarios are that an intelligent system can provide security in an office or also living building. It learns how to behave in a emergency case and because it knows where people are it is able for example to support them by guiding them out of the building or by just shutting down all critical elements. Therefore such a system must have a way to asses itself in terms of behavior, error and success.

Althought these visions sound at the moment not contemporary, we believe that the integration of autonomous and self-adapting control system into our daily life will come true in the near future. This work is, like all related projects, a first step towards this vision.

Chapter 3

Access to a Multi-Sensor Environment

In order to do performance analysis of an intelligent system in a real working and living environment, we need access to all sensors and effectors within rooms, floors or even a whole building. This work was conducted at the Institute of Neuroinformatics (INI) where we had access to the fieldbus of the building to communicate with all common sensors and effectors.

This chapter gives a brief overview of the architecture of the building and explains the concept how devices on the fieldbus can be reached from a normal TCP/IP network.

3.1 Architecture of our Intelligent Building

The environment in which our system runs, consits of many rooms with different size and shapes. In some of them only one person is working whereas in others are many. Figure 3.1 illustrates one of the two floors of the Institute of Neuroinformatics where our system is running.

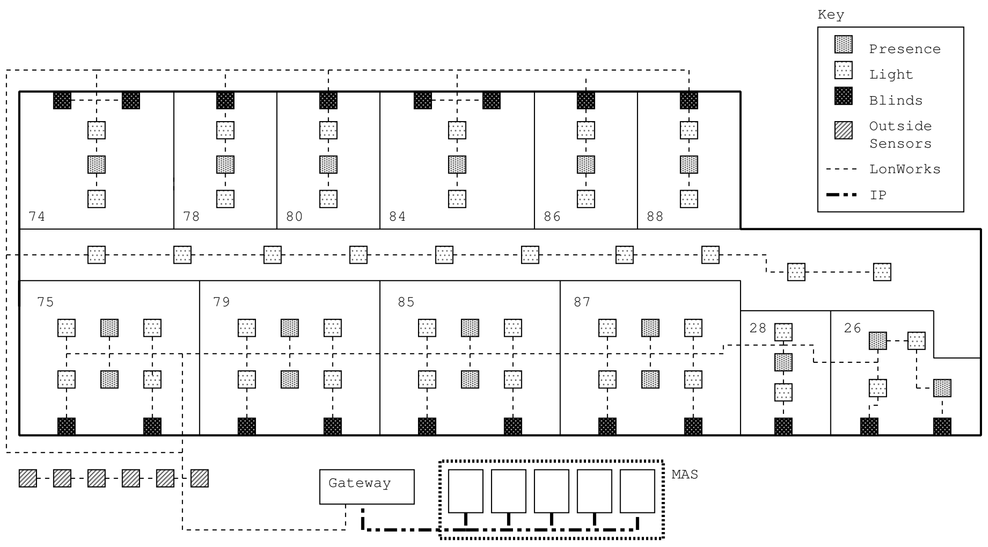


Figure 3.1: The multi-sensor environment within the Institute of Neuroinformatics. Multiple sensors and effectors distributed on different 12 rooms on one floor can be accessed over a fieldbus.

Our system is currently controlling four different rooms on the mentioned floor. But as soon as the testing over a long-term period is successful, we are going to deploy it to other rooms.

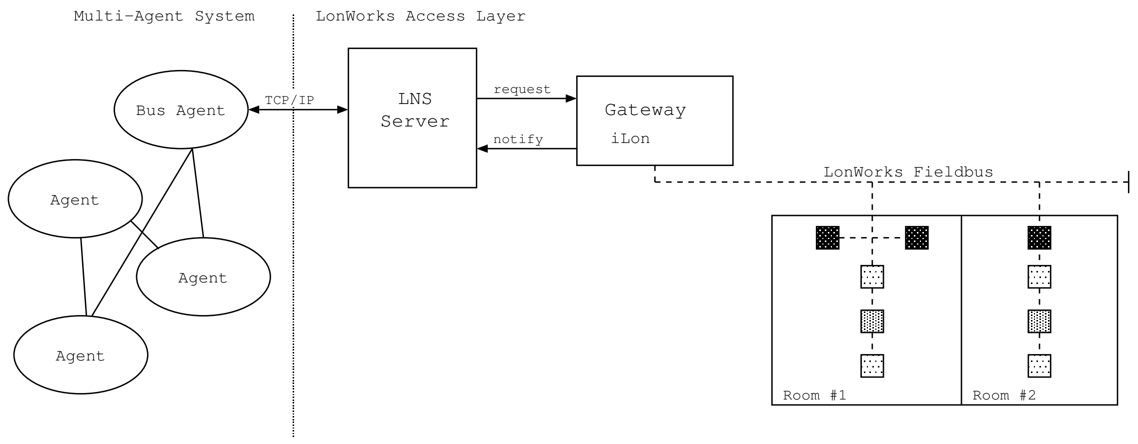


Figure 3.2: LonWorks Gateway iLon which provides the communication with a LNS Server and further this one with the multi agent system.

3.1.1 Properties of Sensors and Effectors

The building where our system is running is equipped with a variety of different sensors and effectors. All of them are common standard devices like in every normal office building and do not have any additional intelligence. Almost all rooms of the building are equipped with wall switches and presence, temperature, illumination and air conditioning sensors. There are also several outside sensors installed around and on top of the building (e.g. temperature, illumination, humidity). Further, each room has different effectors like for example lights, window blinds or air-conditioning.

All of these sensors and effectors have an individual range in which they can sense or act based on the physical constraints. Some of them have a discrete range whereas others have an continuous range like the illumination sensor which is only bound by a upper respectively lower bound. Some of the effectors have also a analog value range but we regard them up to now still as digital devices where we can set them to a discrete number of states.

3.2 Access to Fieldbus LonWorks over a Gateway

All devices are connected together over a fieldbus called LonWorks [loc] which is a product from Echelon. Every device can be set to a specific state by sending it a request message. Obversely, anybody can register itself as a listener to a device and receive an update message as soon as the device changes its state.

Messages on the fieldbus are routed between the different networks by specific LonWork routers. And a central unit has also a set of hard-wired rules that define which sensor can change which effectors (e.g. if wall switch 1 is pressed switch off/on light 2). Therefore it is possible to link all devices in an arbitrarily manner.

We have access to the LonWorks fieldbus via an LonWorks-IP gateway called *iLon* [iLO] that translates messages between the two networks. Therefore it is possible to request and set states of all sensors and effectors from a client in a IP network which uses the mentioned gateway. The *LNS Java API* [LNS] enables us to communicate with a LNS Server which processes and forwards requests or responses to the *iLon* Gateway respectively back to our software.

Part II

Learning and Controlling

Chapter 4

Learning Requirements in a Complex and Dynamic Environment Based on Sparse User Interaction

An intelligent building should take decisions about the different effectors such that the users do not have to make them themself. At first glance, it might look like an easy task to control the building based on a few pre-specified rules. Nowadays buildings are controlled with some manually or semi-automated rules which have to be specified by a human building manager. These building control systems lacks mostly in self-configuring and adapting itself to the current preferences and needs of users. People are often bothered by such systems because they are constantly doing the same action triggered by a few conditions.

A building which is indeed really intelligent should learn from its users, configure itself and adapt to the current preferences and needs of inhabitants without annoying them. Such an autonomous intelligence should act in the background that users are not directly notice its existence because there are no special interaction mechanisms to communicate with the system – this type of intelligence is often classified to ambient intelligence.

We bring up in this chapter some crucial issues which have to be solved by an autonomous intelligence that is able to learn continuously from building its users and satisfy them as much as possible.

4.1 Non-Stationary Environment

Usually, learning occurs on the basis of a pre-defined representation (e.g. number of variables, possible values of these variables), which implies that the underlying structure of the problem itself is static. In the case of an intelligent building, we can not assume such a stability. Structure must be able to detect and incorporate structural changes at any point of time. We think hereby by incorporating new devices or changing the physical structure (e.g. by installing new walls in a multi-office working environment).

Beside the dynamic behavior of the structure we regard users within the building as additional agents who are part of the structure. But different than the normal structure they can behave as a mobile agent due to the fact that they move from one part of the structure to another one within a short time.

The building is occupied by different users who may also come and go. All users have their own desire of an optimal working environment. The preferences of them is constantly changing thus something which have been learnt may be valid at the present time but it may become valid in the future.

4.2 Sparse User Interaction

One aspect that makes learning of a user behavior difficult is that the user instructions are very sparse. Some devices switches their states only a few time per day. To give *sparse* a more precise meaning the table 4.1 shows how many times a device has notified a changed state (we call it a spike). The number of spikes are the accumulated number of user instructions within a time period of 38 day. The data was collected during the project *Adaptive Structure Discovery* [TZR+03]. Especially the blinds were used only a few times which makes a learning even harder. Some of them changed their state only one time within three days.

Room	Device	Spikes	Room	Device	Spikes
Room 86	light 1	77	Room 78	light 1	39
	light 2	73		light 2	38
	blind 1	30		blind 1	11
	presence 1	118		presence 1	180
Room 80	light 1	293	Room 26	light 1	20
	light 2	11		light 2	181
	blind 1	50		blind 1	11
	presence 1	113		blind 2	33
				presence 1	296
				presence 2	195
Room 75	light 1	56	Room 75	blind 1	51
	light 2	14		blind 2	40
	light 3	134		presence 1	772
	light 4	7		presence 2	803

Table 4.1: User instruction data acquired over 38 days in the summer 2003 on five different rooms located on different orientations of the building and also with a different shape.

4.3 No Positive User Feedback

We regard learning from non-explicit feedback as one major issue in making an building intelligent. The user should not notice if he is interacting with an ambient intelligence that implies that there are no special devices or mechanisms with which the user can give a feedback to the controlling system.

The user can only instruct the system with new instructions that are saying how he would like the current state of environment to be. These instructions can be regarded as real instructions or as a punishment if the system has recently changed the state of an effector. An instruction is therefore never a positive feedback (reward) which means that the user is saying he is pleased with the current state.

4.4 Consistency of Different Goals

A user has the goal to reach a maximum comfort in the area where he is working but on the other side, the building should also have the goal to consume as less energy as possible. These two goals will always stay in conflict if a user is present. Our system should find a solution that both goals can be reached at least partially.

Our system is right now equipped with normal presence detectors which can sense a movement inside a specific view range. But they are not able to count people or even to identify different persons. We are working on a personal presence detector which runs on a personal computer and detects if a user is working or is absent. With these sensors we can learn also each individual behavior but on the other side this implies also much more conflicting goals which have to be reached. Imagine a group of users which are working inside the same room and each has a different notion of maximum comfort.

4.5 Acting on Multiple Layers in a Dynamic Environment

A typical commercial building is equipped with numerous different sensors and effectors which are accessable through one or even more fieldbuses. Those systems are typically programmed that each effectors is aware of which sensor it is effecting. An autonomous intelligence should not replace such pre-programmed rules rather it should support the human building manger with the configuring of the overall controlling task.

4.5.1 Multilayered Controlling

Input and output variables of a building are high dimensional, which would make it impractical from a computational point of view to control the whole building with just one controller.

As we have analyzed in our previous project [TZR$^+$03] individual effectors (lights, blinds, etc.) are only influenced by a small subset of all possible input sensors. It is therefore appropriate to control these effectors one a local basis. We use for each of them an individual controller with multiple input dimensions (MISO) also to simplify the design of such a system by considering computational complexity and learning issues. Decisions have to be made in nearly real-time and new instructions must be incorporated immediately. These local controllers enables a fast and efficient controlling due to a small set of input signals.

The assembling of many such small controllers makes it possible to control a whole commercial building. Local controllers should be able to collaborate, exchange knowledge or trying to win against the others.

The whole decision making should be distributed on different layers without any central control that makes a controlling impossible once it would crash. To have a reliable system we want to have multiple small controllers which can act in a sub-part of the building and have to collaborate. Direct decisions by a user should be executed directly be the buildings devices on a lower layer. Our system should act and learn on a higher layer from these instructions in a more abstract way.

Chapter 5

Parallel Fuzzy Control Architecture with a Temporary and Memory-Based Learning Unit

5.1 Overview

An autonomous intelligence which is able to meet the needs and preferences of users in a typical working or living environment has to adapt itself in order to increase users comfort. It must learn from non-explicit interaction with users and must tackles issues like aging, forgetting in a short- and long-term-memory [Zho90]. User interaction in a such environments is typical very sparse where it is a very important requirement that it can assess itself in terms of behavior, error and success.

As we have explained in the last chapter, we have to learn against a non-stationary environment were demands of users and the building itself can change over time and may sometimes be contradictory. As the users are part of the structure and are able to move, change or leave continuously, the building will be used differently by looking at it with a temporary view. A short-term-memory has to learn these preferences and control effectors in such a way that the users are satisfied. Knowledge which has been learnt and is valid at present time may become false in the future. A control and learning instance must therefore forget temporary knowledge if it does not get rewarded.

To learn the long-term behavior of a building, an autonomous intelligence has to look for conditions which occur frequently or regularly, but from an abstract point of view. Knowledge which will be learnt from a long-term-memory was not only valid in the last few days. It should incorporate the general behavior of a building by considering different occupants but rather also seasons. Controlling a building in wintertime differs from another season and vice versa.

We present here an architecture which is able to act in a sub-part of a building and learn from users in order to satisfy them and still trying to reach goals of the building manager (energy consumption, security). We discuss first why and how we applied fuzzy knowledge representation and reasoning to control effectors based on different input sensors. A detailed explanations of the architecture will follow and the next chapters explain two learning algorithms from a general view.

5.2 Approximated Reasoning and Knowledge Representation

An intelligent building has to take decisions in nearly realtime based on a specific knowledge that can be manually predefined or learnt online from instructions given from the users. It is desirable to have a system that uses a human-readable representation of this knowledge. A very common way of representation are rules which can either be manually defined or learnt autonomously. Rules are constituted of a condition and an action part, which will be executed if the condition is true. The condition part can hold multiple subconditions whereas the connection between them specifies if the overall condition is true when at least one subcondition is true or all have to be true.

Effectors of a building are controlled based on a subset of sensors. The value range of sensors have to be discretized before any efficient rule specification can be made. This is because sensors have an continuous value range and it would impractical to have rules which only trigger if a sensor is exactly equal a specific value. A system is only with a small set of rules manageable. The discretization of such sensor values is not an easy task because e.g. the illumination is not escalate from dark to bright. The border has to be fuzzy where it is mixture of dark and bright.

Fuzzy system are suitable to reason in a uncertain or approximated domain, where it is difficult to derive a mathematical model of the environment. Fuzzy logic allows decision making with estimated values under vague or uncertain knowledge. The approximated reasoning in a fuzzy system can be compared to the human cognitive processes whereby the perception is rather a matter of degree than binary. We use this similarity in order to reason about the environment and take the advantage of the fuzzy rule representation.

5.2.1 Fuzzy Logic Controller

A fuzzy logic controller (FLC) takes decisions on a set of fuzzy rules which describe the relationship between different fuzzy variables. A FLC can only process fuzzy input and output values which implies that all crisp input values have to be fuzzified first. A typical interferencing process (see figure 5.1) in a FLC is defined like the following steps:

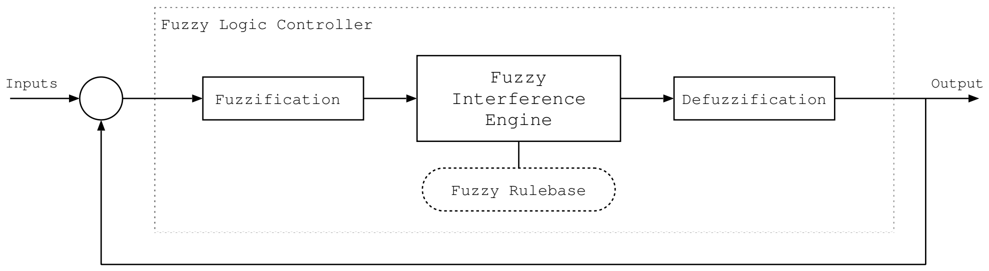


Figure 5.1: Fuzzy interferencing process

- **Fuzzification of input variables:** All crisp inputs values are assigned with a specific degree to the corresponding fuzzy membership functions.
- **Rule evaluation:** All rules which are part of a fuzzy rulebase will be evaluated if they match with the current fuzzified input values. A matching rule is also called as firing rule.
- **Aggregation of the rule outputs:** All firing rules are proposing to execute an action to a specific degree. The aggregation process combines all firing rules into one fuzzy set which represents the fuzzy decision that has to be taken.
- **Defuzzification:** As the decision is still in a fuzzy form, the last step of the fuzzy control process translates a fuzzy decision into a crisp value. The defuzzification is deferred to the last possible point of time because it is not for free. A lot of information will be lost because they can not be represented with a single crisp value.

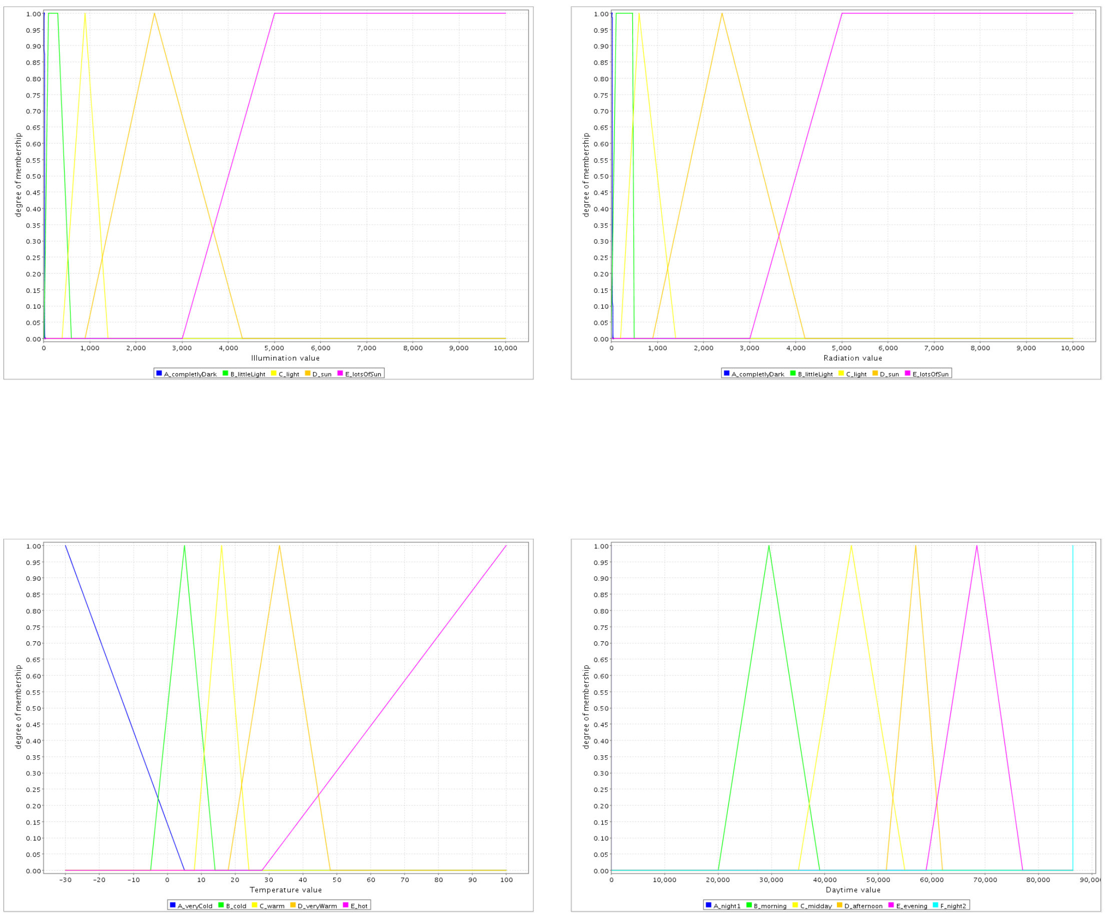
A typical commercial building can sense the environment through several different sensors. A few are simple binary sensors (like presence detectors) but most of them have an analogue value range (like temperature, illumination, etc.). The fuzzification process assigns all crisp sensor values to predefined fuzzy membership functions. A fuzzy input variable can be assigned to different membership function with a specific degree. Such membership functions (MF) have a cognitive label and are mostly pre-defined. MF's for temperature can be defined as cold (less than 5°C), cool (5-15°C), normal (10-25°C), warm (20-35°C) and hot for more than 30°C. A temperature around 22 °C is therefore normal but also warm to a certain degree. Fuzzy logic takes this imprecise knowledge into account where the finite number of fuzzy membership functions can overlap each other.

This vague knowledge can now also be used in fuzzy rules where each condition can be true if a fuzzy variable is in one of several conditional states. A possible rule to control a heating system could be:

```
IF outside temperature IS cold, cool AND inside temperature IS cool
   THEN heating IS slightly increased.
```

5.2.2 Design of Membership Functions

In order to fuzzify all input sensors and effectors of a building, we have do define a set of membership functions for each device type. We use in our system up to now pre-defined functions (Triangular, Shoulder and Trapezoidal [Ful00]) which are static, whereas it would be thinkable to tune their shape during operation. Figure 5.2.2 shows the graphical representation of all MF from each device type.



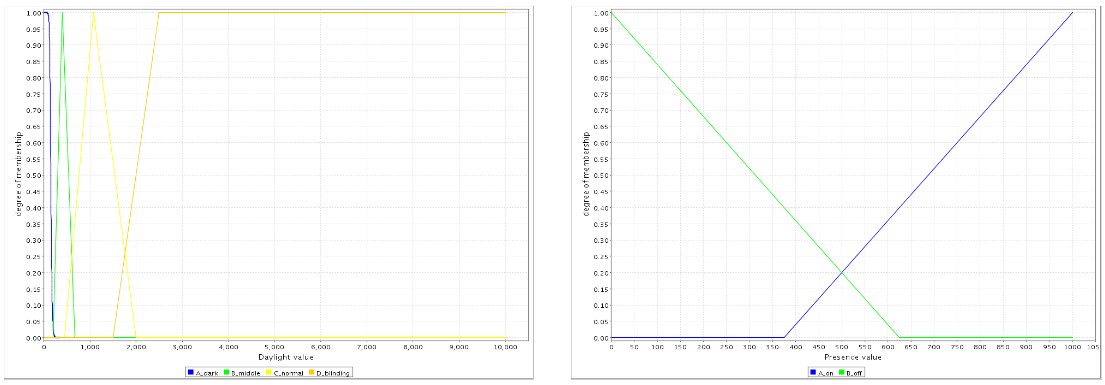


Figure 5.2: Definition of our used fuzzy membership functions.

5.3 Parallel Architecture of a Control Unit

A control unit which is able to learn in a non-stationary domain like working and living environments, has to tackle many challenges. Users can never give positive feedback through common devices which are typically installed in a building. Learning happens from non-explicit feedback without any special mechanism. New instructions have to be incorporated immediately into the knowledge and decisions have to be taken nearly in real-time. As we have already mentioned, there are at least two different levels of knowledge abstractions. One memorizes temporary knowledge which is at the moment true but was maybe different in the past and also in the future. Another, the long-term-memory, should hold knowledge that is valid over a longer period and not forget as fast as a temporary short-term-memory.

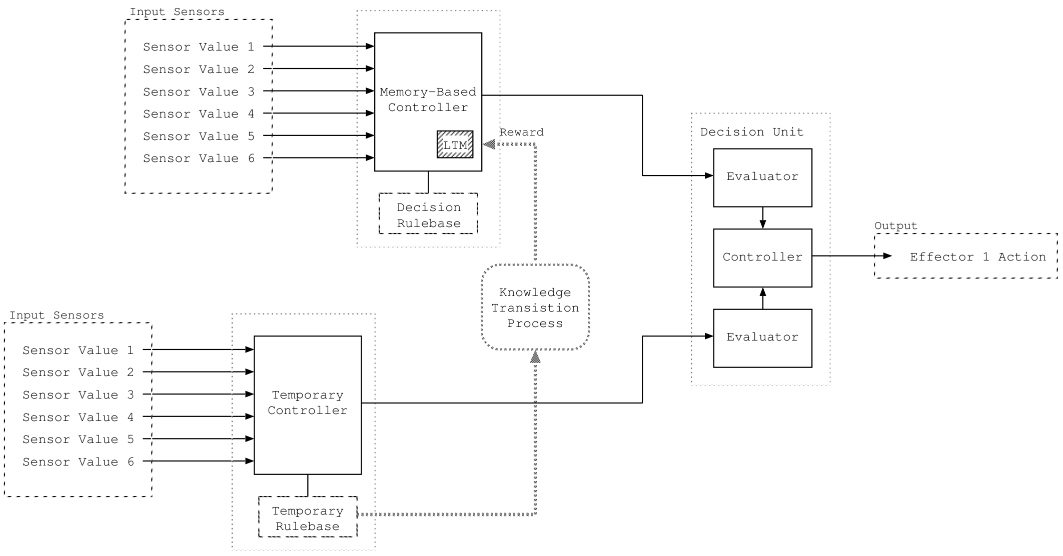


Figure 5.3: Parallel architecture of two controllers where one is learning a rulebase on a more abstract layer than the other.

Figure 5.3 present our proposed fuzzy control architecture which is able to tackle these challenges. Two different controlling and learning units are acting in parallel whereas a decision unit evaluates the actions taken by these two control units. The figure illustrates also the relationship between those two controlling units. Both have the same input sensor values to take decisions.

5.3.1 Temporary Model

The main idea of a temporary controller is to have a *One Shot Learning* which incorporates new knowledge immediately into the definitive ruleset. It has no memory about how important each individual rule is in a previous set of rules and how much it would cost to remove or modify a selection of them. The purpose in our adaptive building intelligence system is mainly that a user can always learn the system a behavior just with one instruction. An instruction given by the user must win against all other decisions proposed by our system because it was only recently learnt from the user. The user should never have the feeling that the system takes control over himself or the environment around him instead of supporting him.

The temporary controller is able to learn online from very sparse user interaction and supports all those users which are new in a building or are using its effectors different than in general. The user can only modify directly the knowledge of the temporary controller. Knowledge which becomes long-term can not be learnt at one instance.

For more details about the algorithm which is used within the temporary controller and its internal architecture we refer to chapter 6.

5.3.2 Memory-Based Model

Beside the temporary controller, another controller should take decisions based on a dynamic rulebase which is gradually adapted to temporary rules that are proved to be valid over a longer period. This controller should learn to adapt its rulebase in a way that it firstly covers new acquired knowledge and secondly still holds rules already learnt. For this reason it must have a memory of a valid and robust set of rules. The goal is further to have as general as possible rules and also the least possible set size.

This controller uses a hybrid architecture which has also been applied in other related projects such as robotics [Bon96], [Hof98],[HKS98] and as well in intelligent buildings [HCCC03]. Often proposed hybrid architectures use the advantages of an expert system in collaboration with neural networks or evolutionary algorithms. A normal expert system is able to make decisions based on a static expert knowledge but is not able to acquire new knowledge and learn from its environment. Neural network and evolutionary algorithms are techniques to learn from examples whereas these can not represent its learnt knowledge in an easy human-readable format. Thus, many attempts are trying to use the advantages of expert systems and learning algorithms to learn a set of rules which are firstly easy understandable and secondly used as an expert knowledge-base.

The big disadvantage of typical learning algorithms is that they require a lot of training samples in order to learn a good classifier which can be represented in rules. We do not have such training samples and our system can not be trained in a simulated environment because the environment is changing consistently (non-stationary). The system has to learn online and can not be stopped once it reaches a unknown state to be trained be again.

Thus, the memory-based controller must be able to adapt its rulebase the least as possible but still cover also new available knowledge. We propose a genetic fuzzy logic controller (GFLC) which modifies its rulebase like the natural evolution would do it. The already learnt rulebase will be modified with genetic operations (reproduction, crossover, mutation) in order to receive a better rulebase. The crucial point here is to define a clear and selective criteria that classifies the outcome of a genetic operation. Genetic algorithms have been proven in many approaches but they do have the disadvantage to consume a lot of time for the genetic search. As we know that not the whole state space is ever reached from a combination of typical sensors we can guide the genetic algorithm in order to reduce the genetic search. More detail about the internal architecture and the whole learning algorithm is explained in chapter 7.

5.3.3 Transition from Temporary into Memorized Knowledge

The process of the verification of new knowledge which can be incorporated into the long-term memory must also happen continuously. Knowledge which has been proven in the temporary controller carries higher-level information and are transfered into the memory-based model where they are incorporated in the already learnt memory. During this transition the rules has to be generalized because the memory learns in a more abstract way.

The following process explains how knowledge can be transfered from the temporary into the memory-based model. The longer a temporary condition will hold the more important is this information and the stronger the transition process. Next to the importance of time there is also the activity which has to be considered. A system which is highly active implies that knowledge is only valid for a short period of time and a generalization is much more difficult to derive.

The Knowledge Transition Process

Step 1 Transform real-valued sample into an instruction (fuzzification of input and action values).

Step 2 Instruction I_t becomes valid in a temporary way and has to prove its trueness.

Step 3 Wait for a new instruction I_{t+1} within a timedelay $t_{verification}$.

Step 4 If a new instruction I_{t+1} was received within the timedelay $t_{verification}$ reject instruction I_t, assign I_{t+1} to I_t and go to step 1.

Step 5 If instruction I_t was not rejected, generate a new instruction I_{mem} with a weight w_{mem} and transfer it into the memory unit.

Step 6 Memory unit accumulates the weight w_{mem} of the new instruction I_{mem} to the already acquired weights w of similar instructions.

Step 7 Wait for a new instruction I_{t+2} within a timedelay $t_{activity}$.

Step 8 If a new instruction I_{t+2} was received reject instruction I_t, assign I_{t+2} to I_t and go to step 1.

Step 9 Go to step 5.

The continuous process of rewarding rules which have been proven valid means that the system will learn from itself. The knowledge transition can only indirectly be controlled by users through changing the current state of the environment. The system is able to generate a positive feedback which is not possible for any user. Such an autonomous system adapts itself to these states in which the system has ever been. The longer and more it is in a specific state the more accurate will the knowledge in this state become.

5.3.4 Decision Making and Instruction Generation

Figure 5.3 illustrates the relationship between the temporary and the memory-based controller. The overall process which describes how decisions are taken and the system receives instructions from the user is explained here in a more formal way. In general, a feedback from a user can be regarded as an instruction or as a indirect punishment of the last taken decision.

The overall process is totally event-triggered and is therefore not periodically executed (Δt_i between instructions I_1,I_2,...,I_n is not constant and furthermore $\Delta t_i \gg 0$). All input crisp values are fuzzified in a first step before they are processed in the system. Both controllers, the temporary and memory-based, take decisions simultaneously if their memory has a knowledge about the current conditions.

Overall control and decision process

Step 0 Wait for a new instruction.

Step 1 Transform real-valued sample into a instruction I_t (fuzzification of input and action values)

Step 2 If fuzzyfied instruction f(I_t) is different to the last one I_{last} transfer instruction I_t to the transition process and assign f(I_t) to I_{last}.

Step 3 If instruction I_t was triggered by a state change of the effector proceed to step 7. Otherwise the instruction was triggered by a sensor update.

Step 4 Check if a decision can be made for new input sensor values of instruction I_t by asking the temporary decision unit and the memorized decision unit. If temporary decision unit has a decision d_{stm} with a truth value t_{stm} above a threshold $thres_{stm}$, set effector to new state given from decision d_{stm}. Go to step 0.

Step 5 If memorized decision unit can give a decision d_{ltm} with a truth value t_{ltm}, set effector to new state given from decision d_{ltm}. Go to step 0.

Step 6 If both decision units can not give a decision, go to step 0.

Step 7 If this instruction was received immediately after the effector was set to a new state, it is a feedback (system delay $t_{sysdelay}$ has to be considered). Go to step 0.
Otherwise it is an instruction from the user.

Step 8 Transfer new instruction I_t to the temporary learning unit. Go to step 0.

More information about the current implementation of such a controller can be found in chapter 10.5.

Chapter 6

Learning a Temporary Fuzzy Rulebase with an Anytime Learning Algorithm

6.1 Overview

This chapter is about the learning algorithm used for the temporary decision unit. It is an online learning algorithm which updates its rulebase incrementally by every new instructions. Theses instructions can only be made explicit by the user. An instruction is constituted of real-valued input values acquired from sensors and a real-valued effector/action value. The input of this learning algorithm are such real-valued instructions whereas the output of the learning algorithm is a set of fuzzy rules. These fuzzy rules can be used in a further step in a fuzzy logic controller to take decisions.

The architecture where this one-shot learning is applied is illustrated in figure 7.1. The temporary controller has two fuzzy logic controller whereas one is learning a dynamic rulebase for the other. The later one takes actually decision about an effector based on fuzzified sensor inputs. The other one has a static rulebase which is mainly for the fuzzyfication of the instructions and further for the generation of new rules that will be incorporated into the dynamic decision rulebase.

The design of this algorithm is inspired by several related approaches [CCSZ21], [CZ16] and our predecessors approach [RJD04].

6.2 One Shot Learning Algorithm

The learning algorithm learns a disjoint set of fuzzy rules from sparse user interactions. Instructions have to be incorporated immediately such that it can be used for any further reasoning. The algorithm generates a set of rules which can be replaced by any contrary instruction. We use here largely the same notation as in [RJD04] and [Ful00].

6.2.1 Definitions

Definition 1 (more general) *A rule R_i is more general then a rule R_j if $\sum_{k=1}^{|E_i|} |E_{ik}| > \sum_{k=1}^{|E_j|} |E_{jk}|$ is true.*

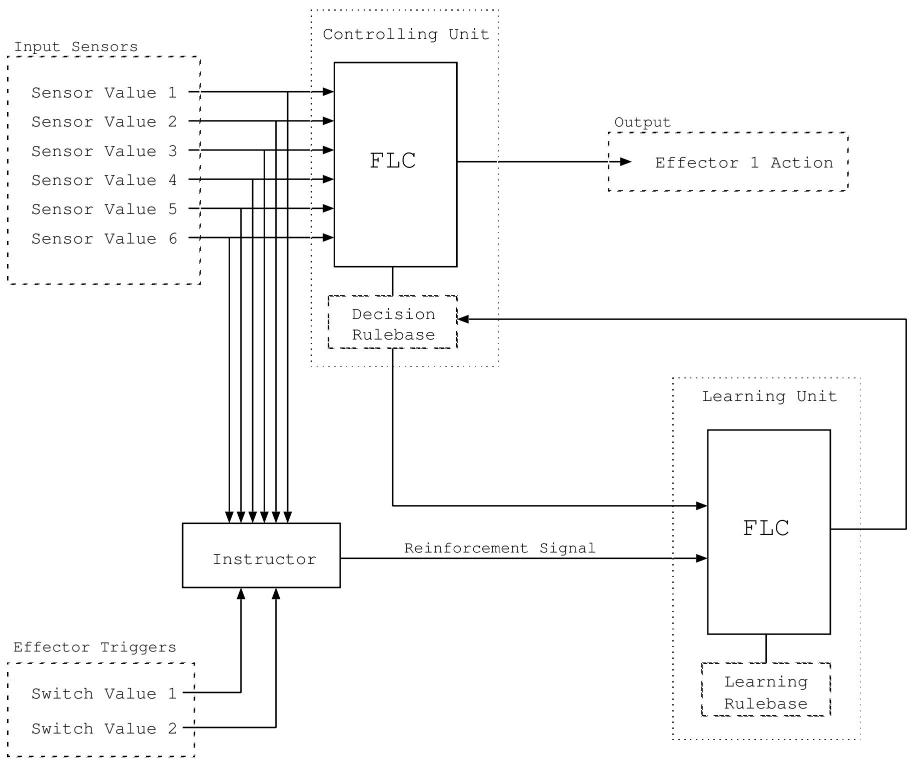


Figure 6.1: Architecture of the temporary controller. One controller takes decision based on a dynamic rulesbase which is learnt from a second controller.

6.2.2 Operators

The algorithm uses the operators **antecedentSubsume**, **overlap** and **reduce**.

Definition 2 (antecedentSubsume) *antecedentSubsume*(R_i, R_j) *is true if the following conditions holds for all* E_{ik} *of* R_i*: For all* $k = 1 \dots N$ *it is true that* $E_{ik} \subset E_{jk}$*.* N *is the number of fuzzy variables. antecedentSubsume*(R_i, R_j) *is true if the above conditions hold and* $y_i \neq y_j$ *or* $y_i = y_j$*.*

Definition 3 (overlap) *The rule* R_i *overlaps rule* R_j *if the following condition is valid for all* E_{ik} *of* R_i*: For all* $k = 1 \dots N$ *it is true that* $|(E_{ik} \cap E_{jk})| \geq 1$*.* N *is the number of fuzzy variables and* $\cap$ *is the intersection of two sets.*

Definition 4 (reduce) *The rule* R_d *can be reduced with rule* R_i *by executing the following operation on all* E_{dk} *of* R_d*: Evaluate for all* $k = 1 \dots N$ *if the reduction* $E_{dk} = E_{dk} - E_{ik}$ *(evaluation by removing all conditions* $E_{ik} \subset E_{dk}$ *from* E_{dk}*) removes the least possible state area whereas* N *is the number of fuzzy variables. A reduction is possible by just reducing one fuzzy variable. Reduce therefore only the fuzzy input dimension* k *evaluated before and assign* $E_{dk} = E_{dk} - E_{ik}$*.*

6.2.3 Algorithm

The following procedure is executed everytime the system receives a new instruction.

Step 0 Wait for a new sample

Step 1 Transform real-valued sample into instruction I_t (fuzzification)

Step 2 Make a fuzzy rule R_i of this instruction I_t and label it also with its weight w

Step 3 Remove all rules R_d which are part of the definitive ruleset ($R_d \in \mathcal{R}_{def}$) and where antecedentSubsume(R_i, R_d) is true.

Step 4 Check if rule R_i has an overlap with any definitive rules ($R_d \in \mathcal{R}_{def}$). If yes, make a set $\mathcal{R}_{red}$ with all rules R_d where overlap(R_i, R_d) is true. Otherwise proceed with step 6.

Step 5 Execute for all rules R_d which are member of set $\mathcal{R}_{red}$ the operation reduce(R_i, R_d) in order that overlap(R_i, R_d) is false.

Step 6 Add new rule R_i with its weight w into definitive ruleset $\mathcal{R}_{def}$.

Step 7 Go to step 0.

Rules which have been generated by this listed algorithm are part of a fuzzy rulebase on which a fuzzy logic controller takes its decisions. The rulebase itself is the temporary memory and will therefore stay always under the dynamic process of forgetting. Once a rule is too old, it will dissapear from the rulebase. A user can therefore reward rules by giving the user a new instruction.

Features of this Algorithm

- online / anytime
- learns incrementally from sparse user interaction
- adapt itself to changing conditions
- learnt rules loose truth and have only a limited lifetime
- rulebase is under the dynamic constraint of forgetting

Chapter 7

Evolutionary Learning of a Generalized Fuzzy Rulebase with a Long-Term Memory

7.1 Overview

In addition to the learning algorithm explained in the last chapter we propose here an evolutionary algorithm which learns a fuzzy rulebase for a memory-based fuzzy logic controller. Rules which have been proved valid in the temporary control unit are an input reward signal for this evolutionary algorithm. Like illustrated in figure 7.1 a memory receives the generalized reward signal if a temporary rule was successfully verified. The memory is divided into s fuzzy areas, whereas s is defined in equation 7.2.

$$N = |\{X_0, X_1, \ldots, X_{N-1}\}| \tag{7.1}$$

$$s = \prod_{k=1}^{N} |\mathcal{L}(X_k)| \tag{7.2}$$

N is the number of all possible fuzzy input variables that can be considered in the condition part of a fuzzy rule. X_i is a possible fuzzy input variable and $\mathcal{L}(X_i)$ is the number of possible fuzzy states of fuzzy variable X_i. More detail about our notation can be found in appendix A. The state space is therefore constituted of s fuzzy areas whereby these can overlap each other based on the design of the individual membership functions.

7.2 Long-Term Memory

After the verification of a temporary rule the memory receives a reward signal with the specific fuzzy rule. It verifies which fuzzy areas are covered by this new rule and accumulates the reward to the corresponding areas. Due to this process these areas will gain importance with such a reward against all others. See figure 7.2 which illustrates how a fuzzy areas of a reward will be determined and the corresponding action weight will be accumulated.

Because a fuzzy rule is constituted of some conditions and an action, the later one has also to be considered. All other actions which are already defined inside a fuzzy area will decrease and only the one which is equal

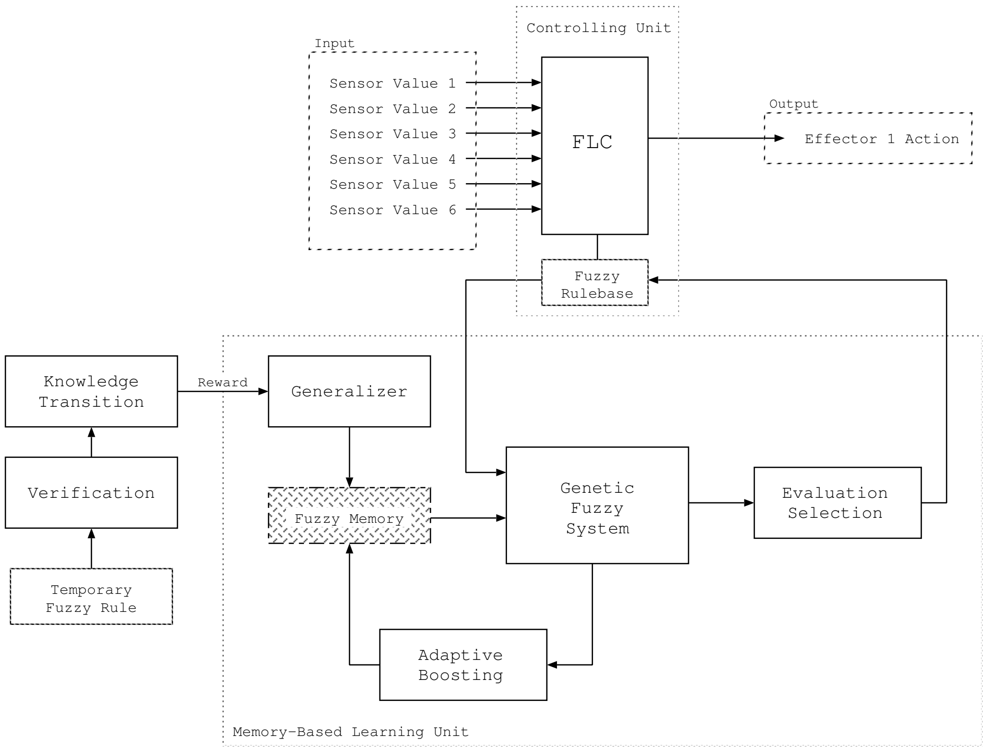
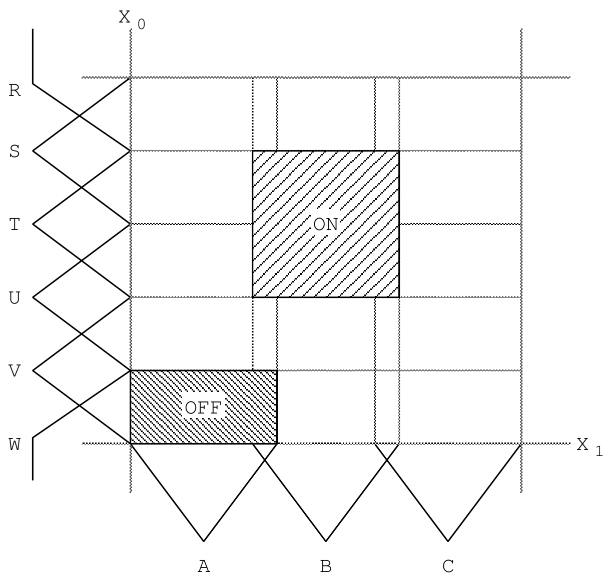


Figure 7.1: Internal architecture of the memory-based controller. Reward signals are first evaluated and generalized before they will be incorporated into the fuzzy memory. A genetic fuzzy system generates with an iterative rule learning approach a fuzzy rulebase on which a common fuzzy logic controller takes decisions.



Reward rules:
if X_0 = T AND X_1 = B then Y = ON
if X_0 = W AND X_1 = A then Y = OFF

Figure 7.2: An example how proved rules from the temporary learning unit can be incorporated as a reward in the memory. All fuzzy areas which are covered by the reward rule and have the same action as the rule's action gets rewarded. The memory accumulates the reward to the already acquired rewards from the past. All other actions of these fuzzy areas are decreased which implies an competition between all actions.

to the rewards action will increase. The weight range of such a fuzzy area is therefore limited with an upper and lower bound whereby the different actions compete against each other to reach the maximum weight.

The memory itself is fuzzy and holds therefore imprecise knowledge about the environment. It serves also as an input for the evolutionary algorithm which has to find an *optimal* set of rules which cover the current memory. Optimal means not only to find a small set of rules which cover the whole state space. Additionally, rules should cover areas where really a knowledge is and should cover consistent areas. A rule which covers different actions is not as good as a rule which covers only uniform fuzzy areas.

Before the memory can be processed by the learning algorithm it has to be transformed into a genetic representation that is easy to evolve by natural genetic operations like crossover, mutation and reproduction. The encoding is similar to the encoding of a normal fuzzy rule which is explained right below.

7.3 Encoding Fuzzy Rules into a Genotype

Beside the encoding of the fuzzy areas also the current decision rulebase has to be transfered in a fuzzy genotype. This genotype has to be evolved that it firstly covers the whole memory but secondly does not differ much from the previous one. A genotype has a population with individual chromosomes where each one has to represent a fuzzy rule.

Therefore a chromosome must hold all conditions and the action of a fuzzy rule and a transformation back to into a fuzzy rule must be possible without any loss. Because the number of fuzzy input variables among all rules inside the same fuzzy logic controller is equal we are able to define the build of a chromosome once. A chromosome holds typically different genes that contain specific information about a specific property. The genes of a fuzzy chromosome are representing conditions and the action of a fuzzy rule. Figure 7.3 illustrates how we define a fuzzy chromosome.

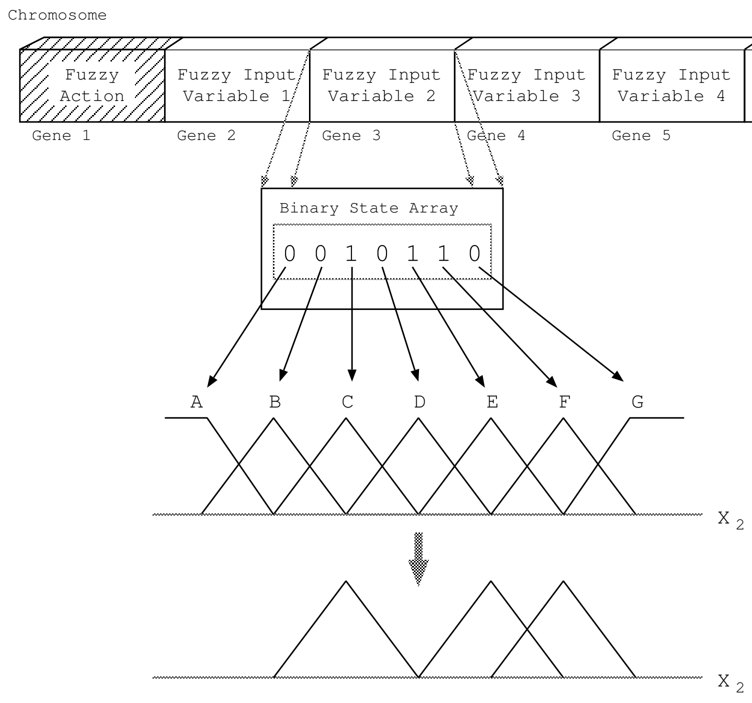


Figure 7.3: Design of a fuzzy chromosome: The first gene is the action of a fuzzy rule and the following ones are fuzzy input variables. A fuzzy gene holds a binary array similar to the size of possible states of the corresponding fuzzy variable. The value of each bit indicates if the fuzzy variable holds a specific state.

7.3.1 Fuzzy Gene

The normal fuzzy gene holds a binary array equal to the number of the possible states of the fuzzy input variable that is represented by this fuzzy gene. Each bit indicates if the fuzzy variable holds a specific state or not. A fuzzy rule R_i gets transformed into a fuzzy chromosome with $|E_i|$ normal fuzzy genes and a special fuzzy action gene. E_i is a rule condition for one fuzzy input variable X_i.

7.3.2 Fuzzy Action Gene

The first gene of a fuzzy chromosome is a special one because it represents the action of a fuzzy rule. It also contains a binary array with a size equal to the number of possible actions. It requires a special treatment because a fuzzy rule can only have one action and therefore in the fuzzy action gene only one bit can be set to 1 in contrast to a normal fuzzy gene that can carry more active bits.

7.4 Genetic Operations Applied to a Fuzzy Genotype

The evolutionary algorithm uses genetic operations inspired by nature. This section explains briefly how these operations can be applied to a fuzzy genotype.

7.4.1 Fuzzy Crossover

A fuzzy crossover selects randomly two parent chromosomes from the population. A locus indicates where the crossover point is. All information of the genes after the locus point will be crossovered. The locus is calculated new for each crossover operation. The exact implementation is explained in chapter 10.

7.4.2 Fuzzy Mutation

A population can evolve due to the crossover operation but once it reaches a maximum and all possibilities are exhausted the fitness of the population is not able to improve more. The fuzzy mutation operation mutates genes in a chromosome with a given probability p_{mutate}. If a gene mutates, all fuzzy states of this gene could be set to a random value or further the mutation rate could also be calculated for the individual states. We use up to now the first mentioned mutation.

7.5 Decoding Fuzzy Rules from a Genotype

Like mentioned in the encoding of a fuzzy rulebase it is a requirement that all fuzzy chromosomes have to be transformed back into a set of real fuzzy rules without any loss. The transformation takes place like the encoding whereas each gene becomes a fuzzy input variable and the first gene in the action of the fuzzy rule.

7.6 Definitions

The evolutionary algorithm uses several functions in order to compare and evolve the population. All these functions are here defined and use the fuzzy notation like in appendix A.

Definition 5 (subsume) *$subsume(a_i, c_j)$ is true if the following conditions holds for all E_{ik} of fuzzy area a_i: For all $k = 1 \dots N$ it is true that $E_{ik} \subset E_{jk}$. N is the number of fuzzy variables (normal fuzzy genes). $subsume(a_i, c_j)$ is only true if the action of $Action(a_i)$ is equal to the action of the chromosome $Action(c_i)$.*

Definition 6 (antecedentSubsume) *antecedentSubsume(a_i, c_j) is true if the following conditions holds for all E_{ik} of fuzzy area a_i: For all $k = 1 \dots N$ it is true that $E_{ik} \subset E_{jk}$. N is the number of fuzzy variables (normal fuzzy genes). antecedentSubsume(a_i, c_j) does not consider the action value and is therefore still true if the action of Action(a_i) is not equal to the action of the chromosome Action(c_i).*

Definition 7 (boosting factor) *The boosting factor of a solution chromosome c_t is calculated based on all fuzzy anchors a where antecedentSubsume(a_i, c_t) is true and n is defined as $|a|$. The boosting factor is a real value between 0.0 and 1.0 and is used to modify the distribution of the samples which have to be covered by fuzzy rules.*

$$E(c_t) = \frac{\sum_{i=1|Act(a_i) \neq Act(c_t)}^{n} w_i}{\sum_{i=1}^{n} w_i} \tag{7.3}$$

Definition 8 (volume) *volume(a_i) of a fuzzy area or a fuzzy chromosome is defined as the product of all active states from each fuzzy variable:*

$$\text{Volume}(a_i) = \prod_{k=1}^{|E_i|} \text{activeStates}(E_{ik}) \tag{7.4}$$

Definition 9 (action) *Act(a_i) of a fuzzy area or a fuzzy chromosome is defined as the action which this area or chromosome represents.*

7.7 Determining the Fitness of Chromosomes

In order to evolve a genotype, the evolutionary algorithm has to calculate the fitness of a possible solution. A fitness indicates how good a solution is and increases also the possibility for this particular solution also to be selected for the next evolution.

As our goal is to find an optimal set of rules which covers the current memory as good as possible the definition of the fitness function is a very curical step. It has mainly three goals:

- **Wrong classified areas:** The solution should cover as few as possible wrong classified fuzzy areas. The fitness of the solution should be decreased more if the wrong classified fuzzy area is an important one which means that the accumulated weight w_i is large. .
- **Amplification:** A solution has the goal to amplify itself in order to incorporate all equal classified fuzzy areas. The amplification takes also the relative weight among all samples into account that means a solution is better if it covers the most important fuzzy areas of a classification.
- **Optimal volume:** The more equal classified fuzzy areas a solution covers the fitter it is. But solutions which incorporates also areas where no knowledge is available loose fitness and also if wrong classified areas are covered by it. The cost to cover a wrong classified area could be bigger than the one to cover a memoryless area.

All these goals are a factor of the the following equation 7.8. In the following equations A is a set with all s possible fuzzy areas (see equ 7.2). a is a set of fuzzy areas $\{a_0, a_0, \dots a_{n-1}\}$ where for all antecedentSubsume(a_i, c_t) is true and n is the power of set a. w_i is the accumulated weight of a fuzzy area a_i and b_i is the classified boosted weight of a_i.

$$err(c_t) = 1 - \frac{\sum_{i=1|Act(a_i) \neq Act(c_t)}^{n} w_i}{\sum_{i=1|Act(a_i) = Act(c_t)}^{n} w_i} \tag{7.5}$$

$$amp(c_t) = \frac{\sum_{i=1|Act(a_i) \neq Act(c_t)}^{n} b_i}{\sum_{i=1}^{n} \sum_{j=1|Act(a_i) = Act(A_j)}^{s} b_i} \tag{7.6}$$

$$vol((c_t) = \frac{\sum_{i=1|Act(a_i)=Act(c_t)}^{n} Volume(a_i)}{Volume(c_t)} \tag{7.7}$$

$$f(c_t) = err(c_t)^q * amp(c_t) * vol(c_t) \tag{7.8}$$

7.8 Adaptive Boosting

The evolutionary algorithm has to find an optimal set of rules which covers the current fuzzy memory. Like explained, the memory is constituted of s different fuzzy areas that have an accumulated weight for all possible actions of the controller. These fuzzy areas are the trainingsamples for the evolutionary algorithm. But before using them, they are classified based on the relative action weight within each area. An area has for example a positive action weight $w_{ON} = 50$ and a negative one $w_{OFF} = 23$ whereas this area will be classified to a positive action with a relative weight $b_{ON} = 27$.

We use an iterative rule learning approach (IRL) [GH97] which is repeatedly invoked to generate a new evolution and incrementally adds the best solution to the new ruleset. The relative weight b_i specifies the relative importance between different fuzzy areas. The fitness function specifies that the best solution is one which covers the most relative important fuzzy areas first. But to be able to find also rules which not have such a high relative importance than others, we decrease the relative importance b_i of all those fuzzy areas which are covered by the fittest solution after a new evolution. We use therefore a fuzzy variant adaptive-boosting algorithm which modifies the distribution of the trainingsamples in order to make a classifier more robust. The design of our boosting algorithm is inspired by [FS96] and [Hof]. Relative importance b_i of fuzzy areas are decreased based on how sufficiently they are covered by a solution (see equ. 7.3 for the exact definition).

7.9 The Algorithm

The evolution of finding an optional set of rules which covers the memory is defined as the following algorithm:

Step 1 Initialize all samples form the memory by setting b_i to the classified relative weight.

Step 2 Evolve the genotype which is the current rulebase $\mathcal{R}_{def}$ for g generations. The genotype can be modified with genetic opeartors like mutation or crossover.

Step 3 Evaluate the fittest solution c_i after the evolution and add it to the set of solutions $\mathcal{R}_{new}$.

Step 4 Calculate the error $\mathrm{E}(c_i)$ of the solution c_i.

Step 5 Decrease b_i of all samples from the memory by the factor $\beta = (1-\mathrm{E}(c_i))$ where subsume(a_i, c_i) is true.

$$b_i(t+1) = \begin{cases} b_i(t) * \beta & \text{if } Act(a_i) = Act(c_i) \\ b_i(t) & \text{if } Act(a_i) \neq Act(c_i) \end{cases} \tag{7.9}$$

Step 6 If all b_i of the sample memory are satisfied and below a threshold t_{boost} go to step 2.

Step 7 Evaluate the set of solutions $\mathcal{R}_{new}$ and select a limited number of solutions from it.

Step 8 Assign the set of solutions $\mathcal{R}_{new}$ to the definitive rulebase $\mathcal{R}_{def}$.

Step 9 Go to step 1.

The algorithm itself is able to adapt itself fast to the long-term memory. The process of forgetting and taking new knowledge into account is matter of the memory. The evolutionary algorithm searches for an optimal set of rules which covers the detailed knowledge of the memory as general as possible.

Features of this Algorithm

- fast converging to the current long-term memory
- learn gradually a fuzzy rulebase
- refines set of fuzzy rules with each new invocation.
- learnt rules loose truth and have only a limited lifetime

Chapter 8

Building Specific Behaviors

The controlling and learning architecture is applied to a subpart of a building. The learning algorithms are indeed in a general form that allows it also to used them in related domains. As usual a controlled system has its own preferences which have to be considered. Most of the problems in our building are caused by the delayed adaption of individual sensors and effectors. Therefore, next to the learning algorithm we are preprocessing delayed behavior which are explained in the following sections.

8.1 User Punishment

Because an effector state change often results in state changes within the environment, the system is after the action situated in an other fuzzy area than before. For example the light is turned off which implies that the daylight in this room decreases.

If the users now punishes a decision with switching on the light again within a very short time period, we have to regard it as a real punishment of the last decision. By using a decision history we are able to determine the conditions of the last decision and generate a temporary punishment into the short-term memory.

8.2 System Toggling

Due to the delayed behavior of a few effectors and therefore also sensors the system can come into a state of toggling from one fuzzy area to the other and vice versa. E.g. the blind needs a lot of time to close or open. All indoor sensors which are dependent of this blind are changing their state also with a delay. Our system is looking online for such behaviors where a few rules are firing periodically within a short time period. Like in the above section we are able to determine with a decision history the conditions of the last decisions and can therefore interupt such a toggling by generating a temporary knowledge.

Part III

Architecture, Design and Implementation

Chapter 9

Framework Architecture

To develop a useful and universal framework we have to define the boundary of the system. There are many different aspects and influences which characterize the architecture of a framework. This chapter tries to point out the main influencing factors of a building system.

9.1 Building Access

A building system consists of many sensors and effectors. Most commercial buildings have an integrated building bus system (like LonWorks [loc] or EIB [eur]) with several well-known devices like switches or presence detectors. Often there is not only one bus system, but also other buses to control e.g. the heating system. Therefore the framework must be able to handle more than one different bus system at the same time. It is also thinkable to connect the framework to virtual software buses like for example instant messaging systems (ICQ [ins]) or a ethernet bus.
Each of such buses operate in their own special way. That implies the necessary of an abstraction of this different bus types with a common access interface.

9.2 Conflicting Goals of Different Interest Groups

A commercial building is occupied by different types of users that have its own interests and goals to maximize their personal comfort. Whereas the management tries to minimize the running costs and the caretaker want to act devices with consideration.
Our framework should offer a possibility that each party can introduce its own objectives.

9.3 Multi Agent System

As we showed above, there are many different aims and requirements. Some of them are dependent and some independent on others. For a better understanding and to breakdown the problem, we want to divide the framework in small software units. Each of these software units is responsible for one specific part of the whole problem. Conventional software design concepts use interfaces and classes to reach a low coupling. But it is still one software.
We design these software units as stand-alone agents. An agent represents a physical or logical part of the building like the physical device bus or an inhabitant. Each agent has its own goals an can interact over a network with other agents to achieve them. We defined for example an energy agent which enforces claims of the power management or a comfort agent which provides maximum of comfort to all inhabitants.
We are using an agent framework, called *ABLE* [Age], from IBM to set up the agent environment. *ABLE*

implements the FIPA [FIP02] and JAS [Fuj02] standard. Appendix C describes the functionality of ABLE in more detail.
The communication between these agents is realized with an asynchronous messaging system. Agents can communicate directly to other agents (by using an agent directory and transport messaging service) or indirectly via a distribution messaging system (DMS).

9.4 Framework Layers

To keep the framework manageable, we divide the whole problem into smaller subproblems. Thereby it is easier to conceive the difficulties and the suited solutions.
We defined the following subparts for our framework:

- Bus abstraction
- Structure abstraction
- Messaging
- Recovering
- Learning and controlling

Some of this subproblems have a relationship to the others, thus we illustrate the dependency graphically in figure 9.1.

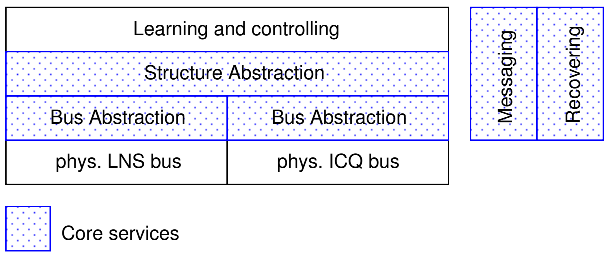


Figure 9.1: Dependency of the framework subparts as a layer model. Messaging and Recovering are used of every agent.

This chapter describes the concept of each part with its agents and the interaction between them.

9.5 Core Services

9.5.1 Messaging

An Agent can communicate directly with an other. But sometimes an agent do not know who is interested in its information. So it is usefull to provide something like a mailing list for different interests. The *MessageDistributionAgent* offer this service. Each agent can create its own topic channel to distribute messages. The *MessageDistributionAgent* delivers the message to all subscribed agents. The concept is comparable to the observer pattern in object oriented software design.

9.5.2 Bus Abstraction

The bus abstraction operate as a gateway between the hardware specific bus systems and other agents. Furthermore it provides a common access interface to higher layered agents.
A building exists of several different devices. Each device type has its own idiosyncrasies and an its special kind to control it. Therefore it is expedient to encapsulate all this and define a general interface.
We defined two layers of encapsulation:

1. Bus abstraction
2. Device abstraction

The **bus abstraction** defines a common access interface for different physical or logical bus systems (LNS bus, ICQ, ABI bus). Each bus system is realized as an own controller.
The **device abstraction** allows a standardized access to all bus devices. Each device type has its own specialized bus-specific implementation (Figure 9.2). All idiosyncrasies of a device are handled in the specific implementation of the device type.

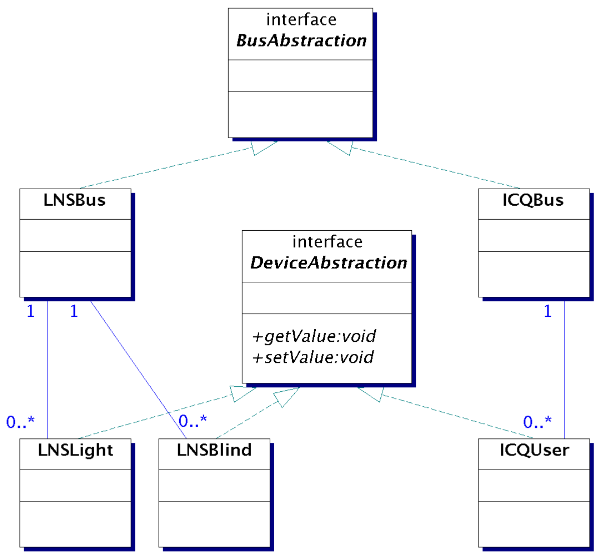


Figure 9.2: An example of a bus abstraction.

Modification Notification

To keep the message traffic as low as possible the *BusAgent* send on every modification of a device value a notification message over the responding topic channel (`TOPIC_PREFIX_DEVICE` + 'device id'). Thus other agents have not to poll the device states but can rather subscribe to the interested topic channels.

Public Interface

The bus abstraction is realized with the implementation of a BusAgent. It accepts only two message types from non-core agents: to get and set the value of a device (9.1).

9.5.3 Structure Abstraction

The structure abstraction represent a structured view to the available devices. It builds up a hierarchical structure of the building. This structure is represented as a tree of clusters whereby each cluster can contain

Message type	**GET_VARIABLE_VALUE**	
Property	P_DEVICE_ID	ID of the interested device
Message type	**SET_VARIABLE_VALUE**	
Property	P_DEVICE_ID	ID of the device which is requested to change its state
Property	P_DEVICE_VALUE	Suggested new value of the device

Table 9.1: Messages to get and set the state of devices.

several devices. Figure 9.3 illustrate an possible structure of an building.

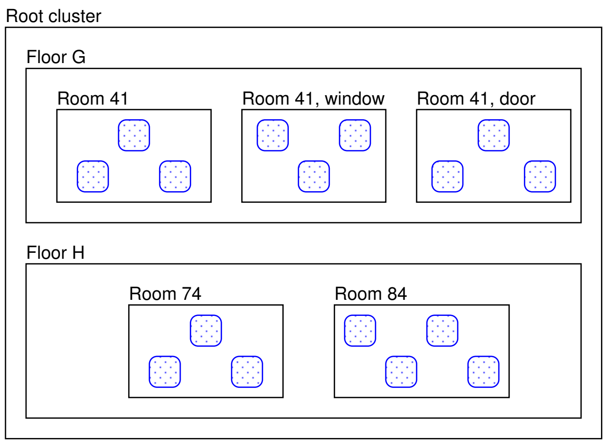


Figure 9.3: Example on a possible hierarchical structure of a building. The devices (blue quadrates) are combined into five clusters (black rectangles). The clusters itself are again combined to more general clusters.

Because building structure can changes and devices are removed, replaced or moved the structure is not constant. The structure abstraction must provide the administration of a dynamic environment.

Public Interface

The implemented *StructureAgent* supports the above named requirements. After an initial startup it contains only one root cluster. For example new devices can be added by sending a `ADD_DEVICE` message to this *StructureAgent*.
Table 9.2 list all supported public messages with the required parameters.

Modification Notification

For each modification of the structure, the *StructureAgent* sends notification messages to all interested agents. For that it uses topic channels of the *DistributionMessageAgent*. Table 9.3 gives an overview of the released messages for each modification instruction. The type of the notification message is always `STRUCTURE_CHANGED` and contains one property `P_EVENT` with the event text. Thereby a depended agent is always well informed about modifications of the structure and can react.

9.5.4 Recovering

To improve the safety and the reliability to whole running system must provide a recovery functionality. For this we designed a global recovery concept, implemented by the *RecoveryAgent*. It sends periodically an `AUTO_SAVE` message to all running agents. This message includes the path and an unique filename in which the agent has to serialize itself. The directory path changes automatically everyday, in order we get a backup

Message type	**ADD_DEVICE**	
Property	P_DEVICE_TYPE	Type of the device. Ex. 'lns.light'
Property	P_DEVICE_NV	The network variable of the device
Property	P_DEVICE_DISPLAYNAME	[Optional] A description/name of the device.
Property	P_DEVICE_OPTIONS	Additional options for the device. Ex. smooth timeout for presence detectors.
Property	P_BUS_TYPE	Identifier of the bus. Ex. 'lns' for the LonWorks-Bus
Property	P_CLUSTER_ID	ID of the cluster which should contain this device. Cluster must already exist.
Message type	**REMOVE_DEVICE**	
Property	P_DEVICE_ID	ID of the device which should be removed
Property	P_CLUSTER_ID	Id of the cluster which contains this device
Message type	**CREATE_CLUSTER**	
Property	P_CLUSTER_ID	ID of the parent cluster in the cluster tree Note: The cluster on the top has the ID 'root'.
Property	P_CLUSTER_DISPLAYNAME	[Optional] A description/name of the cluster.
Property	P_SENDER	AgentDescription of the sender. After successfully creation this sender is notified with the cluster ID.
Message type	**REMOVE_CLUSTER**	
Property	P_CLUSTER_ID	Id of the cluster which should be removed
Message type	**GET_STRUCTURE**	
Property	P_CLUSTER_ID	Returns the structure tree from this cluster.

Table 9.2: Messages to manage the structure.

Event	**ADD_DEVICE**	
Channel	ADD_DEVICE+'cluster id'	Cluster id of the parent cluster
Channel	STRUCTURE_CHANGED	
Event	**REMOVE_DEVICE**	
Channel	REMOVE_DEVICE+'cluster id'	Cluster id of the parent cluster
Channel	STRUCTURE_CHANGED	
Event	**CREATE_CLUSTER**	
Channel	CREATE_CLUSTER+'cluster id'	Cluster id of the parent cluster
Channel	STRUCTURE_CHANGED	
Event	**REMOVE_CLUSTER**	
Channel	REMOVE_CLUSTER+'cluster ids'	This message is sent to different channels. All subcluster channels, the channel of the removed cluster itself and to the channel of the parent cluster
Channel	STRUCTURE_CHANGED	

Table 9.3: Summary of notification messages, which are released after a modification of the structure

history tree.
The *RecoveryAgent* just coordinate the persistencing of all agents, but the effective serialization and restoring functionality is part of each agent itself.
To restore the last persistent state of the system, the *RecoveryAgent* has to be started in an **empty** ABLE platform with the parameter *SystemRecovery*. The agent searches automatically for the latest backup and starts each agent, whereby the order of starting is still the same like the original.

9.5.5 Agent Interaction

The following diagrams show, how a user agent can act with the core services and how it will be processed within the core. A detailed description of the used messages and their properties is listed in table 9.1, 9.2 and 9.3.

Add a New Device

The *StructureAgent* permits to add the same device to different clusters. But rather the *BusAgent* should contains only one instance for each physical device. This intelligence is realized in the *StructureAgent* and does not affect UserAgents. The UserAgent has only to send a message `ADD_DEVICE` to the *StructureAgent*, where the message is forwarded to the *BusAgent*. The *BusAgent* compares the new device with all existing. If there exists already a similar device it returns the description of them to the *StructureAgent*. In the other case, it creates a new device instance. The collaboration with the agents is illustrated in figure 9.4.

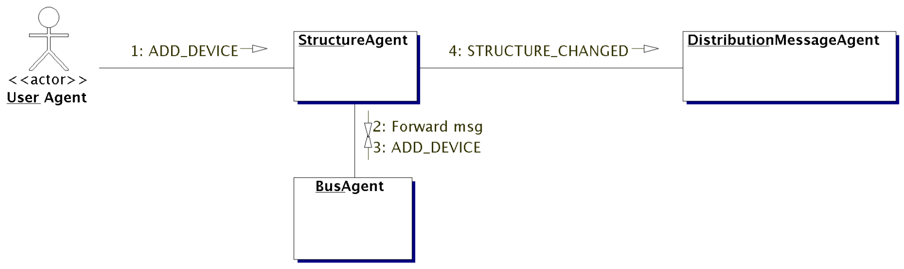


Figure 9.4: Collaboration of the agents to add a new device

Remove a Device

To remove an existing device from a cluster, the UserAgent has to send a message `REMOVE_DEVICE` to the *StructureAgent*, which removes it from the defined cluster. Instances of this device in an other clusters are not affected. If it was the last instance of this device in the current structure, the *StructureAgent* automatically removes the devices from to *BusAgent* (Fig. 9.5).

Add a New Cluster

The message `CREATE_CLUSTER` prompts the *StructureAgent* to create a new subcluster. The *StructureAgent* responses to the requester with a message which contains the new cluster ID. Figure 9.6 illustrate this process in a graphical way.

Remove a Cluster

If a cluster becomes redundant it can be removed by sending a message `REMOVE_CLUSTER` to the *StructureAgent*.

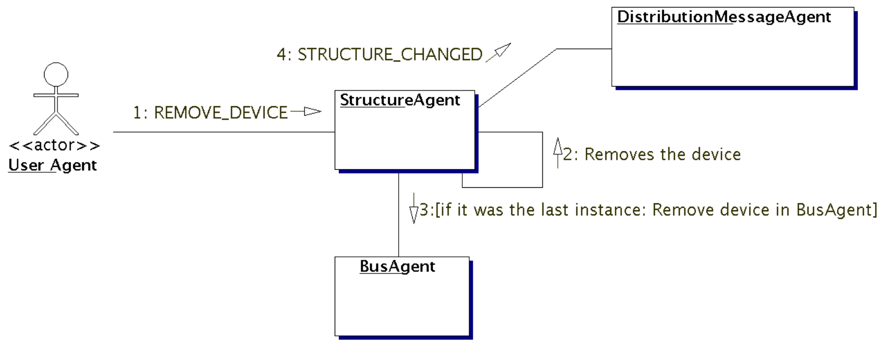


Figure 9.5: Collaboration of the agents to remove an existing device

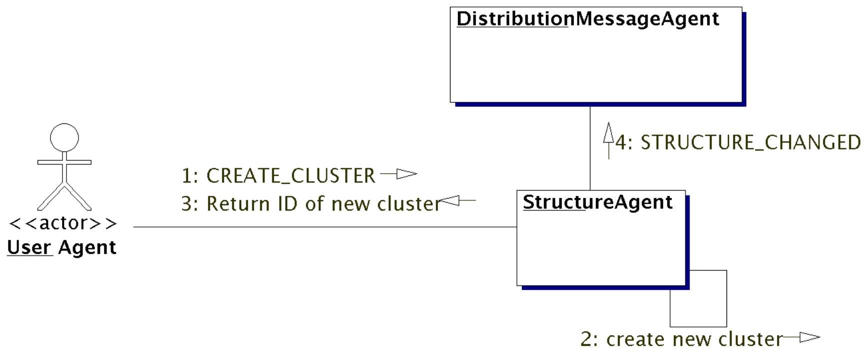


Figure 9.6: Collaboration of the agents to add a new cluster

Note: This command removes also all devices of the cluster and all subclusters with their devices (Fig. 9.7).

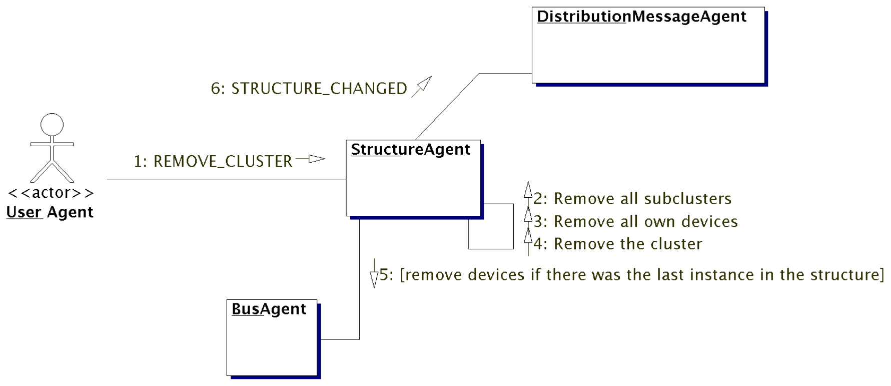


Figure 9.7: Collaboration of the agents to remove an existing cluster

Get and Set Device Values

By sending the message `GET_VARIABLE_VALUE` to the *BusAgent* every agent can ask explicitly for the current state of a device. If the device is an effector, it can also be set to a new state by sending a message `SET_VARIABLE_VALUE`.
Note: Do not use `GET_VARIABLE_VALUE` for polling the state of a device. By a subscription on the responding topic channel, an agent will be well informed about every modification.

9.6 Additional Autonomous Agents

9.6.1 Energy Agent

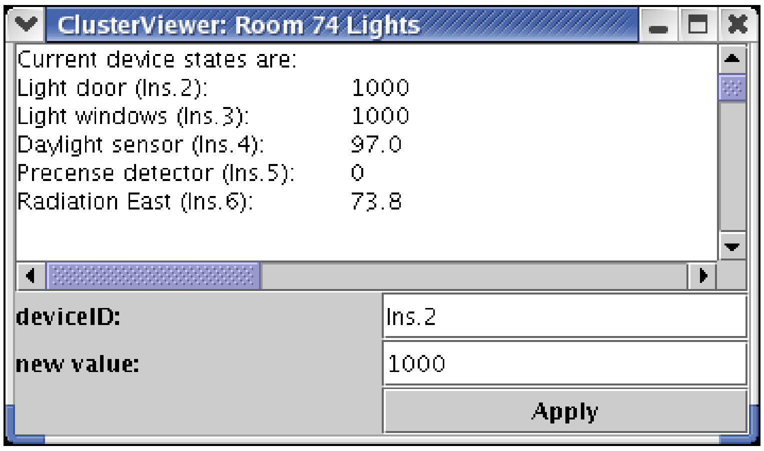
The *EnergyAgent* has a single objective; to save energy. It observes a cluster and checks periodically if there is no person in the room (all presence detectors are low). If this condition is true, it requests all lights within a specific cluster to turn off. The decision to turn off the light is delayed a few seconds. Thus we give other control agents time to turn off the light based on their knowledge. If the light is still on after a timeout, the *EnergyAgent* turns off the lights.

9.6.2 Control Agent

A control agent is started for one cluster. It learns the behavior of the user and controls all effectors of this cluster with the learnt knowledge. The normal learning and controlling algorithm is replaceable.
After startup, the *ControlAgent* registers it to all topic channels of its devices. So it will always be notified about the current state of its devices.

9.6.3 Cluster Viewer Agent

The *ClusterViewerAgent* is a graphical monitor of a cluster. It displays the states of all devices and offers the opportunity to change the state of the effectors (Figure 9.8). Predominantly this agent is used for observing clusters. But as a further step it could be expanded to real control panel.



```
Current device states are:
Light door (Ins.2):            1000
Light windows (Ins.3):         1000
Daylight sensor (Ins.4):       97.0
Precense detector (Ins.5):     0
Radiation East (Ins.6):        73.8
```

deviceID:	Ins.2
new value:	1000

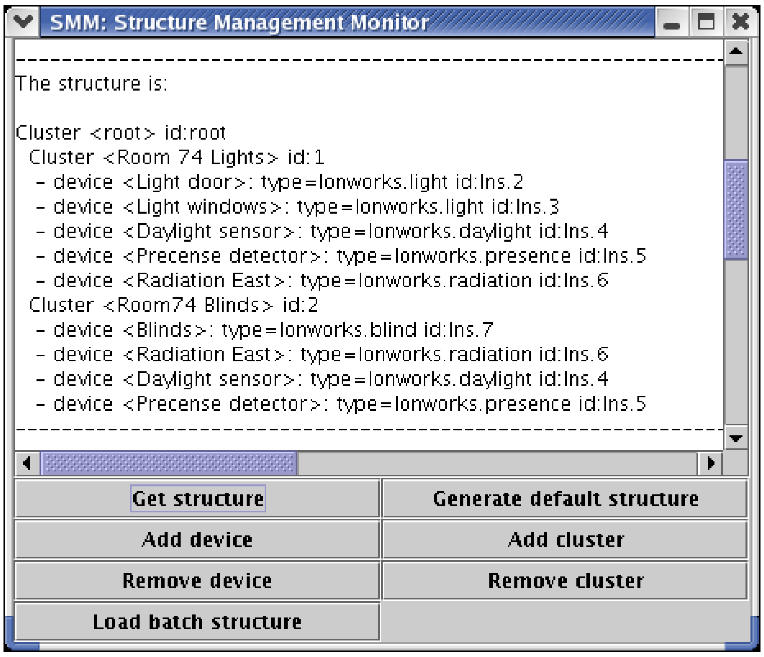
Figure 9.8: Graphical agent to observe and control devices of a cluster

9.6.4 Structure Monitor Agent

This agent arose of the necessity to modify the current structure in an easy way. The *StructureMonitorAgent* provides a graphical interface to add/remove devices and clusters (Fig. 9.9). The graphical user interface is keeped really simple, but it gives a general impression about the opportunities for a further implementation.

9.6.5 Structure Manager Agent

To keep the *StructureAgent* as simple as possible, we add only the basic functions (add/remove device...) to it. But there exists also some more complex tasks to modify the current structure. For example the adding



```
---------------------------------------------------------------
The structure is:

Cluster <root> id:root
 Cluster <Room 74 Lights> id:1
  - device <Light door>: type=lonworks.light id:lns.2
  - device <Light windows>: type=lonworks.light id:lns.3
  - device <Daylight sensor>: type=lonworks.daylight id:lns.4
  - device <Precense detector>: type=lonworks.presence id:lns.5
  - device <Radiation East>: type=lonworks.radiation id:lns.6
 Cluster <Room74 Blinds> id:2
  - device <Blinds>: type=lonworks.blind id:lns.7
  - device <Radiation East>: type=lonworks.radiation id:lns.6
  - device <Daylight sensor>: type=lonworks.daylight id:lns.4
  - device <Precense detector>: type=lonworks.presence id:lns.5
---------------------------------------------------------------
```

Figure 9.9: Graphical agent to manage the current structure

of all devices of one room or the movement of clusters or devices in structure.
All such tasks should be implemented outside of the core, so we created a *StructureManagerAgent*. This agents is waiting for *STRUCTURE_MANAGER_CMD* messages. If it receives such a message, it processes it and waits for a next instruction.

Chapter 10

Design

In this chapter we show the design concept of several agents. For further information read the JavaDoc [TZ03b] and the comments in the source files.

10.1 ABIAgent

Each agent in our framework is derived from the abstract class *ABIAgent*. This class take responsibilities to interact correctly with the ABLE framework. The ABIAgent itself is derived from *AblePlatformDefaultAgent*. For writing a new agent it is only necessary to implement the five abstract methods:

- `protected abstract void customInit(Map params)`
- `public abstract void receiveABIMessage(ABIMessage msg, String type)`
- `protected abstract void autoSave(String fileName, ABIMessage msg)`
- `protected abstract void recovery(String fileName)`
- `protected abstract void beforeQuit()`

The first called method (except the constructor) is `customInit()` in which the agent can be initialized. The map `params` contains all parameters as key-value pairs, which were entered in the console.

The most important method is `receiveABIMessage()`. The communication between the agents is handled by a messaging system. If an agent receives a new message, this method is executed. The frameworks offers an own message class *ABIMessage*. An *ABIMessage* is always defined by a message type (e.g. STRUCTURE_CHANGE, ADD_DEVICE, VARIABLE_CHANGED...). A message contains a parameter list with additional information. This list is realized with a map; contains also always a key-value pair. The message types and parameter keys are defined in the class *MessageConstants* as constants.

Before an agents is quitted, the method `beforeQuit()` is invoked. An agent can deregister here its interests or do some other stuff.

The methods `autoSave()` and `recovery()` implements the persistant function of an agent. The *RecoveryAgent* is responsible to trigger those methods.

10.2 Bus Abstraction

The design of the bus abstraction is divided into two parts. The first one is the general BusAgent which manages several device objects (*ABIDevice*). The other part implements bus specific stuff like the implementation of the *BusController* and the devices. So the *BusAgent* can manage any bus-type because of the defined interfaces (*BusController* and *ABIDevice*). Figure 10.1 shows the design model for a LonWorks-BusAgent implementation.

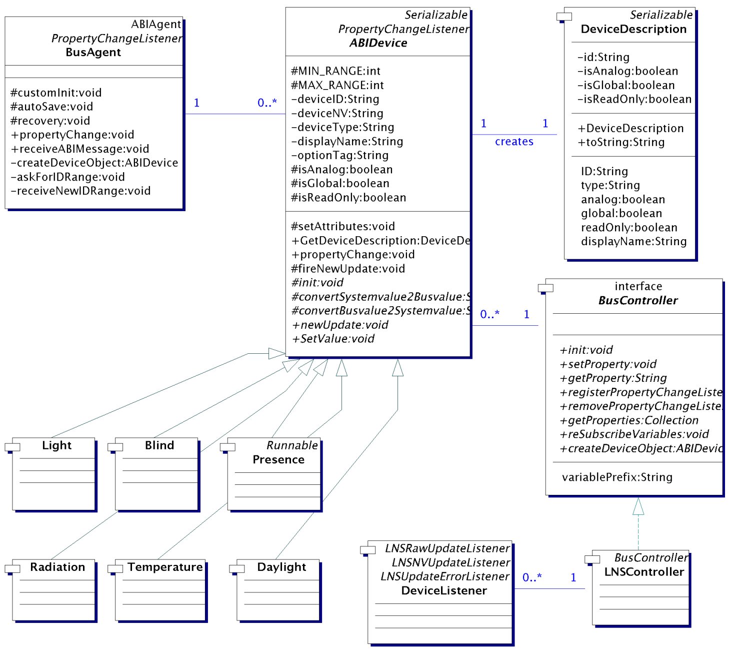


Figure 10.1: Incorporated classes of the BusAgent

The implementation is structured into the following classes:

- `BusAgent`: General default implementation of a BusAgent. It handles all the general stuff and delegate concrete commands to the bus controller or to the devices.
- `ABIDevice`: Interface class for the specific implementation of bus devices
- `DeviceDescription`: This general device description is produced by the specific implementation of a bus device and contains general information about it.
- `Light, Blind, Presence, Daylight, Radiation, Temperature`: This classes are specific implementations of LonWorks devices.
- `BusController`: Each controller type must implement this interface. The BusAgent gets access to the specific controller implementation via this interface.
- `LNSController`: The LNSController is a specific bus implementation for the access to the LonWorks bus.
- `DeviceListener` are used by the LNSController. It implements the required interface of the LonWorks package.

10.2.1 Device ID Management

It is possible to start several *BusAgents*, for each bus type one. But devices of these agents must have a unique ID. Since the *StructureAgent* is always started and there exists only one instance of it, it manages the ID's of the whole system.
After startup of a *BusAgent* it sends an *IDRange* request to the *StructureAgent*. This *StructureAgent* reserves a new ID range for this agent and attach the range to the answer. Now the BusAgent can assign device IDs within this range. If all IDs are spent, the *BusAgent* sends again a *IDRange* request.

10.2.2 Native Bus Implementation

Each type of bus has its own idiosyncrasies. These are implemented in the controller and the device classes. The interfaces *BusController* and *ABIDevice* hide this complexity from the *BusAgent*.
The implementation of the *BusController* is responsible for the communication with the bus system, that means to read sensors and to set effectors. Furthermore it has to notify the device instances about state changes of the sensors.
Each type of physical device is represented by its own implementation of the *ABIDevice*. This implementation maps the device ID to bus specific network variable (NV) and converts bus specific commands (like ′SET_ON 100.0.0′ in the LonWorks bus) to double values which are used in the framework.

10.2.3 Example on the Basis of LonWorks-Bus

Value Change

If a sensor changes its value, a *DeviceListener* will be informed by the LNSServer. This DeviceListener dispatches the notification to the corresponding ABIDevice (e.g. Light in Figure 10.2). There the bus specific command is translated into a system value. Now this system value is distributed via the *MessageDistributionAgent* to all interested agents by the BusAgent (Figure 10.2).

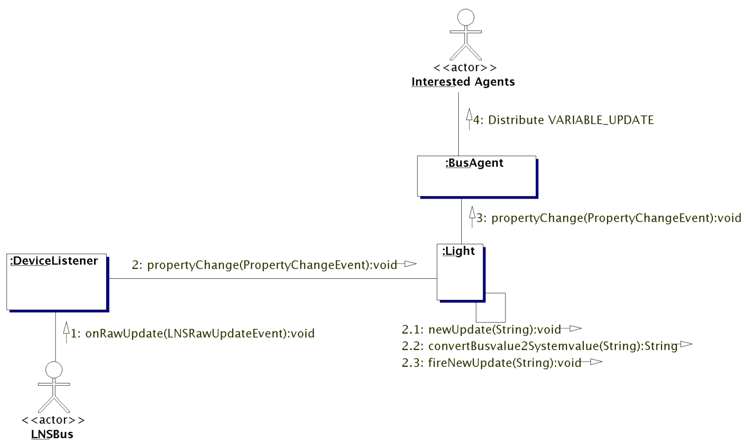

<<actor>>
Interested Agents
4: Distribute VARIABLE_UPDATE
:BusAgent
3: propertyChange(PropertyChangeEvent):void
:DeviceListener
2: propertyChange(PropertyChangeEvent):void
:Light
1: onRawUpdate(LNSRawUpdateEvent):void
2.1: newUpdate(String):void
2.2: convertBusvalue2Systemvalue(String):String
2.3: fireNewUpdate(String):void
<<actor>>
LNSBus

Figure 10.2: Collaboration of classes for a value change

Setting New Effector States

Each agent can change the state of an effector by sending a `SET_VARIABLE_VALUE` message to the corresponding *BusAgent* (Figure 10.3). The *BusAgent* itself gets access to the device implementation by the *ABIDevice* interface and set the new effector value. The device specific implementation is responsible to convert the system value into a correct bus command and send it via the bus controller to the effector device.
Because it is possible that there is more than one running *BusAgents*, it is necessary the find out, which of these agents is the correct one. This problem is solvable, because the device ID starts always with a bus specific prefix (e.g. 'lns' for the LonWorks bus). This functionality is already programmed in `DirectorySupport.getBusAgent(String deviceID)`. This method returns the correct *AgentDescription*, which is necessary to send a message to that *BusAgent*.

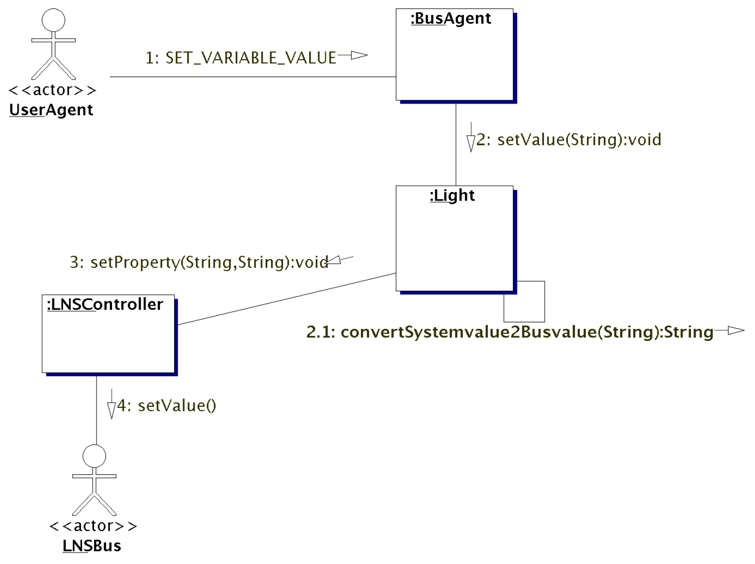


Figure 10.3: Collaboration of classes if a UserAgent sets the value of an effector to a new state.

10.3 Structure Abstraction

The *StructureAgent* is responsible to administrate the structure of clusters and devices. For that it uses the utility class *Structure*. The structure of clusters is strictly hierarchical and starts with one root cluster. Each cluster can contain several subclusters and several devices. It is permitted to add the same sensor device into different clusters (e.g. outdoor radiation sensors). Figure 10.4 visualizes the relationship between the incorporated classes.

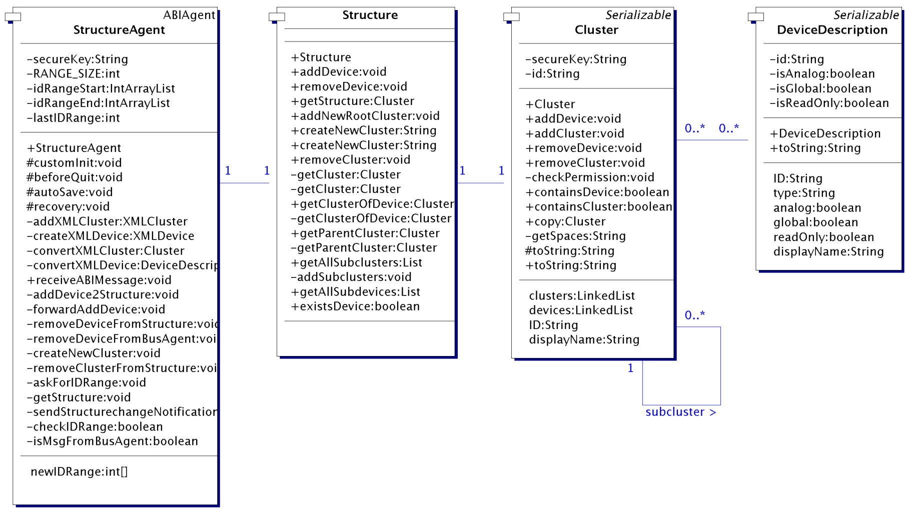


Figure 10.4: Incorporated classes of the StructureAgent

The implementation is structured into the following classes:

- `StructureAgent`: The StructureAgent manages the awarded ID ranges and handles all the incoming messages. Structure operations (add, remove...) are forwarded to the class *Structure*.
- `Structure`: This class manages the structure and make modifications on it. At startup the *Structure* contains one root cluster.
- `Cluster`: A cluster can contain several subclusters and several device descriptions.

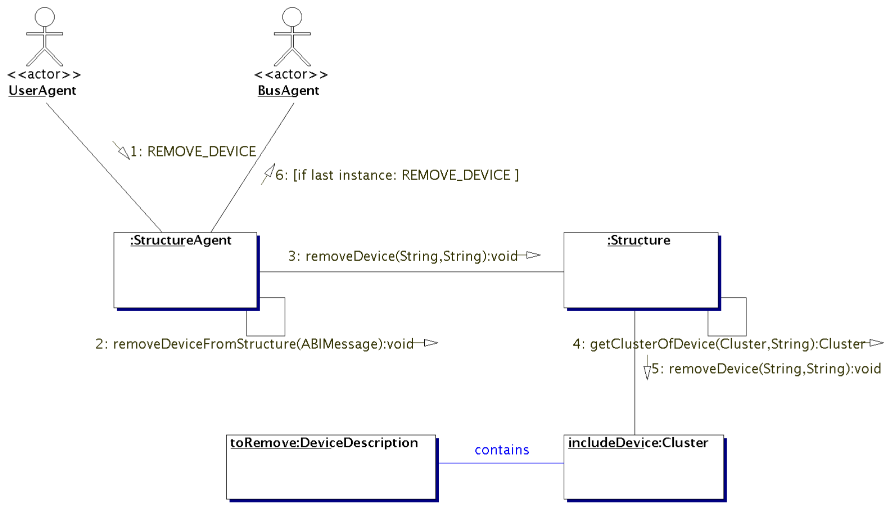
- `DeviceDescription`: Each device is described by an instance of this class. The most important attributes are *ID*, *DisplayName* and *RealOnly*. A device description can be placed in more than one cluster.

10.3.1 Structure Modification

Due to the fact that a device can used in several clusters the administration is also more complex. The following example gives an introduction to the dynamic concept of the StructureAgent. Further information about the interaction with other agents are described in chapter 9.5.5 on page 37.

Remove Device from Cluster

Figure 10.5 shows the regular program flow to remove a device from an existing cluster. A user agent sends the message `REMOVE_DEVICE` to the *StructureAgent* which removes the device from the corresponding cluster. After the successful removal the StructureAgent checks if the device is still used in an other cluster. If the last instance was removed, it send a message to the *BusAgent* to remove this device also there.



<<actor>>
UserAgent
<<actor>>
BusAgent
1: REMOVE_DEVICE
6: [if last instance: REMOVE_DEVICE]
:StructureAgent
3: removeDevice(String,String):void
:Structure
2: removeDeviceFromStructure(ABIMessage):void
4: getClusterOfDevice(Cluster,String):Cluster
5: removeDevice(String,String):void
toRemove:DeviceDescription
contains
includeDevice:Cluster

Figure 10.5: Collaboration of classes to remove a device

10.4 Control Agent

The *ControlAgent* is a general implementation to control the devices of a cluster. During the initial startup of the agent, the structure of the cluster (which should be controlled) is requested. Based on this the *ControlAgent* sets up a StateMemory and a Controller for each effector. So each effector is controlled by its own controller instance. The type of the used controller is defined in the startup parameter `ImplementationClass`. For a better performance the *StateMemory* caches for each device the last known state. To reduce the complexity and to improve the signal quality, the *StateMemory* merges all signal of the presence detectors together to one (by OR gating).
Due to the defined *Controller* interface, the *StateMemory* is able to handle many different learning and controlling algorithms. They just have to implement the defined interface (Figure 10.6).

The implementation is structured into the following classes:

- `ControlAgent`: The ControlAgent interacts as a gateway between the effective implemented controlling algorithm and other agents. A ControlAgent is attached to one cluster. It registers for each included

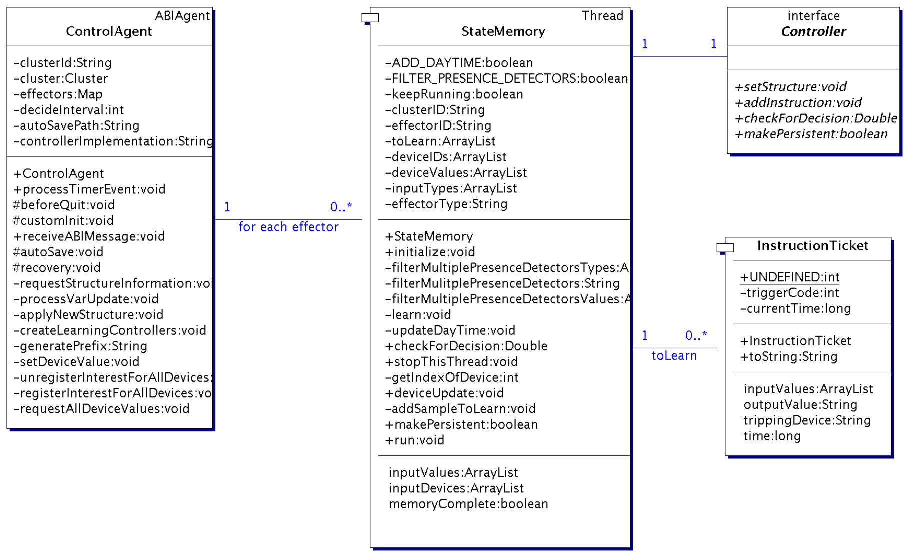


Figure 10.6: Incorporated classes of the ControlAgent

device its interests by the *MessageDistributionAgent* to be informed about any state modification of the devices. The ControlAgent checks periodically if there is a new decision available.

- `StateMemory`: The ControlAgent set up a StateMemory for each effector of the cluster. This memory caches the last known state of each device. On every change of a device state, the StateMemory creates an InstructionTicket to inform the controller.

- `InstructionTicket`: The instruction ticket is like a snapshot of the current environment. Based on these information the controller is able to learn.

- `Controller`: In order to handle different learning/controlling algorithms, we defined this general controller interface. Each developed algorithm must implement it.

10.4.1 Learning

The controlling algorithm has two important tasks: to learn the user behavior (Figure 10.7) and to control the effector state (Figure 10.8). According to the requirements one or both tasks are implemented.
Figure 10.7 visualizes the collaboration of the classes when a device changes its state. On every value change the *ControlAgent* is notified with a `VAR_UPDATE` message (1). The *ControlAgent* informs the *StateMemory* about the changement. The *StateMemory* updates the cached value and creates an *InstructionTicket* to inform the controller about the changement whereby the presence signals are combined to one (4 and 5). This new *InstructionTicket* is placed into an internal buffer (6). An other thread checks periodically this buffer for new instructions and forwards them to the controller (7). The reason for these instruction buffer is, that we want to provide that the method `receiveABIMessage()` in the *ControlAgent* is finished as fast as possible to be ready for new messages. With this concept the proceeding of the message and the learning is thread decoupled.

10.4.2 Decision Making

The `ControlAgent` checks periodically if there is a new decision available for the current environment state (Figure 10.8). If the *Controller* suggests a new decision, the *ControlAgent* sends a `SET_VARIABLE_VALUE` message to the *BusAgent* to control the effector.

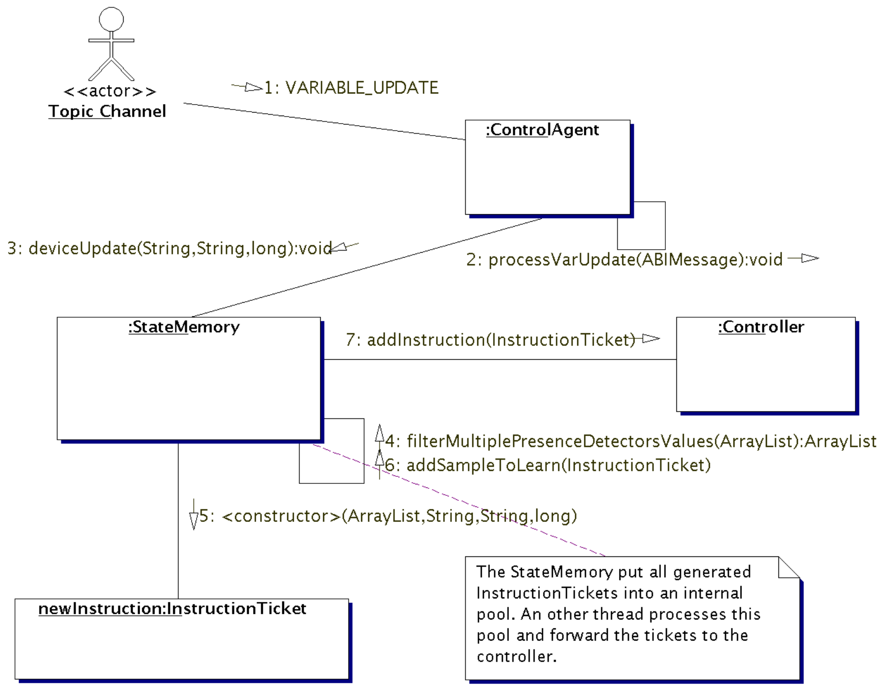

<<actor>>
Topic Channel
1: VARIABLE_UPDATE
:ControlAgent
2: processVarUpdate(ABIMessage):void
3: deviceUpdate(String,String,long):void
:StateMemory
7: addInstruction(InstructionTicket)
:Controller
4: filterMultiplePresenceDetectorsValues(ArrayList):ArrayList
6: addSampleToLearn(InstructionTicket)
5: <constructor>(ArrayList,String,String,long)
newInstruction:InstructionTicket
The StateMemory put all generated InstructionTickets into an internal pool. An other thread processes this pool and forward the tickets to the controller.

Figure 10.7: Feed the control algorithm with instructions about value changes

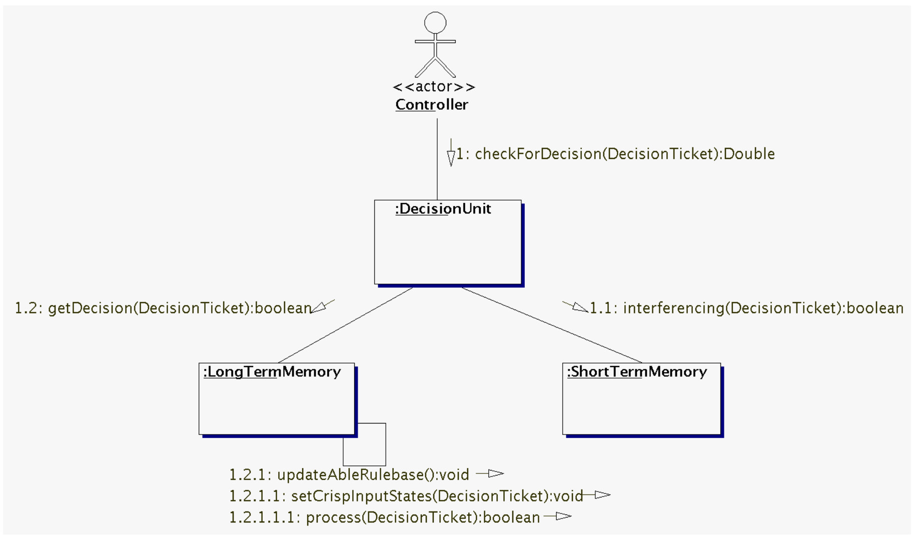

<<actor>>
Controller
1: checkForDecision(DecisionTicket):Double
:DecisionUnit
1.2: getDecision(DecisionTicket):boolean
1.1: interferencing(DecisionTicket):boolean
:LongTermMemory
:ShortTermMemory
1.2.1: updateAbleRulebase():void
1.2.1.1: setCrispInputStates(DecisionTicket):void
1.2.1.1.1: process(DecisionTicket):boolean

Figure 10.8: ControlAgent checks periodically if there is a new decision available

10.5 Genetic Fuzzy Controller

10.5.1 Overview

The *GeneticFuzzyController* is one specific implementation of the *Controller* interface and is used by the *ControlAgent*. This controller implements a learning and a controlling algorithm. Its concept is described in part II. This section explains the design and implementation of the following parts:

- Knowledge storage of temporary and long term memory
- Knowledge rewarding
- Rulebase generation
- Decision making

All processes of the GeneticFuzzyController are based on fuzzified inputs values. So the first step of the controller is always the convertation of the numerical value from the environment (stored in the *InstructionTicket*) into fuzzy values (Figure 10.9).

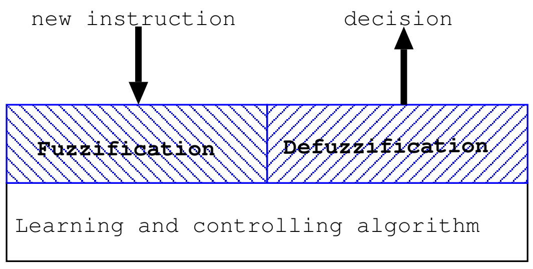


Figure 10.9: Because the algorithm bases on fuzzy logic, all instructions are converted as first into fuzzy values.

10.5.2 Fuzzy Memory

Our algorithm builds up a memory base for temporary and long term behavior. Therefore the algorithm has to gather information about the system behavior. As in chapter 5 described, we collect information about two situations:

- User instructions
- Rewarding of correct controlling behavior

For each of these situations the knowledge is build up resp. decreased. The class *FuzzyAnchor* administrate this knowledge of one atomic fuzzy area. An atomic fuzzy area means, that each edge of this fuzzy area enclose only one fuzzy state. This *FuzzyAnchor* collects for each possible output state the following two characteristic value:

- Truth of the last user action in this fuzzy area: Collected in the attribute `lastActiveDateness`
- "Lifelong" memory of this fuzzy area: Collected in the attribute `dateness`

Figure 10.10 shows the collaboration of the algorithm classes to build up the temporary knowledge. The process is triggered for example by a dissatisfied user, when he is not happy with the current controlling behavior and changes the state of the effector. This produces an *InstructionTicket* for the controller (step

1). The *FuzzyGeneticController* recognize that it is a punishment of the user and will put the new knowledge into the temporary knowledge base. For that it fuzzifies the instruction (step 2.1) and asks the fuzzy anchor pool for all affected anchors (step 2.2). Now it forces each of this anchors with the maximum weight for the new effector state.

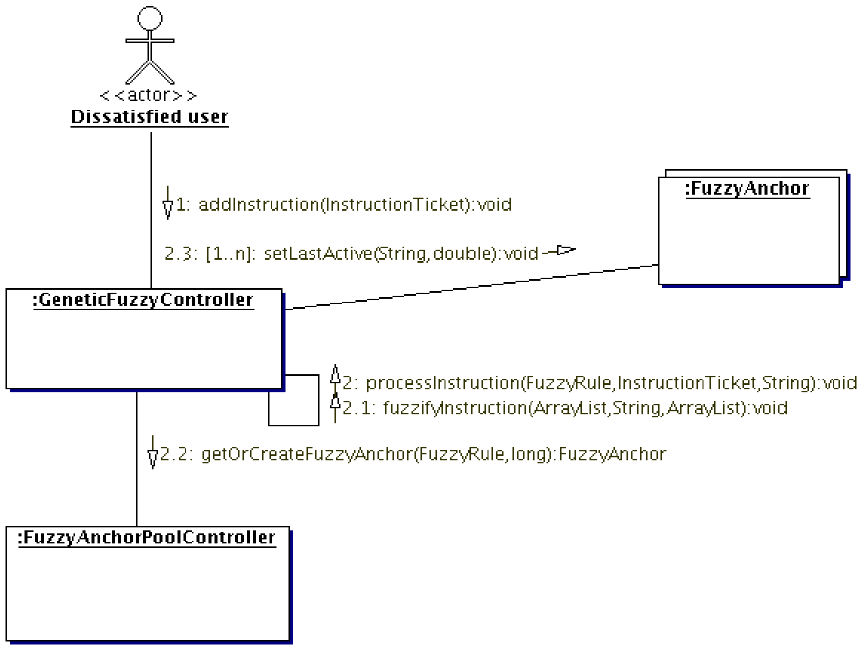

<<actor>>
Dissatisfied user
1: addInstruction(InstructionTicket):void
2.3: [1..n]: setLastActive(String,double):void
:FuzzyAnchor
:GeneticFuzzyController
2: processInstruction(FuzzyRule,InstructionTicket,String):void
2.1: fuzzifyInstruction(ArrayList,String,ArrayList):void
2.2: getOrCreateFuzzyAnchor(FuzzyRule,long):FuzzyAnchor
:FuzzyAnchorPoolController

Figure 10.10: A user instruction triggers the process to build up a temporary knowledge.

10.5.3 Rewarding

Due to the fact that the user normally is active if he is dissatisfied, we never get a direct reward from him. But if there is no feedback from the user, we can interpret this as satisfaction/reward.
The class *Rewarder* is responsible to produce these rewards. It rewards the long term knowledge if the system stays in the same (fuzzy) condition for a longer time. With every state change of the system the timeout is re-triggered (Figure 10.11). This behavior is realized by starting a timeout thread on every state change. When this thread expires, it checks if it is still the newest one. If there was meanwhile a state change the thread terminates without an action. Otherwise it rewards the long term knowledge and starts again.
Each reward increases the weight (dateness) of the current output state and punishes the weights of the other output states.

10.5.4 Evolutionary Rulebase Generation

The rulebase generator combines the gathered long term knowledge of each atomic fuzzy area together in more general rules. We use a genetic algorithm to create this fuzzy rulebase. The algorithm starts with the current rulebase as an initial population and tries to find better rules by crossovering and mutation. We use the framework JGAP [Jav] to evolve a genotype. So we don't care about the implementation of all the selection, we just implemented the following essential functionality:

- Genes
- Fitness function
- Crossover operation

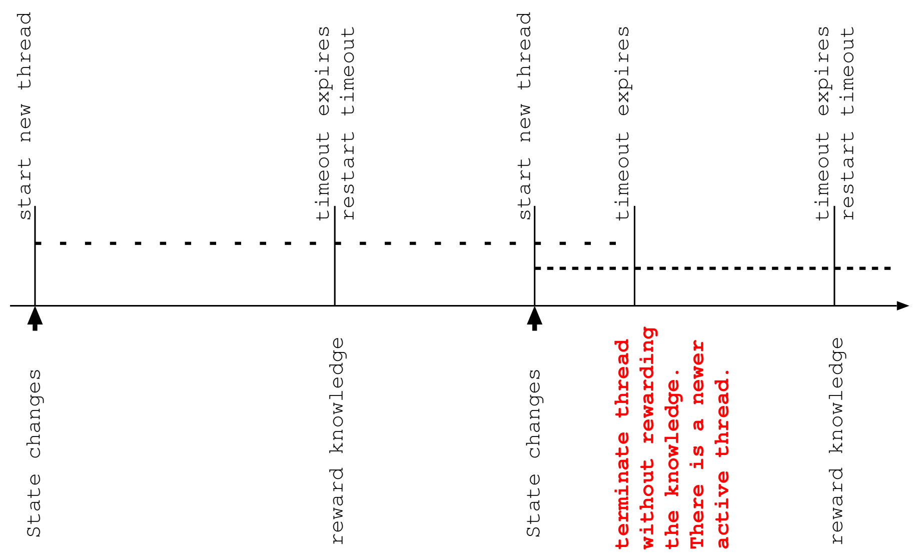


Figure 10.11: Timeout handling for knowledge rewarding.

- Mutation operation

As described in chapter 7 the whole genetic process is divided into three steps: The evolving of a new rulebase, the selection of the best solution and the boosting of the uncovered anchors.
For each loop (Figure 10.12) the best founded solution (rule) is applied to the new population. This loop is repeated until all anchors are covered by a rule or the maximum number of rules is reached. The process is controlled by the *Learner* class.

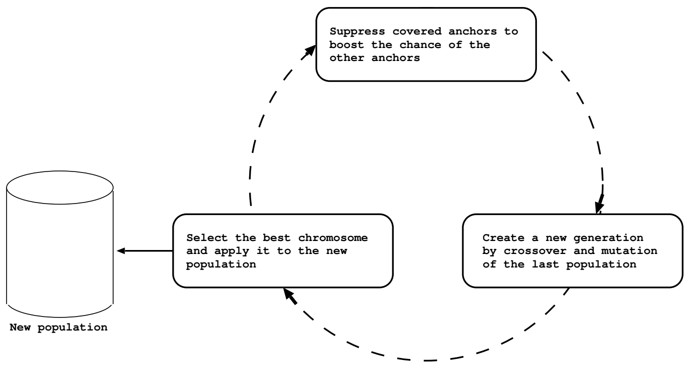

Suppress covered anchors to boost the chance of the other anchors
Select the best chromosome and apply it to the new population
Create a new generation by crossover and mutation of the last population
New population

Figure 10.12: The schematic process of the creation of a new population of rules with a genetic algorithm.

Chromosome

A chromosome encodes the information of a fuzzy rule into a predefined number of genes. Each of this gene encodes the states of one fuzzy dimension. The order of the genes is always the same like the order of the devices in controlled cluster. The method `Learner.translateIntoChromosome()` converts an existing fuzzy rule into such a chromosome. The chromosome class itself is realized by the JGAP framework.

Genes

An instance of the class *FuzzyGene* represents one input dimension of the fuzzy rule. It contains a bit vector (`fuzzyStates`) to encode the valid states of this input. Each bit represents one fuzzy set and is true if the state is included in the fuzzy rule. For example the input dimension radiation is divided into five fuzzy sets (A-E) whereby the rule covers only the first, the second and the last fuzzy set (A,B ad E). So the bit vector of the gene is `11001`.
The first gene of a chromosome represents the output action. This gene has a special character, because it is only possible to set one state simultaneously. This specific behavior is implemented by derive *FuzzyOutputGene* of *FuzzyGene*.

Fitness Function

JGAP permits the usage of an own fitness calculation. The class *BestMatchFitnessFunction* (derived of *org.jgap.FitnessFunction*) calculates with the method `evaluate()` the fitness of a chromosome. Unfortunately the fitness must be represented by an integer value bigger than one. So we calculate internal normal a double value between zero and one, but enlarge it at the end to a value from one to thousand.
The detailed calculation of the fitness function is discussed in the chapter 7 and 13.1.1.

Crossover Operator

The class *FuzzyCrossoverOperator* implements our specific crossover operator in the `evaluate(Chromosome solution)` method. It selects randomly two chromosomes of the population and divide it randomly into two parts. With this four parts, it creates two new chromosomes by crossover. The figure 10.13 illustrate this process.
It is possible that a resulted chromosome is invalid. This could happen, when one or more genes of the chromosome contains no true-bits. In that case the chromosome is not defined in this dimension. We throw away such chromosomes by setting the fitness to the minimum.

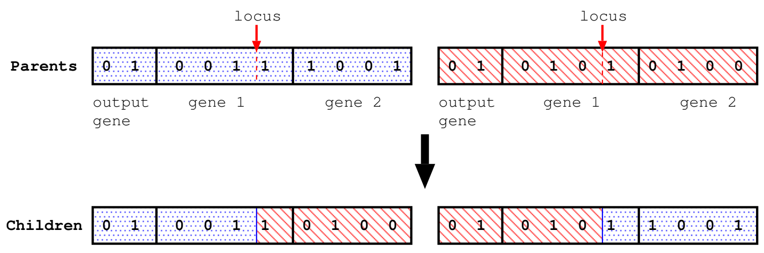


Figure 10.13: Crossover

Mutation

The mutation is not implemented as an own operator. JGAP use for the mutation the method `setToRandomValue()` of the genes. JGAP selects automatically few genes and let them mutate. The above-named difference between an output gene and an input gene is reflected in the different implementation of this `setToRandomValue()` method. A normal gene can contain more than one true-bit whereas an output gene always contains one single true-bit.
The genetic algorithm has to be configured in advance with a specific mutation-rate

Boosting

Without boosting, it is possible that a genetic algorithm will only find one (nearly perfect locale rule). But we are interested to get an optimal set of rule which covers all used states. So we suppress the weight of covered anchors to boost the changes of the other anchors. For that we expand the *FuzzyAnchor* with a boosting value. At the beginning of an evolution, this boosting value is initialized, based on the current datenesses. So we can reduce this boosted weight during the evolution without touch the real gathered knowledge.

10.5.5 Decision Making

On every environment change the controller checks the temporary and the long term rulebase for new decisions. Therefore it creates a *DecisionTicket* with the current system state as crisp and as fuzzy values. Based on these ticket information the temporary and the memorized rulebase can make their decisions. Both of them write their suggested decision into the ticket and then the *DecisionUnit* evaluates and selects the better solution. This is returned to the controller which sends a request to set the effector to the new state. Figure 10.14 illustrate the decision process with a collaboration diagram.

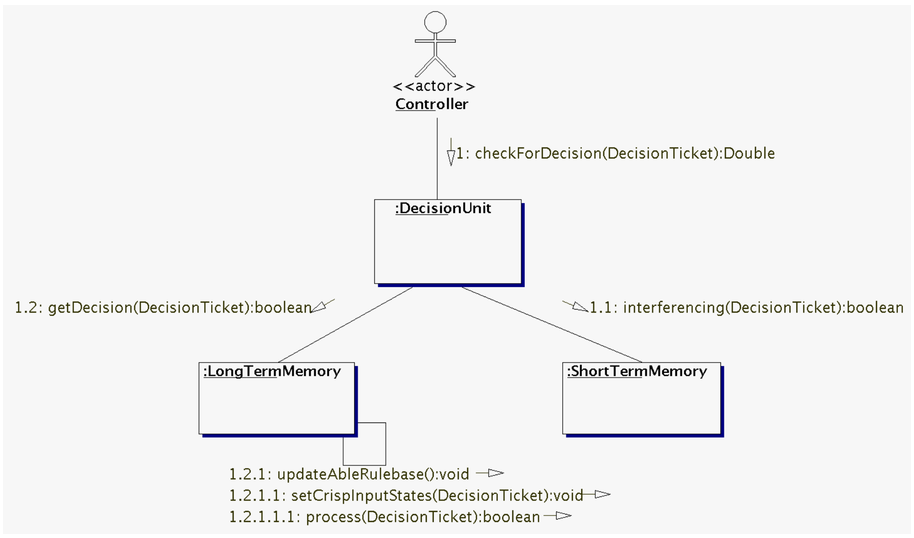

<<actor>>
Controller
1: checkForDecision(DecisionTicket):Double
:DecisionUnit
1.2: getDecision(DecisionTicket):boolean
1.1: interferencing(DecisionTicket):boolean
:LongTermMemory
:ShortTermMemory
1.2.1: updateAbleRulebase():void
1.2.1.1: setCrispInputStates(DecisionTicket):void
1.2.1.1.1: process(DecisionTicket):boolean

Figure 10.14: Classes collaboration for decision making

The controller retain the last few made decision in the *DecisionHistory*. This history is programmed as a ring buffer and is limited to the last 50 decisions. Based on this information the controller can detect two cases of problems:

- The user punishes a recently taken decision
- The system decide within a short period an converse action (System toggle)

In both cases the controller punishes the last decision by setting the temporary knowledge of the ancient state to the new output action.

10.5.6 ABLE Rule Language

The ABLE framework from IBM provides a library of different learning and controlling algorithm. It uses a common syntax to persist this knowledge. This syntax is called ABLE rule language (ARL). It is a matter of human readable representation of the knowledge. The syntax is itself is similar to java. For further information about ARL we refer to [ARL02].

10.5.7 Utility Classes

To keep the main classes as clearly as possible, we use several utility classes. The following brief description about them gives a overview about their functionality. For further information we refer to the JavaDoc [TZ03b].

- `RuleSupport`: The RuleSupporter is used from several other classes to get ABLE fuzzy variables of the environment. It is also able to create ABLE conditional rules.
- `EnvironmentInfo`: General information about the environment are available in the EnvironmentInfo. This class knows e.g. how many input dimensions exist or into how many fuzzy states a specific input dimension is divided.
- `FuzzyRuleSetFactory`: This factory creates for a given environment (several sensors and effectors) the initial ABLE ARL ruleset. This ARL ruleset consists of the fuzzy variables for the devices and the supporter methods to fuzzify a crisp value.
- `FuzzyVariableFactory`: The FuzzyVariableFactory creates concrete fuzzy variable instances from the ABIFuzzyTemplates.
- `ABIFuzzyVariable`: ABLE represent fuzzy variables as an instance of AbleFuzzyVariable. Unfortunately the ABLE class does not support a copy constructor and provides only a limited access to its attributes. So we derived the ABIFuzzyVariable form AbleFuzzyVariable and added the needed functionality.
- `ABIFuzzyTemplate`: An ABIFuzzyTemplate contains the fuzzy set definition of one specific device type. This fuzzy sets are defined in the template ARL-file (fuzzyTemplate.arl). The FuzzyVariableFactory contains for each possible device type an instance of this class.
- `fuzzyTemplate.arl`: This arl file defines the fuzzy sets of each device type.

Part IV

Results and Discussion

Chapter 11

Assumptions

In this chapter we explain our assumptions, which we defined before we developed and further analyzed our new learning and controlling architecture. Because our diploma thesis is very limited in time we had to make a few assumptions which we are trying to prove here in this part as sufficient as possible. However, we believe an extensive and long-term analysis in the future is required in order to understand really to which extend the system increases user comfort and reduces energy consumption.

Our novel learning and controlling architecture assumes:

1. Only a small part of the whole fuzzy state space will be used from a combination of input sensors. Rules which covers unused or even unreachable space are not useful also from a computational point of view. A learning algorithm should focus on these samples which the system has really produced.
2. The design of the fuzzy membership functions is a crucial point in order to control effectors efficiently. Using well designed fuzzy membership functions, we believe each atomic fuzzy area will represent a uniform action (e.g. light = on or light = off).
3. Neighbor areas within the fuzzy state space are dependent on each other. Fuzzy states which are at the one end of a sensor input range are less dependent with one from the other end (temperature = cold and temperature = hot are less dependent than warm and hot).
4. The user is not computable. He wants for the same condition mostly the same action, but the behavior can change temporary.

Within the next chapters of this result part we are trying to prove these assumptions. Due to the fact that the system was only running for less than 2 weeks, we can only present first results and discuss them. A detailed analysis has to prove it in terms of the long-term behavior.

Chapter 12

Acquisition and Evaluation of Temporary and Long-Term Knowledge

12.1 Realization

Within this chapter we explain the parameters and configuration of our learning architecture which was evaluated in a simulated and real-world environment. Parameters are evaluated and defined during an experimental phase and proved to generate good results. A more detailed analysis would be thinkable in order to define them more accurate.

12.1.1 Aging and Forgetting

Acquired knowledge (temporary and long term) is not permanent and static. It is under a continuous modification. The temporary knowledge is especially designed to keep only temporary information. Instructions which have to be learnt instantly but they getting forget also within a short period. On the other side, the long-term memory holds a more solid knowledge. The dynamic behavior can also be defined in terms of aging and forgetting, but the specific knowledge is rather durable.

In our memory base, we administrate the knowledge for the output states of each atomic fuzzy area (fuzzy anchor) separately as a weight. These weights can be rewarded or punished. If one output state of this fuzzy areas is rewarded, all other within this area will automatically decrease (punished).

During a reward, we accumulate a constant reward value to the current weight. The punishment is realized by a decay factor between 0 and 1. The decay factor defines, how long a received reward affects the weight: the nearly this decay factor to 1 the longer a reward will survive.

12.1.2 Rewarding

Because the user is not able to generate a positive feedback (see chapter 4), we have to determine the degree of user satisfaction by ourself. As in chapter 5.3.3 described, we distinguish between two stages: the verification and the activity.

On every state change of the system, the new state must survive the verification state before we generate the first reward. The duration of this stage is defined by the constant time period $t_{verification}$. If the system is still in the same state after the verification, the reward will switch into the activity state. As long as the system stays in this stage, we produce periodically a new reward instruction (time period $t_{activity}$).

The rewarding could also be realized based on the real sensor activity. However, we believe that the time duration for which a subset of the environment is in the same fuzzified conditions is a more crucial indication that users within this environment are satisfied or would otherwise change its state.

12.2 Parameter Configuration

The configuration of the memorized knowledge is defined with four aspects:

- **Decay factor:** The knowledge (acquired as a weight) will decrease periodically (DecayInterval). In fact the accumulated weight is multiplied with this decay factor D.
- **Reward value:** With every reward the knowledge-weight is incremented by this reward value R.
- **Thresholds:** Because the weight never reaches zero, it is necessary to define a threshold. If a weight slides under this level, we do not consider this weight for further calculations. Both memory units (temporary and long term) have their own threshold T_T and T_L.
- **Reward interval:** Each current state of the system, which does not changes for a longer time, can be interpreted as a reward of the controlling behavior. So the knowledge has to be rewarded by the *reward value.* This step is also repeated periodically. The frequency can be adjusted by the parameters $t_{verification}$ and $t_{activity}$. On each reward, all other output states of the fuzzy area are punished (decreased).

We predefined the configuration that the lifetime of a temporary reward is not able to relearn the behavior of the long-term memory. Such a reward has to be generated more frequent or periodically. Therefore, we suggest the following configuration:

- Decay factor (D):	0.985
- Reward value (R):	1.0
- Threshold temporary memory (T_T):	0.7
- Threshold long term memory (T_L):	0.9
- Verification timeout ($t_{verification}$):	30 sec
- Reward interval ($t_{activity}$):	after 5 minutes in the same fuzzy state

For this configuration the figure 12.1 visualizes the relearning of a behavior in the long term knowledge. It illustrates on the right end of the chart a stable perfect learned behavior with the maximum possible weight

$$w_{max} = R/D = 66.67 \tag{12.1}$$

for the first output state and an empty weight for the second. If we now learn always the opposite behavior, the first output decreases with every new reward of the second. The cross point of this two functions is reached after 3.75 hours. After a total of 23.7 hours, the weight of the first output state slides under the threshold T_L.
In contrast, a single reward in the temporary memory has an impact for

$$t_{reward} = log_D(T_T) = 23.59 \text{ rewards} \approx 1.967 \text{ hours} \tag{12.2}$$

So you can reach the crosspoint within two user punishments.

Consider that the above calculated durations are not directly real time durations. A fuzzy area is only rewarded respectively punished if the system is situated in that state. During our analysis, the system was approximately two hours in each used fuzzy area. Based on this approximation, a single reward in the temporary memory survives one day. Furthermore the rewarding interval is not constant. The first reward is released after $t_{verification}$, every further after a time period of $t_{activity}$.

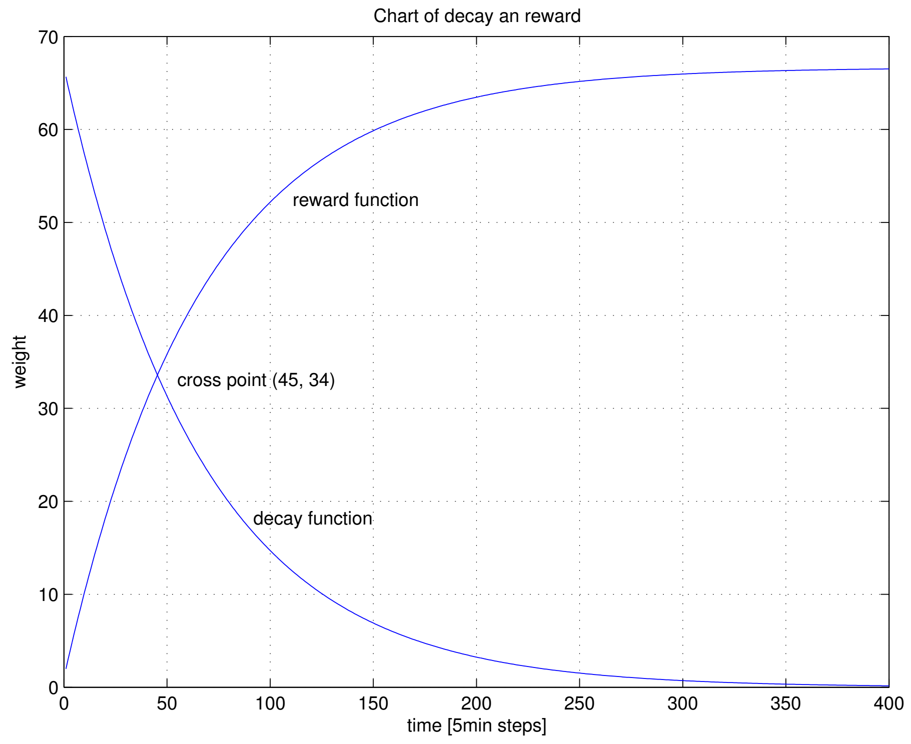


Figure 12.1: Reward and decay function over the time.

12.3 Results

12.3.1 Rewarding

The above mentioned configuration creates rewards if the system stays for five minutes in the same fuzzy area. But if the system changes its state within 5 minutes, the system produces no reward. The duration, in which the system stays in a specific fuzzy state depends a lot on the definition of the fuzzy membership functions.

To analyze the concept of rewarding, we looked at a time period of 40 hours of the running system. Within this term the system reaches 25 different fuzzy areas and produced 557 rewards (Table 12.1). Averaged the system spent totally 1.6 hours in a fuzzy area. Figure A of 12.2 shows the distribution of the rewards in more detail. Two-third of the fuzzy areas received more than ten rewards.

Fuzzy area statistic of a time period of 40h	
Number of reached anchors	25
Number of rewards	557
Number of user punishments	4
Mean summized residence time	1.6116 hours

Table 12.1: Statistic of the system during a test period of 40 hours.

Figure B of 12.2 shows the accumulated relative weights of the reached anchors. The maximum weight is limited to $\pm 66.\bar{6}$. A positive weight represents the action `on`, a negative `off`. Especially the knowledge to turn off the light is well learned (due to the EnergyAgent). Except three states, all others suggests clearly to turn on the light. These three areas have a weight near to zero. In comparison with the number of rewards of these areas, we see that the system almost never stayed in that states.

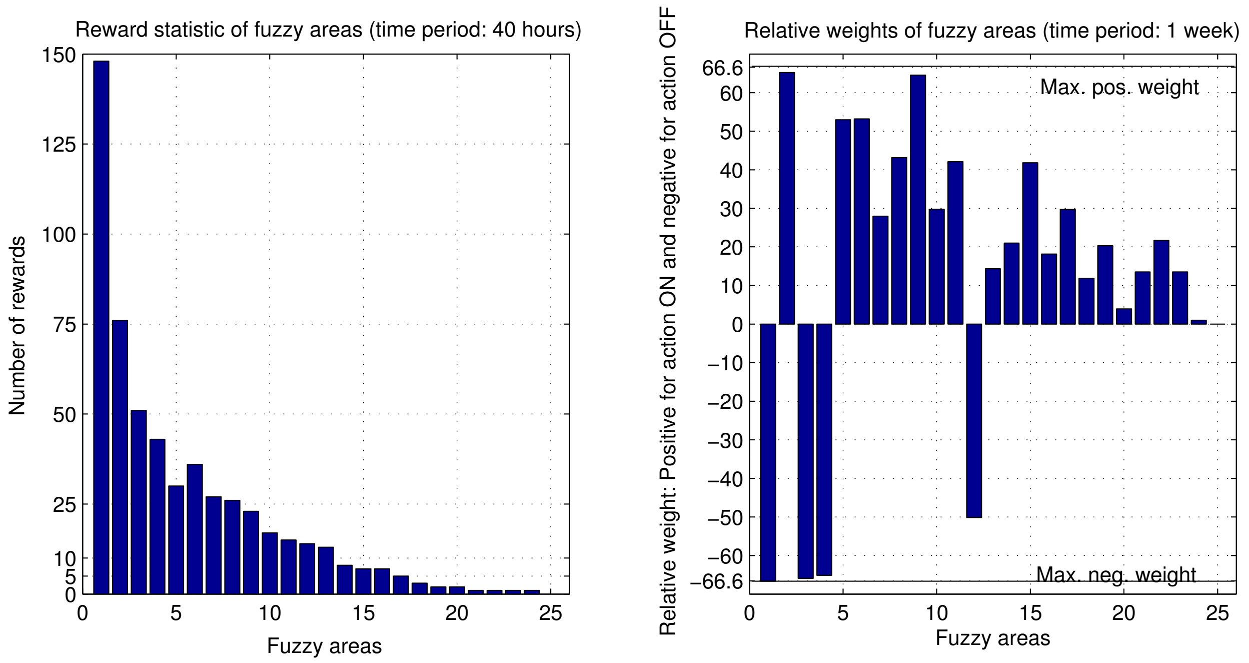


Figure 12.2: Distribution of rewards and relative weights. The order of the fuzzy areas is in both diagrams the same.

12.3.2 Dynamic Behavior of Learnt Knowledge

The learning algorithm operates with the above described concept and configuration since one week. Based on the log data of this period of time we analyzed the dynamics of the knowledge and explain them in the following section.

The first simple example is visualized in figure 12.3. At this particular case, the long-term knowledge to turn off the light develops to the maximum. Of course this is a consequence of the place of the fuzzy area: the time is night and nobody is presence. In that case, our *EnergyAgent* make sure that the lights are turned off. So this knowledge is rewarded every day. The chart does not increase continuously, it looks more like a step. The reason for that is our concept of rewarding. As above described we reward this specific fuzzy anchor only if the system is within that state. Obviously this is only true for a part of the day, because the daytime is also an input dimension.

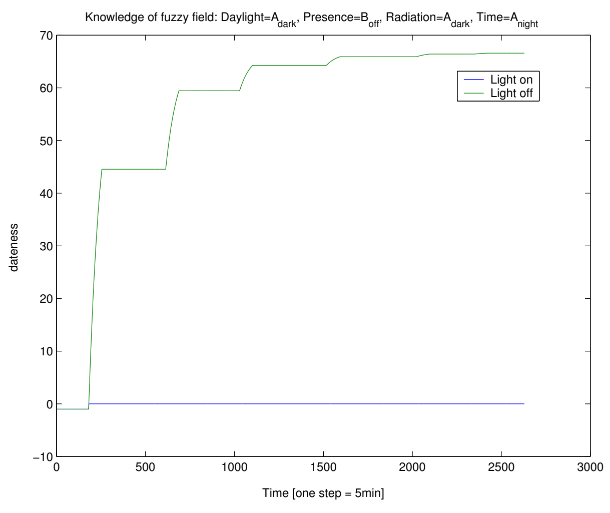


Figure 12.3: This fuzzy anchor learnt nearly perfect to turn of the light, if it is night and nobody is there.

Figure 12.4 shows a real user learnt behavior. This fuzzy area represent the action to turn on the light in the morning if someone is presence and it is dark. Around the time 1600 the behavior of the user changed temporary and the off-weight increased a little. But only after a short period, the behavior was again like before and up-to-then the off-weight decreases with every reward for the on-weight.

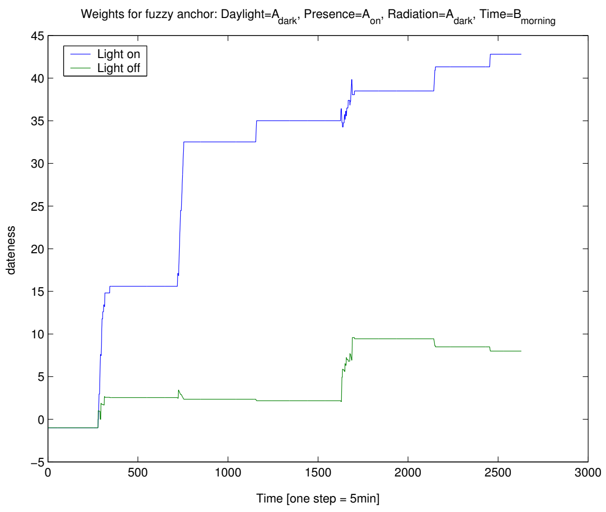


Figure 12.4: Generally the light was turned on. In some converse cases, the off-curve increases for a short time.

Another interesting example is the fuzzy area illustrated in figure 12.5. It represents a fuzzy area with the time constraint midday, but still a dark daylight and radiation. The user learnt firstly to turn off the lights, whereas from another time point on the light should be turned off. With tuned fuzzy membership functions it would be possible to make such areas more sensitive. If both weights are almost equal and not near zero it is also difficult for our learning algorithm to find an optimal rule which covers this area with a good fitness.

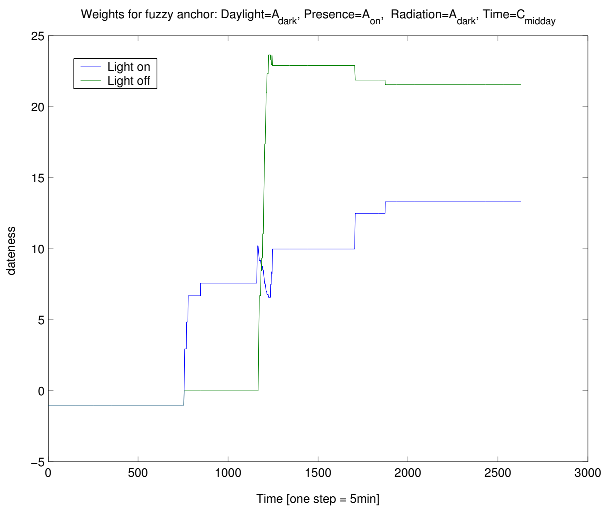


Figure 12.5: This fuzzy anchor relearn its behavior from on to off

Chapter 13

Analysis of Evolutionary Learning Algorithm

After the verification of the fuzzy memory in the last chapter, where we showed how knowledge can become truth and vice versa loose significance, this chapter discusses results of the evolutionary algorithm that learns a fuzzy rulebase based on that long-term memory.

We give first brief information about the setting up and present afterwards results of the learning algorithm applied to a typical classification simulation and then applied to a real commercial office building where people do normal work.

13.1 Realization

The principle and architecture of the fuzzy logic controller which is used for making long-term decisions is explained in detail in chapter 7. We give here explanations of parameters and discuss determined results.

13.1.1 Fitness Function

The fitness function like defined in equ. 7.8 specifies how good a solution is and increases the chance of a good solution to be reselected in the natural selection.

The fitness function contains three factors which all have to be satisfied in order to have a good solution. The first one is an error factor that is near zero if the found solution covers many wrong-classified fuzzy areas. The middle forces the solution to amplify itself as long as it covers not all equal classified areas and the third one judges if the solution covers beside the equal classified also wrong or not classified areas.

In experimental analysis we have seen that the first factor is a crucial one. Because the current fuzzy interference engine is up to now not able to incorporate also a given weight for each rule we don't want overlapping rules which postulate a different action. We weight therefore the error factor more than the other two by square it (see also equ. 7.8).

$$q = 2 \tag{13.1}$$

13.1.2 Boosting Factor

The evolutionary learning algorithm uses an iterative rule learning (IRL) approach where first the most important fuzzy areas have to be covered with a rule. After a new evolution the relative importance b_i of all fuzzy areas which are sufficiently covered by the fittest solution c_i will be decreased. This process modifies the current distribution of the input samples which is done with a boosting algorithm. We use a fuzzy variant of AdaBoost [FS96] and the amount β of decreasing the relative importance b_i of all covered fuzzy areas depends on the accumulated weight of all covered fuzzy areas. If these areas are only well classified areas the error will be small and the decreasing factor β will be nearly 1.0.

$$E(c_t) = \frac{\sum_{i=1|Act(a_i)\neq Act(c_t)}^{n} w_i}{\sum_{i=1}^{n} w_i} \tag{13.2}$$

$$\beta = 1 - E(c_i) \tag{13.3}$$

The relative importance b_i of all covered fuzzy areas will be decreased by the boosting factor β:

$$b_i = b_i * \beta \tag{13.4}$$

13.1.3 Population Size, Number of Generations and Mutation Rate

During the evolution of a given rulebase which has to adapt itself in order to match as good as possible to the current long-term memory, the evolutionary algorithm evolves the fuzzy rules over several generations. If the algorithm has to learn a rulebase from scratch we generate an initial population based on the current fuzzy areas.

We are using the following parameters within the evolutionary algorithm:

- Population size:	60
- Number of generations:	20
- Mutation rate:	1/100

Before every invocation of a new evolution, the genetic encoded fuzzy rulebase will be translated into a population. The population will be filled with the fuzzy rulebase as many times as possible. The genetic operations, the crossover and mutation, which are explained in the implementation chapter 10.5 are specially designed to process fuzzy genes.

13.2 Results in a Simulated Environments

To find out which behavior is learnable with our learning algorithm, we defined an artificial environment with two input dimensions. In our simulation we feed a predefined constant behavior pattern into the system and compare it with the resulting fuzzy rules. The two input dimensions contain six possible states. The output has two possible actions, `on` or `off`.

Of course in a real environment the behavior changes and consequently this simulated constant environments are an ideal world. But we showed in chapter 12 that in many fuzzy areas the behavior of the stored knowledge is the same during a longer time period.

13.2.1 Disjoint Decision Space

The first tested configuration consists of two disjoint classified areas. One turns on and one turns off the light (see figure 13.1). It visualizes the accumulated knowledge. There are nine learnt fuzzy areas with the action A in the top left corner. On the bottom right, there are nine areas with the action B. The white areas have no accumulated knowledge.

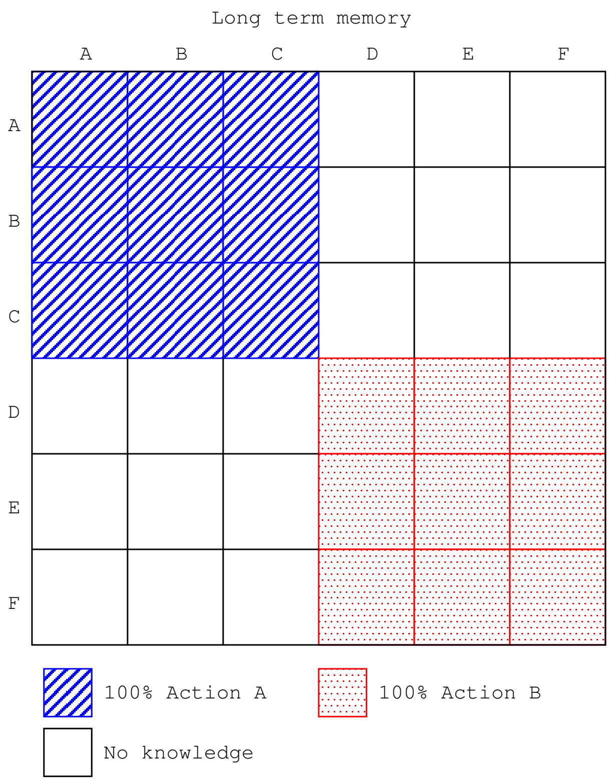


Figure 13.1: 1) The accumulated knowledge for action A and B are totally separated. 2) After several learning iteration the knowledge is exactly represented with two rules.

To analyze our genetic algorithm we start the evolution from scratch. Figure 13.2 shows the development of the determined rulebase. The fuzzy areas in the top left corner are covered within three iterations with exactly one rule. Because the other corner can not reached with the crossover operator the rulebase stay at this level until a mutation is happen. After 30 iterations a rule mutated which reaches the other state space. Within three more iteration this rule is also perfect evolved.

The example above was perfect solvable without any overlap because the equal classified fuzzy areas was arranged as a rectangles. The configuration in figure 13.3 (on the left) consists also of disjoint classified areas, but the optimal solution is not obviously. The optimal solution is between many disjunctive and a few overlapped rules. The found rules (visualized on the right) covers the configuration completely correct. To reduce the number of rules, the algorithm makes use of the empty space (rule Q and R).

Figure 13.4 illustrates the evolve of the rulebase for the above configuration. After only five iterations, the algorithm found the optimal rulebase to cover all 21 used fuzzy areas correctly.

13.2.2 Uncertain Decision Space

Up to now the configuration of the simulated environment was always disjoint. That means that each fuzzy area has accumulated only knowledge for one action. A rule which covers this area is therefore either correct or completely wrong.

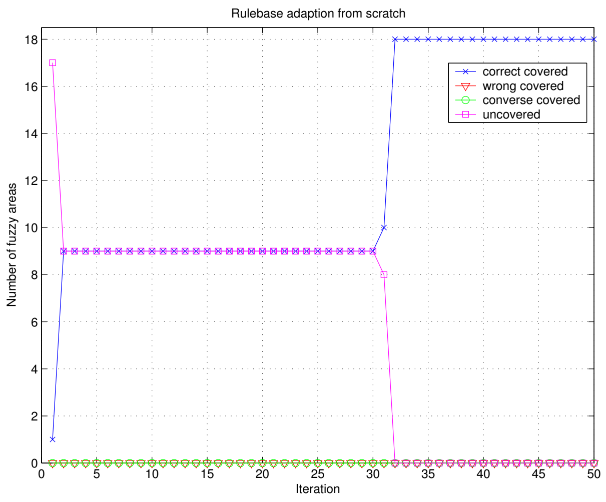


Figure 13.2: Fast evolution of the rulebase in a simulated disjoint environment

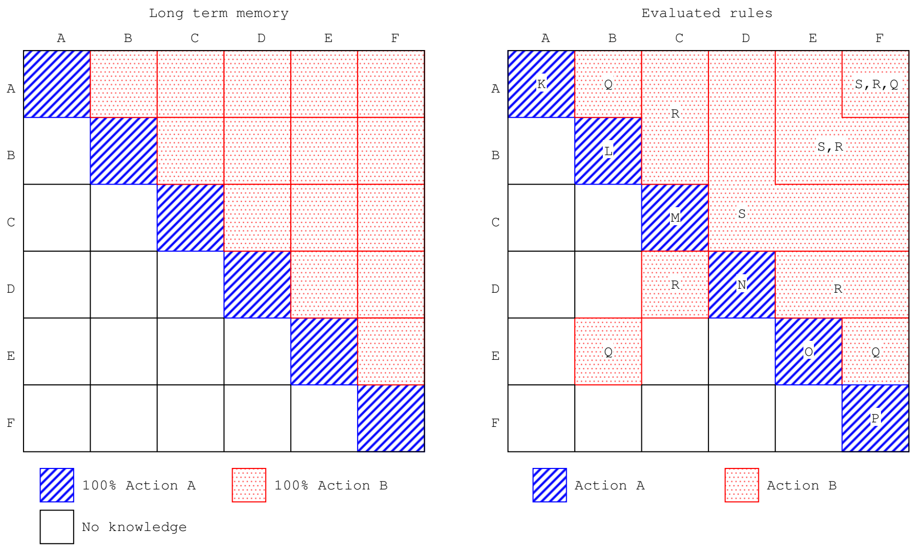


Figure 13.3: 1) The accumulated knowledge for action A and B are totally separated, but the action A is diagonally arranged. This can not be covered by only one rule. 2) After five learning iterations the knowledge is represented with nine rules. The A actions (diagonal) are covered by single rules. The rules to cover the B actions have some overlap to minimize the number of rules.

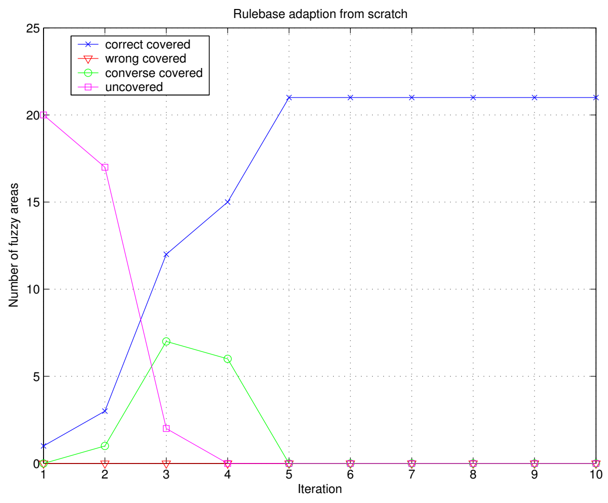


Figure 13.4: Fast evolution of the rulebase in a simulated environment

The configuration in figure 13.5 (left side) posses four uncertain fuzzy areas: (B,B), (B,C), (C,B) and (C,C). For these areas the accumulated knowledge represents the action A with $66.\bar{6}\%$ and the action B with $33.\bar{3}\%$. Although the classifier assigns the action A to these areas, a rule which covers them has an error. But after several iterations, the genetic algorithm extracts the three rules (on th right of figure 13.5) which represent the classified areas correctly.

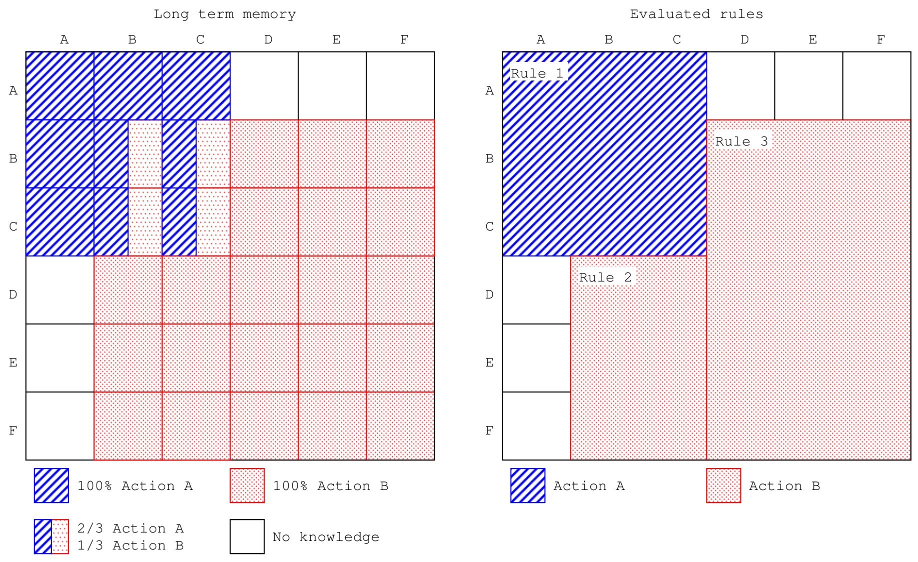


Figure 13.5: 1) The accumulated knowledge of the fuzzy areas (B,B), (B,C), (C,B) and (C,C) is ambiguous. The action A is more certain than the action B. So the classifier assigns the action A to this areas. 2) After several learning iteration the knowledge is exactly represented with three rules.

The evolution of the rulebase to cover this configuration shows an interesting behavior (Figure 13.6). After five iterations all areas are covered by a set of rules, but four fuzzy areas are covered with rules representing action A and B. A more detailed analyze confirms the assumption that these areas are the uncertain four. The algorithm found two rules, one for each possible action, whereby the uncertain space is covered by both. After further 15 iterations the rulebase is optimized by a mutation and finally there is a perfect coverage.

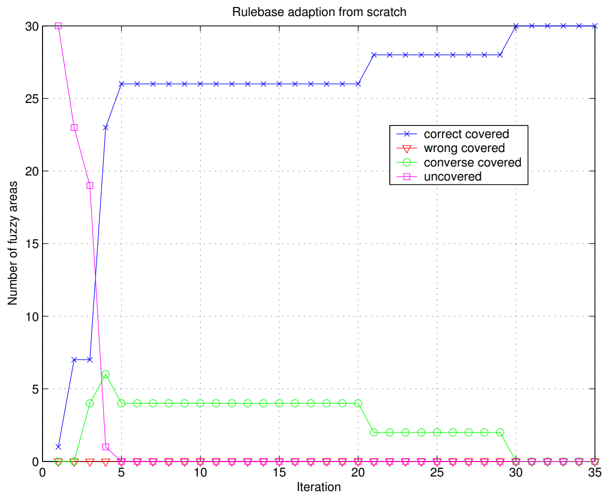


Figure 13.6: Evolution of the rulebase in a simulated uncertain decision environment

13.3 Results in a Real Environment

After a detailed analysis of the evolutionary algorithm in a simulated environment we discuss in this section results obtained when the algorithm was applied to a real environment. Tn the test phase, our intelligent controlling system was applied to three different rooms at the Institute of Neuroinformatics (INI). These rooms differ in size, shape, usage and number of inhabitants. Due to the short time period of our diploma thesis, we can only give here first results during normal operation. A long-term analysis has to be made over several weeks or rather more seasons. We are planning to set up such a test and evaluation period to obtain more data. In a first step a room should be just observed during normal usage. In a second step a second test has to be performed with the same persons that behave similar like in the previous test and the environment conditions should be more or less the same (e.g. in the winter, nearly same temperature and no snow that increases illumination). Only such a setup can give us evidences how good or bad our system performs in a longer time-constraint.

13.3.1 First Results With an Online Evolutionary Learning Based on a Long-Term Memory

As we have seen in a simulated environment, the evolutionary algorithm is able to learn a fuzzy ruleset based on a disjoint or mixed memory. The memory is represented with different fuzzy areas in which the system acts. In all other areas is the system never been and should therefore also not make a decision because it has no idea of its impact.

Usage of the State Space

During the first week of testing, we have seen that the system moves only in a subpart of the whole statespace which we expected with our assumption. The reason for that are mainly the physical constraints of input sensors which are sensing together and situations like `Blind = up, Outdoor Radiation = bright, Indoor Daylight = dark` can never be reached by the system. Another reason is also that we could only look at our test environment during a few weeks at wintertime which had almost the same weather conditions. An analysis during a whole year or maybe just half a year would be more substantial.

Though, table 13.1 shows clearly that the whole state space can never be reached by the system. Learning a rulebase which covers the full state space would be too much, because a rulebase which covers only reachable states would be sufficient.

Room	Effector	Input Sensors	Total Fuzzy Areas	Active Fuzzy Areas
Room 74	light 1	4	240	49 (0.20%)
Room 84	light 1	4	240	37 (0.15%)
Room 54	light 1	4	240	25 (0.10%)

Table 13.1: Relative usage of the fuzzy state space obtained in three rooms during an observation period of 2 weeks.

13.3.2 Online Rulebase Adaption

The fuzzy memory is constituted of different fuzzy areas which are weighted based on a accumulated rewards. The evolutionary algorithm has to learn a fuzzy rulebase on which a fuzzy logic controller is able to take decisions. The algorithm should converge fast if the memory itself has changed dramatically. The incorporation of new knowledge into the memory indeed can not happend from one to the other instance. It is adapting itself continuously with small steps.

Figures 13.7 and 13.8 in room 74 and 54 illustrate the online adaption of the rulebase to the current memory. The evolutionary algorithm is capable of adapting its rulebase in very few learning iterations to the current memory. It is able to find fuzzy rules for most of the fuzzy areas, but generates sometimes also wrong rules due to normal evolution.

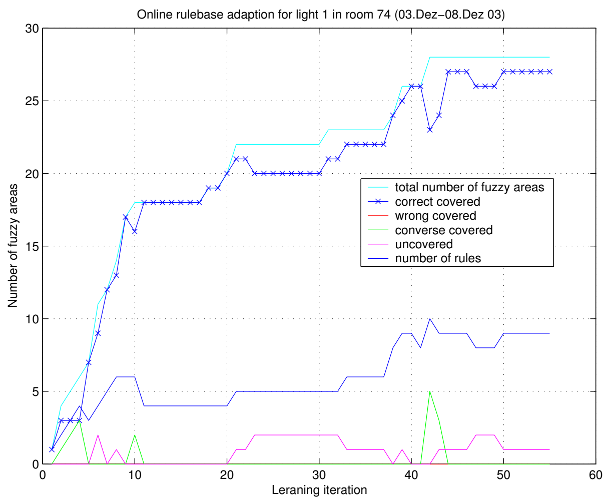


Figure 13.7: Rulebase adaption from scratch in a real environment (room 74). Notice the small number of rules although the set of fuzzy areas increases.

Size and Shape of Fuzzy Rulebase

The shape and size of a fuzzy rulebase learnt with an evolutionary algorithm is heavily dependent on the prespecified parameters of the algorithm. The most important part is the fitness function defined in 7.8 which judges each possible solution within the evolution.

The adaptive boosting algorithm modifies the distribution of inputs samples based on how sufficiently they are covered by a proposed solution. The evolutionary algorithm is invoked as long as samples are uncovered. By each invocation, the fittest solution will become part of the subsequent fuzzy rulebase. Therefore is the size directly dependent on how many times a new evolution is started.

As seen in figures 13.7 and 13.8 is the size of the fuzzy rulebase small if its is consistently adapted to the current memory. For all effectors in all three rooms, the rulebase was never bigger than 11 rules within the test period. But as we have also seen in the simulation results, if the rulebase is learnt from scratch it starts

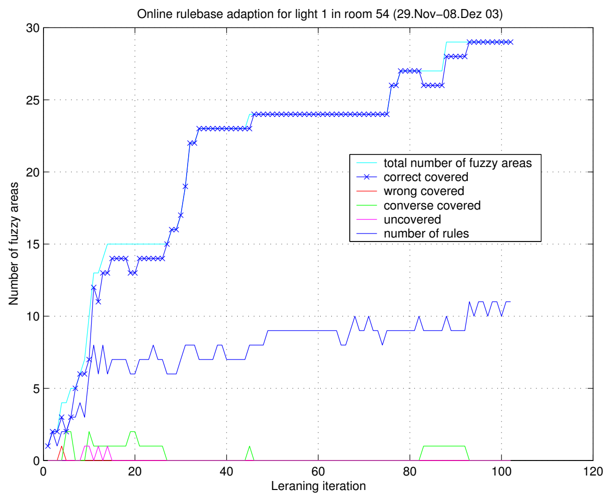


Figure 13.8: Rulebase adaption from scratch in a real environment (room 54). Notice the small number of rules although the set of fuzzy areas increases.

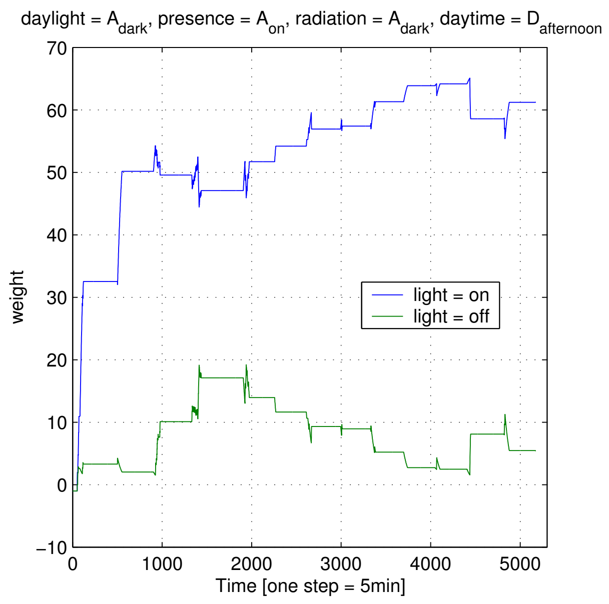
with a high number of rules. This is because we generate an initial rulebase based on the current memory if no rulebase was learnt or is available. The learning algorithm tries to combine afterwards those atomic fuzzy rules.

Behavior in Ambiguous Fuzzy Areas

The number and size of fuzzy areas is directly dependent on the design of the fuzzy membership functions. Figure 13.7 shows in a nice way that the learning algorithm is in fact able to learn almost a perfect rulebase but fails to cover the rulebase without conflicts.

Situations in which such a conflict can occur are critical changeovers from night to day, dark to bright or similar. Imagine, users are coming in the morning and switch on the light whereas the light stays off the other day. By regarding the weight of such fuzzy areas where conflicts occur, one can see that weights for contradictory actions are nearly the same. The left figure on 13.9 shows the trend of a specific fuzzy area where it is difficult to learn rule. This area would have after classification a small relative weight but the error factor of the fitness function decreases a rule which covers this area and also the adaptive boosting will focus more on this difficult areas. On the other side, the right figure of 13.9 shows a typical accumulated weight trend of a specific fuzzy area.

The evolutionary algorithm is able to find for such areas rules, but has to make a trade-off because the error and volume factor of the fitness of its fitness value can be very low. If the system is in an ambigous fuzzy state where the rulebase would have two active rules proposing different actions on the effector we are currently not setting the effector to the new state. Each solution evolved by the evolutionary algorithm will have the same power in the fuzzy interferencing process to fire and trigger an action. However, this process should take also the power of a firing rule into account during the defuzzification. We regard this as a voting-issues where we give different weights to different rules based on the relative importness. A rule which was found within the first evolution of one learning interaction is more important and should have more power than another that was determined in the last evolution. This issue has to be fixed in the future to enable a normal defuzzification.



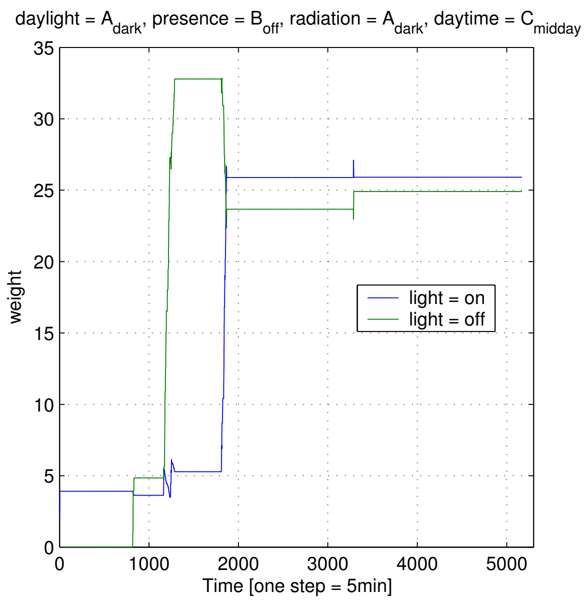


Figure 13.9: The left figure shows a typical trend of a fuzzy area. One action is represented with a higher weight because it occurred more often than the contrary. The right figure illustrates a fuzzy area where both weights are nearly the same and also with a high certainty. In such cases its really almost impossible to learn rules which does not have a conflict. The system can not really judge which action to take. We have to incorporate other preferences such as the cost for switching on the light is a little higher than the one to turn it off.

Part V

Conclusion and Future Work

Chapter 14

Conclusion

14.1 Multi Agent Framework

We implemented an multi-agent framework with a clear architecture which is supporting major issues in order to administrate and control a commercial building. These highly flexible, reliable and general services are distributed on several individual agents. The overall architecture of our multi-agent framework can be described in three abstraction layers:

- Bus abstraction layer
- Structure abstraction layer
- Learning and controlling layer

The framework is now able to handle several different bus types simultaneously whereas all share the same access interface. Thus, it is possible to get sensor information of different building buses or virtual software bus systems. The integration of virtual networks allows it to use also personalized software devices which can run on individual personal computers.

The structure abstraction layer manages a non-stationary structure of all available devices. It provides a dynamic modification of the structure and supports the concurrent adding or removing of devices during runtime.

To provide a high reliability to the framework, a recovery system forces periodically each running agent to make its current state persistant. Because of this, the system can recover itself from its last persistent state.

14.2 Learning and Controlling Architecture

We could prove that the learning and controlling architecture with two parallel units, which takes decisions based on different memories, is able to learn the behavior of different inhabitants. Due to the a temporary and a long-term memory we are able to learn temporary conditions without loosing knowledge which has been rewarded many times in the past.

First analysis has shown, that the comfort of the building's inhabitants can be increased. But it was not possible to test the long-term behavior of this architecture because of the time limit in which this diploma thesis has to be fulfilled. We are planning to test it during the next weeks in order to be really able to judge the performance over a longer period.

The proposed learning algorithms are able to learn either a temporary set of fuzzy rules or build up a long-term memory. The use of several small controlling units made it possible to learn online and reason in nearly realtime. By summarize the features we can argue that our control and learning architecture is able to:

- adapt itself based on the behavior of the inhabitants
- learn and take decisions only on a few fuzzy rules.
- incorporate new temporary knowledge and take decisions in nearly realtime
- adapts gradually a long-term rulebase with a fast evolutionary algorithm to the current long-term memory.

The controller itself is up to now not capable to reason on a dynamic set of input values although the framework supports it. Also the knowledge sharing between multiple controllers is not yet realized.

Chapter 15

Future

Within this chapter we are dealing with possible extentions and issues which we think have to be solved in the near future.

15.1 Multi Agent System

The multi-agent framework proved to offer flexible and still reliable services encapsulated in different agent. During normal operation we encountered several minor issues which have to be realized in order to apply the whole system on a more productive level:

- Enhance *EnergyAgent* with the feature to go in a suspended state or inhibit it to take any control during a specific time period. These issue was mainly the desire of some users which are doing sensitive research experiments where a increasing illumination can make experiments useless.
- Develop user interfaces which can show the current state of the whole system. This will be mainly used for the purpose of demonstrations.

15.1.1 ABLE

During the implementation of the current framework we came across several major problems within the ABLE framework. A distribution of all agents on several computers is possible, but the agent directory does not provide a reliable lookup service.

15.1.2 Simulation

The evaluation of the current controlling and learning architecture depends heavily on the applied environment. We think a simulated environment which approximates the behavior of a small subset of the overall building's structure would be useful in order to evaluate the long-term behavior faster. The small environment has to be analyzed first, before the relative dependency between input sensors and effectors are.

All agents must therefore be able to work with a virtual time within the whole system to allow a speed up of the simulation. We think here of an agent which distributes virtual steps in the multi-agent system.

15.2 Learning

The learning units are not yet compatible with a dynamic size of clusters on which they are running. One has to think of issues which occur if a part of the memory becomes invalid or will be extended.

With the dynamic structure it is possible that devices are moving from one cluster to the other (e.g. a person or mobile robot). If a cluster becomes empty the control agent should recognize it. It can either be suspended and wait for any new occupants in the future or shutdown itself.

Another issue which the dynamic structure implies is that knowledge becomes invalid at one specific point but would be used in another reasoning system that holds a new mobile device. We would like to analyze the extraction of knowledge and if possbile also sharing with others. This could also be used if our system will be deployed to a new cluster where no particular knowledge is available. The size and shape of the cluster is indeed similar to another cluster in which knowledge was already learnt.

15.2.1 Fuzzy Membership Functions

As we have seen in our analysis a accurate design of the fuzzy membership functions is a really important task. The whole learning algorithm relies on this design. The current fuzzy membership functions are an approximation which were made by our predecessors in [RS02]. However, we believe the accuracy of the fuzzy reasoning could be increased if these are defined based on statistical conditions which regard also different seasons.

It would be thinkable to tune them also with a learning algorithm in order to adapt them to the current used range of the sensor. A normal sensor will sense for example during the wintertime in another range than in summertime.

15.2.2 Fuzzy Voting

As we have discussed in our results it is essential to incorporate a fuzzy voting system. Due to imprecise knowledge in the long-term memory it is possible that contrary fuzzy rules will be found which all have the same weight in the fuzzy interferencing. A fuzzy rule which is hard to learn because of such imprecise memory areas must have a smaller voting weight as one that was easily learnt. By regarding also the other mentioned problems of the fuzzy interference engine provied in the ABLE framework, it is maybe worth to take also other engines into consideration or implement an own one.

15.2.3 Personalization

The identification of individual users is possible with the usage of a novel personal PC agent. The learning algorithm can process different presence signals but has not yet an evaluation of the individual desire of each user.

Part VI

Appendix

Appendix A

Brief Explanation of Fuzzy Logic Expressions and Notation

Within this document we are using a fuzzy notation which is largely similar to the ones in [Ful00], [CCSZ21] and [CZ16]. The notation is mainly adapted to the one in [RJD04]. We give here no further introduction into the theory of fuzzy logic. For more information, consult [Ful00] or [RS02] which has a fuzzy primer.

Every fuzzy variable consists of a number of fuzzy sets. Every of these fuzzy sets is identified by a unique name (label) which associates a human readable string with a formal definition of a fuzzy set. All fuzzy sets together define a set of membership functions.

The set of all labels of a fuzzy variable X_0 is denoted as $\mathcal{L}$. Example: $\mathcal{L}(X_0) = \{cold, normal, warm, hot\}$ (which is an example for a temperature fuzzy variable). For a discrete variable Y $\mathcal{L}(Y)$ is the set of all possible values of the discrete variable. Example: $\mathcal{L}(Y) = \{0.0, 1.0, 2.0\}$.

A ruleset consists of a finite number of rules $\mathcal{R}$ (see equation A.1). r specifies the number of rules of the ruleset (equation A.2).

$$\mathcal{R} = \{R_1, R_2, \ldots, R_r\} \tag{A.1}$$

$$r = |\mathcal{R}| \tag{A.2}$$

A rule (equation A.3) consists of a set of sets which specify the labels for every fuzzy variable that need to be true such that the rule becomes true. y_j is a fuzzy or discrete output value of the output variable: $y_j \in \mathcal{L}(Y)$.

$$R_i = ((E_0 \subset \mathcal{L}(X_0), E_1 \subset \mathcal{L}(X_1), \ldots, E_r \subset \mathcal{L}(X_{N-1})), y_j) \tag{A.3}$$

N_i (equation A.4) denotes the number of fuzzy input variables of rule R_i.

$$N_i = |\{X_0, X_1, \ldots, X_{N-1}\}| \tag{A.4}$$

Only sets E_i are valid that satisfy the condition $E_i \in \mathcal{P}(\mathcal{L}(X_i))$, where X_i is a fuzzy variable. In a more human readable form a rule looks like this:

R_i : if X_0 is E_0 and X_1 is E_1 then Y is y_j

The set of conditions E_i of a particular fuzzy variable X_i is meant to be inclusive which means that every $l_{ik} \in E_i$ makes the condition true. l_{ik} denotes a single label of the fuzzy variable X_i.

E_{ij} denotes the set E_j of rule R_i.

Example:
Fuzzy variables: X_0 and X_1. $N = 2$
Labels: $\mathcal{L}_0 = \{A, B, C, D, E\}$, $\mathcal{L}_1 = \{F, G, H\}$
$\mathcal{R} = \{R_0, R_1\}$
Output variable is Y, $\mathcal{L}(Y) = \{y_0, y_1, y_2\}$
$R_0 = ((\{A, B, E\}, \{F, H\}), y_0)$
$R_1 = ((\{D, E\}, \{G\}), y_2)$
$E_{00} = \{A, B, E\}$,$E_{10} = \{D, E\}$,$E_{11} = \{G\}$
The rules of this example could be expressed in boolean notation as:
R_0 : if (X_0 is A or B or E) and (X_1 is F or H) then Y is y_0
R_1 : if (X_0 is D or E) and (X_1 is G) then Y is y_2

Appendix B

Analysis and Extentions of the Previous Learning Algorithm

B.1 Overview

The developed learning and controlling algorithm of the previous project [RJD04] is designed to learn user behavior from sparse data that is acquired in a non-stationary environment. The learning algorithm produces a maximal structure fuzzy rulebase and adapts itself continually. For each new user instruction it use this new knowledge to strengthen or modify the current rulebase.
We applied this algorithm to our new framework. During the reimplementation we extended and modified the algorithm. This chapter gives a brief overview about these extentions.

B.2 Algorithm Extentions

Due to the fact that the algorithm learns based on the really sparse user interactions and it is a one shot learning, it is important to manage the gathered training sets carefully. Driven by this aspect we extended the two central function of the algorithm: the reduction and the amplification of rules. We do not present here the effective implemented algorithm, but rather the problem with the antecedent approach and the concept of our extention. The detailed realization is more complex and described in the JavaDoc in the source code itself.

B.2.1 Considering of Physical Constraints

A fuzzy variable consists of several fuzzy sets (also called fuzzy membership functions). Each of this fuzzy sets represent a range of a physical value, so they have a physical order based on their defined range. We believe that the user's behavior is also dependent on this physical constraints. To consider that, we permit only rules which continuously cover the fuzzy states of an input dimension. That means that it is not allowed to create a rule where the condition for a dimension contains for example only the first and the fourth state. In that case the definition range of this rule is not continuously.

So the valid fuzzy rule

```
presence IS A_on AND radiation IS A_dark, B_middle AND
  daytime IS A_night1, B_morning, E_evening, F_night2
THEN light IS A_on;
```

must be divided into two separate rules:

```
IF presence IS A_on AND radiation IS A_dark, B_middle AND daytime IS A_night1, B_morning
 THEN light IS A_on;

IF presence IS A_on AND radiation IS A_dark, B_middle AND daytime IS E_evening, F_night2
 THEN light IS A_on;
```

B.2.2 Minimal Reduction with Minimal Knowledge Loos

In the case if a new user instruction R_i is in conflict with an already existing rule R_d of the rulebase $\mathcal{R}_{def}$, the rule R_d has to be reduced until the conflict is solved.
Up to now the reduction algorithm removed in each input dimension of the conflicted rules R_d this states which are also defined in the user instruction R_i. This eliminates all conflicts but it is not the minimal possible reduction. The example in figure B.1 illustrates this on the basis of a two dimensional rule space with an existing rule. The blue circle represents the converse user instruction R_i. To solve this conflict on the fuzzy area (D,C), the algorithm removes the state D from the first dimension (reduction A, visualized as an arrow in figure B.1) and the state C from the second dimension (reduction B).
With this proceeding we loose more knowledge than required. The conflict can already be solved, if we reduce only one of dimension. Furthermore some reduction cross out gathered training sets. In our example (figure B.1) the reduction A removes less rule space but more knowledge, whereas the reduction B flushes no knowledge but reduce the rule size more than the reduction A.

Our new reduction algorithm consider this both aspects in the following order:

1. Keep as much training samples as possible
2. Reduce the rule size as less as possible

We realize this by reduce each possible reduction and choose the reduction which keeps the most knowledge. If there are more than one solution, we select the reduction which has the higher rule volume.

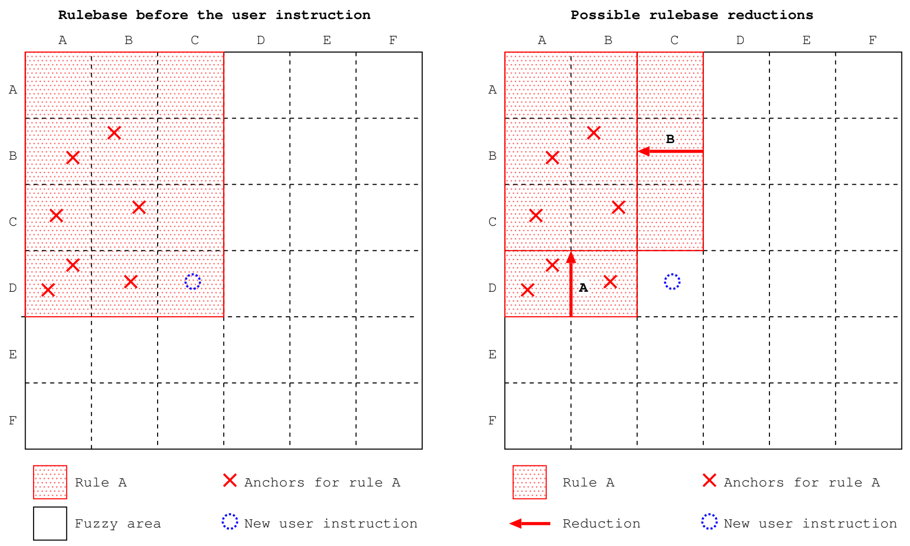


Figure B.1: Example of a reduction.

B.2.3 Maximal Local Amplification

The algorithm amplifies the knowledge of a new user instruction R_i in the rule space to covers the whole state space as soon as possible. This amplification was realized with the algorithm proposed in [CCSZ21]. The simplified concept of this amplification algorithm works as following:

```
FOR EACH dimension d DO
  amplifyDimesionToMaximum(d)
END
```

This solution of amplification is quite fast, but does not find the maximal amplification. The reason is that the algorithm amplify firstly the first dimension as much as possible. In the next iteration it tries to amplify the second dimension based on the previous found amplification of the first dimension. In our example (Figure B.2) this algorithm would not be able to find the maximal blue stripped amplification, because the first iteration expands the instruction R_i (D,C) in the first dimension to the maximum amplification ({A,B,C,D,E},C). Based on this amplification there is no expansion possible for the second dimension.
Further more this algorithm privilege the first dimension. Therefore the order of the input dimensions have an crucial effect which we can not judge or define in advance.

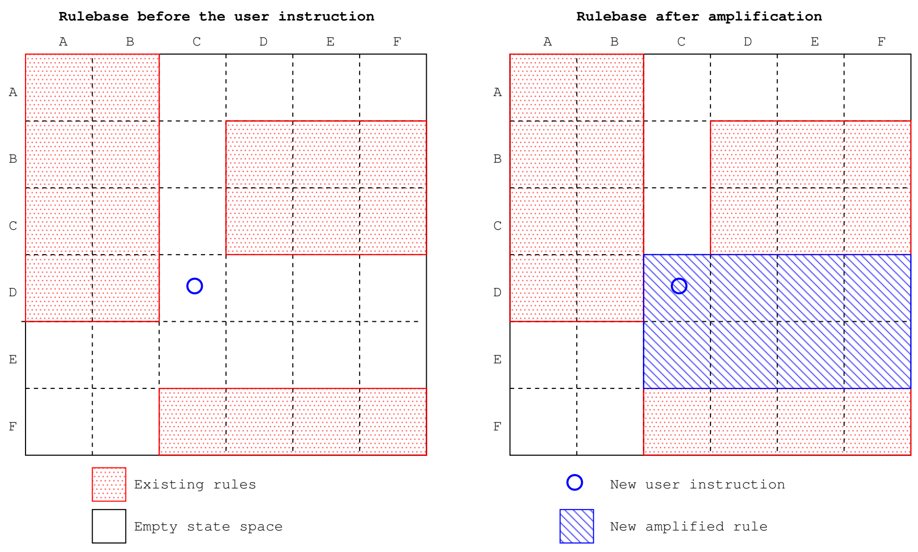


Figure B.2: An example of the amplification of a user instruction to the maximum possible size.

Our implementation of the amplification algorithm tries for each dimension to achieve the maximal expansion. We apply the amplified rule with the maximal volume to the definitive rulebase $\mathcal{R}_{def}$.

B.3 Conclusion

After we had adapted the algorithm with the above described modifications and extensions to our new framework, we started to control several rooms for two weeks to collect information about the behavior. In this section we list our conclusions due to the analyze of the gathered data.

B.3.1 Sparse User Interaction

As also described in our term project, the user interaction is really sparse (Chapter 4). Especially in a well controlled room, the user should never take a decision.
Furthermore, it is not possible for the user to produce a reward instruction. If the user changes the state

of an effector, it is always a punishment of the current rulebase because the actual active rule suggests the converse action.
Based on that perception, we can say that the user has no possibility to introduce a positive feedback to the system. It is necessary the find an additional way to get this positive feedbacks from the user.

B.3.2 Unused Rules

Within this two weeks the learning algorithm produced 40 rules whereas only five are regulary used. The other rules are retreated into an unused space which will not be reached by the system again. In general we can say that the current algorithm produces a small set of used rules, but keeps a huge among of unreachable rules.

B.3.3 One Shot Learning

Because the algorithm is a one shot learning algorithm, it has no knowledge about statistic values. A new converse user instruction replaces (or reduces) an existing rule also if the rule is strengthened by several training samples. This behavior makes sense because a learning algorithm has to respect the current user behavior (at least temporary). But on the other side, it inhibits the learning of a long term behavior.
We believe that it is necessary to design a learning algorithm in two parts. One learns temporary behavior of the user and control a room based on that knowledge. But this knowledge is also only temporary and should be forgotten after a while. The other part learns the general long term behavior. The long term knowledge can be permanent, but it must also be able to relearn behaviors.

Appendix C

Agent Building and Learning Environment

Agent Building and Learning Environment (ABLE) is a Java framework, component library, and productivity tool kit for building intelligent agents using machine learning and reasoning. It provides mechanisms to distribute agents to different computers. ABLE is a research project of the IBM T.J. Watson Research Center and is offered by IBM on alphaworks.

Within our framework we are using only a small subset of the whole ABLE services:

- Agent services
- Messaging transport system
- Fuzzy engine
- Fuzzy rule sets

C.1 Problems

Our framework uses several different specific ABLE services whereas most of them work more or less reliable. However, there are some crucial bugs in their framework which we fixed with an own solution.We give here a brief overview of current bugs and explain also our solution to them.

Directory Service Returns Null Pointer

ABLE supports the distribution of agents on several platforms. The directory service provides a query interface to search agents with a directory lookup. This service is used by several agents and works only well, if all agents are running on the same platform as lifecycle, naming, directory and transport service. But if an new agent is started on a different platform (which can still be on the same computer within another virtual machine) the query returns for some agents a null pointer instead of the agent description. This can be the case for remote agents as well as for local ones.

Due to this problem e.g. the RecoveryAgent is not able to backup all agents or the AgentMonitor is not able the give information about all running agents distributed on several platforms.
This bug is still unsolved!

Dynamically Adding of Rules

ABLE provides to create new fuzzy rules and to add it to the rulebase at runtime. That works well without any exception, but the new rule will never fire. It looks that it rulebase is not reinitialized correctly after adding a new rule. We solved the problem with a work around: after adding the new rule, the rulebase will be saved into an ARL file and then reparsed. Afther that, the rulebase will be initialized correctly.

Memory Leak of AbleRuleSet

It seems that the class AbleRuleSet has a memory leak. If we create and release instances of this class more than 260 times, we get an OutOfMemory exception. As we are creating just one AbleRuleSet on startup it is not really a problem up to now.

Mystic ARL Comment Modification

Each block, variable or rule of an ARL file contains an comment. Because the parser and this ARL-file writer does not work exactly identical, the comment string gets with each read/write cycle a white space more at the beginning and end. We simply remove this white spaces after reloading the ARL file.

Internal Used Fuzzy Sets Are Visible

Each fuzzy variable is represented by the ABLE class AbleFuzzyVariable. With the method `getFuzzySets()`, we get a list of all defined fuzzy sets. But this list also includes some internal fuzzy sets which are generated from the ABLE framework. Fortunately the name of this additional sets end always with 'NPS' and it is easy to filter them.

Order of Fuzzy Sets Is Not Defined

The order of the returned fuzzy sets does not match with the definition in the ARL file. Each fuzzy set starts therefore with a capital letter which defines the real position. We know therewith that the fuzzy set Ć_middayśhould be placed at the third position.

C.2 Fuzzy OR Operator Is Not Supported

ABLE does still not provide the fuzzy OR operator. We use instead an AND condition which can be seen as a approximation. Within fuzzy logic it is not exactly true like in normal boolean logic. We used the approach from the predecessor work [RS02].

Part VII

Glossary and Bibliography

Glossary

ABI	Adaptive Building Intelligence
ABLE	Agent building and learning network
Agent	An *agent* is a computer system that is situated in some environment, and that is capable of *autonomous action* in this environment in order to meet its design objectives ([WJ95]).
AHA	Adaptive Home Automation
AI	Artificial intelligence
ARL	ABLE rule language
DAI	Distributed artificial intelligence
EA	Evolutionary Algorithm
EIB	European installation bus
ETH	Swiss Federal Institute of Technology
FIPA	Foundation for Intelligent Physical Agents
FLC	Fuzzy Logic Controller
GA	Genetic Algorithm
GFLC	Genetic Fuzzy Logic Controller
HSR	University of applied science Rapperswil, Switzerland
IB	Intelligent Building
IE	Intelligent Environment
INI	Institute of Neuroinformatics, University and ETH Zurich, Switzerland
JAS	Java Agent Specification
LNS	Lon Network Server
LonWorks	Field bus network protocol / standard
MAS	Multi agent system
MDA	Message Distribution Agent
ML	Machine learning
UNIZH	University of Zurich, Switzerland

Bibliography

[Age] Agent building and learning environment (able). http://www.alphaworks.ibm.com/tech/able.

[ARL02] IBM Thomas J. Watson Research Center. *ABLE Team, Able Rule Language (ARL) Documentation*, 2002.

[BDY99] Magnus Boman, Paul Davidsson, and Håkan L. Younes. Artificial decision making under uncertainty in intelligent buildings. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 65–70, 1999. ftp://ftp.dsv.su.se/users/mab/uai99.ps.

[Bon96] Andrea Bonarini. Evolutionary Learning of Fuzzy rules: competition and cooperation. In W. Pedrycz, editor, *Fuzzy Modelling: Paradigms and Practice*, pages 265–284. Norwell, MA: Kluwer Academic Press, 1996.

[CCSZ21] J. L. Castro, J. J. Castro-Schez, and J. M. Zurita. Learning maximal structure rules in fuzzy logic for knowledge acquisition in expert systems. *Fuzzy Sets and Systems*, 101:331–342, 1999/2/1.

[CZ16] J. L. Castro and J. M. Zurita. An inductive learning algorithm in fuzzy systems. *Fuzzy Sets and Systems*, 89:193–203, 1997/7/16.

[DB00] Paul Davidsson and Magnus Boman. Saving energy and providing value added services in intelligent buildings: A MAS approach. In *ASA/MA*, pages 166–177, 2000.

[eur] European Installation Bus. http://www.eiba.com.

[FIP02] FIPA Abstract Architecture Specification (XC00001K). Technical report, Foundation For Intelligent Physical Agents, Geneva, Switzerland, 2002.

[FS96] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.

[Fuj02] Fujitsu Laboratories. JAS Agent Services (JSR-87) Specification, 2002.

[Ful00] Robert Fuller. Neural fuzzy systems. In *Advances in Soft Computing Series*. Springer-Verlag, Berlin/Heildelberg, 2000. ISBN : 3-7908-1256-0.

[GH97] A. Gonzalez and F. Herrera. Multi-stage genetic fuzzy systems based on the iterative rule learning approach, 1997.

[HCCC03] Hani Hagras, Victor Callaghan, Martin Colley, and Graham Clarke. A hierarchical fuzzy-genetic multi-agent architecture for intelligent buildingsnext term online learning, adaptation and control. *Information Sciences*, 150(1-2):33–57, 2003.

[HKS98] F. Hoffmann, T. Koo, and O. Shakernia. Evolutionary design of a helicopter autopilot, 1998.

[Hof] Frank Hoffmann. Boosting a genetic fuzzy classifier.

[Hof98] Frank Hoffmann. Incremental tuning of fuzzy controllers by means of an evolution strategy. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 843–851, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[iLO] LonWorks Networking Platform. http://www.echelon.com/products/internet/ilon1000.htm.

[ins] Instant messaging service icq. http://web.icq.com.

[Jav] Java genetic algorithm package. http://jgap.sourceforge.net/.

[LNS] LNS HMI Developer's Kit for the Java Platform. http://www.echelon.com/products/development/lns_java/.

[loc] LonWorks Networking Platform. http://www.echelon.com/products/lonworks/.

[Moz98] M.C. Mozer. The Neural Network House: An Environmnet that Adapts to its Inhabitants. In M. Coen, editor, *Proceedings of the American Association for Artificial Intelligence Spring Symposium on Intelligent Environments*, pages 110–114, Menlo Park, CA, 1998. AAAI Press.

[Moz99] M.C. Mozer. An intelligent environment must be adaptive. *IEEE Intelligent Systems*, 14(2), 1999.

[RJD04] U. Rutishauser, J. Joller, and R. Douglas. Control and learning of ambience by an intelligent building. *IEEE Transactions on System,Man and Cybernetics Part A, Submitted*, 2004.

[RS02] U. Rutishauser and A. Schaefer. Intelligent buildings – a multi-agent approach. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2002.

[TZ03a] J. Trindler and R. Zwiker. Adaptive building intelligence – an approach to adaptive discovery of functional structure. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2003.

[TZ03b] J. Trindler and R. Zwiker. *Java Documentation of ABI framework*, 2003.

[TZR$^+$03] J. Trindler, R. Zwiker, U. Rutishauser, J. Joller, and R.Douglas. Discovery of logical structures in a multisensor environment based on sparse events. In *Proceedings of Workshop on Ambient Intelligence, 8th National Congress of Italien Association for Artificial Intelligence, Pisa*, 2003.

[WJ95] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[Zho90] H. Zhou. CSM: A Computational Model of Cumulative Learning. *Machine Learning*, 5:383–406, 1990.